


Sage para Estudiantes de Pregrado

Gregory V. Bard

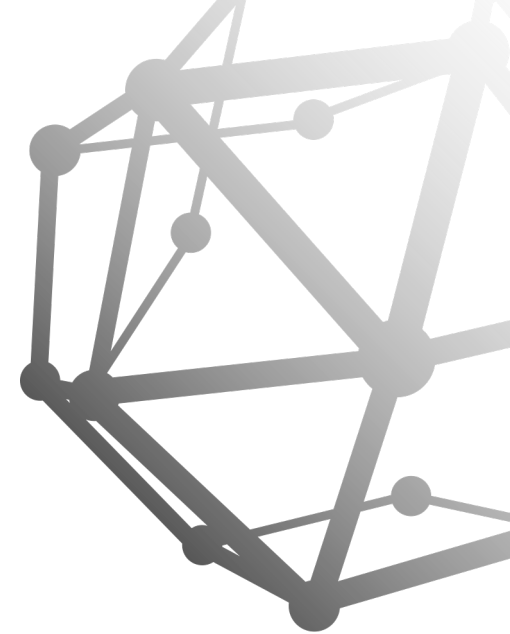
 <https://orcid.org/0000-0001-9460-6086>
Menomonie, Wisconsin, United States
bardg@uwstout.edu

Traducción al español:

Diego Sejas Viscarra

 <https://orcid.org/0000-0002-0368-2161>
Cochabamba, Bolivia
dsejas.math@pm.me

Con el cariño más grande, dedico este libro a mi papá, quien ha sido mi mayor guía y compañía en la vida; que ha recorrido conmigo los largos y oscuros caminos del subconsciente, y ahora comparte conmigo la luz en la cima más alta. No lo habría podido hacer sin ti.



Índice general

Prefacio	XIII
Agradecimientos del traductor	XVII
Agradecimientos del autor (del año 2014)	XIX
Special thanks / Agradecimientos especiales	XXIII
Introducción. Cómo usar este libro.	XXV
Capítulo 1. ¡Bienvenido a Sage!	1
1.1. Usando Sage como una calculadora	1
1.2. Usando Sage con funciones elementales	3
1.3. Usando Sage para trigonometría	8
1.4. Usando Sage para graficar en dos dimensiones	9
1.4.1. Controlando el área de visualización de un gráfico	13
1.4.2. Superponiendo varias gráficas en una sola imagen	16
1.5. Matrices y Sage, parte uno	19
1.5.1. Una primera experiencia con matrices	19
1.5.2. Complicaciones al convertir sistemas lineales	20
1.5.3. Obteniendo la RREF en Sage	21
1.5.4. Resumen básico	24
1.5.5. La matriz identidad	24
1.5.6. Un reto de práctica para el lector	25
1.5.7. Matrices de Vandermonde	25
1.5.8. Los casos semirraros	26
1.6. Definiendo nuestras propias funciones en Sage	28
1.7. Usando Sage para manipular polinomios	32
1.8. Usando Sage para resolver problemas simbólicamente	35
1.8.1. Resolviendo ecuaciones con una variable	35
1.8.2. Resolviendo ecuaciones con varias variables	36

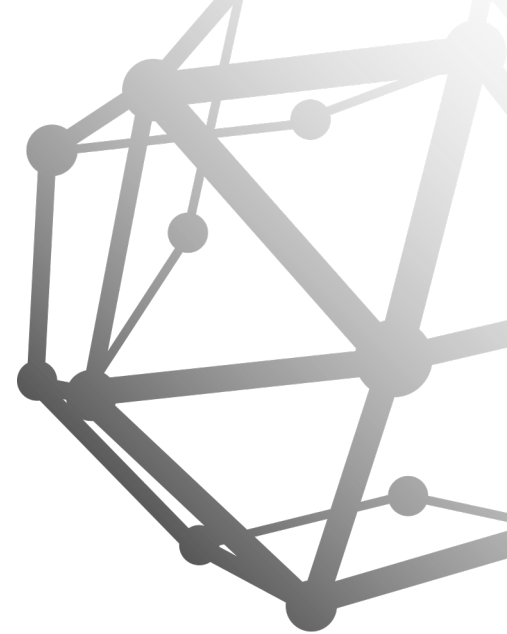
1.8.3. Sistemas de ecuaciones lineales	37
1.8.4. Sistema de ecuaciones no lineales	37
1.8.5. Casos avanzados	41
1.9. Usando Sage para resolver problemas numéricamente	41
1.10. Consiguiendo ayuda cuando la necesitemos	44
1.11. Usando Sage para calcular derivadas	46
1.11.1. Graficando $f(x)$ y $f'(x)$ juntas	47
1.11.2. Derivadas de orden superior	48
1.12. Usando Sage para calcular integrales	48
1.13. Compartiendo los resultados de nuestro trabajo	57
1.14. Los varios usos del comando <code>show()</code>	59
1.15. Un detalle técnico sobre funciones en Sage	61
Capítulo 2. Proyectos divertidos usando Sage	65
2.1. Microeconomía: Determinando el precio de venta	66
2.2. Biología: Arterias obstruidas y la Ley de Poiseuille	70
2.3. Optimización industrial: Transportando taconita	72
2.4. Química: Balanceando reacciones con matrices	74
2.4.1. Los fundamentos	74
2.4.2. Cinco ejemplos interesantes	75
2.4.3. La forma lenta: derivando el método	75
2.4.4. Balanceando de la manera rápida	77
2.5. Física: Projectiles balísticos	78
2.5.1. Un primer ejemplo de trayectorias balísticas	78
2.5.2. Un segundo ejemplo de trayectorias balísticas	80
2.5.3. Fuego contra-batería	81
2.5.4. El desafío: Un cohete multietapa	82
2.6. Criptología: El ataque $p - 1$ de Pollard contra RSA	82
2.6.1. Los fundamentos: Números B -lisos y $B!$	83
2.6.2. La teoría detrás del ataque	84
2.6.3. Calculando $2^{B!} \bmod N$	85
2.6.4. El desafío: Hacer que todo esto ocurra en Sage	86
2.6.5. Protecciones contra el ataque factorial de Pollard	86
2.7. Miniproyecto sobre gráficas vectoriales de un campo eléctrico	87
Capítulo 3. Técnicas avanzadas de graficación	89
3.1. Anotando gráficos por claridad	89
3.1.1. Etiquetando los ejes de los gráficos	89
3.1.2. Cuadriculados y gráficas estilo calculadora	90
3.1.3. Añadiendo flechas y texto	92
3.1.4. Graficando una integral	93
3.1.5. Líneas punteadas y quebradas	94
3.2. Gráficas de algunas funciones hiperactivas	95
3.3. Gráficas polares	96
3.3.1. Ejemplos de gráficas polares	97
3.3.2. Problemas que pueden ocurrir ocasionalmente	98
3.4. Graficando una función implícita	100
3.5. Gráficas de contorno y conjuntos de nivel	102
3.5.1. Una aplicación a la termodinámica	104
3.5.2. Aplicación a la microeconomía (ecuaciones de Cobb-Douglas)	107
3.6. Gráficas paramétricas 2D	108

3.7. Gráficas de campos vectoriales	110
3.7.1. Gradientes y gráficas de campos vectoriales	111
3.7.2. Una aplicación de la física	112
3.7.3. Gradientes vs. gráficas de contorno	115
3.8. Gráficas log-log	116
3.9. Situaciones raras	117
 Capítulo 4. Características avanzadas de Sage	 123
4.1. Usando Sage con ecuaciones y funciones multivariadas	123
4.2. Trabajando con fórmulas grandes en Sage	124
4.2.1. Finanzas personales: hipotecas	125
4.2.2. Física: Gravitación y satélites	128
4.3. Derivadas y gradientes en cálculo multivariado	129
4.3.1. Derivadas parciales	129
4.3.2. Gradientes	130
4.4. Matrices en Sage, parte dos	130
4.4.1. Definiendo algunos ejemplos	130
4.4.2. Multiplicación y exponenciación matricial	131
4.4.3. Resolviendo sistemas por izquierda y por derecha	132
4.4.4. Matrices inversas	134
4.4.5. Calculando el núcleo de una matriz	135
4.4.6. Determinantes	137
4.5. Operaciones vectoriales	137
4.6. Trabajando con enteros y Teoría de Números	140
4.6.1. El máximo común divisor y el mínimo común múltiplo	140
4.6.2. Más acerca de los números primos	142
4.6.3. Acerca de la función phi de Euler	142
4.6.4. Los divisores de un número	144
4.6.5. Otro uso para tau	145
4.6.6. Aritmética modular	146
4.6.7. Lectura adicional sobre teoría de números	147
4.7. Algunos comandos menores de Sage	147
4.7.1. Redondeo, pisos y techos	147
4.7.2. Combinaciones y permutaciones	148
4.7.3. Las funciones trigonométricas hiperbólicas	149
4.8. Calculando límites expresamente	149
4.9. Gráficas de dispersión en Sage	151
4.10. Haciendo nuestras propias regresiones en Sage	154
4.11. Calculando en octal, binario y hexadecimal	156
4.12. ¿Puede Sage resolver el Sudoku?	157
4.13. Midiendo la velocidad de Sage	158
4.14. Números gigantes y Sage	158
4.15. Usando Sage y \LaTeX , parte uno	159
4.16. Matrices en Sage, parte tres	160
4.16.1. Introducción a autovectores	160
4.16.2. Encontrando autovalores de manera eficiente en Sage	163
4.16.3. Factorizaciones matricial	164
4.16.4. Resolviendo sistemas lineales de forma aproximada con mínimos cuadrados	164
4.17. Calculando los polinomios de Taylor o de MacLaurin	166
4.17.1. Ejemplos de Polinomios de Taylor	167

4.17.2.	Una aplicación: Entendiendo cómo cambia g	168
4.18.	Minimizaciones y multiplicadores de Lagrange	169
4.18.1.	Optimización sin restricciones	170
4.18.2.	Optimización restringida por medio de multiplicadores de Lagrange	171
4.18.3.	Un ejemplo de multiplicadores de Lagrange en Sage	171
4.18.4.	Algunos problemas aplicados	173
4.19.	Sumas infinitas y series	174
4.19.1.	Verificando identidades de sumatorias	175
4.19.2.	La serie geométrica	176
4.19.3.	Usando Sage para guiar una demostración con sumatorias	177
4.20.	Fracciones continuas en Sage	178
4.21.	Sistemas de desigualdades y Programación Lineal	179
4.21.1.	Un ejemplo simple	179
4.21.2.	Características convenientes en la práctica	181
4.21.3.	El poliedro de un programa lineal	183
4.21.4.	Programación lineal entera y variables booleanas	183
4.21.5.	Lectura adicional sobre programación lineal	183
4.22.	Ecuaciones diferenciales	184
4.22.1.	Algunos ejemplos sencillos	184
4.22.2.	Un problema con valor inicial	186
4.22.3.	Graficando un campo de pendientes	188
4.22.4.	El problema del torpedo: Trabajando con parámetros	189
4.23.	Transformadas de Laplace	191
4.23.1.	Transformando de $f(t)$ a $L(s)$	191
4.23.2.	Calculando la Transformada de Laplace de “la forma larga”	192
4.23.3.	Transformando de $L(s)$ a $f(t)$	193
4.24.	Cálculo vectorial en Sage	194
4.24.1.	Notación para funciones con valores vectoriales	194
4.24.2.	Calculando la matriz Hessiana	195
4.24.3.	Calculando el Laplaciano	196
4.24.4.	La matriz Jacobiana	197
4.24.5.	La divergencia	198
4.24.6.	Verificando una vieja identidad	199
4.24.7.	La rotacional de una función con valores vectoriales	200
4.24.8.	Desafío: Verificar algunas identidades sobre la rotacional	203
4.24.9.	Integrales múltiples	203
4.25.	Usando Sage y \LaTeX , parte dos	204
4.25.1.	El paquete Sage \TeX	204
4.25.2.	Usando el paquete Sage \TeX	205
4.25.3.	La documentación de Sage \TeX	208
Capítulo 5.	Programando en Sage y Python	209
5.1.	Repetición sin fastidio: El bucle <code>for</code>	210
5.1.1.	Usando Sage para generar tablas	210
5.1.2.	Formateando cuidadosamente la salida	211
5.1.3.	Usando listas arbitrarias	212
5.1.4.	Bucles con acumuladores	212
5.1.5.	Usando Sage para encontrar un límite numéricamente	213
5.1.6.	Bucles <code>for</code> y Polinomios de Taylor	215
5.1.7.	Si el lector ya sabe programar...	216

5.1.8. Si el lector aún no sabe programar...	216
5.2. Escribiendo subrutinas	216
5.2.1. Un ejemplo: trabajando con monedas	217
5.2.2. Desafío: Una caja registradora	219
5.2.3. Un ejemplo: Diseñando acuarios	219
5.2.4. Desafío: Un silo cilíndrico	221
5.2.5. Combinando bucles y subrutinas	221
5.2.6. Otro desafío: Sumando una sucesión	222
5.3. Bucles y el Método de Newton	222
5.3.1. ¿Qué es el Método de Newton?	222
5.3.2. El Método de Newton con un bucle <code>for</code>	224
5.3.3. Probando el código	225
5.3.4. Representación numérica vs. exacta	227
5.3.5. Trabajando con parámetros opcionales y mandatorios	227
5.3.6. Devolviendo un valor y anidando subrutinas	229
5.3.7. Desafío: Encontrar rectas tangentes paralelas	231
5.3.8. Desafío: El Método de Halley	232
5.4. Una introducción al control de flujo	232
5.4.1. Control de verbosidad	232
5.4.2. Interludio teórico: Cuando el Método de Newton enloquece	234
5.4.3. Deteniendo el Método de Newton antes	237
5.4.4. La lista de comparaciones	239
5.4.5. Desafío: ¿Cuántos primos menores que o iguales a un x dado hay?	239
5.5. Más conceptos sobre control de flujo	239
5.5.1. Levantando una “excepción”	239
5.5.2. La construcción <code>if-then-else</code>	241
5.5.3. Un desafío sencillo: El procesamiento del fin del semestre, parte 1	243
5.5.4. Un desafío más difícil: El procesamiento del fin del semestre, parte 2	243
5.6. Bucles <code>while</code> versus bucles <code>for</code>	244
5.6.1. Una cuestión sobre factoriales	244
5.6.2. Desafío: Encontrar el número primo siguiente	244
5.6.3. El Método de Newton con un bucle <code>while</code>	245
5.6.4. El impacto de las raíces repetidas	246
5.6.5. Factorización mediante división al tanteo	247
5.6.6. Minidesafío: División al tanteo, deteniendo antes	249
5.6.7. Desafío: Una caja registradora mejorada	250
5.7. Cómo funcionan los arreglos y las listas	251
5.7.1. Listas de puntos y graficación	251
5.7.2. Desafío: Creando el comando <code>plot</code>	252
5.7.3. Operaciones con listas	252
5.7.4. Bucles a través de un arreglo	254
5.7.5. Desafío: Reporte de ganancia de la compañía	255
5.7.6. Promediando un puñado de números	255
5.7.7. Minidesafío: Promediando con y sin descarte de notas	257
5.7.8. Minidesafío: Duplicando la nota más alta	257
5.7.9. Desafío: El procesamiento del fin de semestre, parte 3	257
5.7.10. Algo útil: Devolviendo solamente raíces reales, racionales o enteras	257
5.7.11. Notación alternativa para listas	258
5.8. ¿A dónde ir a partir de aquí?	260
5.8.1. Otros recursos sobre programación en Python	260

5.8.2. ¿Qué hemos dejado fuera de este libro?	261
Capítulo 6. Construyendo páginas web interactivas con Sage	263
6.1. Nuestros ejemplos	263
6.2. El proceso de seis etapas para construir interactivos	264
6.3. El interactivo de la recta tangente	265
6.4. Un desafío para el lector	270
6.5. El interactivo del acuario óptimo	270
6.6. Selectores y casillas de verificación	271
6.7. El interactivo de la integral definida	272
Apéndice A. ¿Qué hacer cuando uno está frustrado?	275
Apéndice B. Haciendo la transición a CoCalc.com	281
B.1. ¿Qué es CoCalc?	281
B.2. Hojas de cálculo de Sage en CoCalc	282
B.3. Otras características de CoCalc	282
Apéndice C. Otros recursos sobre Sage	285
Apéndice D. Sistemas lineales con infinitas soluciones	289
D.1. El ejemplo de apertura	289
D.2. Otro ejemplo	291
D.3. Explorando más profundo	293
D.3.1. Una interesante reexaminación de las soluciones únicas	293
D.3.2. Dos ecuaciones y cuatro incógnitas	293
D.3.3. Dos ecuaciones y cuatro incógnitas, pero sin soluciones	294
D.3.4. Cuatro ecuaciones y tres incógnitas	294
D.4. Ideas erróneas	294
D.5. Definiciones formales de la REF y la RREF	295
D.6. Notaciones alternativas	296
D.7. Interpretaciones geométricas en tres dimensiones	296
D.7.1. Visualización de los anteriores casos	297
D.7.2. Interpretaciones geométricas en dimensiones superiores	297
D.8. Notación paramétrica	297
D.9. Soluciones a los desafíos de este apéndice	298
Apéndice E. Instalando Sage en una computadora personal	299
Apéndice F. Índice de comandos por nombre y por sección	305
Índice alfabético	319



Prefacio

Quien se dedica a las ciencias y a las ingenierías debe, inevitablemente, recurrir a una computadora. No es solamente el hecho que una máquina siempre está dispuesta a realizar por nosotros los cálculos tediosos, sino también que el avance de la ciencia en pleno siglo 21 exige un paso acelerado que solo puede alcanzarse con la computadora. Hoy en día, el alfabetismo computacional es tanto un requisito como saber leer y escribir, e incluso la matemática misma.

Hoy en día existen muchos sistemas computacionales que están a la altura del desafío, desde software de alto costo como Matlab, Mathematica, Magma, Maple, hasta software gratuito como Scilab, Octave, Sage, Maxima, etc. Sin embargo, de entre todos ellos, destaca Sage, un magnífico sistema de álgebra computacional, que ha demostrado no solo ser útil como herramienta para aprendizaje de las matemáticas, sino también para la investigación de alto nivel.

Este libro del Profesor Gregory Bard, originalmente publicado en inglés por la Sociedad Matemática Americana, es una de las mejores guías introductorias a Sage, y ha tenido gran éxito entre aquellos que desean iniciarse en este lenguaje de programación y ambiente de desarrollo matemático. El libro mismo ha ganado el reconocimiento de la *Open Textbook Initiative* del Instituto Americano de Matemáticas.

Para el traductor, el hecho que este material tan importante no esté disponible en español, ha sido una situación lamentable e inaceptable. Como Ingeniero Matemático y profesor de matemáticas a nivel universitario, he recurrido a este libro en innumerables ocasiones, ya sea para resolver problemas matemáticos o para crear materiales de estudio interactivos. Sin embargo, a lo largo de los años, he visto a muchos de mis alumnos y colegas carecer de acceso a este libro, no por cuestiones monetarias —ya que el libro está disponible en internet para descarga gratuita—, sino simplemente por estar escrito en inglés. Esta situación me ha impulsado a la tarea de realizar esta traducción, colaborando con el Prof. Bard para tener esta primera edición en español.

Es la esperanza del autor y el traductor que que esta edición beneficie a los estudiantes y profesionales de habla hispana, y que aumente en gran medida el alcance del libro a lo largo del mundo.

¿Qué es Sage? *SageMath*, o simplemente *Sage*, es un software matemático, lenguaje de programación y ambiente de desarrollo creado por el Profesor William Stein (<https://www.wstein.org>) y muchos otros voluntarios. Nacido con el propósito establecido de convertirse en “una alternativa viable *open-source* (de código abierto) a Magma, Maple, Mathematica y Matlab”, Sage ha superado sus orígenes y se ha convertido en una herramienta indispensable a muchos niveles de la formación profesional, desde los estudios de colegio, pasando por los universitarios, hasta la investigación matemática de alto nivel.

La genialidad del Prof. Stein consistió en no reinventar la rueda al crear un software desde cero, sino tomar los mejores programas y librerías matemáticos de código abierto disponible —como Maxima, GAP, R, Numpy, SciPy, junto con otros casi 200 más—, y unirlos mediante el lenguaje de programación Python. El resultado es una de las mejores piezas de software en el mundo.

Sage es gratis, libre y abierto; puede ser descargado sin costo, redistribuido sin restricción y modificado sin permiso. Estas características son de suma importancia, pues eso no solo nos permiten hacer mejoras o correcciones, sin tener que esperar a que una compañía lo haga por nosotros, sino que permite que otras personas revisen y puedan reproducir o validar nuestro trabajo. La *revisión, validación y reproducibilidad* de resultados son requisitos indispensables para la ciencia: ¿cómo podemos confiar en que un resultado está correcto si no lo podemos revisar? En este aspecto, Sage es superior a software comercial como Magma, Mathematica, Matlab y Maple.

Usamos Sage 9.1, o superior, y Python 3 Sage es un software que está mejorando constantemente. Este libro se ha escrito con las últimas mejoras en mente. En particular, a partir del primero de enero de 2019, el lenguaje Python 2 ya no es mantenido de manera oficial, por lo que, a partir de la misma fecha, Sage usa Python 3 como lenguaje base. Existen mejoras importantes que este cambio introduce y están consideradas en este libro. Por ejemplo, la sentencia `print` ha cambiado a una función `print`. Lo que esto significa es que, lo que antes se escribía como `print x`, ahora se escribe como `print(x)`. Parece ser cambio pequeño, pero es mucho más profundo e importante de lo que uno puede apreciar con esta simple explicación.

Si el lector está usando una versión antigua de Sage, basada en Python 2, debe añadir la instrucción

Código de Sage

```
1 from __future__ import print_function
```

como primera línea de sus programas, para poder usar el código de este libro. Esta instrucción indica a Python que debe usar la nueva función `print` en lugar de la antigua sentencia.

El libro también ha sido escrito pensando en una de las más innovaciones más útiles de Python 3: la posibilidad de usar guiones bajos en números. Por ejemplo, lo que antes se escribía como `x=54321`, ahora puede escribirse de manera más clara y cómoda en la forma `x = 54_321`. Esta característica estará disponible a partir de Sage 9.1. Si el lector usa una versión de Sage inferior, debe abstenerse de usar dichos guiones bajos en los números.

Para conocer qué versiones de Sage y Python se están usando, el lector puede usar el comando

Código de Sage

```
1 banner()
```

¿Qué temas cubre este libro? Sage para estudiantes de pregrado ha sido pensado como una guía introductoria al mundo de la resolución de problemas y la programación científica por medio de Sage. Como el título mismo indica, está escrito teniendo en mente a los estudiantes de pregrado que están iniciando sus estudios en matemáticas, física, química o alguna de las ingenierías. Sin embargo, el material debe ser adecuado para estudiantes de posgrado e incluso para investigadores que desee aprender a usar y programar Sage.

El material cubierto incluye gran parte de las materias Cálculo Univariado, Cálculo Multivariado y Álgebra Lineal, así como partes elementales de Ecuaciones Diferenciales, Teoría de Números y Modelamiento. El libro no pretende desarrollar la teoría detrás de estas áreas de estudio. En cambio, el lector encontrará una explicación detallada de cómo trabajar en estas áreas con Sage. Como requisito mínimo para entender el material presentado, el lector deberá completar al menos un primer semestre —incluso, tal vez, dos— de estudios universitarios.

El contenido entero se desarrolla en el contexto de aplicaciones, especialmente en el capítulo 2, donde se presentan proyectos para resolver problemas en diferentes áreas aplicativas. En general, el libro echa mano de conceptos de microeconomía, física, química, biología, criptografía, balística, optimización de transporte

industrial y otros menores. No es requisito que el lector tenga conocimientos previos sobre estas cuestiones para que pueda hacer una lectura efectiva. Más al contrario, el lector se hará un poco más sabio en estas aplicaciones al leer estas páginas.

En cuanto a programación se refiere, el libro cubre aproximadamente la mitad del contenido usual de una clase de Ciencia Computacional I. Sin embargo, esto debe ser más que suficiente para la mayoría de los lectores. Para los demás, que quieran aprender aun más sobre ciencia computacional y programación científica, este libro será un excelente primer encuentro con estos temas.

La página web del libro Esta edición en español cuenta con su propia página web:

www.sage-para-estudiantes.com.

Ahí, el lector encontrará materiales adicionales, como programas de Sage, páginas web interactivas (también creadas con la ayuda de Sage), aplicaciones al mundo real y muchos otros. El sitio también tiene disponible varias versiones de este libro: a colores; en blanco y negro (ideal para imprimir ahorrando en tinta); optimizado para pantallas de laptops; una versión en línea, y otros.

A lo largo de este libro mencionaremos varias direcciones electrónicas. Como resulta ser la naturaleza de la internet, es probable que alguna de ellas ya no esté disponible o haya cambiado al momento en que el lector lea estas líneas. De ser así, el autor y el traductor harán un esfuerzo razonable para tener una lista de direcciones electrónicas actualizadas en la página del libro.

El autor cuenta con una página web personal, donde se pueden encontrar los materiales adicionales correspondientes a la edición en inglés:

<http://gregorybard.com/Sage.html>.

El traductor se ha esforzado en tener todos estos disponible en español en el sitio web de esta edición.

Si el lector encuentra un error, ya sea ortográfico, conceptual o de programación, y desea reportarlo, puede recurrir a

www.sage-para-estudiantes.com/errata,

donde encontrará un formulario para llenar un reporte.

Innovaciones que introduce esta edición Muchas cosas han cambiado desde que se publicó la primera edición en inglés de este libro. Esta edición se ha actualizado en consecuencia. Por ejemplo, muchas direcciones de internet que se mencionaban ya no son válidas o han sido cambiadas. La constante evolución de Sage y Python han presentado nuevas y mejores formas de trabajar. Es imposible crear un libro libre de errores, y la primera edición en inglés tenía algunos que sobrevivieron al proceso de revisión, por lo que nos hemos esforzado en eliminarlos completamente ahora.

Como mencionamos arriba, Sage es software libre y de código abierto. En esta edición, hemos aprovechado estas características para introducir una mejora importante. Este libro está construido con una mezcla de \LaTeX (el estándar internacional para publicaciones científicas) y Sage, usando código escrito especialmente para este propósito. Muchos de los elementos de esta edición, ya sean fórmulas, texto o gráficos, son generados de manera automática por Sage mismo cuando el libro es compilado. De ahora en adelante, la actualización y mantenimiento del texto contenido en estas páginas será mucho más fácil; también, estamos seguros que los cálculos y resultados mostrados están libres de errores humanos, pues ningún humano los realizó. Si algún dato de un ejemplo o ejercicio fuese a cambiar en una edición futura, todas las gráficas y los resultados serán actualizados por Sage de manera automática, en concordancia, sin intervención humana.

Esta una innovación importante, pues nos permitirá actualizar y mejorar el libro y sus resultados de manera eficiente, evitando así el fenómeno llamado *decaimiento del código*, que ocurre cuando una larga sucesión de cambios pequeños son hechos a algún software, resultando en que viejos códigos ya no son funcionales o funcionan de manera diferente. Un ejemplo de este fenómeno se dio con el cambio de Python 2 a Python 3.

Más aun, para lograr la mezcla adecuada de \LaTeX y Sage en este libro, se ha escrito nuevo código de Sage, o se ha modificado parte del existente. Esta es una de las ventajas de tener un software de código abierto y libre. Estas innovaciones serán publicadas después de este libro, de manera que estén disponibles para el público en general. El lector puede ver la página

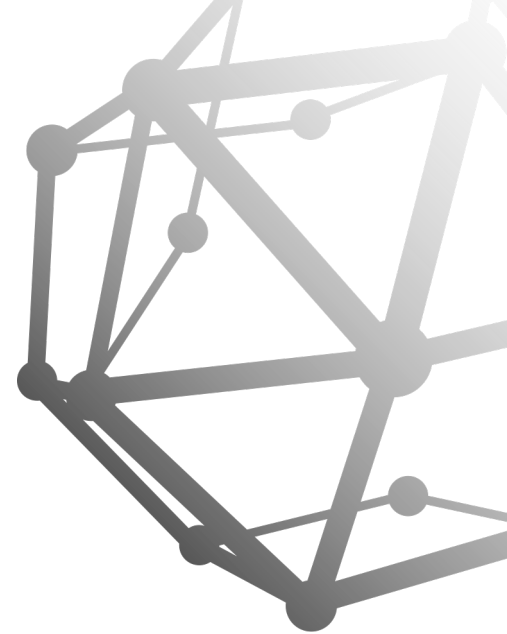
`www.sage-para-estudiantes.com`

para estar informado sobre este asunto.

Esta edición añade la sección 1.14, la sección 1.15 y la sección 4.25. La primera da una explicación clara de las formas en que Sage entiende y maneja las funciones matemáticas. La segunda constituye una explicación detallada de algunos usos adicionales del comando `show`. Finalmente, la tercera permitirá al lector que sepa programar en \LaTeX , cómo construir documentos similares a este libro, donde Sage se encargará de hacer los cálculos y transcribir los resultados.

Esta edición también incorpora una de las características más solicitadas en la publicación original: un índice alfabético. Este, junto con el índice de comandos en el apéndice F en la página 305, proveerá una guía importante para el lector casual o quien desee hacer una consulta puntual.

Una innovación final es el nuevo estilo de esta edición. Se ha rediseñado completamente la apariencia del libro. Desde los encabezados de los capítulos, pasando por las secciones, subsecciones y otras partes del texto. El código se ha coloreado de acuerdo a la sintaxis de Sage y Python, y se lo ha enmarcado para que sea más claro cuál es su extensión. Todo esto, además de un sinnúmero de pequeñas mejoras. Se ha puesto mucho esfuerzo en mejorar la estética de esta edición. Esperamos que esto haga más agradable su lectura.



Agradecimientos del traductor

No puedo empezar a escribir estos agradecimientos si no es mencionando al autor del libro, el Profesor Gregory Bard. Le estoy eternamente agradecido por confiarme la traducción de su obra. Desde el primer día, hasta el momento de escribir estas palabras, el Prof. Bard ha depositado en mí la mayor de las confianzas y me ha permitido la mayor de las libertades en esta tarea. Por ello, me siento muy conmovido. Con él, hemos compartido el gran entusiasmo de crear innovaciones y mejoras en este libro. Él siempre se ha mostrado paciente conmigo y dispuesto a ayudar. Para mí ha sido todo un privilegio trabajar bajo su supervisión y tutela. Todo ha sido una magnífica experiencia.

La Sociedad Matemática Americana (www.ams.org) posee los derechos de publicación de este libro. Ellos fueron muy amables en permitirme realizar esta traducción y darme mucha libertad en mi trabajo. De la misma manera, nos han permitido poner esta edición libre en internet para descarga gratuita. Les estoy muy agradecido por su consideración y confianza. Debo agradecer en especial —en orden alfabético— a Erin M. Buck (Asistente Editorial de la AMS), Sergei Gelfand (Jefe de Publicaciones de la AMS) e Ina Mette (Adquisiciones de Libros de la AMS) por su apoyo a esta edición.

Nadie puede seguir adelante sin que haya alguien que le diga que puede seguir adelante. Durante la traducción de esta obra, mi papá ha sido una guía moral, espiritual e intelectual, no solo con este libro, sino en todos los aspectos de mi vida que han ocurrido durante este trabajo. Las largas charlas que hemos tenido me han enriquecido como persona y como profesional, de maneras que ni siquiera puedo empezar a explicar al lector.

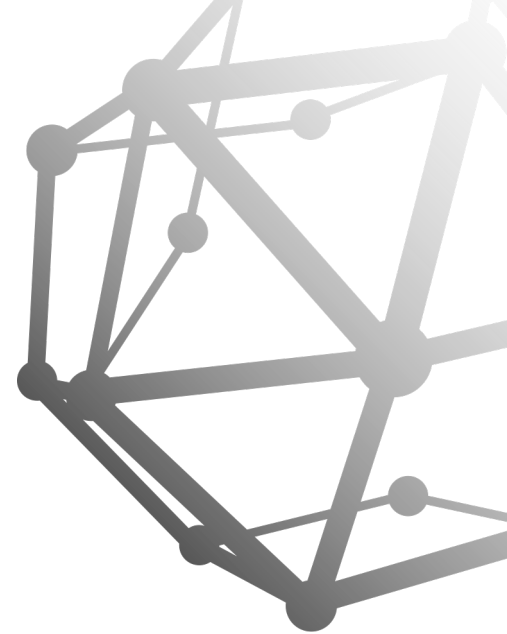
Agradezco a la Dra. Kathia Butrón por su amistad, por su apoyo moral para realizar este trabajo, y por compartir conmigo la alegría de haberlo completado. Una amiga que me ha apoyado tanto y ayudado a mirar el universo con nuevos ojos, siempre estaré en deuda con ella.

El apoyo moral y la amistad son importantes cuando uno escribe. Agradezco a mis amigos César Andrés Cabrera Cabero y José Manuel Muñoz (orden alfabético, no necesariamente de importancia) por el apoyo moral y el interés en esta obra. Y también les estoy agradecido por los momentos de diversión cuando el trabajo se tornaba pesado.

Un agradecimiento especial va para el Prof. William Stein, el genio creador y la mente maestra de Sage, quien merece la mayor admiración y el eterno agradecimiento de los usuarios. Agradezco también a William Stein, así como Harald Schilly y Hal Snyder por corregir y actualizar el apéndice B. Esta ha sido un gran contribución a esta edición.

Por supuesto, debo agradecer a la Comunidad Sage, los cientos de programadores alrededor del mundo que han trabajado en crear este magnífico software. De una manera u otra sus contribuciones a Sage han hecho posible este libro y su traducción. Ellos son quienes han programado o mejorado el código fuente; guiado a los usuarios y al traductor mismo por medio del foro de preguntas y respuestas `ask.sagemath.org`; creado tutoriales y documentación. Ellos están al frente de las muchas formas en las que uno puede contribuir a un proyecto gratis, libre y de código abierto, como Sage.

Aunque su cronograma no le permitió hacer una revisión más completa de los capítulos 5 y 6, el Dr. Oswaldo Larreal hizo una lectura de 16 páginas del libro y realizó sugerencias útiles en base a ello, algunas de las cuales ahora forman parte de esta edición.



Agradecimientos del autor (del año 2014)

Primero, me gustaría agradecer a William Stein, el creador de Sage, por su incentivo, apoyo y acceso a recursos de supercomputación que han hecho mi investigación criptoanalítica posible. El Prof. Stein es el fundador de Sage, su arquitecto y organizador principal, y la fuente de entusiasmo para la comunidad Sage —que en este punto incluye cientos de personas en muchos países de varios continentes—. Muchas de las líneas originales de Sage mismo, así como la vasta mayoría de CoCalc, fueron personalmente programados por el Prof. Stein.

Este libro empezó como “Una Guía para Sage” mientras estaba enseñando por cinco semanas en Beijing, en la Chinese Academy of Sciences, Academy of Mathematics and Systems Science, Key Laboratory for Mathematics Mechanization, durante el verano de 2010. Estoy muy agradecido porque se me dio esa oportunidad especial de trabajar con estudiantes tan avanzados. Estaba enseñando con otro libro que he escrito, *Algebraic Cryptanalysis*, publicado por Springer en 2009. Di varias horas de instrucción en Sage, pero se hizo claro que tener instrucciones escritas sería más útil para los estudiantes. Uno puede hojear a través de ellas mientras trabaja con Sage. Y, por lo tanto, nació el documento que eventualmente se convertiría en este libro. Estoy extraordinariamente agradecido por la oportunidad de haber visitado esa excelente institución. Cuando partí de ahí, este documento tenía 37 páginas, pero a partir de pequeñas bellotas crecen poderosos robles.

Como muchos investigadores que inventan un nuevo algoritmo como parte de su tesis de doctorado en la Universidad de Maryland, yo tuve que escribir código para probar que mi algoritmo es efectivo, correcto y eficiente. Sin embargo, tal código es frecuentemente descartado después que el título de PhD es conferido. Me gustaría agradecer a Martin Albrecht, quien me presentó por primera vez a la comunidad Sage, y quien llevó a la inclusión de mi código de tesis en las librerías de Sage, en lo que se ha convertido en el proyecto M4RI. Sin Martin, nunca hubiera escuchado de Sage, ni el código de mi tesis sería usado por gente a lo largo de este mundo. Resulta que M4RI es un proyecto en evolución, continuamente actualizado por diferentes investigadores y, probablemente, hoy contiene muy poco de mis líneas originales. Ese es un ejemplo de cómo el software puede evolucionar. Se puede leer sobre el algoritmo original en el capítulo 9 de *Algebraic Cryptanalysis*.

También me gustaría agradecer a Robert Beezer y Robert Miller por alentarme a usar Sage en mi enseñanza e investigación, respectivamente. En particular, el sobresaliente (y gratis) libro electrónico de Robert Beezer, *A First Course in Linear Algebra*, fue una inspiración para mí mientras escribía este documento.

El *Sage Cell Server* (*Sevidor de Celdas de Sage*) ha hecho Sage accesible a una audiencia mucho más amplia de lo que antes se soñó posible. Incluso para aquellos expertos en Sage, la experiencia de usuario está ahora enormemente mejorada. Este importante y difícil trabajo ha sido realizado por Jason Grout, Ira Hanson, Steven

Johnson, Alex Kramer y William Stein. Originalmente, Sage era adecuado para estudiantes de posgrado, profesores y probablemente graduados de matemáticas. Entonces vino el Servidor Sage Notebook, trayendo Sage a estudiantes de pregrado a media carrera. Ahora, gracias al Servidor Sage Cell, es apropiado usar Sage incluso en clases de Precálculo, si se desea.

La Sociedad Matemática Americana fue extremadamente flexible conmigo en cuestión de los plazos, así como las muchas imágenes en este texto. Producir un libro requiere un número de pasos sorprendentemente grande, y estoy agradecido por el apoyo de Ina Mette, la editora de adquisiciones, y Marcia Almeida, su asistente. Estoy particularmente agradecido por el estímulo enérgico y entusiasta de Ina Mette para convertir lo que una vez fue una pequeña guía en un extenso libro (impreso). Este proyecto no existiría sin su incentivo.

Me gustaría agradecer a Andrew Novocin por enseñarme cómo incluir archivos *.png en documentos de \LaTeX , lo que me ahorró el tiempo de convertir manualmente los formatos de archivos de la armada de capturas de pantalla que llenan la primera edición de este libro. De otra manera, habría abandonado este proyecto entero.

Ningún autor puede corregir su propio trabajo efectivamente, pero este es particularmente el caso para mí. Estoy agradecido por la ayuda de los revisores Joseph Robert Bertino, Joseph Loeffler y Thomas Suiter, quienes han hecho cada uno sugerencias y correcciones a porciones de este documento. De particular mérito, Thomas Suiter hizo el índice de comandos, y Joseph Bertino ha soportado noblemente (sin queja alguna) una enorme carga de trabajo de mi parte a través de los últimos años —tanto de este libro como de otros proyectos—. Por supuesto, cualquier error que permanece es mi responsabilidad.

Mi primer trabajo después de obtener mi PhD de la Universidad de Maryland fue en la Universidad Fordham, en El Bronx, NY, donde fui asignado a uno de los “Profesorados Asistentes Visitantes Peter M. Curran” de cuatro años. Hice algún progreso en este proyecto durante mi tiempo en Fordham, después de regresar de mi verano en China. Recibí apoyo y sugerencias de Shaun Ault, Melkana Brakalova, Armand Brumer, Michael Burr, Bill Campbell, Janusz Golec, Maryam Hastings, William Hastings, Nick Kintos, Robert Lewis, Damian Lyons, Ian Morrison, Leonard Nissim, Cris Poor, Benjamin Shemmer, Shapoor Vali y Kris Wolff. El libro de Fred Marotto, *Introduction to Mathematical Modeling Using Discrete Dynamical Systems*, publicado por Cengage en 2005, fue también una inspiración para este trabajo, aunque no tuve la oportunidad de discutir nuestras áreas de interés común muy frecuentemente. Jack Kamis, Mila Martynovsky y Michael “Misha” Zigelbaum me proporcionaron valiosos y prácticos consejos sobre enseñanza.

En particular, Shapoor Vali y yo escribimos nuestros libros casi al mismo tiempo, así que soportamos la lucha de escribir capítulos, borradores para revisión y elaboración de propuestas de libros simultáneamente. Ambos hemos cruzado un gran desierto, no sin pena y sufrimiento, y ahora podemos disfrutar del oasis al otro lado.

Muchos estudiantes de pregrado en Fordham me dieron una gran cantidad de sugerencias, correcciones, oportunidades para mejoras e ideas. A riesgo de accidentalmente excluir a alguien, deseo mencionar a los siguientes estudiantes de Fordham: James Compagnoni, Gray Crenshaw, Dan DiPasquale, Patrick Fox, Alex Golec, Simon Kaluza, Kyle Kloster, Peter Muller, Luigi Patruno, Seena Vali y Sean Woodward. Mucho ánimo me fue dado por Ed Casimiro, por lo que estoy agradecido.

Después que el puesto de cuatro años en Fordham había terminado, tuve la gran suerte de ser contratado en la Universidad de Wisconsin–Stout, el politécnico del sistema de la Universidad de Wisconsin. Esta fue una gran bendición para mí, pues recibí mi título de pregrado en Ingeniería Computacional y de Sistemas del Instituto Politécnico Rensselaer en 1999 y, similarmente, mi padre recibió su título de pregrado en Ingeniería Mecánica del ETH en Zürich (el politécnico de Suiza), allá en 1951. De acuerdo a esto, retornar a la atmósfera de un politécnico era como volver a casa. Aunque las tres instituciones están lejos de ser trillizos idénticos, el noble objetivo de preparar estudiantes aplicados para convertirse en ingenieros, científicos y administradores, o miembros de otras profesiones técnicas lucrativas, crea una actitud de enfoque en lo pragmático, en el trabajo duro y en la exactitud. Esta “actitud politécnica” es reflejada a través del espectro del rango académico, desde el rector hasta los estudiantes de primer año. Genera respeto por las matemáticas en general y el cálculo en particular. En la UW Stout, las matemáticas son vistas como un puente a importantes cursos relacionados con la carrera, y no como una barrera para la graduación, como eran frecuentemente vistas en Fordham, tanto por los estudiantes como por los administradores de alto rango.

La vasta, vasta mayoría de este libro fue escrita mientras trabajaba para UW Stout. Tanto el decano Jeff Anderson como el decano Charles Bomar proporcionaron apoyo administrativo, financiero y emocional para este proyecto. El grado de cooperación de nuestro *Departamento de Matemáticas, Estadística y Ciencia Computacional* (MSCS) es extremadamente raro y conmovedor. A menos que esté muy equivocado, cada uno de los miembros del departamento han contribuido a este libro a través de sugerencias, correcciones, comentarios, reemplazándose en clases cuando estaba lejos en una conferencia, o dándome ánimos. Enseñar 24 créditos por año en un clima completamente hostil para la vida humana¹, mientras intentamos mantener alguna fracción de vida de investigación, nos ha unido. Con gran respeto, escribo sus nombres aquí en este momento: Anne Antonippillai, Wan Bae, Alexander Basyrov, Christopher Bendel, Seth Berrier, Jeffrey Boerner, Diane Christie, Steven Deckelman, Brent Dingle, Seth Dutter, Jeanne Foley, Eugen “Andre” Ghenciu, Petre “Nelu” Ghenciu, Nasser Hadidi, Matt Horak, Ayub Hossain, Amitava Karmaker, Deborah Kruschwitz-List, Terrence Mason, Valentina “Diana” Postelnicu, Laura Schmidt, Loretta Thielman, Peter Thielman, Keith Wojciechowski y Ming-shen Wu. Adicionalmente, los anteriormente miembros del cuerpo docente, Joy Becker, John Hunt, Benjamin Jones, Dennis Mikkelson y Eileen Zito han proporcionado consejo, incentivo o material de curso. Más aun, muchos de nuestros contratados temporalmente se han hecho cargo voluntariamente de cursos fenomenalmente desagradables o de las obligaciones elevadas de la docencia de tiempo completo, sin compensación financiera adicional, y son, por lo tanto, particularmente merecedores de alabanzas: Jim Church, Brian Knaeble, Shing Lee, Olga Lopukhova, John Neiderhauser, Mark Pedersen y Dennis Schmidt. El jefe del departamento, Chris Bendel, es particularmente dedicado y un excepcional pacificador que ha creado un ambiente de colaboración, cooperación y respeto mutuo, que es frecuentemente soñado pero casi nunca encontrado, en la vida académica. En el MSCS, trabajamos juntos para asistirnos unos a otros, a diferencia de otro departamento en nuestro edificio, que no nombraré explícitamente, donde los profesores antiguos han hecho un hobby el aplastar las carreras y destruir los sueños de sus propios profesores nuevos.

Fuera de mi departamento en UW Stout, Alan Block, Mark Fenton, Julie Furst-Bowe, Sue Foxwell, Jennifer Grant, Mary Hopkins-Best, Michael Levy, Steve Nold, Marlann Patterson, y Todd Zimmermann me han dado un generoso aliento. También me gustaría agradecer al Departamento de Biología entero, quienes han establecido un estándar muy alto de estudio, de enseñanza interactiva, de hábil escritura de concesiones, de clases innovadoras y enfocadas en el laboratorio, y quienes nos han enseñado a todos a involucrar profunda y materialmente a los estudiantes de pregrado en la investigación.

Sería habitual listar aquí a los estudiantes de la UW Stout que han enriquecido mi enseñanza y me han dado retroalimentación útil. Tal lista alargaría seriamente este libro, sin embargo. He tenido mucha suerte en que se me han asignado cursos altamente aplicados, con contenido extremadamente aplicable a la práctica profesional tanto en ingeniería como administración. Se verá este efecto en el cuerpo de este libro, pues mis ejemplos están sacados de la física y la economía, en concordancia con las aspiraciones profesionales de mis estudiantes. La vasta mayoría de mis alumnos, en algún punto u otro, me han dado sugerencias, una corrección, o me han mostrado alguna nueva aplicación de las matemáticas a sus campos de estudio elegidos. He estudiado y enseñado en otras instituciones, pero este fenómeno es uno que solo he encontrado en la UW Stout y en Oxford, en el Reino Unido, cuando era un estudiante visitante ahí. Como no me es posible listar a todos los estudiantes por nombre, les agradeceré por número de curso: Math-123, *Matemática Finita y Financiera*; Math-154, *Cálculo II*; Math-380/580, *Criptografía*; Math-447/747, *Análisis Numérico II*, y CS-480/680, *Seguridad Computacional*.

Previamente a enseñar en Fordham tuve la oportunidad de ser un profesor ajunto en la Universidad Americana, en Washington DC. Me gustaría agradecer a Michael Gray, Angela Wu, Nate Harshman y Michael Black por su apoyo, incentivo y enseñanza, así como al antiguo alumno ya graduado Alexander Ivanov, por su estímulo y energía personal.

¹Esta no es una exageración. Dos ejemplos tendrán que bastar. El primer día de clases en 2014, es decir el 28 de enero, cuando estaba saliendo de casa para ir al campus, weather.com dio una temperatura de -22°F en el termómetro y -41°F para la sensación térmica. Para mis lectores del sistema métrico, eso es $-29,4^{\circ}\text{C}$ en el termómetro y $-40,6^{\circ}\text{C}$ para la sensación térmica. El 2 y 3 de mayo de 2013, tuvimos 17 pulgadas de nieve (eso es 43,2 cm).

Además de los profesores en las universidades donde he estado enseñando, los profesores de la Universidad Columbia y la Universidad de la Ciudad de Nueva York (CUNY), dos de las mejores instituciones en “la ciudad” y la nación, me han concedido gran ayuda. He disfrutado de las conversaciones tanto sobre investigación como enseñanza con David Allen, David Bayer, Maria Chudnovsky, Kenneth Kramer, Hunter Johnson y Vladimir Shpilrain.

En física y química, he tenido el tremendo beneficio de consultar con expertos. Para física, fue Gabriel Hannah y, para química, fue Gergely Sirokman. Ellos han revisado cuidadosamente sus secciones relevantes, y por lo tanto, el lector puede estar seguro que estas están libres de errores, con una excepción: en los problemas de física, no he llevado las unidades a través de las fórmulas y en las ecuaciones, como es requerido por esta disciplina. En matemática aplicada, generalmente no hacemos esto y sería altamente desorientador y tedioso para mí modificar este libro ya completado para corresponder con las convenciones de los físicos.

He recibido gran cantidad de apoyo en este proyecto, incluyendo sugerencias detalladas, de aquellos listados arriba, así como de Martin Brock, Bruce Cohen, Carmi Gressel, Marshall Hampton, Benjamin Jones, David Joyner, Craig Larson, Andrey Novoseltsev, Clement Pernet y Susan Schmoyer.

Mi antiguo tutor de tesis y amigo de confianza Lawrence Washington, me ha inspirado con sus textos extremadamente exitosos. Él ha puesto una vara muy alta, a la cual el resto de nosotros aspiramos. Frecuentemente recomiendo *Cryptography with Coding Theory*, de Wade Trappe y Lawrence Washington, a cualquiera que quiera aprender criptografía —desde estudiantes de segundo año de pregrado hasta docentes con antigüedad—.

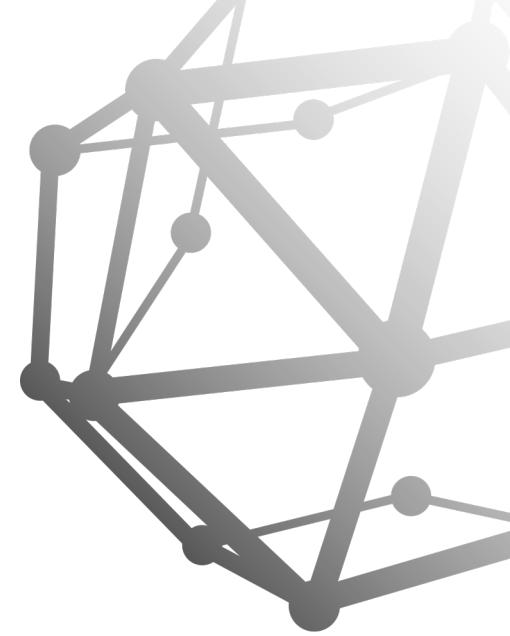
La comunidad WeBWorK ha trabajado de cerca con la comunidad Sage. WeBWorK es el competidor open-source (de código abierto) y libre de los sistemas de tarea en línea dirigidos por las casas de publicación de textos, y está avalada por la Asociación Matemática de América. Numerosas horas de las vidas de los voluntarios de WeBWorK han sido dedicadas a escribir y manipular este sistema, así como programar problemas matemáticos. Por la integración con Sage, la comunidad esta en gran deuda con John Travis, del Mississippi College, quien contribuye activamente a ambos proyectos.

El financiamiento de Sage proviene en gran parte de la Fundación Nacional para la Ciencia, y en particular de las divisiones listadas abajo. La comunidad Sage está agradecida por el apoyo de las concesiones de la Fundación Nacional para la Ciencia:

- DMS-0545904, Division of Mathematical Sciences, “CAREER: Cohomological Methods in Algebraic Geometry and Number Theory”;
- DMS-0555776, Division of Mathematical Sciences, “Explicit Approaches to Modular Forms and Modular Abelian Varieties”;
- DMS-0713225, Division of Mathematical Sciences, “SAGE: Software for Algebra and Geometry Experimentation”;
- DMS-0757627, Division of Mathematical Sciences, “FRG: L-functions and Modular Forms”;
- DMS-0821725, Division of Mathematical Sciences, “SCREMS: The Computational Frontiers of Number Theory, Representation Theory, and Mathematical Physics”;
- DMS-0838212, Division of Mathematical Sciences, “EMSW21-RTG: Research Training Group on Inverse Problems and Partial Differential Equations”;
- DMS-1015114, Division of Mathematical Sciences, “Sage: Unifying Mathematical Software for Scientists, Engineers, and Mathematicians”;
- DUE-1022574, Division of Undergraduate Education, “Collaborative Research: UTMOST: Undergraduate Teaching in Mathematics with Open Software and Textbooks”;
- ACI-1147463, Division of Advanced CyberInfrastructure, “Collaborative Research: SI2-SSE: Sage-Combinat: Developing and Sharing Open Source Software for Algebraic Combinatorics”;

así como financiamiento de Microsoft, Sun Microsystems, Enthought Inc. y Google.

Como mencioné antes, la comunidad Sage es enorme, con cientos de desarrolladores en muchos países de varios continentes. A pesar del apoyo financiero mencionado arriba, casi todo el trabajo es realizado de forma enteramente voluntaria y sin pago alguno. Por el sacrificio del tiempo libre por parte de tanta gente, la comunidad Sage entera está en extremo agradecida.



Special thanks

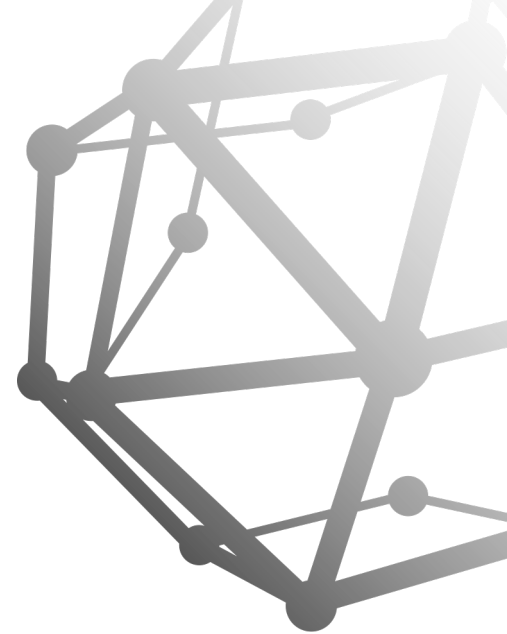
The task of proofreading is rather tedious in general, and particularly so in the case of a mathematics textbook. Proofreading this book required not only fluency in both English and Spanish, but also knowledge of Sage, and expertise in mathematics at an advanced level. Prof. Ivelisse Rubio of the University of Puerto Rico was especially helpful by volunteering to help us locate proofreaders. She introduced us to Andrés Ramos Rodríguez, Carlos E. Seda, and Jaziel Torres Fuentes. We are deeply thankful to César Andrés Cabrera Cabero, a friend of the translator, for being so kind of taking care of the proofreading of a large part of this book. These four proofreaders have helped the text become better, and for that the translator and the author express our gratitude. They deserve special acknowledgements for their dedicated work of searching for errors and suggesting corrections. All of them performed this task completely voluntarily and without compensation.

Nonetheless, errors are inevitable, as any author knows. If any errors have escaped our joint attention, then please report them to the translator, Diego Sejas Viscarra, at dsejas.math@pm.me

Agradecimientos especiales

La tarea de revisar un libro es bastante tediosa en general, y particularmente lo es en el caso de un texto de matemáticas. La revisión de este libro requirió no solo fluidez en inglés y español, sino también conocimiento sobre Sage y pericia en matemáticas en un nivel avanzado. La Profa. Ivelisse Rubio de la Universidad de Puerto Rico fue especialmente importante al ofrecerse a ayudarnos a encontrar revisores. Ella nos presentó a Andrés Ramos Rodríguez, Carlos E. Seda y Jaziel Torres Fuentes, quienes revisaron porciones de este libro. Estamos profundamente agradecidos con César Andrés Cabrera Cabero, un amigo del traductor, por ser tan amable de hacerse cargo de revisar una parte considerable del libro. Estos cuatro revisores han ayudado a que el texto sea mejor, y por esa razón el autor y el traductor expresamos nuestra gratitud. Ellos merecen un reconocimiento especial por su dedicada labor al buscar errores y proponer correcciones. Todos ellos ejecutaron esta tarea de manera completamente voluntaria y sin compensación.

Sin embargo, los errores son inevitables, como cualquier autor lo sabe. Si cualquier error ha escapado a nuestra atención conjunta, entonces pedimos al lector que por favor los reporte al traductor, Diego Sejas Viscarra, en la dirección `dsejas.math@pm.me`



Introducción. Cómo usar este libro.

Como el competidor open-source (de código abierto) y libre de software costosos como Maple, Mathematica, Magma y Matlab, Sage ofrece a cualquiera con acceso a un navegador web la habilidad de usar software matemático de última generación, y exponer los resultados propios a otros, frecuentemente con gráficos espectaculares. Estoy seguro que el lector encontrará Sage mucho más fácil de usar que una calculadora graficadora, así como mucho más poderoso.



No es necesario leer este documento entero, como no es necesario leer un diccionario de tapa a tapa. Este libro es ideal para el autodidacta, pero si al lector se le asigna este como parte de una clase, entonces el profesor indicará qué secciones corresponden con lo que se necesita leer.

Para otros lectores, quienes estudien este libro independientemente de una clase, tengo los siguientes consejos:

- El capítulo 1 contiene las bases —lo que el lector realmente necesita saber—. La mayor parte de las tareas comunes en Sage que se exponen ahí fueron cuidadosamente elegidas para ser tan intuitivas como sea posible, y con notación tan próxima como se pueda a cómo las matemáticas son hechas en un pizarrón o en papel y lápiz. Recomiendo que el lector nunca intente leer más de tres secciones en un día; de otra manera, hay mucho que la mente debe absorber a la vez como para que la experiencia se mantenga divertida y fresca. Sin embargo, 1–2 secciones por día deberían ser fácilmente digeribles.

Nota: Personalmente, recomiendo leer el capítulo 1 y entonces empezar a experimentar por uno mismo. La mejor manera de aprender un nuevo paquete de software es usarlo recreativamente. Usando la tabla de contenidos (cerca del principio del libro), o el índice de comandos y el índice alfabético (al final del libro), siempre se puede identificar cómo realizar alguna tarea que aún no se ha aprendido. Si el lector se siente confundido en algún punto, asegúrese de ver el apéndice A, “¿Qué hacer cuando uno está frustrado?”

- El capítulo 2 contiene proyectos en los que se puede usar Sage para resolver problemas matemáticos como los que suelen aparecer en otras áreas de estudio —criptografía, física, química, biología y otros—. Estos proyectos están principalmente diseñados para ser tratables después de leer el capítulo 1; sin embargo, en algunos casos, otra sección de un capítulo superior será también requerida. Por supuesto, en esos casos, la dependencia es claramente indicada. Estos proyectos están pensados para ser asignados a lo largo de fines de semana. No son lo suficientemente largos para ser proyectos semestrales, pero son muy largos como para ser problemas de tarea para el día siguiente.
- El capítulo 3 trata exclusivamente acerca de la creación de hermosas gráficas e imágenes con Sage. He colocado ahí todo lo que me fue posible; sin embargo, algunos tipos de gráficos, incluyendo los tridimensionales, animados y multicolores, no pueden ser representados fácilmente en un libro. Por lo tanto, esos tipos de imágenes son cubiertos en el apéndice únicamente electrónico de este libro, “Graficando a color, en 3D, y animaciones”, disponible para descarga gratuita en la página web

www.sage-para-estudiantes.com

- El capítulo 4 es enorme y contiene pequeñas secciones que discuten temas matemáticos avanzados. Sage puede tratar una gran cantidad de problemas. Este capítulo fue diseñado de manera que las secciones individuales pueden ser leídas de forma independiente. No hay necesidad de leerlas en orden, a menos que el lector así lo desee.
- El capítulo 5 trata acerca de cómo programar en Python usando Sage y viceversa. Como sistema de álgebra computacional, Sage está construido sobre ese lenguaje de programación. Por lo tanto, se pueden usar comandos de Python en Sage. En ocasiones, esto es realmente útil —por ejemplo, se pueden escribir ciclos `for` para hacer tablas muy fácilmente—. En otras situaciones, es conveniente escribir programas de computadora enteros en Python usando Sage como interfaz. Esto es más cierto cuando se usa CoCalc.
- El capítulo 6 cubre la creación de páginas web interactivas (llamadas también *interactivos*, *applets* o *apps*) usando Sage. El proceso es notablemente directo. Propongo un método de seis etapas que he estado usando personalmente. El lector puede hacer interactivos extremadamente interesantes por medio de Sage, y apuesto a que estará sorprendido por lo fácil que es construir sus propias apps.
- El apéndice A es “¿Qué hacer cuando uno está frustrado?”, donde presento una lista de preguntas que el lector debería hacerse a sí mismo si queda atascado en una situación en la que Sage se reusa a responder a los comandos en la forma que nos gustaría. En casi todos los casos, uno de los puntos listados presentará una forma de resolver la frustración.
- El apéndice B trata acerca de ganar familiaridad con CoCalc y algunas de sus características. La computación en la nube es una nueva y excitante forma de hacer computación usando la internet, en lugar de una máquina local, como lugar para almacenar nuestros archivos.

Nota: Generalmente, los usuarios querrán cambiar a CoCalc después de completar el capítulo 1, antes de abordar los proyectos en el capítulo 2 o el trabajo avanzado de los capítulos 4–6. Considero que el capítulo 3 puede leerse antes o después de aprender CoCalc, sin ninguna desventaja.

- El apéndice C es una colección de mayores recursos sobre Sage, para aquellos lectores a quienes les gustaría aprender más de lo que este libro contiene.
- El apéndice D es mi respuesta al hecho que los estudiantes frecuentemente no tienen facilidad con los sistemas de ecuaciones lineales con infinitas soluciones. Este concepto es frecuentemente aprehendido como una idea, pero los detalles no siempre están ahí cuando necesito que los conozcan. Por lo tanto, he decidido ser hiperexhaustivo y dedicar un gran número de páginas a explicar este tema (ciertamente difícil). Espero que resulte útil para los estudiantes.
- El apéndice E es lo que se debe leer si uno desea saber cómo instalar Sage en una computadora personal.
- El apéndice F es un índice de 16 páginas (!) de todos los comandos en este libro, para referencia del lector. Agradecimientos especiales para Thomas Suiter, quien preparó la primera versión de este.

- El apéndice únicamente electrónico en línea cubrirá los temas de graficación a color, animaciones y gráficas 3D. Esos temas no son adecuados para un libro estático (no interactivo), y por lo tanto, no pueden ser impresos en estas páginas. El lector lo puede descargar de la página web www.sage-para-estudiantes.com.

¡Empecemos! ¡Ahora mismo! Sin el más mínimo titubeo, abramos un navegador de internet en este mismo instante. Escribamos la siguiente URL:

<https://sagecell.sagemath.org/>

Ahora estamos conectados al “Sage Cell Server” (“Servidor de Celdas de Sage”, al cual nos referiremos como “Servidor Sage Cell”). Veremos un gran campo de texto ahí (llamado *celda*) y un botón etiquetado “Evaluate” (“Evaluar”). En esa celda, escribimos lo siguiente:

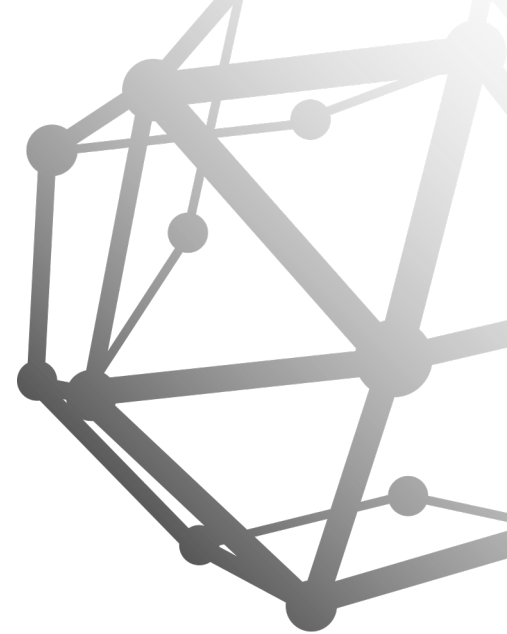
```
Código de Sage
1 solve(x^2 - 16 == 0, x)
```

y presionamos “Evaluate”. Seguramente el lector recibió la respuesta

```
[x == -4, x == 4]
```

la cual es claramente el conjunto correcto de dos soluciones a la ecuación $x^2 - 16 = 0$. Acabamos de resolver nuestro primer problema matemático con Sage. ¡Felicidades!

¡Bienvenido a Sage! Ahora está listo para empezar a leer el capítulo 1.



1

¡Bienvenido a Sage!

Las matemáticas son algo que hacemos, no algo que solo observamos. Por ello, la mejor manera de aprender cualquier rama de las matemáticas es realmente *empezar* e intentar las cosas. ¡Lo mismo es cierto para aprender Sage! Solo debemos conectarnos al “Sage Cell Server” (Servidor de Celdas de Sage) y sumergirnos en ello, trabajando los ejemplos de las siguientes páginas.

<https://sagecell.sagemath.org/>

1.1 Usando Sage como una calculadora

Antes que nada, siempre podemos usar Sage como una calculadora. Por ejemplo, si escribimos $2+3$ y presionamos “Evaluate” (“Evaluar”), entonces veremos que la respuesta es 5.

Las expresiones pueden hacerse tan complicadas como se desee. Para calcular el valor final en una inversión con interés simple del 6 % anual por 90 días, para un monto inicial de \$ 900, sabemos que la fórmula es $A = P(1 + rt)$, de manera que solo debemos escribir

Código de Sage

```
1 900 * (1 + 0.06 * (90 / 365))
```

y presionar “Evaluate”. Entonces veremos que la respuesta es 901,516 853 932 584 o \$ 901,52, después de redondear al centavo más próximo.

Nótese que el símbolo para la adición es el signo más (“+”) y para la sustracción es el guion (“-”). El símbolo para la multiplicación es el asterisco (“*”) y el símbolo para la división es el símbolo de quebrado o slash (“/”). Estas son las mismas reglas usadas por MS-Excel y muchos lenguajes de programación, incluyendo C, C++, Perl, Python y Java.

Para el interés compuesto necesitaremos usar exponentes. El símbolo para la exponenciación es la tilde circunfleja (“^”). Por ejemplo, consideremos el 12 % compuesto anualmente sobre un préstamo a sola firma por tres años, para un monto inicial de \$ 11000. Sabemos que la fórmula es $A = P(1 + i)^n$, así que debemos escribir

Código de Sage

```
1 11000 * (1 + 0.12)^3
```

y presionar “Evaluate”. Veremos entonces que la respuesta es 15 454,208 000 000 0 o \$ 15 454,21, después de redondear al centavo más próximo.

Por cierto, en lugar de usar el botón “Evaluate”, podemos simplemente presionar Shift+Enter en el teclado y tendremos el mismo efecto.

Advertencia: Una forma común de notación decimal, especialmente en el mundo de habla inglesa, es usar una coma cada tres dígitos (por ejemplo, escribiendo 11,000 en lugar de 11 000 o también 1,000,000 en lugar de 1 000 000). Es muy importante no usar esta notación en Sage. Si el lector está acostumbrado a escribir 11,000 para el número once mil, debe escribirlo como 11000 en Sage.

Para aquellos que no gusten de la tilde circunfleja ^ para la potenciación, también es posible usar dos asteriscos seguidos (“**”). Por ejemplo, podemos escribir

Código de Sage

```
1 11000 * (1 + 0.12)**3
```

lo que constituye una regresión al lenguaje de programación Fortran¹, que fue uno de los más populares durante las décadas de 1970 y 1980.

¿Qué hacemos si cometemos un error? Supongamos que en realidad queríamos trabajar con \$ 13 000 en lugar de \$ 11 000 en el ejemplo anterior. Simplemente hacemos click en el error y lo corregimos con el teclado, cambiando 11 por 13. Presionamos “Evaluate” nuevamente, y eso es todo.

Agrupando símbolos Cuando hay múltiples grupos de paréntesis en una fórmula, a veces los matemáticos usan corchetes como una especie de “superparéntesis”. Resulta que Sage necesita los corchetes para otros objetos —como las listas, por ejemplo—, de manera que debemos usar siempre paréntesis para agrupación dentro de las fórmulas. Por ejemplo, digamos que necesitamos evaluar

$$550 \frac{[1 - (1 + 0,05)^{-30}]}{0,05}.$$

No debemos escribir

Código de Sage

```
1 550 * [1 - (1 + 0.05)^(-30)] / 0.05
```

sino

Código de Sage

```
1 550 * (1 - (1 + 0.05)^(-30)) / 0.05
```

donde los corchetes se convierten en paréntesis.

Algunos libros de matemáticas antiguos usan llaves (“{” y “}”) como auxiliares para los paréntesis y los corchetes. Estos también, si son usados para agrupación en una fórmula, deben convertirse en paréntesis. Resulta que Sage, así como los libros de matemáticas modernos, usan las llaves para denotar conjuntos.

A propósito, la fórmula del ejemplo anterior no es artificial; es el valor de un préstamo al 5 % compuesto anualmente por 30 años, con un pago anual de \$ 550. La fórmula es llamada “el valor presente de una anualidad”.

¹Una nota histórica irrelevante: Dado que reescribir software es un proceso muy difícil y muy largo, muchas piezas importantes de software permanecieron escritas en FORTRAN hasta hace unos pocos años, y lo mismo ocurrió para software de negocios escrito en COBOL. Cada nuevo lenguaje de programación incorpora características de generaciones previas de lenguajes para hacer más fácil su aprendizaje, por lo que, de vez en cuando, uno ve raíces de lenguajes de programación realmente antiguos.

Tres errores que hacen más del 90 % de los errores comunes en Sage Existen tres errores que se cometen frecuentemente al usar Sage.

Supongamos que queremos expresar $11\,000x + 1200$. Una forma de hacerlo es escribir

Código de Sage

```
1 11000 * x + 1200
```

El primer error que se comete ocasionalmente es olvidar el asterisco entre 11000 y x . En Sage, se debe incluir ese asterisco, pues ese es el símbolo de la multiplicación.

El segundo error es que uno frecuentemente escribe mal los números largos como 11000. Algunos usuarios acostumbrados a la notación americana/inglesa, tienden a escribir 11,000, lo cual no es un número válido en Sage. Quien acostumbra usar la notación europea continental, escribiría 11.000, que Sage entiende como el número 11. Finalmente, la notación en español tiende a causar dos problemas. Por un lado, existe el peligro de olvidar escribir un dígito (o más), como 1100 en lugar de 11000. Este peligro se hace mayor cuando el número es más grande. Para evitar eso, la Real Academia Española (RAE) recomienda el uso de un espacio cada tres dígitos para tener una ayuda visual², como al escribir 11 000. Desafortunadamente, esta notación tampoco es válida en Sage.

Actualmente, empezando con la versión 9.1, Sage ha adoptado una notación, que heredó del lenguaje Python 3, en la que se pueden usar guiones bajos entre los dígitos de un número. Por ejemplo, podemos escribir $11\,000x + 1200$, como hicimos arriba, o podemos escribir también

Código de Sage

```
1 11_000 * x + 1_200
```

Esta notación exige no usar guiones bajos al principio o al final de un número, ni tener un grupo de dos o más de ellos entre dos dígitos. Fuera de estos cuidados, esta notación nos permite usar guiones bajos en la posición que deseemos dentro de un número. Pero nosotros convendremos en usarlos como separación cada tres dígitos, para tener mayor claridad en la escritura de nuestro código. Si el lector usa una versión de Sage anterior a la 9.1, debe abstenerse de usar estos guiones bajos.

El tercer error que se suele cometer es tener paréntesis desbalanceados. Cualquier expresión en Sage debe tener el mismo número de “(” como de “)” —ni más ni menos—.

Una lista más completa es dada en el apéndice A, “¿Qué hacer cuando uno está frustrado?”, en la página 275.

1.2 Usando Sage con funciones elementales

Ahora discutiremos cómo trabaja Sage con raíces cuadradas, logaritmos, exponenciales y otras funciones similares.

Raíces cuadradas Las funciones estándares “de colegio” están predefinidas en Sage. Por ejemplo, si escribimos

Código de Sage

```
1 sqrt(144)
```

y presionamos “Evaluate”, obtendremos la respuesta 12.

A partir de este punto, ya no se dirá ‘presionamos “Evaluate”’, para evitar la fatiga por repetición. Se asumirá que el lector sabe qué hacer.

²En realidad, la regla es un poco más complicada. Por ejemplo, para un número entero, se usan pequeños espacios cada tres dígitos, a menos que el número solo tenga cuatro dígitos; las partes entera y decimal de un número se tratan por separado, etc. Para mayores detalles, el lector puede consultar el Diccionario Panhispánico de Dudas en la dirección <http://lema.rae.es/dpd/>.

Dado que Sage prefiere las respuestas exactas, si intentamos

Código de Sage

```
1 sqrt(8)
```

obtenemos $2\sqrt{2}$. Si realmente necesitamos decimales, usamos en cambio

Código de Sage

```
1 N(sqrt(8))
```

y obtenemos 2.82842712474619, que es una aproximación decimal.

La función $N()$, que también puede escribirse $n()$, convertirá lo que está dentro de sus paréntesis en un número real, de ser posible. Usualmente eso es una expansión decimal. Así, a menos que lo que hay dentro de los paréntesis sea un entero³, entonces se obtendrá una aproximación.

Por otro lado, Sage asumirá que cualquier expresión decimal es una aproximación. Así que, por ejemplo,

Código de Sage

```
1 sqrt(3.4)
```

resultará en 1.84390889145858 sin la necesidad de usar $N()$. De la misma manera, si escribimos

Código de Sage

```
1 sqrt(4.0)
```

Sage nos mostrará el resultado 2.00000000000000.

Raíces de orden superior Las raíces de orden superior se pueden calcular de la misma manera que las potencias. Por ejemplo, para encontrar la raíz sexta de 64, hacemos

Código de Sage

```
1 64^(1/6)
```

para obtener 2.

Las constantes especiales π y e Frecuentemente en matemáticas necesitamos las constantes especiales π y e . Resulta que π está predefinida en Sage como `pi` (ambas letras minúsculas) y e está predefinida como `e` (nuevamente minúscula).

Podemos hacer algo así como

Código de Sage

```
1 numerical_approx(pi, digits=200)
```

para obtener varios dígitos de π o, similarmente,

Código de Sage

```
1 numerical_approx(sqrt(2), digits=200)
```

³Para ser matemáticamente correctos, deberíamos decir “un entero o una fracción con denominador expresable como el producto de una potencia de 5 y una potencia de 2”. Por ejemplo, fracciones con denominadores como 25, 16 y 80 pueden escribirse exactamente como decimales, mientras que con denominadores como 3, 15 y 14, no es posible. Nótese que $25 = 5^2$ y $16 = 2^4$, así como $80 = 2^4 \times 5$. Estos denominadores tienen solamente al 2 y al 5 como factores primos. Por otro lado, $3 = 3$ y $15 = 3 \times 5$, mientras que $14 = 2 \times 7$. Como podemos ver, estos denominadores tienen factores primos distintos de 2 y 5. Si el lector encuentra esto interesante, debería leer “Trabajando con enteros y Teoría de Números” en la página 140.

para obtener una expansión de alta precisión de $\sqrt{2}$. En estos casos obtenemos 200 dígitos. También podemos abreviar como

```
Código de Sage
1 N(sqrt(2), digits=200)
```

como hicimos antes. Las funciones `N()` y `n()` (son la misma) son abreviaciones de `numerical_approx` (“aproximación numérica”).

Completando con TAB Este es un buen momento para presentar una fantástica característica de Sage. Cuando estamos escribiendo un comando largo, como “`numerical_approx`” y en medio camino olvidamos el resto (¿es `approximation`?, ¿es `approximations`?, ¿es `appr`?), entonces podemos presionar la tecla TAB. Si solo un comando en la librería entera tiene ese prefijo, entonces Sage completará el resto por nosotros; caso contrario, nos presentará una lista de sugerencias. Esta es una característica muy útil cuando no podemos recordar los comandos exactos. Veremos otros ejemplos de características de ayuda integradas en la sección 1.10 en la página 44.

Aunque ya lo mencionamos antes, vale la pena recordar que, cuando estemos cansados de usar el botón “Evaluate” todo el tiempo, podemos simplemente presionar Shift y Enter a la vez. Esto tiene el mismo efecto.

¿Sage diferencia entre mayúsculas y minúsculas? Probablemente todos hemos pasado por la situación en la que al ingresar una contraseña, descubrimos que ha sido rechazada por no tener las mayúsculas o minúsculas correctas —por ejemplo, probablemente tenemos la tecla “CAPS-LOCK” o “BloqMayús” activada—. De la misma manera, Sage reconoce que `Pi` es diferente que `pi` y `Sin` es diferente de `sin`.

Las únicas cuatro excepciones conocidas por el autor son `True` y `False`, así como `i` y `n()`. No haremos uso extensivo de `True` y `False` hasta el capítulo 5, pero vale la pena mencionar que podemos escribirlos como `true` y `false`, respectivamente, y Sage los reconocerá como iguales.

Por otro lado, probablemente el lector haya escuchado que $\sqrt{-1}$ es llamada la “unidad imaginaria” y denotada por i . En Sage, podemos escribir `i` o `I`, y en ambos casos reconocerá que nos referimos a $\sqrt{-1}$. Si el lector no ha escuchado de los números imaginarios o i , puede ignorar este hecho sin perjuicio.

Finalmente, como hemos visto ya, podemos escribir `n(sqrt(2))` o `N(sqrt(2))`, y Sage los tratará de manera idéntica.

Hasta donde el autor sabe, esta característica solo se aplica a `True`, `False`, $\sqrt{-1}$ y `N`. En todos los demás casos, el uso de mayúsculas es consecuente.

Exponenciales Así como

```
Código de Sage
1 2^3
```

nos da 2^3 y similarmente

```
Código de Sage
1 3^3
```

nos da 3^3 , si deseamos expresar e^3 solo debemos escribir

```
Código de Sage
1 e^3.
```

Para una aproximación decimal, podemos hacer

```
Código de Sage
1 N(e^3)
```

También vale la pena mencionar que algunos libros escriben

$$\exp(5 \cdot 11 + 8)$$

en lugar de

$$e^{5 \cdot 11 + 8},$$

y Sage también acepta esta notación. Por ejemplo,

```
Código de Sage
1 exp(5 * 11 + 8) - e^(5 * 11 + 8)
```

se evalúa como 0. (No debemos olvidar el asterisco entre el 5 y el 11.)

Logaritmos Por supuesto, Sage conoce los logaritmos —pero debemos ser cuidadosos—. Existen varios tipos de logaritmos, incluyendo el logaritmo común (en base 10), el logaritmo natural, el logaritmo binario y el logaritmo de cualquier otra base que deseemos. En colegio, “log” se refiere al logaritmo común y “ln” se refiere al logaritmo natural. Sin embargo, a partir de la materia de cálculo, “log” indica el logaritmo natural. Dado que Sage fue pensado principalmente para trabajos a nivel universitario y superior, entonces es simplemente natural (broma intencional) que se haya decidido que el logaritmo natural sea “log”.

Entonces, para calcular el logaritmo natural de 100 escribimos

```
Código de Sage
1 N(log(100))
```

para el logaritmo común de 100 escribimos

```
Código de Sage
1 N(log(100, 10))
```

y para el logaritmo binario de 100 escribimos

```
Código de Sage
1 N(log(100, 2))
```

lo cual por supuesto podemos generalizar. Por ejemplo, para encontrar el logaritmo de 100 en base 42 escribimos

```
Código de Sage
1 N(log(100, 42))
```

No olvidemos que Sage es excelente en encontrar respuestas exactas. Intentemos

```
Código de Sage
1 log(sqrt(100^3), 10)
```

y obtendremos 3, la respuesta *exacta*.

Un ejemplo de finanzas Supongamos que depositamos \$ 5000 en una cuenta que gana 4,5 % compuesto mensualmente. Nos gustaría saber cuándo el total alcanzará los \$ 7000. Usando la fórmula $A = P(1+i)^n$, donde P es la cantidad invertida, A es el monto final, i es la razón del interés por mes y n es el número de periodos de composición (número de meses, para este caso), tenemos

$$\begin{aligned} A &= P(1+i)^n \\ 7000 &= 5000(1+0,045/12)^n \\ 7000/5000 &= 5000(1+0,045/12)^n \\ 1,4 &= (1+0,045/12)^n \\ \log 1,4 &= \log(1+0,045/12)^n \\ \log 1,4 &= n \log(1+0,045/12) \\ \frac{\log 1,4}{\log(1+0,045/12)} &= n \end{aligned}$$

Así, en Sage debemos escribir

Código de Sage

```
1 log(1.4) / log(1 + 0.045/12)
```

para obtener 89.8940609330801. De manera que ahora sabemos que se necesitarán 90 meses o, mejor aun, 7 años y 6 meses.

Este es un ejemplo de *solución asistida por computadora*. Adicionalmente, Sage también está dispuesto a resolver el problema de principio a fin sin intervención humana. Veremos esto a detalle en la página 44.

Matemáticas puras y raíces cuadradas A continuación presentamos un par de notas sobre las raíces cuadradas, que serán de particular interés para los estudiantes de matemáticas, o cualquiera que desee trabajar con números complejos. Si el lector lo prefiere, puede saltar este subtítulo y el siguiente sobre números complejos, y continuar con “Usando Sage para trigonometría” en la página 8 o “Usando Sage para graficar en dos dimensiones” en la página 9.

En el espíritu de la absoluta pomposidad, sabemos que $(-2)^2 = 4$ al igual que $2^2 = 4$. Así que, si deseamos todas las raíces cuadradas de un número podemos usar

Código de Sage

```
1 sqrt(4, all=True)
```

para obtener

```
[2, -2]
```

donde los corchetes denotan una lista. Cualquier sucesión de números separados por comas y rodeados de corchetes es una lista. Veremos otros ejemplos a lo largo de este libro.

Un primer vistazo a los números complejos Los que deseen trabajar con números complejos pueden escribir

Código de Sage

```
1 sqrt(-4, all=True)
```

para obtener

```
[2*I, -2*I]
```

donde, como ya mencionamos, la letra mayúscula I representa $\sqrt{-1}$, la unidad imaginaria, que también puede escribirse como una i minúscula si lo deseamos.

Aunque Sage es excelente en análisis complejo y puede producir hermosos gráficos de funciones de una variable compleja, no entraremos en esos detalles aquí. Los gráficos a colores de funciones complejas serán tratados en el apéndice únicamente electrónico de este libro, “Graficando en color, en 3D, y animaciones”, disponible para descarga gratuita en la página web del libro, www.sage-para-estudiantes.com.

1.3 Usando Sage para trigonometría

Algunos estudiantes no están familiarizados con la trigonometría o están en algún curso que no guarda relación con ella. Si es así, se puede saltar esta sección con confianza e ir directamente a la siguiente, que empieza en la página 9.

Los comandos para trigonometría son muy intuitivos, pero es importante recordar que Sage trabaja en radianes. De manera que si deseamos saber el valor del seno de $\pi/3$, debemos escribir

Código de Sage

```
1 sin(pi / 3)
```

con lo que obtenemos $1/2\sqrt{3}$.

Como habíamos notado antes, esta es una respuesta exacta y no una simple aproximación decimal (Sage está predominantemente orientado a las respuestas exactas). Esto puede resultar irritante si, por ejemplo, pedimos el coseno de $\pi/12$. En ese caso, escribimos

Código de Sage

```
1 cos(pi / 12)
```

y obtenemos $1/4\sqrt{6} + 1/4\sqrt{2}$, lo que es particularmente insatisfactorio si esperábamos un decimal. En cambio, con

Código de Sage

```
1 N(cos(pi/12))
```

obtenemos 0.965925826289068, una buena aproximación decimal.

Veremos que Sage es bastante perceptivo cuando se trata de saber cuándo las funciones fallarán. En particular, tratemos de evaluar la tangente en una de sus asíntotas verticales, digamos

Código de Sage

```
1 tan(pi / 2)
```

lo que producirá la útil respuesta “Infinity” (infinito).

Las funciones trigonométricas “recíprocas”, cotangente, secante, cosecante, que son tan importantes en cálculo pero molestas en las calculadoras de bolsillo, están predefinidas en Sage. Se las denota por *cot*, *sec* y *csc*, respectivamente.

Las funciones trigonométricas inversas también están disponibles. Se usan de la misma manera que las funciones trigonométricas. Por ejemplo, si escribimos

Código de Sage

```
1 arcsin(1 / 2)
```

recibimos la respuesta

$1/6\pi$

como era de esperarse. Similarmente,

Código de Sage

```
1 arccos(1/2)
```

produce

$1/3\pi$.

Las abreviaciones usuales son todas conocidas y usadas por Sage. Aquí tenemos una lista completa:

Notación matemática	Comando forma larga	Comando forma corta
$\sin^{-1}(x)$	<code>arcsin(x)</code>	<code>asin(x)</code>
$\cos^{-1}(x)$	<code>arccos(x)</code>	<code>acos(x)</code>
$\tan^{-1}(x)$	<code>arctan(x)</code>	<code>atan(x)</code>
$\cot^{-1}(x)$	<code>arccot(x)</code>	<code>acot(x)</code>
$\sec^{-1}(x)$	<code>arcsec(x)</code>	<code>asec(x)</code>
$\csc^{-1}(x)$	<code>arccsc(x)</code>	<code>acsc(x)</code>

También podemos usar Sage para graficar las funciones trigonométricas. Trataremos eso en la sección titulada “Usando Sage para graficar en dos dimensiones”, en la página 12.

Convirtiendo entre grados y radianes Recordemos de la clase de trigonometría que, para convertir de radianes a grados, basta multiplicar por $180/\pi$, mientras que para convertir de grados a radianes, simplemente multiplicamos por $\pi/180$. De acuerdo a esto, lo que escribimos para convertir $\pi/3$ a grados es

Código de Sage

```
1 (pi/3) * (180/pi)
```

que produce 60, mientras que para convertir 60 grados a radianes, debemos escribir

Código de Sage

```
1 60 * (pi/180)
```

que produce $1/3\pi$.

Una forma de recordar esta técnica es pensar que un transportador tiene 180 grados y parece una porción de pastel. Ahora bien, las palabras “porción” y “pastel” ambas empiezan con la letra “p”, y π es su equivalente en el alfabeto griego. Entonces, por ejemplo, 90 grados hacen la mitad de una *porción de pastel* (un transportador), mientras que 60 grados hacen la tercera parte de una *porción de pastel*, y por ello son lo mismo que $\pi/2$ y $\pi/3$, respectivamente. Similarmente, si vemos $\pi/6$, sabemos que es la sexta parte de un transportador (el pedazo de pastel), por lo que equivale a 30 grados.

1.4 Usando Sage para graficar en dos dimensiones

La curva favorita del autor es la parábola, así que comencemos ahí. Escribimos

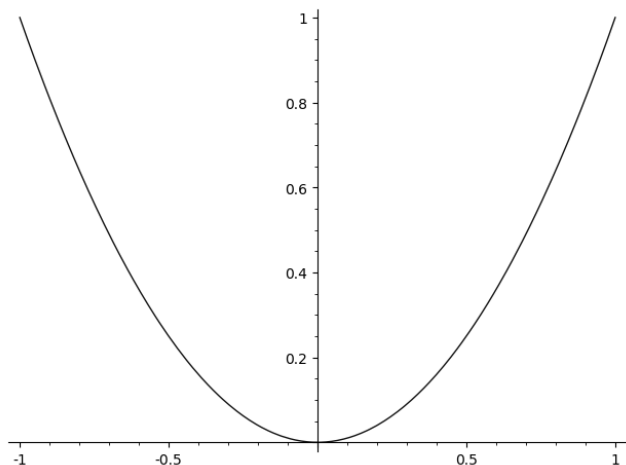
Código de Sage

```
1 plot(x^2)
```

y obtenemos una atractiva gráfica definida en el intervalo

$$-1 < x < 1,$$

el cual es el dominio por defecto para el comando `plot`. Sage ajustará el rango del eje y como sea necesario para mostrar la gráfica completa en ese dominio. El resultado se ve a continuación:

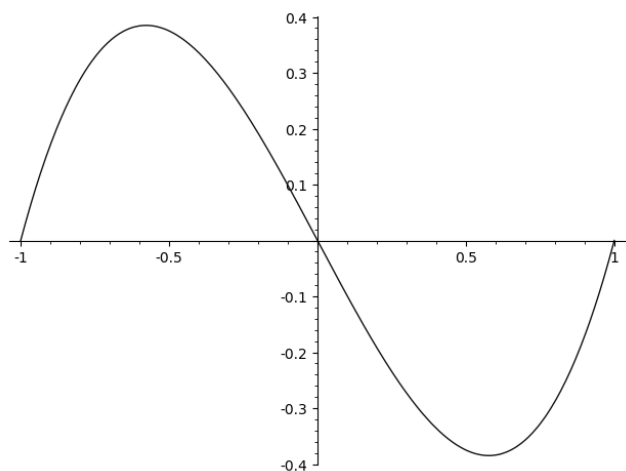


De manera similar podemos escribir

Código de Sage

```
1 plot(x^3 - x)
```

que resulta en una gráfica elegante y atractiva como vemos a continuación:

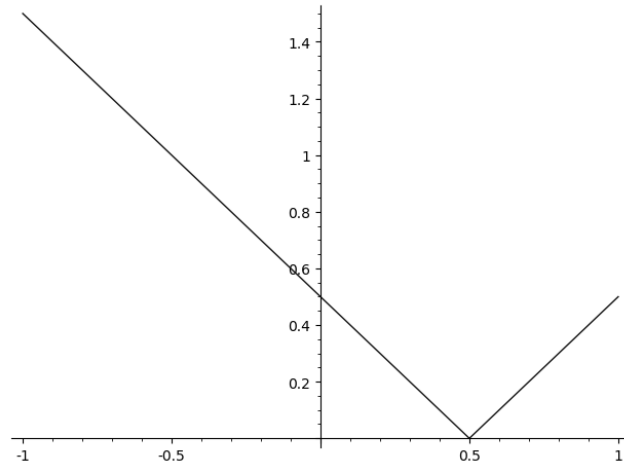


Para la función $|x - 1/2|$ podemos escribir

Código de Sage

```
1 plot(abs(x - 1/2))
```

lo que produce el siguiente gráfico:

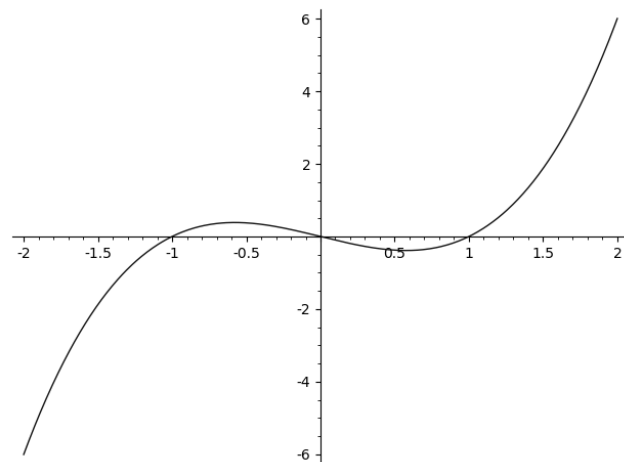


¿Qué hacemos si deseamos un rango de variación diferente para el eje x ? Por ejemplo, para graficar en $-2 < x < 2$, debemos escribir

Código de Sage

```
1 plot(x^3-x, -2, 2)
```

y obtenemos el gráfico deseado, es decir:



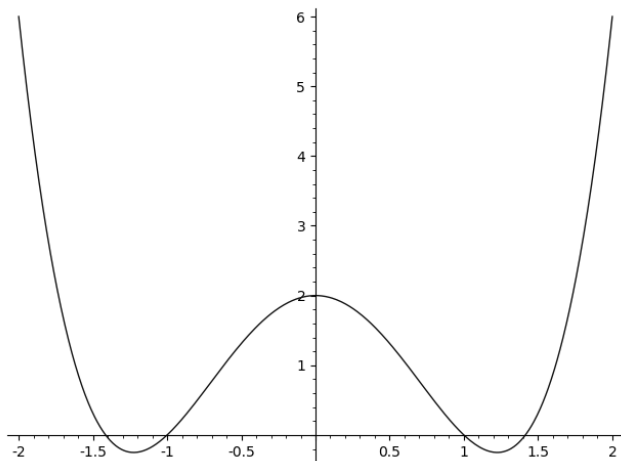
Ahora es buen momento para mencionar que siempre se pueden guardar en un archivo los gráficos que Sage ha generado. Solo debemos hacer click derecho en la imagen mostrada en el navegador web y guardar el archivo en la computadora. Luego podemos importar la imagen resultante a cualquier reporte o artículo que estemos escribiendo.

Un gráfico muy interesante se obtiene con

Código de Sage

```
1 plot(x^4 - 3*x^2 + 2, -2, 2)
```

(Pero nuevamente tengamos cuidado con el asterisco entre 3 y x en “ $3*x^2$ ”: ¡recibiremos un mensaje de error si lo olvidamos!) El resultado es



Otro punto con el que debemos tener cuidado es que, para graficar $y = \sin(t)$, debemos introducir

Código de Sage

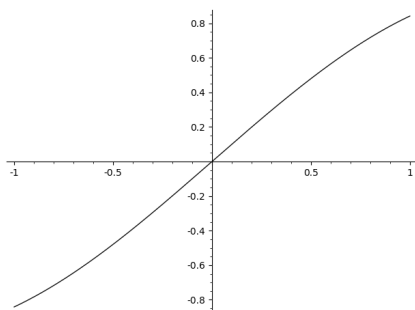
```
1 plot(sin(x))
```

o, para apreciarlo mejor aun,

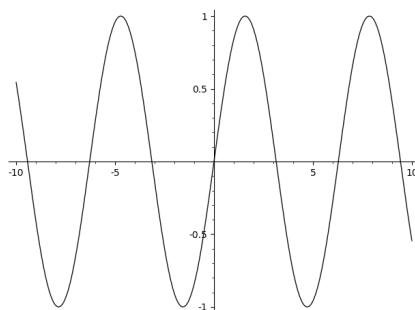
Código de Sage

```
1 plot(sin(x), -10, 10)
```

Esto es porque el comando `plot` no espera la variable t , sino x . Las imágenes que obtenemos con estos dos últimos comandos son



`plot(sin(x))`



`plot(sin(x), -10, 10)`

De hecho, si escribiésemos

Código de Sage

```
1 plot(sin(t))
```

Sage respondería con el mensaje

```
NameError: name 't' is not defined
```

porque en este caso, ignora el significado de “ t ”. Una forma alternativa de lidiar con esta situación se discute en la página 101.

1.4.1 Controlando el área de visualización de un gráfico

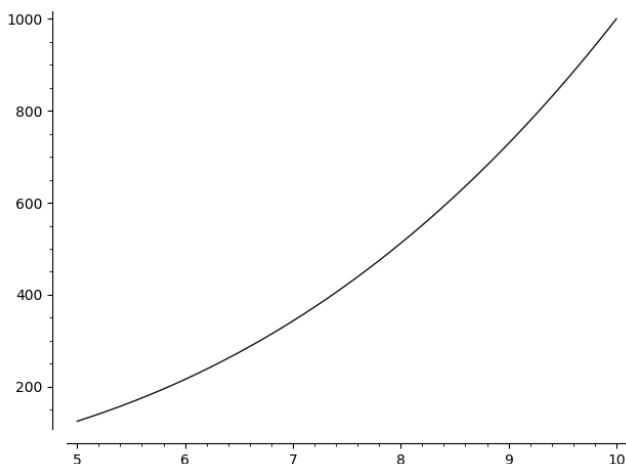
La mayor parte del tiempo, un gráfico como se genera por defecto va a ser exactamente como deseamos —pero no siempre—. La excepción más común es una función con asíntotas verticales. Aquí vamos a discutir cómo ajustar el área de visualización.

Yendo “fuera de escala” Consideremos la gráfica de x^3 desde $x = 5$ hasta $x = 10$, que está dada por

Código de Sage

```
1 plot(x^3, 5, 10)
```

Como podemos imaginar, la función va desde $5^3 = 125$ hasta $10^3 = 1000$, por lo que el origen debería de aparecer muy abajo en el gráfico. Sage nos muestra lo siguiente:



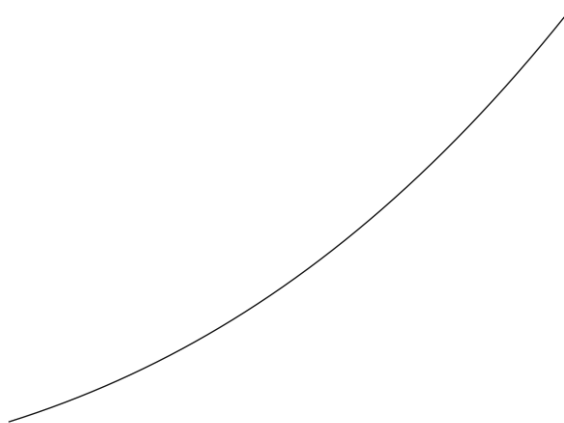
El indicio de que la ubicación del eje x en pantalla no es donde normalmente estaría, es que los ejes no se intersectan. Esto es para decirnos que el origen se encuentra lejos. Cuando los ejes sí se intersectan en pantalla, entonces el origen es (en la realidad y en la pantalla) donde se cortan.

También, si lo deseamos, podemos esconder los ejes con

Código de Sage

```
1 plot(x^3, 5, 10, axes=False),
```

y eso produce



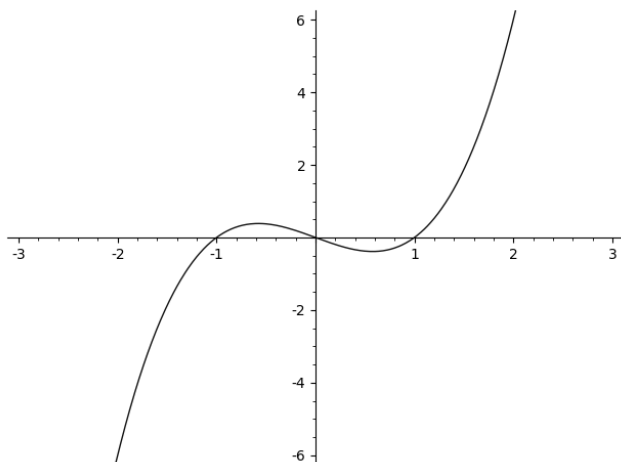
que es considerablemente menos informativo.

Forzando el rango del eje y de una gráfica Si por alguna razón deseamos restringir el rango en el eje y de una gráfica, de manera que esté acotado entre dos valores, podemos hacerlo fácilmente. Por ejemplo, para mantener $-6 < y < 6$, podemos hacer

Código de Sage

```
1 plot(x^3-x, -3, 3, ymin=-6, ymax=6)
```

con lo que obtenemos



Esta es una característica importante, porque normalmente Sage pretende mostrar la gráfica entera. Como consecuencia, se asegurará que el eje y sea lo suficientemente largo para incluir cada punto, desde el mínimo hasta el máximo. Para algunas funciones, el mínimo, el máximo o ambos podrían ser enormes.

Gráficas de funciones con asíntotas verticales La forma en que Sage (y todas las herramientas de álgebra computacional) determina la gráfica de una función es generar un gran número de puntos en el intervalo de las x s, y entonces evaluar la función en cada uno de esos puntos. Para ser precisos, si deseamos graficar $f(x) = 1/x^2$ entre $x = -4$ y $x = 4$, la computadora podría elegir 10 000 valores aleatorios de x entre esos dos números, encontrar los valores y correspondientes al evaluar $f(x)$, y finalmente dibujar los puntos en las ubicaciones apropiadas de la imagen.

Así que, si la gráfica tiene una asíntota vertical, entonces los valores serán muy grandes cerca de ella. Debido a esto, cuando graficamos una función racional, debemos estar seguros de restringir los valores de y . Por ejemplo, comparemos los siguientes gráficos, que produjimos con los estos comandos:

Gráfica 1:

Código de Sage

```
1 plot(1/(x^3-x), -2, 2)
```

Gráfica 2:

Código de Sage

```
1 plot(1/(x^3-x), -2, 2, ymin=-5, ymax=5)
```

Gráfica 3:

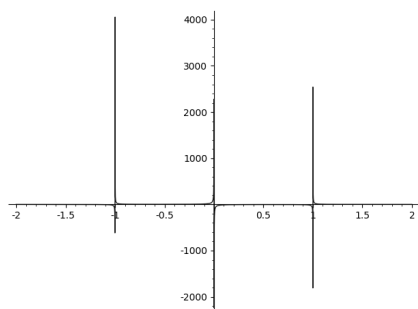
Código de Sage

```
1 plot(1/(x^3-x), (x, -2, 2), detect_poles=True, ymin=-5, ymax=5)
```

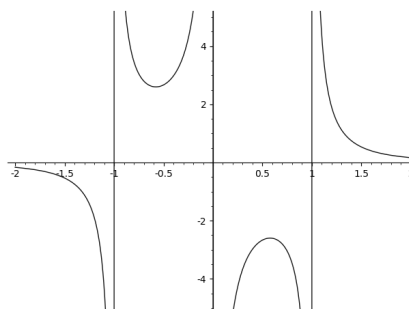
Gráfica 4:

Código de Sage

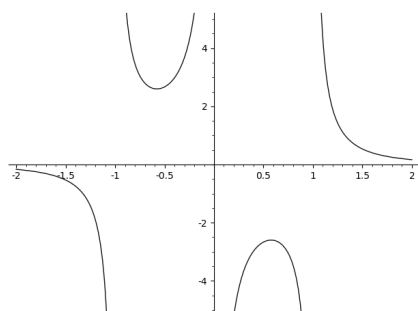
```
1 plot(1/(x^3-x), (x, -2, 2), detect_poles='show', ymin=-5, ymax=5)
```



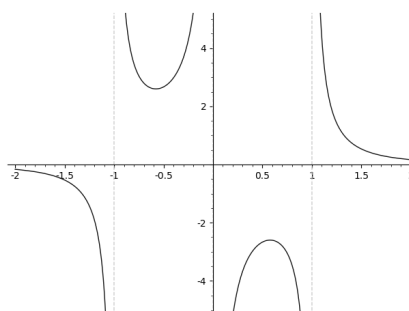
Gráfica 1



Gráfica 2



Gráfica 3



Gráfica 4

Como podemos ver, la primera gráfica es un desastre total. La segunda elimina los valores muy grandes y muy pequeños en el eje y, pero aún intenta conectar los varios “miembros” de la gráfica. Cuando ajustamos la opción “detect_poles” a True, entonces Sage deducirá que los pedazos no están conectados. Finalmente, vemos en el cuarto caso que lo mismo ocurrirá si ajustamos “detect_poles” a 'show', pero, adicionalmente, Sage nos mostrará las asíntotas verticales de manera que no se confundan con la gráfica.

Queda aclarar un pequeño punto. ¿Notó el lector que en los comandos listados arriba, 'show' está entre comillas, pero True no lo está? Eso se debe a que True y False ocurren tan frecuentemente en matemáticas, que son palabras clave predefinidas en Sage. Sin embargo, 'show' ocurre con menor frecuencia y por lo tanto no está predefinida, y debemos ponerla en comillas.

Las comillas simples se usan para delimitar lo que se conoce como *cadenas de caracteres*. Son básicamente conjuntos de letras que forman un texto. Como alternativa, Sage nos permite usar las comillas dobles para este mismo propósito. Por ejemplo, podríamos escribir "show". No existe ninguna regla que nos obligue a usar una u otra forma; depende del gusto personal. En este libro usaremos comillas simples, por ser estas más fáciles de escribir en el teclado del autor.

1.4.2 Superponiendo varias gráficas en una sola imagen

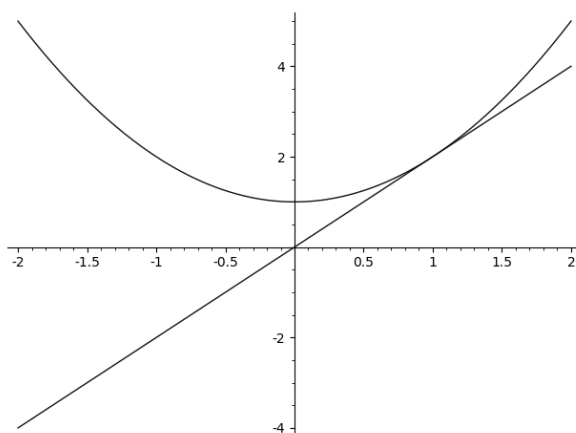
Ahora veremos cómo superponer gráficas. Veremos que Sage concibe esto como una “suma” de imágenes, incluso usando el signo más “+”.

Un ejemplo de Cálculo: La recta tangente Supongamos que deseamos obtener una imagen de $f(x) = x^2 + 1$ en el intervalo $-2 < x < 2$ y la recta tangente a esa parábola en $x = 1$. Dado que $f(1) = 2$, y $f'(1) = 2$, sabemos que la recta debe pasar por el punto $(1, 2)$ y tendrá pendiente 2. No es difícil determinar que la ecuación de tal recta es $y = 2x$. La cuestión aquí es que queremos ambas gráficas en la misma imagen. El comando para esto es

Código de Sage

```
1 plot(2*x, -2, 2) + plot(x^2+1, -2, 2)
```

con lo que obtendremos la siguiente imagen:



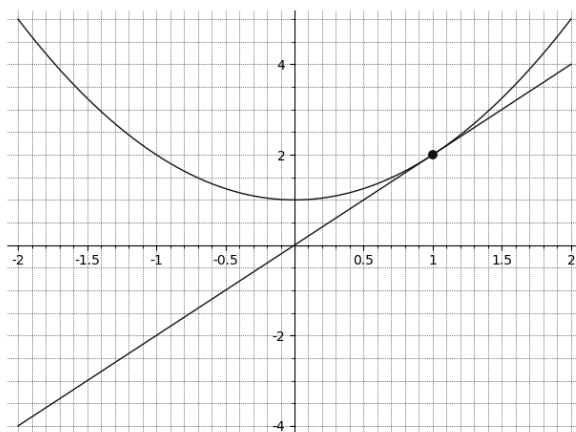
Así, sumar dos gráficas hace una combinación o superposición de ambas. El signo de suma indica a Sage que debe graficar las dos curvas, una encima de la otra.

Ahora, podemos hacer una mejora añadiendo un punto en el punto de tangencia $(1, 2)$, y tal vez añadiendo un cuadrículado. El comando para este propósito es

Código de Sage

```
1 plot(2*x, -2, 2, gridlines='minor') + plot(x^2+1, -2, 2) + point((1,2),
↪ size=50)
```

Obtendremos con ello la imagen siguiente:



Como podemos ver, hemos añadido el punto usando el comando `point` y este también fue superpuesto usando el símbolo de suma. El argumento `gridlines='minor'` nos da un cuadrículado muy satisfactorio, que puede ser de mucha utilidad en la preparación de gráficos de alta calidad y fáciles de leer para reportes de laboratorio, uso en la clase o artículos para publicación. Discutiremos otras maneras de realizar cuadrículados, así como otras maneras de anotar gráficos en la sección 3.1 en la página 89. Este gráfico específico aparecerá con muchos más detalles y anotaciones en la página 93.

A propósito del comando anterior, es decir la suma de las dos gráficas más un punto, es importante sea introducido por completo en la misma línea de código. No podemos tener un salto de línea. En este libro no se pudo escribir en una sola línea debido a que la página es finita. Debemos estar seguros que no haya un salto de línea cuando escribamos el comando.

Un ejemplo más complejo Supongamos que estuviésemos investigando polinomios, y deseásemos conocer el efecto de variar el último número de

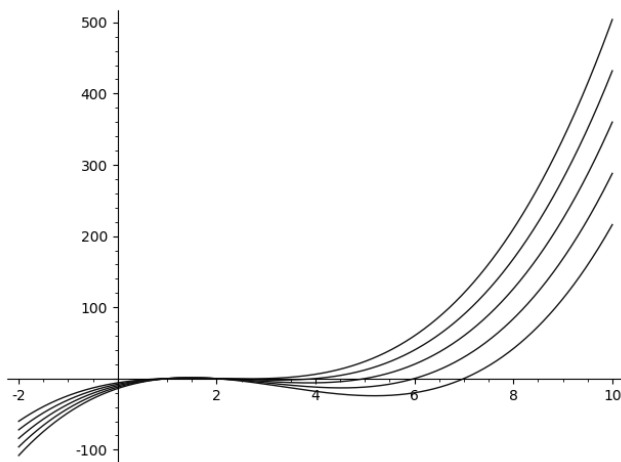
$$y = (x - 1)(x - 2)(x - 5)$$

en su gráfica. (En otras palabras, queremos saber qué ocurre si cambiamos el “5” en esa ecuación por otro número.) Entonces, tal vez podríamos considerar $(x - 3)$ y $(x - 4)$ como reemplazos de $(x - 5)$, así como tal vez $(x - 6)$ y $(x - 7)$. Esto puede ser hecho con

Código de Sage

```
1 plot((x-1)*(x-2)*(x-3), -2, 10) + plot((x-1)*(x-2)*(x-4), -2, 10) +
  ↪ plot((x-1)*(x-2)*(x-5), -2, 10) + plot((x-1)*(x-2)*(x-6), 2, 10) +
  ↪ plot((x-1)*(x-2)*(x-7), -2, 10)
```

donde se ha elegido el dominio, desde $x = -2$ hasta $x = 10$, de manera arbitraria. Una vez más, cuando ingresemos el comando anterior, es importante que esté en una (muy larga) línea única. Si lo hicimos de manera correcta, obtenemos

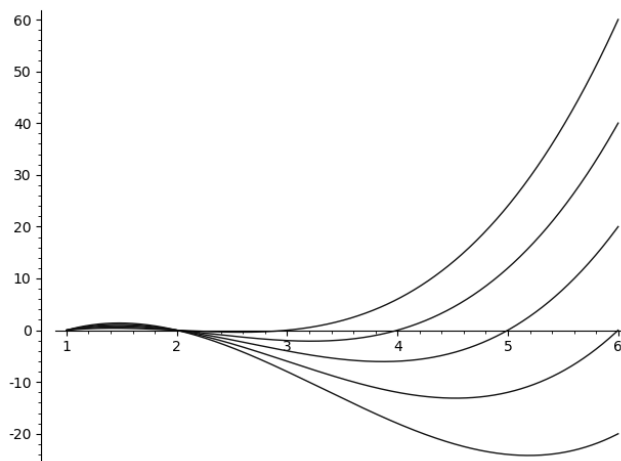


Tal parece que debemos ampliar un poco el gráfico para ver más detalle. Si quisiéramos cambiar el dominio de $-2 < x < 10$ a algo como $1 < x < 6$, solo necesitaríamos cambiar cada uno de los -2 s por 1 s y cada uno de los 10 s por 6 s. Así que escribiríamos

Código de Sage

```
1 plot((x-1)*(x-2)*(x-3), 1, 6) + plot((x-1)*(x-2)*(x-4), 1, 6) +
  ↪ plot((x-1)*(x-2)*(x-5), 1, 6) + plot((x-1)*(x-2)*(x-6), 1, 6) +
  ↪ plot((x-1)*(x-2)*(x-7), 1, 6)
```

y obtendríamos como resultado



Hay una cuestión importante aquí. Debemos ser cuidadosos de hacer los cambios correctamente. Y este requerimiento ya de por sí desafiante se hace mucho peor debido a que el código es difícil de leer. Sin embargo, muy pronto aprenderemos cómo hacerlo más legible, y esto a su vez hará que sea más fácil de modificar.

En cualquier caso, ahora podemos entender mejor lo que el último número del polinomio hace: controla la ubicación, o coordenada x , de la última intersección de la curva con el eje x .

Nuevamente, recordamos un tecnicismo al cual debemos prestar atención. No debemos introducir saltos de línea entre los `plot`s y los signos de suma. Así que, si el lector tiene problemas ejecutando las dos últimas instrucciones, debe asegurarse primero que no introdujo dichos saltos en medio. Este es un ejemplo de código que es totalmente funcional, pero difícil de leer o alterar, y ciertamente no tan fácil de entender. Para Sage, cada una de estas instrucciones debe ser escrita en una sola línea, de manera similar a un párrafo largo en un procesador de texto—podemos usar el ajuste automático de línea (donde el texto *parece* abarcar varias líneas), pero no introducir saltos de línea (con la tecla ENTER). Así, no podemos escribir

Código de Sage

```
1 plot((x-1)*(x-2)*(x-3), 1, 6)
2 + plot((x-1)*(x-2)*(x-4), 1, 6)
3 + plot((x-1)*(x-2)*(x-5), 1, 6)
4 + plot((x-1)*(x-2)*(x-6), 1, 6)
5 + plot((x-1)*(x-2)*(x-7), 1, 6)
```

lo cual Sage interpreta como cinco comandos distintos en lugar de uno solo extenso.

Sin embargo, hay una salida elegante de este dilema. Podemos escribir lo siguiente, que produce el gráfico deseado.

Código de Sage

```
1 P1 = plot((x-1)*(x-2)*(x-3), 1, 6)
2 P2 = plot((x-1)*(x-2)*(x-4), 1, 6)
3 P3 = plot((x-1)*(x-2)*(x-5), 1, 6)
4 P4 = plot((x-1)*(x-2)*(x-6), 1, 6)
5 P5 = plot((x-1)*(x-2)*(x-7), 1, 6)
6
7 P = P1 + P2 + P3 + P4 + P5
8 P.show()
```

De seguro que todos estaremos de acuerdo que este es un código mucho más legible.

Tópicos adicionales sobre gráficas y graficación Hasta este punto solo hemos arañado la superficie de la graficación en Sage. Existe una librería muy rica de varios tipos de gráficas e imágenes que Sage puede producir para nosotros. Un capítulo entero de este libro está dedicado a “Técnicas avanzadas de graficación”, empezando en la página 89. No es necesario leer el capítulo entero en orden. Se puede usar el índice o la tabla de contenidos para seleccionar exactamente el tipo de gráfica deseado, y solo leer la sección correspondiente.

Para el lector que sienta curiosidad acerca de cómo Sage en realidad genera las imágenes, hablamos acerca de ello en la página 251, pero puede que sea necesario leer la mayor parte del capítulo 5 para poder seguir la discusión.

1.5 Matrices y Sage, parte uno

En esta sección aprenderemos cómo resolver sistemas de ecuaciones lineales usando matrices en Sage. Esta sección asume que el lector nunca ha trabajado con matrices o, alternativamente, las ha olvidado. Los expertos en Álgebra Lineal pueden saltar a la subsección 1.5.3 en la página 21. Por otro lado, algunos lectores no tendrán ningún interés específico en matrices y pueden saltar a la sección 1.6 en la página 28, donde aprenderemos a definir nuestras propias funciones en Sage.

1.5.1 Una primera experiencia con matrices

Supongamos que deseamos resolver el siguiente sistema de ecuaciones:

$$\begin{aligned} 3x - 4y + 5z &= 14 \\ x + y - 8z &= -5 \\ 2x + y + z &= 7 \end{aligned}$$

Primero debemos convertirlo en la siguiente matriz:

$$A = \left[\begin{array}{ccc|c} 3 & -4 & 5 & 14 \\ 1 & 1 & -8 & -5 \\ 2 & 1 & 1 & 7 \end{array} \right].$$

Notemos que los coeficientes de las x s aparecen todos en la primera columna, los coeficientes de las y s aparecen en la segunda columna, y los coeficientes de las z s aparecen en la tercera columna. La cuarta columna almacena los términos independientes (las constantes). De esa manera hemos encapsulado y abreviado toda la información del problema a resolver. Más aun, observemos que las sumas están representadas por coeficientes positivos y las restas por coeficientes negativos. A propósito de todo esto, la línea vertical entre la tercera y cuarta columnas es solamente decorativa —su propósito es mostrar que la cuarta columna es “especial” en el sentido que contiene las constantes, mientras que las otras columnas representan las variables x , y y z —.

Las matrices tienen una forma especial llamada la “forma escalonada reducida por filas”, frecuentemente abreviado como *RREF*⁴ por sus siglas en inglés. La RREF es, desde cierta perspectiva, la descripción más simple del sistema de ecuaciones que es aún verdadera. La forma escalonada reducida por filas de esta matriz A es

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{array} \right].$$

⁴**Nota del traductor:** “Reduced Row Echelon Form”. Sage tiene el comando `rref` para obtener la forma escalonada reducida por filas de la matriz, así que usamos la abreviación en inglés para recordarlo.

Podemos traducir esto literalmente como

$$1x + 0y + 0z = 3$$

$$0x + 1y + 0z = 0$$

$$0x + 0y + 1z = 1$$

o, en notación más simple,

$$x = 3, \quad y = 0, \quad z = 1,$$

la cual es la solución. Tomémonos un momento ahora, introduzcamos estos tres valores en el sistema original, y veamos que lo satisfacen.

1.5.2 Complicaciones al convertir sistemas lineales

Al leer la discusión previa, uno puede pensar que lo que estamos haciendo es algún tipo de “solución milagrosa”, rápidamente capaz de resolver cualquiera de un gran número de problemas tediosos que, con papel y lápiz, suelen tomar mucho tiempo y esfuerzo.

En efecto, el álgebra matricial es una solución milagrosa ahora que tenemos las computadoras. Antes de ellas, los problemas grandes de álgebra lineal eran considerados extremadamente desagradables. Sin embargo, ahora que tenemos computadoras, y dado que estas pueden realizar operaciones de manera esencialmente instantánea, este tópico es una excelente forma de resolver enormes sistemas de ecuaciones lineales. A su vez, los matemáticos pueden abordar problemas importantes de la ciencia y de la industria, pues los problemas del mundo real frecuentemente tienen muchas variables.

Este método, sin embargo, es mejor descrito como “semiautomático” más que “automático”. La razón de esto es que un ser humano primero debe convertir muy cuidadosamente un problema verbal en un sistema de ecuaciones lineales. No cubriremos ese aspecto aquí. El segundo paso es convertir ese sistema en una matriz, lo cual no es algo demasiado trivial. De hecho, generalmente ocurre que los alumnos cometen errores en este paso, así que invertiremos un poco de tiempo con un ejemplo detallado.

Consideremos este sistema de ecuaciones:

$$2x - 5z + y = 6 + w$$

$$5 + z - y = 0$$

$$w + 3(x + y) = z$$

$$1 + 2x - y = w - 3x$$

Tal como está presentado, este sistema tiene las siguientes “trampas”:

- En la primera ecuación, las variables están desordenadas, lo cual es extremadamente común. Frecuentemente, estudiantes y profesionales fallan en notar esto y colocan los coeficientes equivocados en los lugares equivocados. Siempre podemos usar el orden que deseemos, siempre y cuando mantengamos este orden en cada una de las otras ecuaciones. Sin embargo, para evitar cometer un error, los matemáticos frecuentemente ponen las variables en orden alfabético.
- También en la primera ecuación, la variable w está en el lado equivocado y debemos moverla a la izquierda del signo de igualdad. Es necesario que todas las variables se posicionen en un solo lado del signo igual, y que todas las constantes estén al otro lado.

Con estas dos enmiendas, la primera ecuación es ahora

$$-w + 2x + y - 5z = 6.$$

- La segunda ecuación tiene la constante en el lado equivocado, así que debemos moverla al otro lado del símbolo de igualdad, recordando cambiar su signo.
- Como si fuera poco, x y w no están presentes en la segunda ecuación. Asumimos que sus coeficientes son cero.

Con estas dos enmiendas, la segunda ecuación es ahora

$$0w + 0x - y + z = -5.$$

- La tercera ecuación tiene paréntesis—eso no es conveniente. Recordamos que $3(x + y) = 3x + 3y$.
- También en la tercera ecuación, la variable z está en el lado equivocado. Una vez más, debemos tener cuidado de convertir $+z$ en $-z$ al cruzar la igualdad.
- Un tercer defecto en la tercera ecuación es que no existe una constante específica. Asumiremos que esta es cero.

Al corregir estos tres defectos, la tercera ecuación se convierte en

$$w + 3x + 3y - z = 0.$$

- El verdadero “delincuente” de este sistema es la cuarta ecuación. Tenemos w en el lado equivocado y, lo que es peor, hay dos ocurrencias de la variable x . Cuando movamos $-3x$ de la derecha a la izquierda de la igualdad, se convertirá en $+3x$ y se combinará con el $+2x$ que se encuentra ya ahí, para formar $+5x$. Como si no fuese suficiente, la constante está en el lado equivocado. Finalmente, pero no menos importante, z está totalmente ausente.

Corrigiendo estos defectos, tenemos

$$-w + 5x - y + 0z = -1.$$

Con estas ecuaciones (modificadas) en mente, tenemos la matriz

$$B = \left[\begin{array}{cccc|c} -1 & 2 & 1 & -5 & 6 \\ 0 & 0 & -1 & 1 & -5 \\ 1 & 3 & 3 & -1 & 0 \\ -1 & 5 & -1 & 0 & -1 \end{array} \right].$$

Sería muy tedioso calcular a mano la RREF correspondiente —y sería aun peor tener que resolver el sistema de ecuaciones manualmente sin matrices, usando álgebra ordinaria—. Sin embargo, usando Sage calcularemos fácilmente la RREF en la siguiente subsección. Por ahora, solamente diremos que la solución es

$$w = -107/7, \quad x = -12/7, \quad y = 54/7, \quad z = 19/7,$$

que podemos verificar fácilmente ahora.

1.5.3 Obteniendo la RREF en Sage

Esta subsección nos mostrará cómo calcular la forma escalonada reducida por filas de una matriz, presumiblemente con el objeto de resolver un sistema de ecuaciones lineales. Hacerlo en Sage requiere solamente cuatro pasos, el último de los cuales es opcional.

Primero definiremos la matriz

$$A = \left[\begin{array}{ccc|c} 3 & -4 & 5 & 14 \\ 1 & 1 & -8 & -5 \\ 2 & 1 & 1 & 7 \end{array} \right],$$

la cual proviene de la penúltima subsección. Haremos esto con la instrucción

Código de Sage

```
1 A = matrix(3, 4, [3,-4,5,14,1,1,-8,-5,2,1,1,7])
```

La clave para entender este comando de Sage es notar que el primer número es la cantidad de filas y el segundo es la cantidad de columnas.⁵ A continuación viene la lista de entradas de la matriz (los números que la componen), separadas por comas y delimitadas por corchetes.

⁵Esta es una forma de recordarlo: las columnas de una matriz, como las columnas de un edificio, son verticales; las filas, por proceso de eliminación, son horizontales. Pero también podemos pensar que estas últimas son como las filas de personas esperando a ser atendidas en una ventanilla: horizontales. De la misma manera, construir una matriz es como construir un edificio: primero deben aparecer las *filas* de trabajadores para que luego puedan erigir las *columnas*.

A propósito de esto, es importante mantener la definición de la matriz A en el Servidor Sage Cell mientras estemos trabajando con ella; no podemos prescindir de ella hasta entonces. Cada vez que presionamos “Evaluate”, mandamos lo que hayamos introducido en la celda al servidor para su evaluación. Este no recordará la definición de la matriz A si la removemos. En contraste, CoCalc (anteriormente SageMathCloud) tiene una memoria más permanente.

Nótese que Sage no responde cuando recibe este comando. Esto se debe a que solo estamos indicando la definición de la matriz A , pero no estamos dando ninguna instrucción para operar sobre ella.

El segundo paso es verificar que la matriz que hemos introducido es la que deseábamos introducir. Este paso no es opcional, pues los errores al teclear son extremadamente comunes. Para hacer esto, solo debemos escribir `print(A)` en una línea propia, bajo la definición de A , y presionar “Evaluate”. La matriz es mostrada y así podemos verificar que hemos tecleado correctamente.

El tercer paso consiste en calcular la RREF de la matriz. Esto lo hacemos con el comando

Código de Sage

```
1 print(A.rref())
```

a lo que Sage responderá con la forma escalonada reducida por filas, o

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{array} \right].$$

El cuarto paso es verificar el trabajo hecho. Esto es opcional, pero altamente recomendable. Primero escribimos (en una línea propia)

Código de Sage

```
1 print(3*3 + -4*0 + 5*1)
```

y veremos que la respuesta es 14, tal como esperábamos. A continuación escribimos

Código de Sage

```
1 print(1*3 + 1*0 - 8*1)
```

y vemos que la respuesta es -5 , como se esperaba. La última ecuación se verifica de manera similar. ¡No olvidemos que es crucial verificar las tres ecuaciones! Si no lo hacemos, fallaríamos en detectar una “solución impostora” como

$$x = 30, \quad y = 29, \quad z = 8,$$

que satisface las dos primeras ecuaciones, pero no la tercera.

Ahora consideremos el ejemplo “complicado” de la subsección anterior. La matriz correspondiente era

$$B = \left[\begin{array}{cccc|c} -1 & 2 & 1 & -5 & 6 \\ 0 & 0 & -1 & 1 & -5 \\ 1 & 3 & 3 & -1 & 0 \\ -1 & 5 & -1 & 0 & -1 \end{array} \right],$$

que podemos introducir en Sage con

Código de Sage

```
1 B = matrix(4, 5, [-1,2,1,-5,6,0,0,-1,1,-5,1,3,3,-1,0,-1,5,-1,0,-1])
```

Con la matriz B siendo construida a partir de una lista de números tan larga, debemos ser muy cuidadosos de teclearlos correctamente, sin omitir o duplicar ninguno. Verificamos la correctitud de B escribiendo `print(B)` en una línea propia y viendo si hemos introducido exactamente lo que deseábamos introducir.

Antes de continuar, aprendamos algo de notación. Algunas veces deseamos referirnos a una posición dentro de la matriz. Por ejemplo, el 6 en la primera fila, quinta columna de B , puede ser escrito B_{15} . Similarmente, el 5 en la cuarta fila, segunda columna de B , puede ser escrito como B_{42} . Así también, los dos -5 s están localizados en B_{14} y B_{25} . Recordemos que en esta notación primero se escribe el número de la fila y luego el de la columna.

En Sage, la numeración de las filas y las columnas de una matriz empieza en 0 y no en 1. Esto significa que la esquina superior izquierda es $B[0,0]$ en Sage, pero B_{11} en matemáticas. Similarmente, B_{14} es escrito $B[0,3]$ y B_{23} es escrito $B[1,2]$. Esta anomalía es algo que Sage heredó del lenguaje de programación Python y no puede ser cambiado fácilmente, pues el primero se ejecuta sobre el segundo.

A propósito del tema, una forma alternativa de definir una matriz muy larga es como sigue:

Código de Sage

```
1 B = matrix([[ -1,2,1,-5,6], [0,0,-1,1,-5], [1,3,3,-1,0], [-1,5,-1,0,-1]])
```

Este código es una lista de listas. Cada lista de cinco números entre corchetes y separados por comas representa una fila; así que hay cuatro de ellas, encerradas en corchetes y separadas por comas, que representan la matriz entera. Dado que este formato delata el número de filas y columnas, no necesitamos colocar “4, 5” para informar a Sage el tamaño de B . Siempre podemos usar cualquiera de los dos formatos con el que nos sintamos más cómodos.

Ahora que B ha sido ingresada (por cualquiera de los métodos anteriores), debemos añadir el comando `print(B.rref())` en una línea propia para ver que la RREF es

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -107/7 \\ 0 & 1 & 0 & 0 & -12/7 \\ 0 & 0 & 1 & 0 & 54/7 \\ 0 & 0 & 0 & 1 & 19/7 \end{array} \right],$$

lo que nos da la respuesta final de

$$w = -107/7, \quad x = -12/7, \quad y = 54/7, \quad z = 19/7$$

para el sistema original de ecuaciones.

Finalmente, debemos revisar nuestro trabajo. Por ejemplo, verificamos la primera ecuación con estas líneas

Código de Sage

```
1 print(2*(-12/7) - 5*(19/7) + 54/7)
2 print(6 + -107/7)
```

las cuales nos dan ambas $-65/7$. Como siempre, el punto aquí no es que el resultado es $-65/7$, sino que ambas dan el mismo valor. Así, la primera ecuación se satisface. Las otras tres ecuaciones se verifican de manera similar. Pero no olvidemos que realmente debemos reemplazar cada uno de los cuatro valores en cada una de las cuatro ecuaciones originales. Por ejemplo, la “solución impostora”

$$w = -139/7, \quad x = -8/7, \quad y = 64/7, \quad z = 29/7$$

satisface las tres primeras ecuaciones, pero no la cuarta.

Si el lector ha leído la subsección 1.5.2, recordará que se requirió bastante esfuerzo para obtener esta matriz B . Esos muchos pasos podrían haber contenido un error si hubiésemos sido descuidados. Así que tenemos otra razón más para verificar nuestro trabajo.

1.5.4 Resumen básico

Hemos aprendido cómo usar los comandos `matrix` y `rref` para determinar la RREF de una matriz y, por lo tanto, para resolver sistemas de ecuaciones lineales. Realizando todos estos pasos a la vez, el autor suele escribir:

Código de Sage

```
1 B = matrix(4, 5, [-1,2,1,-5,6,0,0,-1,1,-5,1,3,3,-1,0,-1,5,-1,0,-1])
2 print('Problema:')
3 print(B)
4 print('Respuesta:')
5 print(B.rref())
```

Las líneas de código “`print('Pregunta:')`” y “`print('Respuesta:')`” son completamente opcionales, pero ayudan a hacer la salida más legible. También incentivan a revisar que se ha introducido la matriz que se quería introducir, sin ningún error. De alguna forma, el autor siempre comete un error de tecleo, que debe ser corregido. La salida producida es la siguiente:

```
Problema:
[-1  2  1 -5  6]
[ 0  0 -1  1 -5]
[ 1  3  3 -1  0]
[-1  5 -1  0 -1]
Respuesta:
[  1  0  0  0  0 -107/7]
[  0  1  0  0  0 -12/7]
[  0  0  1  0  0  54/7]
[  0  0  0  0  1  19/7]
```

1.5.5 La matriz identidad

Observemos las RREFs que hemos calculado hasta ahora. ¿Puede el lector ver un patrón? o, mejor dicho, ¿un patrón en todas las columnas excepto la última?

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{array} \right] \quad y \quad \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -107/7 \\ 0 & 1 & 0 & 0 & -12/7 \\ 0 & 0 & 1 & 0 & 54/7 \\ 0 & 0 & 0 & 1 & 19/7 \end{array} \right].$$

En efecto, podemos apreciar que todas las columnas (exceptuando la última) tienen ceros por todas partes, excepto por una muy notable diagonal de unos, es decir, en las posiciones de $A_{11}, A_{22}, A_{33}, \dots$. Este es un patrón que ocurre muy frecuentemente, por lo que no es sorpresa que tenga un nombre: los matemáticos llaman a esto la “matriz identidad”.

Consideremos una situación en la que una persona requiere mostrar su “ID” (su identificación o identidad)—por ejemplo, cuando es detenido para recibir una multa, o para entrar a un bar, o para hacer un trámite—. La ID rápidamente muestra los datos más importantes de esa persona, como su nombre, fecha de nacimiento, género, dirección, etc. De manera similar, y desde un punto de vista intuitivo, la matriz identidad muestra los datos más importantes de un sistema de ecuaciones lineales. Cuando esta matriz aparece, se pueden leer los valores de las incógnitas en la última columna de la RREF.

1.5.6 Un reto de práctica para el lector

Usando Sage, resuelva el siguiente sistema de ecuaciones lineales. Verifique sus respuestas con una calculadora.

$$3481x + 59y + z = 0,87$$

$$6241x + 79y + z = 0,61$$

$$9801x + 99y + z = 0,42$$

Nota: Este problema puede parecer artificial, pero en realidad es un problema aplicado que visitaremos nuevamente muy pronto. Como muchos problemas basados en aplicaciones, las soluciones tienen decimales, lo que esperamos que no desanime a nadie. Veremos la respuesta en poco.

1.5.7 Matrices de Vandermonde

En Economía y en Ciencias Empresariales en general, se suele hablar frecuentemente de la curva de precio-demanda. Después de todo, conforme el precio de un producto se eleva, la demanda debería bajar, mientras que si el precio disminuye, la demanda debería aumentar, al menos bajo condiciones estándares.

Dicha curva en general solo puede ser calculada encuestando a compradores potenciales. La técnica estándar es seleccionar, digamos, 2000 personas, mostrarles el producto y preguntarles si lo comprarían o no. Se les obsequia el producto como pago por haber participado en la encuesta.⁶ Tal vez 20 diferentes precios serán propuestos, siendo cada precio usado con 100 personas.

Este proceso producirá 20 puntos de datos, que son suficientes. Sin embargo, a veces es mejor contar con una función real: típicamente una función cuadrática es buscada, y hay buenas razones para ello.⁷

Imaginemos que un amigo nuestro con una compañía recién inaugurada ha hecho una encuesta, pero con 3 precios distintos y 300 personas, para un mismo elegante producto nuevo.

- A un precio de \$ 59, la encuesta indica que un 87 % lo comprarían.
- A un precio de \$ 79, la encuesta indica que un 61 % lo comprarían.
- A un precio de \$ 99, la encuesta indica que un 42 % lo comprarían.

Toda función cuadrática se puede escribir bajo la forma

$$f(x) = ax^2 + bx + c,$$

pero la cuestión es cómo podemos determinar los coeficientes a , b y c para este caso particular. Con ese objetivo en mente, y asumiendo que $f(x)$ modela bien la relación precio-demanda, entonces realmente debe ocurrir que $f(59) = 0,87$, $f(79) = 0,61$ y $f(99) = 0,42$. Así que podemos escribir las siguientes ecuaciones:

$$a(59^2) + b(59) + c = 0,87$$

$$a(79^2) + b(79) + c = 0,61$$

$$a(99^2) + b(99) + c = 0,42$$

Curiosamente, estas ecuaciones son lineales, lo que no necesariamente esperábamos dado que estamos trabajando con una función cuadrática. Todo lo que debemos hacer es escribir la matriz

$$\begin{bmatrix} 59^2 & 59 & 1 & | & 0,87 \\ 79^2 & 79 & 1 & | & 0,61 \\ 99^2 & 99 & 1 & | & 0,42 \end{bmatrix}$$

y pedir a Sage que calcule la RREF. ¡Sin embargo, es probable que el lector ya lo haya hecho! Se pedía calcular la solución

$$a = 0,000\,087\,5, \quad b = -0,025\,075\,0, \quad c = 2,044\,837\,5$$

como práctica en la subsección 1.5.6 en la página 25.

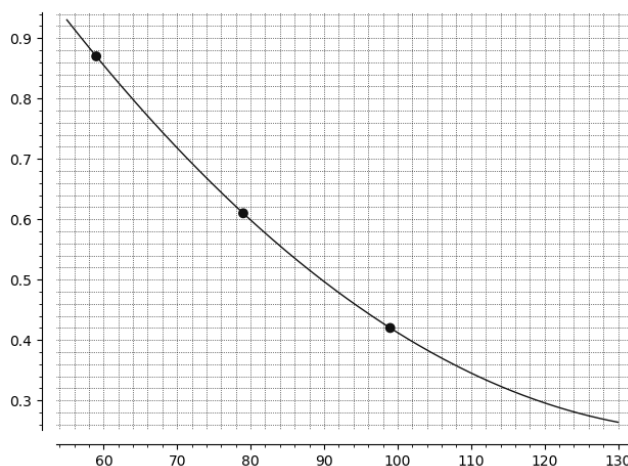
⁶No sorprende que esta técnica sea más comúnmente usada para equipos electrónicos pequeños o comida, y menos común entre fabricantes de autos deportivos de lujo.

⁷Digamos solamente que ajustar un polinomio de grado superior a puntos de datos es enteramente posible, pero extremadamente imprudente, debido a un fenómeno llamado “sobreajuste” (“overfitting”).

Finalmente, concluimos que la función buscada es

$$f(p) = 0,000\,087\,5p^2 - 0,025\,075\,0p + 2,044\,837\,5.$$

El gráfico correspondiente se observa abajo:



Solo para calmar la curiosidad, el código que produjo esta imagen fue

Código de Sage

```
1 f(x) = 0.000_087_5*x^2 - 0.025_075*x + 2.044_837_5
2
3 plot(f, 55, 130, gridlines='minor') + point([59, 0.87], size=50) +
  ↪ point([79, 0.61], size=50) + point([99, 0.42], size=50)
```

La técnica general para ajustar un polinomio de grado n a $n + 1$ puntos usando una matriz se debe a Alexandre-Théophile Vandermonde (1735–1796), y por lo tanto las matrices asociadas son llamadas *matrices de Vandermonde* en su honor. Por ejemplo, para ajustar un polinomio de grado cuatro a cinco puntos dados $(x_1, y_1), (x_2, y_2), \dots, (x_5, y_5)$, usaríamos la matriz

$$\begin{bmatrix} (x_1)^4 & (x_1)^3 & (x_1)^2 & (x_1) & 1 & y_1 \\ (x_2)^4 & (x_2)^3 & (x_2)^2 & (x_2) & 1 & y_2 \\ (x_3)^4 & (x_3)^3 & (x_3)^2 & (x_3) & 1 & y_3 \\ (x_4)^4 & (x_4)^3 & (x_4)^2 & (x_4) & 1 & y_4 \\ (x_5)^4 & (x_5)^3 & (x_5)^2 & (x_5) & 1 & y_5 \end{bmatrix}.$$

1.5.8 Los casos semirraros

Previamente, cada uno de nuestros ejemplos nos dio una solución única. Ahora examinaremos dos casos semirraros. Estos ocurren si existe una fila de ceros al fondo de la RREF, terminando a la derecha con un cero o con un número no nulo.

El sistema de ecuaciones lineales

$$x + 2y + 3z = 7$$

$$4x + 5y + 6z = 16$$

$$7x + 8y + 9z = 24$$

resulta en la matriz

$$C1 = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 4 & 5 & 6 & 16 \\ 7 & 8 & 9 & 24 \end{array} \right],$$

la cual ingresamos a Sage con

Código de Sage

```
1 C1 = matrix(3, 4, [1,2,3,7,4,5,6,16,7,8,9,24])
```

A continuación, para encontrar su RREF escribimos

Código de Sage

```
1 C1.rref()
```

y recibimos la respuesta

```
[ 1  0 -1  0]
[ 0  1  2  0]
[ 0  0  0  1]
```

Esto a su vez se traduce en

$$\begin{aligned}x - z &= 0 \\ y + 2z &= 0 \\ 0 &= 1\end{aligned}$$

La última ecuación del sistema anterior dice explícitamente que $0 = 1$, ¡lo que claramente no es verdad!. No existe forma de satisfacer ese requerimiento. Como consecuencia, esta ecuación no tiene soluciones. Así, en lugar que la “respuesta” sea una solución o incluso una infinidad de ellas, la respuesta es la oración “este sistema de ecuaciones no tiene soluciones”. Ese será así cuandoquiera que la RREF tenga una fila de ceros excepto en la última columna, donde se encuentra un número no nulo: el sistema no tiene soluciones.

Otro caso interesante se da cuando cambiamos el 24 por un 25 en la última ecuación. Ello resulta en la matriz

$$C2 = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 4 & 5 & 6 & 16 \\ 7 & 8 & 9 & 25 \end{array} \right],$$

que la introducimos a Sage con

Código de Sage

```
1 C2 = matrix(3, 4, [1, 2, 3, 7, 4, 5, 6, 16, 7, 8, 9, 25])
```

Ahora, para encontrar la “solución”, escribimos

Código de Sage

```
1 C2.rref()
```

y recibimos

```
[ 1  0 -1 -1]
[ 0  1  2  4]
[ 0  0  0  0]
```

Como podemos ver, esta RREF tiene una fila de ceros terminando en cero. Eso siempre indica una infinidad de soluciones, un punto que aclararemos pronto. Algunos profesores permiten la respuesta “el sistema tiene infinitas soluciones”, y en ciertas aplicaciones es irrelevante proveer de una forma de describir esas soluciones. Sin embargo, en ocasiones desearemos conocer cuáles son estas. Esto es especialmente cierto en problemas relacionados con la geometría. Aunque el espacio de soluciones es infinito, resulta que tiene lo que los matemáticos llaman *grados de libertad*. Esto significa que hay *variables libres* que pueden ser fijadas en cualquier valor que

deseemos. Pero una vez hecho esto, las demás variables quedan determinadas por esta elección. Este proceso es explicado en el apéndice D, junto con muchos otros hechos acerca de los sistemas lineales con infinitas soluciones.

¿Cuándo existen infinitas soluciones? Así que, ¿qué es lo que una fila de ceros realmente nos dice? La mayor parte del tiempo nuestros sistemas de ecuaciones tendrán dos ecuaciones con dos incógnitas, tres ecuaciones con tres incógnitas, cuatro ecuaciones con cuatro incógnitas, y así sucesivamente. El término técnico para esto es “sistema lineal cuadrado”. Un sistema lineal cuadrado resulta en una matriz de 2×3 , 3×4 , 4×5 , 5×6 o $n \times (n + 1)$. La regla que estamos por presentar funcionará siempre que tengamos uno de estos sistemas.

Esta es la regla de oro: si vemos la señal indicativa de “no hay soluciones” (el primer caso semirraro), entonces ignoramos toda la demás información y declaramos que no las hay. Por otro lado, si hay una fila de ceros, entonces tendremos infinitas soluciones.

Sin embargo, ¡tengamos cuidado! Esta regla no aplica si hay “muy pocas” o “demasiadas” ecuaciones, es decir, si tenemos un sistema rectangular. Por ejemplo, en la apéndice D.3 en la página 293, veremos algunos contraejemplos que demuestran que la regla de oro es falsa si el sistema es rectangular. Los sistemas lineales con pocas ecuaciones aparecen en algunos problemas interesantes de geometría analítica, y los sistemas lineales con demasiadas ecuaciones aparecen en criptoanálisis —y fueron usados por el autor en su tesis de doctorado—

Finalmente, debemos indicar que en el apéndice D aprenderemos una técnica, relativamente sencilla, que muestra exactamente cómo manejar sistemas de ecuaciones lineales con una infinidad de soluciones, independientemente del número de filas y columnas. El truco ahí es ligeramente más complicado, pero siempre funciona. Más aun, este nos da unas fórmulas para generar, de entre las infinitas que existen, tantas soluciones particulares como deseemos. Esas fórmulas también sirven para otros propósitos, como verificar si un candidato a solución efectivamente lo es.

No imaginemos que los sistemas lineales con infinitas soluciones son extremadamente raros; aparecen ocasionalmente en las aplicaciones. Sin embargo, son lo suficientemente poco comunes como para no tener que cargar con esos detalles aún y poder esperar hasta el apéndice D.

1.6 Definiendo nuestras propias funciones en Sage

Es extremadamente fácil definir nuestras propias funciones⁸ en Sage. Es una de las características más brillantes del diseño de Sage implementada por sus desarrolladores. Podemos definir nuestras propias funciones usando exactamente la misma simbología que usaríamos normalmente en papel o en un pizarrón.

Definiendo una función Por ejemplo, si deseamos trabajar con $f(x) = x^3 - x$, entonces escribimos

Código de Sage

```
1 f(x) = x^3 - x
2 f(2)
```

donde hemos definido $f(x)$ en la primera línea, y hemos solicitado el valor de $f(2)$ en la segunda. Por supuesto, podríamos preguntar por $f(3)$ o cualquier otro. Similarmente, si queremos trabajar con $g(x) = \sqrt{1 - x^2}$, escribimos

⁸Para aclarar, nos referimos a funciones en el sentido matemático, tal como $f(x) = x^3$. Sin embargo, los programadores experimentados deben conocer sobre definir funciones en un lenguaje de programación —a veces llamadas *procedimientos*, *métodos* o *subrutinas*—. Discutiremos sobre ello en la sección 5.2 en la página 216.

Código de Sage

```
1 g(x) = sqrt(1 - x^2)
2 g(1/4)
```

donde hemos definido $g(x)$ y preguntado por el valor al que $g(1/4)$ evalúa. Igual de fácil, podríamos pedir el valor de $g(1/2)$ o cualquier otro número en el dominio de g . Para evaluar varios valores a la vez, podemos usar el comando `print`:

Código de Sage

```
1 g(x) = sqrt(1 - x^2)
2 print(g(0.1))
3 print(g(0.01))
4 print(g(0.001))
5 print(g(0.000_1))
6 print(g(0.000_01))
7 print(g(0.000_001))
```

Si el lector ya empezó o cursó la asignatura de cálculo, puede ver que la salida de los anteriores comandos ayuda a evaluar

$$\lim_{x \rightarrow 0^+} g(x) = 1$$

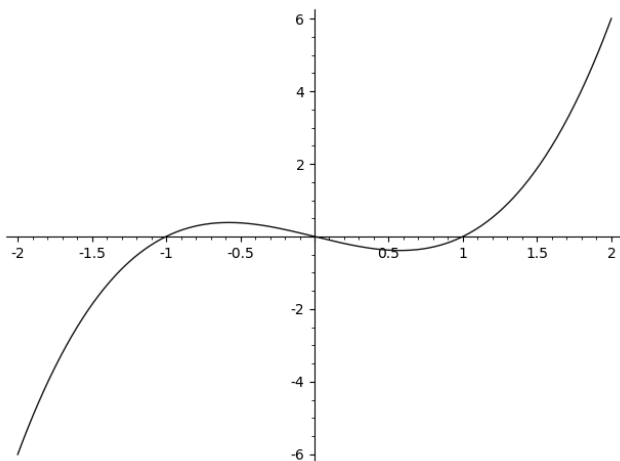
numéricamente, o también ayuda a confirmar la respuesta que se obtiene algebraicamente. Si aún no ha estudiado cálculo, puede ignorar esta observación.

Graficando funciones Ahora que sabemos cómo definir $f(x)$ y $g(x)$, podemos graficarlas:

Código de Sage

```
1 f(x) = x^3 - x
2 plot(f, -2, 2)
```

Esto da el resultado esperado:



De manera similar, si escribimos

Código de Sage

```
1 g(x) = sqrt(1 - x^2)
2 plot(g, 0, 1)
```

que es equivalente a

Código de Sage

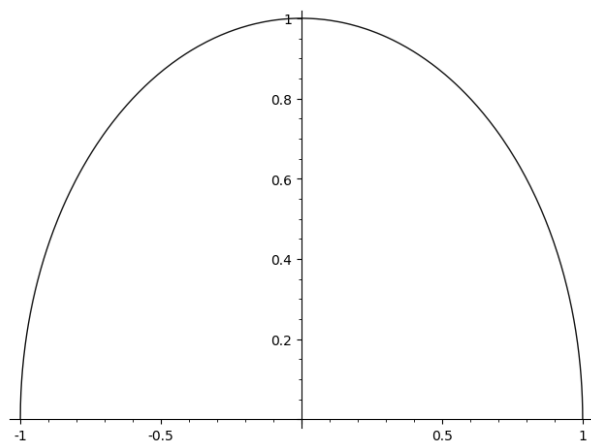
```
1 g(x) = sqrt(1 - x^2)
2 plot(g(x), 0, 1)
```

tenemos la gráfica de $g(x)$ en el dominio $0 \leq x \leq 1$. Finalmente, como habíamos visto antes, si escribimos

Código de Sage

```
1 g(x) = sqrt(1 - x^2)
2 plot(g)
```

que es una forma abreviada —omitiendo el intervalo del eje x —, entonces Sage asumirá por defecto que deseamos el intervalo $-1 \leq x \leq 1$. Esto produce la gráfica



Recordemos que debemos mantener la definición de $f(x)$ escrita mientras trabajemos con ella, y lo mismo aplica para $g(x)$. Esta es una característica del servidor de Sage Cell: tiene lo que podríamos llamar “amnesia inmediata” y trata cada ocasión que uno presiona el botón “Evaluate” como un universo matemático diferente, olvidando todo lo de la anterior ocasión. Esto se ha hecho así para beneficio del principiante, pues hace muy fácil la corrección de errores de tecleo. Simplemente se corrige el error y se vuelve a evaluar. CoCalc (anteriormente SageMathCloud) trabaja de manera ligeramente diferente, como explicaremos en la página 281.

Usando variables intermedias En ocasiones necesitaremos usar muchas veces el mismo valor, así que desearemos almacenarlo de alguna forma. Por ejemplo, si escribimos 355/113 muy frecuentemente, entonces mejor podemos escribir

Código de Sage

```
1 k = 355 / 113
```

que silenciosamente (sin imprimir nada en pantalla) almacena este valor en la variable k . Luego esta puede ser usada en cualquier línea sucesiva en Sage Cell, tal como

Código de Sage

```
1 k = 355 / 113
2 print(2 + k)
3 print(1 + k)
```

lo que nos muestra los valores correctos de $581/113$ y $468/113$. Si volvemos atrás y cambiamos el valor de k , sin embargo, debemos asegurarnos de pulsar el botón “Evaluate” para reevaluar las fórmulas.

Si reemplazamos $2+k$ por $2.0+k$, obtendremos una aproximación decimal. Si una expresión matemática está compuesta enteramente de enteros y números racionales, Sage proveerá una respuesta exacta; si un decimal es introducido, proveerá una aproximación decimal en su lugar.

La técnica de las variables intermedias es muy útil, por ejemplo, en física, para valores como la aceleración debida a la gravedad o la velocidad de la luz, que no cambian durante un problema.

Resulta que este k particular es una aproximación de π , así como la más conocida $22/7$, solo que mucho mejor. Si quisiésemos encontrar el error relativo de k como una aproximación de dicha constante, recordaríamos la fórmula

$$\text{error relativo} = \frac{\text{aproximación} - \text{verdadero}}{\text{verdadero}}$$

y escribiríamos

Código de Sage

```
1 k = 355 / 113
2 N((k - pi) / pi)
```

con lo que veríamos que el error relativo es alrededor de 84,9 partes por cada mil millones. Nada mal. Esta aproximación fue encontrada por el astrónomo chino Zu Chongzhi, también conocido como Tsu Ch'ung-Chih. Podemos comparar $355/113$ con $22/7$ mediante

Código de Sage

```
1 N((22/7 - pi) / pi)
```

que nos muestra que esta mucho más común aproximación de π tiene un error relativo de aproximadamente 402,5 partes por cada millón. ¡Eso es mucho peor! De hecho, $22/7$ es 4740,9 veces peor, así que es una verdadera lástima que no se suela enseñar $355/113$ en lugar de $22/7$.

Composición de funciones También es posible componer funciones fácilmente en Sage. Consideremos las funciones

$$f(x) = 3x + 5 \quad \text{y} \quad g(x) = x^3 + 1.$$

Tal vez deseamos trabajar con $f(g(x))$. La mejor manera de hacer esto es definir una tercera función, $h(x) = f(g(x))$. Entonces podríamos querer imprimirla en pantalla, así como sus valores en $x = 1$, $x = 2$ y $x = 3$. El código de Sage para realizar todo esto es bastante directo:

Código de Sage

```
1 f(x) = 3*x + 5
2 g(x) = x^3 + 1
3 h(x) = f(g(x))
4
5 print(h(x))
6 print(h(1))
7 print(h(2))
8 print(h(3))
```

Eso produce la salida

```
3*x^3 + 8
11
32
89
```

como se quería. Sin embargo, si reemplazamos la tercera línea con $h(x) = g(f(x))$, entonces obtenemos una salida completamente diferente:


```
(3*x + 5)^3 + 1
513
1332
2745
```

En efecto, no hay razón para esperar que $f(g(x)) = g(f(x))$ en general. La forma de verificar este caso particular es desarrollar ambas composiciones algebraicamente. Calculamos

$$\begin{aligned} f(g(x)) &= 3(g(x)) + 5 \\ &= 3(x^3 + 1) + 5 \\ &= 3x^3 + 3 + 5 \\ &= 3x^3 + 8, \end{aligned}$$

que es completamente diferente a

$$\begin{aligned} g(f(x)) &= g(3x + 5) \\ &= (3x + 5)^3 + 1 \\ &= 27x^3 + 135x^2 + 225x + 125 + 1 \\ &= 27x^3 + 135x^2 + 225x + 126. \end{aligned}$$

Funciones multivariadas En esta sección hemos tratado con funciones de una sola variable. Sage puede, igual de fácil y simple, manejar funciones de varias variables, como por ejemplo

$$f(x, y) = (x^4 - 5x^2 + 4 + y^2)^2.$$

Cubriremos este tema en detalle en la sección 4.1 en la página 123.

1.7 Usando Sage para manipular polinomios

Ahora exploraremos algunas aplicaciones de la notación funcional de la sección anterior a los polinomios. Primero, veamos que Sage puede realizar álgebra de polinomios muy fácilmente. Trabajaremos en este caso con

$$\begin{aligned} a(x) &= x^2 - 5x + 6, \\ b(x) &= x^2 - 8x + 15. \end{aligned}$$

Si escribimos el siguiente código:

```
Código de Sage
1 a(x) = x^2 - 5*x + 6
2 b(x) = x^2 - 8*x + 15
3
4 a(x) + b(x)
```

obtenemos la respuesta correcta, $2x^2 - 13x + 21$. Como siempre, es muy importante no olvidar el asterisco entre el 5 y la x , así como el que está entre 8 y x . Similarmente, si cambiamos el signo $+$ entre $a(x)$ y $b(x)$ por un signo $-$, obtenemos la resta de polinomios $3x - 9$. Como podemos imaginar, el producto puede ser calculado convenientemente con el símbolo $*$:

Código de Sage

```

1  a(x) = x^2 - 5*x + 6
2  b(x) = x^2 - 8*x + 15
3
4  a(x) * b(x)

```

Pero este último pedazo de código produce una respuesta sin desarrollar, aunque innegablemente correcta:

$$(x^2 - 5x + 6)(x^2 - 8x + 15)$$

En su lugar, podemos hacer lo siguiente:

Código de Sage

```

1  a(x) = x^2 - 5*x + 6
2  b(x) = x^2 - 8*x + 15
3
4  g(x) = a(x) * b(x)
5  g(x).expand()

```

y obtenemos la forma algebraicamente desarrollada de $g(x)$. En álgebra computacional, esta se llama la *forma expandida*. La forma expandida de nuestro polinomio $g(x)$ es

$$x^4 - 13x^3 + 61x^2 - 123x + 90$$

También podemos escribir la última línea de nuestro código anterior como `expand(g(x))`, que es una forma equivalente.

Si en lugar de `g(x).expand()` escribimos `g(x).factor()`, obtenemos la siguiente salida:

$$(x - 2)(x - 3)^2(x - 5)$$

que es la forma factorizada del polinomio $g(x)$ —una especie de forma “inversa” o “contraria” a la expandida—. También podemos factorizar polinomios de manera más directa con

Código de Sage

```

1  a(x) = x^2 - 5*x + 6
2  factor(a(x))

```

que nos da la respuesta correcta $(x - 2)(x - 3)$. Alternativamente, podemos ser incluso más directos escribiendo

Código de Sage

```

1  factor(x^2 - 8*x + 15)

```

para obtener $(x - 3)(x - 5)$.

Podemos calcular el *máximo común divisor*, o \gcd ⁹ de dos polinomios. Por ejemplo, el siguiente código:

Código de Sage

```

1  a(x) = x^2 - 5*x + 6
2  b(x) = x^2 - 8*x + 15
3
4  gcd(a(x), b(x))

```

nos provee de la respuesta $x - 3$. Por supuesto, conocíamos esta por las factorizaciones que encontramos antes: el factor común de $(x - 2)(x - 3)$ y $(x - 3)(x - 5)$ es claramente $x - 3$.

⁹**Nota del traductor:** Una vez más, usamos la abreviación basada en las siglas en inglés, pues Sage cuenta con un comando llamado `gcd`, lo que hará que sea más fácil de recordar.

Debemos hacer notar que estos ejemplos son muy triviales porque hemos usado polinomios cuadráticos. El siguiente polinomio es uno que nadie querría factorizar a mano —aunque es cierto que el *teorema de las raíces enteras*¹⁰ nos daría la respuesta solo con papel y lápiz, y una buena medida de paciencia.

Código de Sage

```
1 factor(x^4 - 60*x^3 + 1_330*x^2 - 12_900*x + 46_189)
```

Con referencia al teorema de las raíces enteras, podemos escribir `factor(46189)`, y obtener la factorización

$11 * 13 * 17 * 19$

para ayudarnos a encontrar las raíces de ese polinomio de grado cuatro. El comando `factor` opera factorizando tanto polinomios como enteros. El comando `divisors(46_189)`, como alternativa, enumera todos los enteros positivos que dividen 46 189. Naturalmente esta es una lista más larga que la salida de `factor`, que solo muestra los divisores primos (y sus exponentes, si los hay). El lector puede comparar las salidas de ambos comandos, si lo desea.

La composición de polinomios tampoco es un problema, como ya vimos. El código mostrado a continuación compondrá dos polinomios y producirá la respuesta en tres formas distintas. Es notable que este sea nuestro primer “programa” real en Sage, en el sentido que es la primera vez hemos usado más de dos o tres comandos a la vez.

Código de Sage

```
1 a(x) = x^2 - 5*x + 6
2 b(x) = x^2 - 8*x + 15
3
4 f(x) = a(b(x))
5
6 print('Forma directa:')
7 print(f(x))
8 print('Forma expandida:')
9 print(f(x).expand())
10 print('Forma factorizada:')
11 print(f(x).factor())
```

En particular, nuestro programa produce la siguiente salida:

```
Forma directa:
(x^2 - 8*x + 15)^2 - 5*x^2 + 40*x - 69
Forma expandida:
x^4 - 16*x^3 + 89*x^2 - 200*x + 156
Forma factorizada:
(x^2 - 8*x + 13)*(x - 2)*(x - 6)
```

El código anterior muestra cuán útil puede ser etiquetar partes de la salida con comandos `print`, en caso que dicha salida tenga más de uno o dos ítems. Tal etiquetado (a veces llamado “anotación de la salida”) es una buena práctica general mientras aprendemos a abordar tareas más y más complejas en Sage.

Tal como vimos en la sección 1.6, obtenemos un resultado distinto si cambiamos $f(x) = a(b(x))$ por $f(x) = b(a(x))$. ¡Adelante, vale la pena verificarlo!

¹⁰Si $p(x)$ es un polinomio con coeficientes enteros, entonces toda raíz entera debe ser un divisor del término constante. La mayoría de los números tienen una cantidad relativamente modesta de divisores, por lo que es relativamente fácil simplemente intentar con cada uno de ellos y de este modo tener una lista completa de las raíces enteras de $p(x)$. Nótese que este método incluso funciona con polinomios de grado cinco o superior, que de otra manera son difíciles de resolver. También existe un *teorema de las raíces racionales*, pero no lo trataremos aquí.

1.8 Usando Sage para resolver problemas simbólicamente

Cuando una computadora ha resuelto un problema por nosotros, el resultado puede estar expresado en una de dos formas: numérica o simbólica. Cuando resolvemos numéricamente, obtenemos una expansión decimal para una (muy buena) aproximación de la respuesta. Cuando resolvemos simbólicamente, obtenemos una respuesta exacta, frecuentemente en términos de radicales u otras funciones complicadas. Todo esto se reduce a decir que las soluciones de

$$\frac{x^2}{2} - x - 2 = 0$$

son $-\sqrt{5} + 1$ y $\sqrt{5} + 1$, o decir que las soluciones son

$$-1,236\,067\,977\,499\,79 \quad \text{y} \quad 3,236\,067\,977\,499\,79.$$

Las dos primeras son un ejemplo de solución simbólica, mientras que las segundas forman una numérica. Frecuentemente, una solución numérica es deseada. Sin embargo, si la respuesta es una fórmula, entonces una simbólica es la manera de proceder. En esta sección trataremos la forma simbólica, y la siguiente tratará de la forma numérica.

1.8.1 Resolviendo ecuaciones con una variable

Primero tratemos con ecuaciones con una sola variable. Podemos escribir

Código de Sage

```
1 solve(x^2 + 3*x + 2 == 0, x)
```

para indicar que queremos resolver $x^2 + 3x + 2 = 0$ con respecto a x , y entonces obtenemos

```
[x == -2, x == -1]
```

¡Es importante recordar el asterisco! Para ser precisos, escribir

Código de Sage

```
1 solve(x^2 + 3x + 2 == 0, x)
```

sería incorrecto debido a la ausencia del asterisco entre “3” y “x”.

Otro ejemplo, más ilustrativo de solución simbólica, es

Código de Sage

```
1 solve(x^2 + 9*x + 15 == 0, x)
```

que produce la salida

```
[x == -1/2*sqrt(21) - 9/2, x == 1/2*sqrt(21) - 9/2]
```

A propósito, notemos cómo las soluciones están separadas por comas y rodeadas de corchetes. Este es un ejemplo de una lista, una notación que Sage heredó del lenguaje de programación Python.

Una manera de realmente apreciar la diferencia entre simbólico y numérico es comparar

Código de Sage

```
1 3 + 1/(7 + 1/(15 + 1/(1 + 1/(292 + 1/(1 + 1/(1 + 1/6))))))
```

que devuelve $1354394/431117$, con

Código de Sage

```
1 N(3 + 1/(7 + 1/(15 + 1/(1 + 1/(292 + 1/(1 + 1/(1 + 1/6))))))
```

que devuelve 3.14159265350241 . Este valor puede ser fácilmente confundido con π . De hecho, es una aproximación realmente buena con un error relativo de $2,78 \cdots \times 10^{-11}$.

No es necesario restringirnos a polinomios o a la variable x . También podemos escribir

Código de Sage

```
1 var('theta')
2 solve(sin(theta) == 1/2, theta)
```

para obtener

```
[theta == 1/6*pi]
```

Pero notemos que de hecho hay una infinidad de soluciones para esta ecuación. Por ejemplo,

$$\theta \in \left\{ \dots, \frac{31\pi}{6}, \frac{23\pi}{6}, \frac{19\pi}{6}, \frac{11\pi}{6}, \frac{7\pi}{6}, \frac{\pi}{6}, \frac{5\pi}{6}, \frac{13\pi}{6}, \frac{17\pi}{6}, \frac{25\pi}{6}, \frac{29\pi}{6}, \dots \right\}$$

son varios valores de θ que satisfacen la ecuación $\sin(\theta) = 1/2$.

Declarando variables Puede ser confuso ver el comando `var('theta')` arriba. Su propósito es informar a Sage que el valor de esta variable aún no es conocido. Hemos usado variables antes, pero les habíamos asignado valores. Cuando deseamos que una variable represente una cantidad desconocida, debemos usar `var`. La excepción es la variable x , que siempre está predefinida y no necesitamos declararla —Sage asume que x es una incógnita, a menos que se le asigne previamente un valor—.

1.8.2 Resolviendo ecuaciones con varias variables

Ahora usemos Sage para redescubrir la fórmula de resolución de ecuaciones cuadráticas. Primero debemos declarar nuestras variables:

Código de Sage

```
1 var('a b c')
```

Entonces escribimos

Código de Sage

```
1 solve(a*x^2 + b*x + c == 0, x)
```

que nos devuelve

```
[x == -1/2*(b + sqrt(b^2 - 4*a*c))/a, x == -1/2*(b - sqrt(b^2 - 4*a*c))/a]
```

lo que contiene algunos términos en un orden poco usual, pero es indudablemente correcto.

Por supuesto, es probable que el lector conozca la fórmula cuadrática muy bien (¡al menos esperamos que así sea!). Sin embargo, probablemente muy pocos conozcan la Fórmula Cúbica de Cardano. En la subsección 1.8.5 en la página 41, veremos cómo Sage puede “redescubrir” la fórmula de Cardano al instante cuando se procede de manera similar.

Declarando varias variables a la vez Por cierto,

Código de Sage

```
1 var('a b c')
```

es meramente una abreviación para

Código de Sage

```
1 var('a')
2 var('b')
3 var('c')
```

También se pueden usar las siguientes formas equivalentes:

```
var("a, b, c") var("a b c") var('a b c') var('a, b, c')
```

El lector puede elegir la que le resulte más cómoda.

1.8.3 Sistemas de ecuaciones lineales

Es posible resolver varias ecuaciones simultáneamente, ya sean lineales o no. Un caso sencillo es resolver

$$x + b = 6$$

$$x - b = 4$$

lo que puede hacerse con los comandos

Código de Sage

```
1 var('b')
2 solve([x+b == 6, x-b == 4], x, b)
```

donde hemos introducido las ecuaciones como una lista.

En general, podemos delimitar cualquier tipo de datos con “[” y “]”, y separar las entradas con comas para crear una lista en Sage.

También es importante notar que no representa ningún problema “declarar” una variable (como b) muchas veces. Dos comandos `var` no interfieren uno con el otro. Sin embargo, tampoco es necesario hacerlo así, porque una vez hecha la declaración, Sage no olvidará la variable.

Un ejemplo más típico de sistema lineal es

$$9a + 3b + 1c = 32$$

$$4a + 2b + 1c = 15$$

$$1a + 1b + 1c = 6$$

Para resolverlo, debemos escribir

Código de Sage

```
1 var('a, b, c')
2 solve([9*a + 3*b + c == 32, 4*a + 2*b + c == 15, a + b + c == 6], a, b, c)
```

y Sage nos dará la respuesta

```
[[a == 4, b == -3, c == 5]]
```

la cual podemos verificar fácilmente como correcta. Si prestamos atención, notaremos que este es exactamente el mismo sistema lineal que usaríamos si se nos pidiera encontrar la parábola que conecta los puntos $(3, 32)$, $(2, 15)$ y $(1, 6)$, es decir, $f(x) = 4x^2 - 3x + 5$. Implícitamente, este es otro ejemplo de matriz de Vandermonde (excepto que no usamos una matriz aquí), como se definió en la subsección 1.5.7 en la página 25.

Los sistemas de ecuaciones lineales pueden resolverse con matrices. De hecho, Sage es realmente útil para trabajar con ellas, y puede evitarnos mucho del tedio normalmente asociado con el álgebra matricial. Eso lo discutimos en la sección 1.5 en la página 19.

1.8.4 Sistema de ecuaciones no lineales

Mientras que las ecuaciones lineales son muy fáciles de resolver con matrices, el caso no lineal es usualmente mucho más difícil. Primero entraremos en calor con una sola ecuación, aunque una altamente no lineal. Trate-mos de resolver

$$x^6 - 21x^5 + 175x^4 - 735x^3 + 1624x^2 - 1764x + 720 = 0.$$

Usamos el comando

Código de Sage

```
1 solve(x^6 - 21*x^5 + 175*x^4 - 735*x^3 + 1_624*x^2 - 1_764*x + 720 == 0, x)
```

que nos da

```
[x == 5, x == 6, x == 4, x == 2, x == 3, x == 1]
```

Recibimos una lista de seis respuestas porque este polinomio de grado seis tiene seis raíces.

En esta ocasión, la solución fue fácil de leer, pero, de no ser así, también podemos acceder a cada respuesta individualmente. Si introducimos

Código de Sage

```
1 respuesta = solve(x^6 - 21*x^5 + 175*x^4 - 735*x^3 + 1_624*x^2 - 1_764*x +
  ↪ 720 == 0, x)
```

entonces podemos escribir `print(respuesta[0])` o `print(respuesta[1])`, para obtener la primera o la segunda respuesta, respectivamente. Para obtener la quinta entrada de la lista escribimos `print(respuesta[4])`. De manera similar a la numeración de los elementos de una matriz, esta forma de acceder a los elementos de una lista se debe a que Sage está construido sobre Python, que enumera las entradas empezando con 0, no 1.

Consideremos ahora el siguiente problema sugerido por el Prof. Jason Grout, de la Universidad Drake. Para resolver

$$\begin{aligned} p + q &= 9 \\ qy + px &= -6 \\ qy^2 + px^2 &= 24 \\ p &= 1 \end{aligned}$$

escribimos

Código de Sage

```
1 var('p q y')
2 eq1 = p+q == 9
3 eq2 = q*y + p*x == -6
4 eq3 = q*y^2 + p*x^2 == 24
5 eq4 = p == 1
6 solve([eq1, eq2, eq3, eq4], p, q, x, y)
```

lo que produce

```
[[p == 1, q == 8, x == -4/3*sqrt(10) - 2/3, y == 1/6*sqrt(10) - 2/3], [p
== 1, q == 8, x == 4/3*sqrt(10) - 2/3, y == -1/6*sqrt(10) - 2/3]]
```

Esto no es más que una lista de listas, pero es un enredo difícil de leer. Usando la técnica que aprendimos cuando analizamos el polinomio de grado seis más arriba, reemplazamos la última línea con

Código de Sage

```
1 respuesta = solve([eq1, eq2, eq3, eq4], p, q, x, y)
```

y ahora podemos escribir

Código de Sage

```
1 print(respuesta[0])
2 print(respuesta[1])
```

para acceder a las soluciones individualmente:

```
[p == 1, q == 8, x == -4/3*sqrt(10) - 2/3, y == 1/6*sqrt(10) - 2/3]
[p == 1, q == 8, x == 4/3*sqrt(10) - 2/3, y == -1/6*sqrt(10) - 2/3]
```

Esta es la forma en que Sage nos dice:

- solución #1: $p = 1, q = 8, x = \left(\frac{-4}{3}\right) \sqrt{10} + \frac{-2}{3}, y = \frac{1}{6} \sqrt{10} + \frac{-2}{3};$
- solución #2: $p = 1, q = 8, x = \frac{4}{3} \sqrt{10} + \frac{-2}{3}, y = \left(\frac{-1}{6}\right) \sqrt{10} + \frac{-2}{3}.$

Dado que en este caso hay solo dos soluciones, no podemos solicitar una tercera. En efecto, si escribimos `print(respuesta[2])`, obtendremos la respuesta

```
IndexError: list index out of range
```

con lo que Sage trata de decirnos que ya nos ha dado todas las respuestas a este problema —la lista no contiene un tercer elemento—.

Tratemos ahora de hallar las intersecciones de la hipérbola $x^2 - y^2 = 1$ con la elipse $x^2/4 + y^2/3 = 1$. Escribimos

Código de Sage

```
1 var('y')
2 solve([x^2 - y^2 == 1, x^2/4 + y^2/3 == 1], x, y)
```

y obtenemos

```
[[x == -4/7*sqrt(7), y == -3/7*sqrt(7)], [x == -4/7*sqrt(7), y ==
3/7*sqrt(7)], [x == 4/7*sqrt(7), y == -3/7*sqrt(7)], [x == 4/7*sqrt(7), y
== 3/7*sqrt(7)]]
```

que nuevamente es prácticamente ilegible. Una vez más cambiamos el último comando por

Código de Sage

```
1 respuesta = solve([x^2 - y^2 == 1, x^2/4 + y^2/3 == 1], x, y)
```

y escribimos

Código de Sage

```
1 print(respuesta[0])
2 print(respuesta[1])
3 print(respuesta[2])
4 print(respuesta[3])
5 print(respuesta[4])
```

que nos da cuatro respuestas y un mensaje de error:

```
[x == -4/7*sqrt(7), y == -3/7*sqrt(7)]
[x == -4/7*sqrt(7), y == 3/7*sqrt(7)]
[x == 4/7*sqrt(7), y == -3/7*sqrt(7)]
[x == 4/7*sqrt(7), y == 3/7*sqrt(7)]
IndexError: list index out of range
```

Con esto, Sage nos dice que

- solución #1: $x = \left(\frac{-4}{7}\right) \sqrt{7}, y = \left(\frac{-3}{7}\right) \sqrt{7};$
- solución #2: $x = \left(\frac{-4}{7}\right) \sqrt{7}, y = \frac{3}{7} \sqrt{7};$

- solución #3: $x = \frac{4}{7}\sqrt{7}, y = \left(\frac{-3}{7}\right)\sqrt{7};$
- solución #4: $x = \frac{4}{7}\sqrt{7}, y = \frac{3}{7}\sqrt{7};$

pero que no existe una “solución #5”.

Dado que hallar las raíces de un polinomio (o encontrar las soluciones a una ecuación) es una tarea matemática común, existen otras formas de presentar las soluciones. Por ejemplo, existe código que nos mostrará cada raíz en su propia línea, y podemos elegir mostrar solamente las raíces reales, solo las racionales o solo las enteras. Describiremos esto a detalle en la subsección 5.7.10 en la página 257.

No necesitamos limitarnos a polinomios. Podemos escribir

Código de Sage

```
1 solve(log(x^2) == 5/3, x)
```

y recibimos

```
[x == -e^(5/6), x == e^(5/6)]
```

las cuales claramente satisfacen la ecuación dada. Recordemos que `log` representa el logaritmo natural, como explicamos en la página 6.

También podemos hacer

Código de Sage

```
1 solve(sin(x + y) == 0.5, x)
```

y obtenemos

```
[x == 1/6*pi - y]
```

Aunque claramente esta no es la respuesta que esperábamos, sí es totalmente correcta. Por ejemplo, si $x = 5\pi/6 - y$, entonces tenemos

$$\begin{aligned}\sin(x + y) &= \sin\left(\frac{5\pi}{6} - y + y\right) \\ &= \sin\left(\frac{5\pi}{6}\right) \\ &= 1/2.\end{aligned}$$

Números complejos Normalmente esperamos que un polinomio de grado ocho tenga ocho raíces en el plano complejo, a menos que algunas de ellas estén repetidas. Por lo tanto, si preguntamos “¿cuáles son las raíces de $x^8 + 1$?”, seguramente esperamos ocho de ellas. Estas son las ocho raíces octavas de -1 en el conjunto de los números complejos. Podemos encontrarlas con

Código de Sage

```
1 solve(x^8 == -1, x)
```

que produce

```
[x == (1/2*I + 1/2)*sqrt(2)*(-1)^(1/8), x == I*(-1)^(1/8), x == (1/2*I - 1/2)*sqrt(2)*(-1)^(1/8), x == -(-1)^(1/8), x == -(1/2*I + 1/2)*sqrt(2)*(-1)^(1/8), x == -I*(-1)^(1/8), x == -(1/2*I - 1/2)*sqrt(2)*(-1)^(1/8), x == (-1)^(1/8)]
```

dándonos ocho números complejos distintos, cada uno de los cuales tiene a -1 como su octava potencia. Estos pueden ser calculados (a mano) más lentamente con la Fórmula de DeMoivre, pero para Sage este es un problema fácil. Copiemos y peguemos la primera raíz anterior a la celda de Sage y elevémosla a la octava potencia, con la línea

Código de Sage

```
1 ((1/2*I + 1/2) * (-1)^(1/8) * sqrt(2))^8
```

y Sage nos devolverá la respuesta -1 .

1.8.5 Casos avanzados

Para entender por qué no se suele pedir a los estudiantes de pregrado que memoricen la Fórmula Cúbica de Cardano, intentemos escribir

Código de Sage

```
1 solve(x^3 + b*x + c == 0, x)
```

lo que nos da dicha fórmula para un polinomio cúbico mónico deprimido. Un polinomio cúbico es llamado *deprimido* si el coeficiente cuadrático es cero, y mónico significa que el coeficiente principal (aquí, el cúbico) es uno. La versión completa de la fórmula cúbica será dada por

Código de Sage

```
1 var('d')
2 solve(a*x^3 + b*x^2 + c*x + d == 0, x)
```

que es simplemente demasiado complicada. Inténtelo y lo verá. De hecho, es tan complicada que uno tendría que confesarse incapaz de usarla.

Una fórmula incluso más desagradable es la fórmula general para polinomios de grado cuatro. De seguro que cualquiera de tales polinomios puede escribirse como

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = 0,$$

así que podemos pedirle a Sage que lo resuelva con

Código de Sage

```
1 var('a0 a1 a2 a3 a4')
2 solve(a4*x^4 + a3*x^3 + a2*x^2 + a1*x + a0 == 0, x)
```

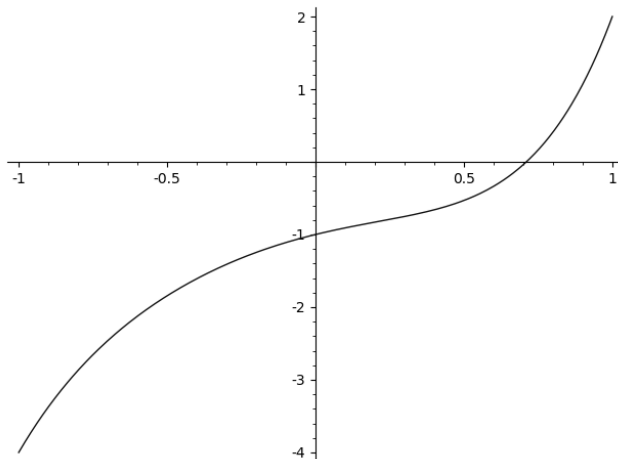
y obtendremos una fórmula de proporciones ridículas.

1.9 Usando Sage para resolver problemas numéricamente

Los valores x tales que $f(x) = 0$ son comúnmente conocidos como las “raíces” o “ceros” de $f(x)$. Supongamos que deseamos conocer el valor de alguna raíz de

$$f(x) = x^5 + x^4 + x^3 - x^2 + x - 1,$$

un polinomio cuya gráfica se muestra abajo.



Más aun, supongamos que estamos interesados en una particular entre -1 y 1 . Este podría ser el caso porque graficamos el polinomio (con o sin ayuda de Sage) y vimos que existe un cero en esa región; podría ser que notamos que $f(-1) = -4$ y $f(1) = 2$, así que $f(x)$ claramente cruza el eje x en algún punto entre -1 y 1 , aunque no sabemos exactamente por cuál; tal vez un problema en algún texto simplemente nos pide encontrar el cero entre esos puntos.

Para resolver este problema, solo necesitamos escribir

Código de Sage

```
1 find_root(x^5 + x^4 + x^3 - x^2 + x - 1, -1, 1)
```

y Sage nos dirá que

0,710 434 255 786 912 5

es la raíz buscada. Esta es una aproximación numérica, pues los polinomios de grado cinco tienen una propiedad muy especial —si el lector no la conoce, puede preguntar a su profesor de matemáticas—. Sin embargo, dado que la computadora ejecuta varios miles de millones de instrucciones por segundo, Sage invierte el tiempo computacional en refinar el resultado. Como consecuencia, es una aproximación que es confiable hasta una precisión de alrededor de 10^{-16} , que es la diez mil billonésima parte de la unidad. Esta es una buena aproximación de acuerdo a cualquier estándar.

Por otro lado, resulta que nuestro polinomio no tiene raíces entre -1 y 0 . Podría ser que eso se nos dijo, o tal vez lo graficamos (con o sin ayuda de Sage). Si escribimos

Código de Sage

```
1 find_root(x^5 + x^4 + x^3 - x^2 + x - 1, -1, 0)
```

Sage nos dirá

RuntimeError: f appears to have no zero on the interval

que se traduce como “Error en tiempo de ejecución: f parece no tener un cero en el intervalo”, ¡la cual es una respuesta perfectamente honesta para el caso!

Si no tuviésemos idea de dónde se ubica la raíz, podríamos escribir

Código de Sage

```
1 find_root(x^5 + x^4 + x^3 - x^2 + x - 1, -10^12, 10^12)
```

Es decir, pedimos a Sage buscar la raíz entre menos/más un billón.

Problemas altamente sofisticados pueden ser resueltos. Aquí tenemos un favorito: si alguien nos pregunta acerca de $x^x = 5$, entonces podríamos intentar determinar dónde $x^x - 5 = 0$, escribiendo

Código de Sage

```
1 find_root(x^x - 5, 1, 10)
```

o, equivalentemente,

Código de Sage

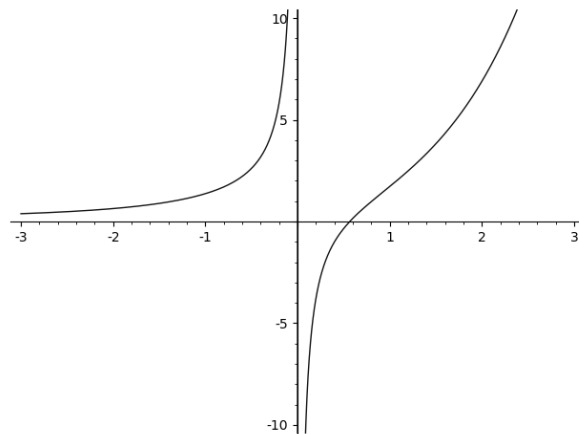
```
1 find_root(x^x == 5, 1, 10)
```

Otro problema muy interesante es determinar dónde $e^x = 1/x$, lo que hacemos encontrando un cero de $e^x - 1/x = 0$. Podríamos apoyarnos con

Código de Sage

```
1 plot(e^x - 1/x, -3, 3, ymin=-10, ymax=10)
```

obteniendo la gráfica



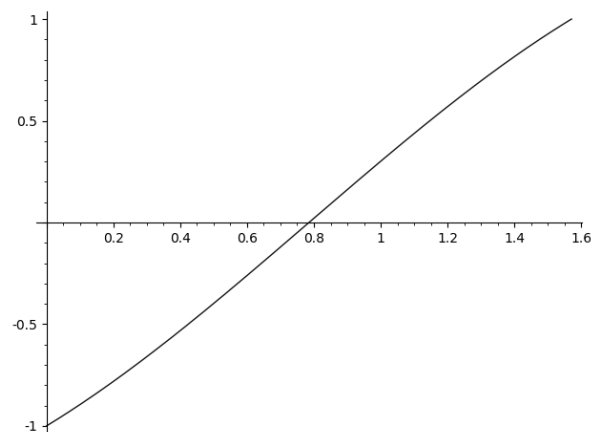
donde veríamos que la raíz se encuentra entre -1 y 1 . Esto significa que deberíamos escribir

Código de Sage

```
1 find_root(e^x - 1/x, -1, 1)
```

Y así encontraríamos que el cero buscado es $x = 0,567\,143\,290\,409\,795\,4$.

Por último, pero no menos importante, otro favorito es encontrar dónde $\sin(x) = \cos(x)$. Resulta ser el caso que hay una tal x entre 0 y $\pi/2$, que podemos confirmar graficando:



Ahora, podemos pedir la raíz de $\sin(x) - \cos(x)$ o, alternativamente, también podemos simplemente escribir

Código de Sage

```
1 find_root(cos(x) == sin(x), 0, pi/2)
```

la cual es solo una notación más sencilla.

Revisitando un viejo problema de interés compuesto Más arriba, en la página 7, vimos un problema sobre interés compuesto y cómo hallar cuántos meses se requieren para convertir \$ 5000 en \$ 7000. Ahora veamos cómo resolver ese mismo problema con Sage en una sola línea, en lugar del anterior enfoque que requería de un humano para trabajar con la parte algebraica.

La primera línea de nuestro análisis era

$$7000 = 5000(1 + 0,045/12)^n,$$

que traducida al lenguaje de Sage resulta en

Código de Sage

```
1 7_000 == 5_000 * (1 + 0.045 / 12)^n
```

Entonces debemos escribir

Código de Sage

```
1 var('n')
2 find_root(7_000 == 5_000 * (1 + 0.045/12)^n, 0, 10_000)
```

Nótese que la segunda línea indica que estamos buscando un valor de n que satisfaga la ecuación, y que dicho valor debe estar entre 0 y 10 000. Esto parece razonable, pues 10 000 meses es un poco más de 833 años —seguramente no tardará tanto—. Recordemos que si estamos acostumbrados a la notación inglesa para numerales, debemos tener cuidado de no escribir 10 ,000, pues Sage prohíbe dicho uso, como vimos en la página 2. El primer comando (`var`) es solo para declarar n como una variable, lo que estudiamos en la página 36. En cualquier caso, obtenemos la respuesta 89,89406093308006 como antes, y vemos que 90 meses serán requeridos.

1.10 Consiguiendo ayuda cuando la necesitemos

Hasta este punto hemos aprendido más que unos cuantos comandos, y es importante discutir cómo podemos usar las “características de ayuda” incorporadas en Sage. Existen varias, cada una diseñada con un propósito diferente.

Una de las más comúnmente usadas es llamada *autocompletado con TAB* (primero descrita en la página 5), la cual está diseñada para cuando no recordemos el nombre de algún comando. Por ejemplo, tal vez estemos dudando si el comando correcto es `find_root`, `findroot`, `find_roots`, `root_find`, o algo completamente distinto. Escribimos `find` y presionamos la tecla TAB en el teclado (sin presionar “Evaluate”). Entonces recibiremos una lista de opciones, la mayoría de las cuales no guardan relación con nuestras intenciones, pero una de ellas es el comando buscado, `find_root`. Alternativamente, si hubiésemos llegado hasta `find_r` antes de presionar TAB, entonces habría una única opción, en cuyo caso Sage simplemente completaría el comando por nosotros.

Ahora imaginemos que efectivamente recordamos el nombre de un comando —digamos `solve`—, pero no su sintaxis exacta. Entonces podemos escribir `solve?`, sin dejar espacios entre el nombre y el signo de interrogación, y presionamos “Evaluate”. En este caso se nos mostrará una gran cantidad de información útil en inglés sobre el comando. Podemos leer todo el texto completo, pero usualmente es más rápido simplemente saltar a la sección de ejemplos y ver cómo se hacen las cosas ahí, imitando lo que encontremos conforme necesitemos. Esta larga colección de información es llamada “docstring”, la abreviación en inglés para “texto de documentación”.

Existe otra forma de usar el operador `?`. Si ponemos dos de ellos después del nombre de un comando (nuevamente, sin espacios añadidos), como en `N??`, entonces veremos el código fuente completo (usualmente escrito en el lenguaje de programación Python) asociado a ese nombre. En este caso, veremos el código en Python para el comando `N()` que convierte expresiones algebraicas exactas en números de punto flotante. Por supuesto, el código es muy avanzado y requiere mucho más conocimiento de Python del que este libro pretende impartir —simplemente rasgaremos la superficie en el capítulo 5—.

A pesar de todo, es importante que esta característica (de exhibir el código fuente) esté presente, por razones filosóficas. Frecuentemente, debemos saber *cómo* un programa matemático realizará su tarea (es decir, cómo trabajará internamente) para que podamos decidir la mejor forma de dar formato y preprocesar nuestros datos, previamente al uso del programa. Esta necesidad surge frecuentemente en todo tipo de situaciones y contextos, y tener acceso al código fuente es esencial. También permite que las ideas inmersas en el código matemático sean usadas por otros programadores en sus propios programas. De esta manera, la comunidad *open-source* (de código abierto) vive como una especie de “mente colmena” compartiendo sus ideas ampliamente. Por otro lado, el software matemático inevitablemente contendrá errores o *bugs*. Un colega del autor encontró un *bug* en el software Mathematica y lo notificó a la compañía vía correo electrónico. El error fue reconocido, pero tres años después este permanecía aún sin arreglo. En Sage, uno simplemente puede arreglar el *bug* y enviar el cambio hecho para revisión. Si se determina que es válido y útil, este será incorporado en futuras versiones. En ocasiones el proceso puede tomar tan poco como seis semanas.

Esto nos trae a un punto importante: tenemos que conocer el nombre del comando —o al menos una parte— para poder usar cualquiera de estos tres métodos para conseguir ayuda. ¿Qué pasa si no podemos recordarlo? Esto ocurre frecuentemente, y existen varias soluciones.

La documentación oficial de Sage puede encontrarse en la dirección

`https://doc.sagemath.org`

Existen varios recursos en formatos HTML y PDF, muchos de ellos en varios idiomas (incluyendo español), aunque la documentación más completa y actualizada está en inglés. En particular, el *Manual de Referencia de Sage* oficial puede ser encontrado en inglés (no existe en español) en la dirección

`https://doc.sagemath.org/html/en/reference/index.html`

¡pero no es para principiantes! Esté advertido, el manual tiene literalmente miles de páginas. Es una enciclopedia: uno no lo lee de principio a fin, sino solamente lo que necesita. Por suerte hay un campo de búsqueda “Quick Search” a mano izquierda para ayudarnos.

Los tutoriales serán mucho más útiles para la mayoría de los lectores. Algunos están organizados por área matemática y otros por la audiencia objetivo. Existe una colección de tutoriales de Sage, algunos de los cuales se encuentran listados en el apéndice C en la página 285.

Existe también una manera de hacer una búsqueda de todos los docstrings de todos los comandos en Sage. Sin embargo, esto usualmente produce un gran número de resultados. Escribamos, por ejemplo,

Código de Sage

```
1 search_doc('laplace')
```

y veremos una respuesta inmensa. No es exactamente legible para humanos. Aquí está una línea particular de esa salida:

```
en/constructions/calculus.html:228:<div class="section"
id="laplace-transforms">
```

Esta se refiere a una página web. El número en medio, entre el par de dos puntos “:”, indica un número de línea —número uno es la línea superior, número dos es la segunda desde arriba, etc—. La información después del segundo símbolo de dos puntos “:” está escrito en lenguaje HTML.¹¹ Si el lector conoce HTML¹¹, entonces podrá usar esa información.

¹¹Si no lo conoce, tal vez quiera considerar aprenderlo. El lenguaje HTML es extremadamente útil y fácil. Ser capaz de crear páginas web propias es una excelente habilidad, además de altamente comercial.

Hasta ahora, `search_doc` no nos ha dicho nada útil, pero la información antes de los primeros dos puntos es muy valiosa. Ahí tenemos el nombre de un archivo dentro de la jerarquía de directorios que contiene la documentación de Sage. Usualmente Sage es usado a través de un servidor remoto, tal como Sage Cell Server o CoCalc (alias SageMathCloud), sin embargo uno puede hacer una instalación local, lo que significa tener Sage contenido completamente en una computadora personal, y no comunicándose con un servidor remoto. Esto no es recomendable, sino para usuarios expertos. ¿Qué es lo que, entonces, debemos hacer el resto de nosotros (que usamos Sage en nuestro navegador web o en CoCalc) para obtener la información? Debemos ir a la siguiente dirección URL, que es esencialmente la información que `search_doc` no proveyó, pero con algunas adiciones:

`https://doc.sagemath.org/html/en/constructions/calculus.html`.

Como podemos ver, simplemente hemos antepuesto `https://doc.sagemath.org/html/` (la dirección de la documentación de Sage en HTML) a la dirección devuelta por `search_doc` para construir la URL.

1.11 Usando Sage para calcular derivadas

El comando para la derivada es simplemente `diff`. Por ejemplo, para hallar la derivada de $f(x) = x^3 - x$, escribimos

Código de Sage

```
1 diff(x^3 - x, x)
```

para obtener la respuesta

$3x^2 - 1$

O para $f(x) = \sin(x^2)$ escribimos

Código de Sage

```
1 diff(sin(x^2), x)
```

y obtenemos

$2x \cos(x^2)$

En ocasiones habremos definido una función por adelantado, como

Código de Sage

```
1 g(x) = e^(-10*x)
```

así que podremos escribir

Código de Sage

```
1 diff(g(x), x)
```

para obtener la derivada, que naturalmente es

$-10e^{-10x}$

Pero también podemos escribir

Código de Sage

```
1 gprima(x) = diff(g(x), x)
```

que es una forma silenciosa (no se imprime en pantalla), pero nos permite escribir código como `gprima(2)` o `gprima(3)`, lo que por supuesto nos dice los valores $g'(2)$ y $g'(3)$. Otra forma alternativa de la derivada es

Código de Sage

```
1 g(x) = e^(-10*x)
2 g(x).derivative()
```

No podemos, sin embargo, definir una función llamada $g'(x)$. Esto se debe una vez más a que Sage esta construido sobre el lenguaje Python, para el cual el apóstrofo tiene otro significado predefinido. Esto es muy desafortunado; pero por suerte siempre podremos escribir `gprima(x)` o `g_prima(x)`, o cosas similares, de manera que no constituirá una barrera para nuestros propósitos.

Sage también puede calcular derivadas complicadas. Por ejemplo,

Código de Sage

```
1 diff(x^x, x)
```

nos provee la respuesta

$$x^x(\log(x) + 1)$$

Ahora bien, si al lector le interesa dar a sus habilidades de cálculo una práctica, aunque una probablemente intensa, puede tratar de ver cómo calcularía la derivada de x^x con papel y lápiz. A propósito de esto, es importante recordar que `log` se refiere al logaritmo natural, no al logaritmo común, como explicamos en la página 6.

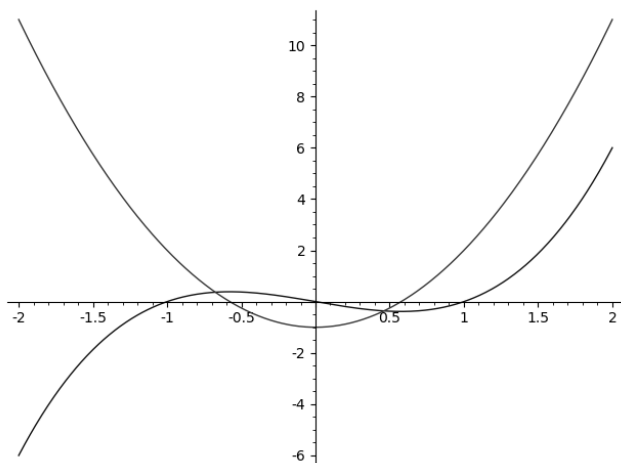
1.11.1 Graficando $f(x)$ y $f'(x)$ juntas

Uno de los trozos de código favoritos del autor es

Código de Sage

```
1 f(x) = x^3 - x
2 fprima(x) = f(x).derivative()
3 plot(f(x), -2, 2, color='blue') + plot(fprima(x), -2, 2, color='red')
```

Esto grafica $f(x) = x^3 - x$ y $f'(x) = 3x^2 - 1$ en una misma imagen, la primera en color azul y la segunda en color rojo. Podemos cambiar la función que estamos analizando al cambiar la primera línea del código anterior. La gráfica es como sigue:



1.11.2 Derivadas de orden superior

Aunque podemos hallar la segunda derivada usando `diff` dos veces, existe una forma alternativa muy conveniente:

Código de Sage

```
1 diff(x^3 - x, x, 2)
```

nos da la segunda derivada

$$6*x$$

La tercera derivada o de órdenes superiores tampoco son problema. Por ejemplo, la tercera derivada puede ser hallada con

Código de Sage

```
1 diff(x^3 - x, x, 3)
```

Los lectores a los que no les moleste teclear más, pueden hacer también

Código de Sage

```
1 derivative(x^3 - x, x, 2)
```

lo que es más claro que `diff`.

1.12 Usando Sage para calcular integrales

Lo único que debemos entender para calcular integrales en Sage es que en realidad hay tres tipos de estas: la integral numérica aproximada, la integral definida exacta y la integral indefinida. Los primeros dos tipos resultan en números, obteniéndose de manera aproximada o exacta, respectivamente. El tercer tipo es la antiderivada simbólica, lo que significa que la respuesta es una función —de hecho, una función cuya derivada es la que escribimos originalmente—. Si esto resulta confuso, consideremos un ejemplo sencillo.

Un ejemplo de integración Usando la integral, ¿cómo calcularíamos el área entre el eje x y la gráfica de $f(x) = x^2 + 1$ en el intervalo $3 \leq x \leq 6$? Pues bien, esa integral sería

$$\int_3^6 (x^2 + 1) dx = \left(\frac{1}{3}x^3 + x \right) \Big|_3^6 = \left(\frac{1}{3}6^3 + 6 \right) - \left(\frac{1}{3}3^3 + 3 \right) = 72 + 6 - (9 + 3) = 66,$$

el cual es un número. Aquí lo hemos hecho analíticamente, usando propiedades del cálculo. Esta es una integral definida exacta: es definida porque resulta en un número y es exacta porque no hicimos ninguna aproximación.

Por otro lado,

$$\int (x^2 + 1) dx = \frac{1}{3}x^3 + x + C$$

es una integral indefinida: es decir, produce una función, no un número. Por supuesto, frecuentemente calculamos una integral indefinida en el proceso de calcular una definida; sin embargo, ambos tipos constituyen categorías distintas.

La integral indefinida La integral indefinida es la más simple. Por ejemplo, si quisiéramos conocer

$$\int x \sin(x^2) dx$$

simplemente escribiríamos

Código de Sage

```
1 integral(x * sin(x^2), x)
```

que da como resultado

$$-1/2 * \cos(x^2)$$

De manera similar, para calcular

$$\int \frac{x}{x^2 + 1} dx$$

escribiríamos

Código de Sage

```
1 integral(x / (x^2 + 1), x)
```

lo que nos da

$$1/2 * \log(x^2 + 1)$$

Más acerca de +C Consideremos el conjunto de funciones cuyas derivadas son $3x^2$. Existe un infinito número de ellas, incluyendo $x^3 + 5$, $x^3 - 8$, $x^3 + 81$, $x^3 + \sqrt{\pi}$ y así sucesivamente, todas ellas diferenciándose en solamente una constante. Es por esa razón que los profesores de cálculo insisten en que se escriba “+C”, como en la expresión

$$\int 3x^2 dx = x^3 + C,$$

cuando se calcula la integral indefinida. Sin embargo, Sage no sabe acerca del uso de +C. Esto es por una buena razón: tiene que ver con el hecho de que no es muy sencillo escribir una función de Python que represente una familia infinita de funciones. Por ejemplo, si definimos $f(x) = \int 3x^2 dx$, entonces ¿qué valor debería devolver Python al preguntar por $f(2)$?, ¿cómo puede saber Sage a qué miembro de la familia nos estamos refiriendo?

Con esto en mente, siempre debemos llevar registro de las +Cs. El manejo correcto de +C es una tarea clave en uno de los proyectos de este libro (acerca de proyectiles balísticos), en la sección 2.5 en la página 78.

La integral definida Alternativamente, podríamos poner un límite inferior de 0 y un límite superior de 1 en la integral:

$$\int_0^1 \frac{x}{x^2 + 1} dx.$$

Para que Sage calcule esta integral definida por nosotros, escribimos

Código de Sage

```
1 integral(x / (x^2 + 1), x, 0, 1)
```

y veremos que la respuesta es

$$1/2 * \log(2)$$

Integrales imposibles La palabra “imposible” es peligrosa en matemáticas, pero debemos decir que algunas integrales son famosamente imposibles. Las dos más famosas son la Integral de Fresnel

$$\int_0^x \sin\left(\frac{\pi t^2}{2}\right) dt$$

que es importante en óptica, y la Integral Gaussiana

$$\int_0^y \frac{2}{\sqrt{\pi}} e^{-x^2} dx$$

que es de suma importancia en probabilidad. ¿A qué nos referimos con “imposible” aquí? No existe una función construida de una cantidad finita de sumas, restas, multiplicaciones, divisiones, raíces, exponenciales, logaritmos, funciones trigonométricas o sus inversas¹² tal que su derivada sea la función de Fresnel o la Gaussiana.

¡Sin embargo, en matemáticas aplicadas necesitamos calcular estas integrales! En particular, la Gaussiana es extraordinariamente importante en estadística. Lo que se hace entonces es dividir el intervalo de integración en muchas, muchas regiones. Varios rectángulos o trapezoides definidos por estas regiones pueden ser usados para aproximar el área entre el eje x y la curva. Las áreas numéricas de estas “subáreas” son sumadas para dar una aproximación numérica de la integral. (Por esta razón es llamada “integración numérica”.) Cada uno de los rectángulos o trapezoides puede producir una pequeña cantidad de error, pero cuando se suman sus áreas, la esperanza es que estos errores se cancelen un poco entre sí. Probablemente el lector usó esta técnica en algún punto de la clase de cálculo, o la usará en un futuro.

Métodos incluso más sofisticados para la integración numérica incluyen el uso de polinomios en lugar de trapezoides, y Sage usa técnicas extraordinariamente intrincadas para lograr la mejor aproximación posible. Estas aproximaciones son un tema antiguo y profundo, remontándose a la Regla de Simpson, quien usó pequeñas parábolas en lugar de trapezoides estrechos. Esto le permitió obtener aproximaciones bastante precisas usando muchos menos cálculos de lo que la Regla del Trapecio requiere. Dado que Simpson trabajó en la primera mitad del siglo XVIII, este es un logro notable.

Lo anterior puede haber resultado un poco confuso, pero todo el punto de usar Sage es que este se ocupará de estos detalles por nosotros, siempre y cuando sepamos qué es lo que queremos pedirle.

Veamos un ejemplo específico. Consideremos la función

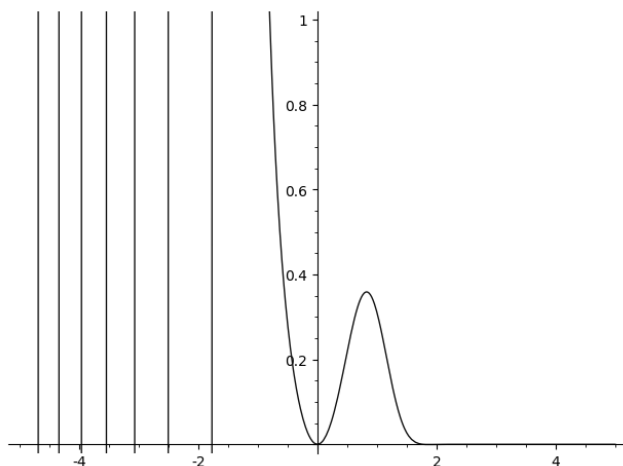
$$f(x) = e^{-x^3} \sin(x^2),$$

que fácilmente puede verse que es continua. Podemos graficarla con el comando

Código de Sage

```
1 plot(exp(-x^3) * sin(x^2), -5, 5, ymin=0, ymax=1)
```

Obtenemos la siguiente imagen



Esta función parece ser interesante, pero calcular su integral es realmente frustrante. El autor no puede calcularla con papel y lápiz, y de seguro que el lector tampoco podrá. Aunque efectivamente existe una función $g(x)$, cualquiera que sea esta, tal que su derivada es $f(x) = e^{-x^3} \sin(x^2)$ —una consecuencia lógica del análisis real y del hecho que f es continua— el aspecto práctico del asunto es que no hay forma de escribir $g(x)$ como una fórmula finita. Para ser realmente precisos, no existe tal función $g(x)$, construida solamente de una cantidad finita de sumas, restas, multiplicaciones, divisiones, raíces, exponenciales, logaritmos, funciones

¹²También deberíamos incluir las funciones trigonométricas hiperbólicas y las trigonométricas inversas.

trigonométricas, sus inversas o sus parientes hiperbólicos, de manera que $g'(x) = f(x)$. En otras palabras, $f(x) = e^{-x^3} \sin(x^2)$ es integrable, pero la integral no puede ser escrita en términos de funciones elementales. Sin embargo, sí es posible escribir una tal función $g(x)$ como una serie infinita en notación Sigma mayúscula.

Regresando nuevamente a Sage, dado que

$$\int e^{-x^3} \sin(x^2) dx$$

no tiene expresión sencilla, entonces cuando escribimos

Código de Sage

```
1 integral(exp(-x^3) * sin(x^2), x)
```

Sage responde con

```
integrate(e^(-x^3)*sin(x^2), x)
```

lo que es básicamente una admisión de derrota. Sin embargo, no debemos temer, pues podemos calcular respuestas numéricas de manera aproximada —con muy alta precisión—, lo que explicamos a continuación.

Integración numérica en Sage Una integral nefasta, favorita entre los profesores de cálculo, es

$$\int t^{20} e^t dt,$$

donde el 20 puede ser cualquier otro número decentemente grande. Esta surge al exponer la función Gamma de Euler, pero no estamos particularmente interesados en ese tema ahora. En cualquier caso, podríamos escribir

Código de Sage

```
1 integral((t^20) * (e^t), t)
```

pero entonces se nos respondería

```
(t^20 - 20*t^19 + 380*t^18 - 6840*t^17 + 116280*t^16 - 1860480*t^15 +
27907200*t^14 - 390700800*t^13 + 5079110400*t^12 - 60949324800*t^11 +
670442572800*t^10 - 6704425728000*t^9 + 60339831552000*t^8 -
482718652416000*t^7 + 3379030566912000*t^6 - 20274183401472000*t^5 +
101370917007360000*t^4 - 405483668029440000*t^3 + 1216451004088320000*t^2
- 2432902008176640000*t + 2432902008176640000)*e^t
```

lo cual es lo suficientemente extenso como para no poder lograr tener una idea de lo que significa. Si, en cambio, tuviésemos límites de integración como en

$$\int_2^3 t^{20} e^t dt,$$

entonces escribiríamos

Código de Sage

```
1 integral((t^20) * (e^t), t, 2, 3)
```

y obtendríamos

```
121127059051462881*e^3 - 329257482363600896*e^2
```

en su lugar, ¡lo que ya es una excelente respuesta! Si realmente deseamos una solución numérica, debemos usar un viejo truco, el comando `N()`, y escribir

Código de Sage

```
1 N(integral((t^20) * (e^t), t, 2, 3))
```

lo cual nos devuelve el número

8.79797452800000e9

que, escrito en una forma más conocida, es 8,797 974 528 000 00e9. Esto resalta las diferencias y similitudes de los varios tipos de resultados. Si hemos olvidado el comando `N()`, podemos ver la página 4.

Este último número que hemos calculado pareciera ser un entero, pues termina en 528 cuando desarrollamos la parte de 10^9 . Pero al observar la respuesta exacta, vemos claramente que no, no es un entero. Sabemos esto porque e^2 y e^3 están involucrados, y e es un número irracional. Cuandoquiera que tratemos con cálculos computacionales, es de crítica importancia llevar cuenta de cuáles resultados son aproximaciones y cuáles son exactos. Los números irracionales pueden expresarse de manera exacta en ocasiones (tal como hicimos aquí usando e^2 y e^3), pero esa es la excepción, no la regla. En cualquier caso, este párrafo no tiene nada que ver con Sage, así que retornemos al tema en cuestión.

Hablando de aproximaciones, también podemos saltar a la integral numérica muy rápidamente, sin pasar por la simbólica. De seguro que podemos calcular esta integral:

$$\int_0^1 (x^3 - x) \, dx = \left(\frac{1}{4}x^4 - \frac{1}{2}x^2 \right) \Big|_0^1 = \left(\frac{1}{4}1^4 - \frac{1}{2}1^2 \right) - \left(\frac{1}{4}0^4 - \frac{1}{2}0^2 \right) = \frac{1}{4} - \frac{1}{2} = -\frac{1}{4}.$$

Sin embargo, si deseamos pedirle a Sage que la calcule de forma numérica sin pasos intermedios, entonces debemos escribir

Código de Sage

```
1 numerical_integral(x^3 - x, 0, 1)
```

que nos devolverá la salida

(-0.24999999999999997, 2.775557561562891e-15)

donde el primer número es la mejor aproximación que Sage tiene para la respuesta, y el segundo número es la incertidumbre del cómputo. En este caso, esta incertidumbre es la 2,77 mil billonésima parte de la unidad, lo que es realmente impresionante.

Por supuesto, Sage puede realizar esta integral igual de bien usando los comandos que hemos aprendido momentos atrás. Lo interesante de la integración numérica son los casos en los que no se puede calcular la integral indefinida porque, como en la Integral de Fresnel, no puede ser escrita en términos de funciones elementales, pero se pueden hallar buenos estimados numéricos. Consideremos nuevamente

$$\int_1^3 e^{-x^3} \sin(x^2) \, dx,$$

pero notemos que ahora hemos añadido límites de integración, de manera que se pueda realizar numéricamente. Escribimos

Código de Sage

```
1 numerical_integral(exp(-x^3) * sin(x^2), 1, 3),
```

lo que nos devuelve

(0.07799701126208781, 8.684049685199504e-16)

donde, como antes, el primer número representa la respuesta, mientras que el segundo es la incertidumbre. En este caso, la incertidumbre es la 8,68 diez mil billonésima parte de la unidad —nuevamente muy impresionante—.

Integración por fracciones parciales Una cuestión muy importante que surge en problemas relacionados a *ecuaciones diferenciales* y *cálculo II* es la integración por fracciones parciales. Si deseamos conocer la descomposición en fracciones parciales de

$$\frac{x^3 - x}{x^2 + 5x + 6},$$

entonces debemos hacer

Código de Sage

```
1 f(x) = (x^3 - x) / (x^2 + 5*x + 6)
```

seguido de

Código de Sage

```
1 f(x).partial_fraction()
```

para recibir la respuesta

$$x + 24/(x + 3) - 6/(x + 2) - 5$$

que es un resultado correcto. De hecho, acabamos de determinar que

$$\frac{x^3 - x}{x^2 + 5x + 6} = x + \frac{24}{x + 3} - \frac{6}{x + 2} - 5.$$

Sin embargo, también podemos tomar un atajo y directamente pedir

Código de Sage

```
1 integral((x^3 - x) / (x^2 + 5*x + 6), x)
```

o, mejor aun, ya que hemos definido la función f ,

Código de Sage

```
1 integral(f(x), x)
```

lo que, en ambos casos, resulta en

$$1/2*x^2 - 5*x + 24*\log(x + 3) - 6*\log(x + 2)$$

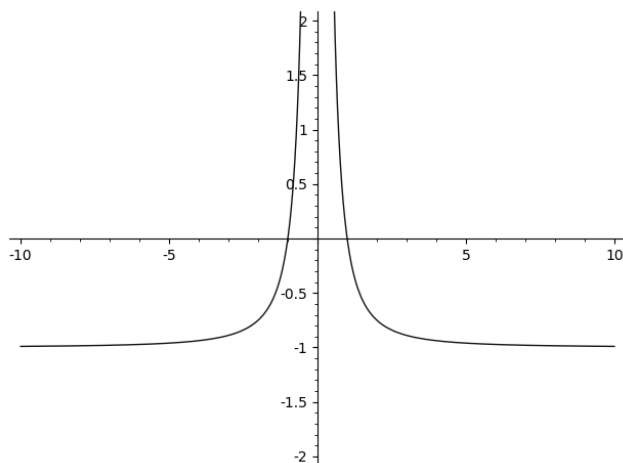
Integrales impropias Aunque la integral

$$\int \left(-1 + \frac{1}{x^2}\right) dx = -x - \frac{1}{x} + C$$

es matemáticamente válida y el lado derecho de la igualdad puede ser evaluado en $x = -1$ y $x = 1$, no existe una respuesta finita a

$$\int_{-1}^1 \left(-1 + \frac{1}{x^2}\right) dx,$$

pues esta integral impropia es divergente. La gráfica nos ayudará a revelar por qué es este el caso. He aquí la gráfica de $-1 + 1/x^2$:



que hemos generado con

Código de Sage

```
1 plot(-1 + 1/x^2, -10, 10, ymin=-2, ymax=2)
```

Sage está consciente de todo esto y dirá

$-x - 1/x$

en respuesta a

Código de Sage

```
1 integral(-1 + 1/x^2, x)
```

pero en cambio responderá

ValueError: Integral is divergent.

a la petición

Código de Sage

```
1 integral(-1 + 1/x^2, x, -1, 1).
```

Para demostrar que la integral anterior en efecto diverge, observemos que

$$\int_{-1}^{-1} \left(-1 + \frac{1}{x^2}\right) dx = \int_{-1}^0 \left(-1 + \frac{1}{x^2}\right) dx + \int_0^1 \left(-1 + \frac{1}{x^2}\right) dx.$$

Consideremos la integral de 0 a 1 anterior, calculando en realidad desde a hasta 1, donde a es ligeramente mayor que 0:

$$\begin{aligned} \int_a^1 \left(-1 + \frac{1}{x^2}\right) dx &= \left(-x - \frac{1}{x}\right) \Big|_a^1 \\ &= \left(-1 - \frac{1}{1}\right) - \left(-a - \frac{1}{a}\right) \\ &= a - 2 + \frac{1}{a}. \end{aligned}$$

Como podemos ver, al aproximarse a a 0 en el conjunto de los reales positivos, el valor de $a - 2 + 1/a$ se incrementa más allá de cualquier cota. Más precisamente, podemos escribir

$$\lim_{a \rightarrow 0^+} \left(a - 2 + \frac{1}{a}\right) = \infty,$$

lo que a su vez implica

$$\lim_{a \rightarrow 0^+} \int_a^1 \left(-1 + \frac{1}{x^2} \right) dx = \infty.$$

Un razonamiento similar muestra que esto también es verdad para la integral de -1 a 0 . Por lo tanto, la integral original es igual a $\infty + \infty = \infty$.

Más integrales impropias Otro tipo de integral impropia es

$$\int_2^{\infty} \frac{1}{x^2} dx$$

que puede ser escrita en Sage como

Código de Sage

```
1 integral(1 / x^2, x, 2, oo)
```

a lo que nos responde $1/2$. Otro ejemplo es

$$\int_{-\infty}^{\infty} e^{-x^2} dx$$

que en Sage lo escribimos

Código de Sage

```
1 integral(exp(-x^2), x, -oo, oo)
```

y nos da la respuesta correcta de $\sqrt{\pi}$.

La clave aquí es simplemente que “oo” es un código especial para “infinito”. Está formado por dos letras “o”, y la lógica detrás de esto es que, visto de reojo, “oo” se parece mucho al símbolo de infinito.

La función erf y las integrales Entre los matemáticos y especialmente entre los estadísticos, la integral

$$\int_0^y \frac{2}{\sqrt{\pi}} e^{-x^2} dx = \text{erf}(y)$$

es extraordinariamente importante. Esta es la Integral Gaussiana de la que hablamos antes, en particular, indicando que no puede ser expresada como una combinación finita de funciones elementales. Esta es tan importante que se le ha dado un nombre propio, “erf”. Un nombre amigable y fácilmente pronunciable, erf significa “Función Error”.

Entonces, si escribimos en Sage

Código de Sage

```
1 integral((2 / sqrt(pi)) * exp(-x^2), x, -oo, 2)
```

este responderá

$$(\text{sqrt}(\pi) * \text{erf}(2) + \text{sqrt}(\pi)) / \text{sqrt}(\pi)$$

lo cual es totalmente correcto, pues

$$\begin{aligned}\int_{-\infty}^2 \frac{2}{\sqrt{\pi}} e^{(-x^2)} dx &= \int_{-\infty}^0 \frac{2}{\sqrt{\pi}} e^{(-x^2)} dx + \int_0^2 \frac{2}{\sqrt{\pi}} e^{(-x^2)} dx \\ &= 1 + \operatorname{erf}(2) \\ &= \frac{\sqrt{\pi}}{\sqrt{\pi}} (1 + \operatorname{erf}(2)) \\ &= \frac{\sqrt{\pi} \operatorname{erf}(2) + \sqrt{\pi}}{\sqrt{\pi}}.\end{aligned}$$

Probablemente el lector considere que $1 + \operatorname{erf}(2)$ es una respuesta más compacta y sencilla. El autor se siente inclinado a estar de acuerdo con ello, y desconoce por qué Sage no simplifica su respuesta final. En cambio, siempre es posible pedir una simplificación de un resultado una vez que se ha calculado, usando el comando `full_simplify()` o su alias `simplify_full()`:

Código de Sage

```
1 mi_integral = integral((2 / sqrt(pi)) * exp(-x^2), x, -oo, 2)
2 mi_integral.full_simplify()
```

Esto nos dará el $\operatorname{erf}(2) + 1$ que esperábamos.

En cualquier caso, el punto es que ahí afuera existen algunas funciones cuyas integrales no pueden escribirse en términos de funciones elementales, pero que Sage calcula de todas formas, usando la función `erf`.

El comando `assume` y las integrales Este ejemplo fue identificado por Joseph Loeffler, quien lo trajo a la atención del autor. Digamos que queremos evaluar la integral

$$\int_1^x \frac{t^3 + 30}{t} dt$$

y en consecuencia escribimos los comandos de Sage

Código de Sage

```
1 var('t')
2 integrate((t^3 + 30) / t, t, 1, x)
```

(Este es un buen momento para indicar que `integrate` es sinónimo de `integral`.) Resulta que recibiremos un gigantesco mensaje de error como respuesta. Debemos recordar que en Sage, las últimas líneas de un mensaje de error son las que nos importan. En este caso, estas son:

```
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(x-1>0)', see 'assume?' for more
details)
Is x-1 positive, negative or zero?
```

La queja aquí, enviada a Sage desde el software Maxima, está justificada porque la función $f(t) = (t^3 + 30)/t$ “explota” en $t = 0$ debido a la división entre cero. La integral es, por lo tanto, impropia si 0 se encuentra entre 1 y x . Sin embargo, si $x - 1 > 0$, o, en español simple, si $x > 1$, entonces eso claramente no ocurrirá. De esta manera, Sage tiene plena justificación al pedirnos que declaremos explícitamente esta suposición.

Sigamos el curso de acción recomendado por Sage, añadiendo el comando `assume` antes de `integrate`. Ahora tenemos

Código de Sage

```
1 var('t')
2 assume(x > 1)
3 integrate((t^3 + 30) / t, t, 1, x)
```

que produce la respuesta

$$1/3*x^3 + 30*\log(x) - 1/3$$

como esperábamos.

El comando `assume` es lo suficientemente raro como para que muchos usuarios experimentados de Sage no sepan que existe. Es poco probable que surja en nuestro trabajo diario. Sin embargo, veremos otro ejemplo de su uso en la página 176, en una sumatoria, y en la página 192 en una integral relacionada a la Transformada de Laplace.

1.13 Compartiendo los resultados de nuestro trabajo

En esta sección estudiaremos siete formas efectivas de compartir los resultados de nuestro trabajo en Sage. Estas pueden ser muy útiles para la colaboración científica, presentar un trabajo a un miembro de la facultad, recibir ayuda de un amigo, o simplemente explicar un concepto a alguien.

Cuando tenemos un trabajo hecho en Sage Cell, podemos presionar el botón “Share” (“Compartir”). Entonces veremos desplegarse opciones para crear un permalink (enlace web permanente), crear un link (enlace web) corto temporal y crear un “código de barras 2D”, también llamado “código QR”.

El permalink es el favorito del autor; esta opción genera una URL enorme, y nos muestra que esta es válida y funcional al recargar la página con esa dirección. Copiamos esta URL del navegador web, del campo donde usualmente escribiríamos `https://sagecell.sagemath.org/`, teniendo mucho cuidado en resaltar la dirección completa. Esta URL puede entonces ser usada en documentos, correos electrónicos, en chats e incluso en Facebook. El permalink de hecho codifica todo el código de Sage en la ventana y, por lo tanto, dicho código no es almacenado en ningún lugar, y no puede ser perdido, borrado o destruido. Mientras retengamos la URL completa sin modificar, esta siempre funcionará, incluso años después. El permalink suele ser largo y poco grato a la vista. Esto no es sorpresa, pues este codifica nuestro bloque entero de código de Sage. De hecho, puede ser tan largo que llegue a causar problemas en algunos navegadores web o procesadores de correo electrónico. En casi todos los casos, superará los 140 caracteres y así no podrá ser compartido en un mensaje de texto o en Twitter.

El link corto temporal, en cambio, es más pequeño y menos intimidante para un humano. Siendo corto, es menos probable que sea truncado en tránsito cuando se envíe por varios medios. Sin embargo, el link no contiene una copia completa de nuestro código; este es almacenado en el servidor y la URL lo recupera. No hay ninguna garantía de cuánto tiempo funcionará el link: tal vez un día, tal vez un año, o tal vez para siempre, pero podría ser solo una hora. La experiencia del autor es que frecuentemente funcionará hasta varios días después de ser generado. En general, si la longitud y la elegancia no son problema, entonces el permalink debería ser usado siempre, y no el link corto.

Estos dos anteriores son los métodos favoritos del autor para compartir código de Sage. El tercer método es el gran código de barras 2D producido cuando presionamos el botón “Share”. Este de hecho codifica el permalink y por lo tanto funcionará con todos los teléfonos inteligentes que puedan usar estos códigos. La mayoría de los teléfonos requerirán instalar una app¹³, como “QR Code Reader”, para ser capaces de procesar los códigos de barras, que a veces son llamados “Códigos QR”, “Códigos Quick Response (de respuesta rápida)” o “Códigos de Barras Matriciales”. Si fotografiamos la imagen generada por el servidor Sage Cell usando una de tales apps,

¹³Realmente se requiere una app, sin embargo. De otro modo, solo se tiene la imagen de un código QR, que es totalmente inútil.

entonces se abrirá el navegador web y se nos presentará el mismo resultado que con el permalink. ¡Ni siquiera necesitaremos presionar “Evaluate”! Aquí tenemos un ejemplo de código QR:



El cuarto método para compartir nuestro trabajo en Sage es cuando hacemos un gráfico (como aprendimos en la sección 1.4 en la página 9). Si deseamos incluir esta imagen en cualquier tipo de documento, o enviarlo en un correo electrónico, entonces simplemente guardamos el archivo de imagen desde el navegador web —de la misma manera en que guardaríamos cualquier otra imagen—. En la mayoría de los navegadores, esto implica hacer click derecho, seleccionar “Guardar imagen como...” e introducir un nombre para el archivo en el lugar adecuado. La imagen se almacenará por defecto en el formato “portable network graphics” o *.png. Este funcionará en virtualmente todos los programas que permitan importar imágenes. Alternativamente, algunas revistas matemáticas y de física prefieren el formato “encapsulated postscript” o *.eps. Estos archivos pueden generarse con comandos como

Código de Sage

```
1 P = plot(x^3 - x, -2, 2)
2 P.save('migrafica.eps')
```

lo que nos mostrará un link para iniciar la descarga de la imagen `migrafica.eps`.

El quinto método simplemente consiste en resaltar el código de Sage, y copiarlo y pegarlo en un correo electrónico, en CoCalc (SageMathCloud) o en el software de gestión del curso de la universidad, como D2L, por ejemplo. Enviar el código por correo electrónico puede ahorrarnos tiempo y dificultades.

El sexto método es imprimir la ventana del Servidor Sage Cell en papel. Usamos el comando de impresión de nuestro navegador para imprimir la página tal como haríamos con cualquier otra. Esta es una forma que usa el autor en sus clases para que sus alumnos envíen un problema de la tarea que ha sido resuelto usando Sage. Si se desea, mediante un poco de navegación y redimensionado de la página web, frecuentemente se pueden hacer visibles el bloque de código de Sage y su salida al mismo tiempo, siendo posible obtener una buena captura de pantalla (screenshot) de nuestro trabajo, que también puede imprimirse.

Finalmente, pero no menos importante, el séptimo método consiste en seguir pasos similares a los requeridos para imprimir una captura de pantalla de Sage Cell, pero en lugar de usar papel, seleccionamos “Guardar como PDF” del menú de impresión del navegador. Este PDF es esencialmente lo mismo que se obtiene con el método anterior, pero en forma electrónica en lugar de física.

La colaboración científica es uno de los grandes placeres de trabajar en las áreas conocidas conjuntamente como STEM¹⁴. Es la esperanza del autor que el lector frecuentemente tenga oportunidad de compartir con otros, no solo sus ideas, sino los detalles matemáticos y cuantitativos profundos de su trabajo. Dado que Sage ha sido desarrollado por cientos de personas distribuidas por todo el mundo, no es sorpresa que existan muchas formas predefinidas de intercambio en Sage. CoCalc (SageMathCloud) tiene incluso más métodos de intercambio, incluyendo proyectos públicos, acceso por invitación solamente y chat en vivo. Podremos aprender tales características en la página web de CoCalc.

¹⁴STEM: Science, Technology, Engineering and Mathematics o, en español, Ciencia, Tecnología, Ingeniería y Matemáticas.

1.14 Los varios usos del comando show()

El comando `show()` es muy flexible y útil, y es uno de los favoritos del autor. Ya vimos, en la subsección 1.4.2 en la página 18, que podemos usarlo para mostrar la imagen resultante de superponer varias gráficas. También vimos en la página 120 que podemos usar `show()` para forzar el rango del eje x que se mostrará en una gráfica (aun cuando estemos fuera del dominio de una función). Ahora veremos uno de los usos más frecuentes que le da el autor: para mostrar fórmulas o funciones grandes y complicadas.

Por ejemplo, cuando trabajamos con hipotecas, una de las fórmulas que se usa frecuentemente es para calcular el “valor presente de una anualidad decreciente”, que denotamos como VP . Este es simplemente el valor monetario de una fiel promesa de entregar c dólares, por m intervalos de tiempo igualmente espaciados por año, durante t años, empezando dentro de $1/m$ -ésima parte de un año a partir del momento presente. Usando r para denotar la tasa de interés nominal, la fórmula es

$$PV = c \frac{1 - (1 + r/m)^{-mt}}{r/m}.$$

Para poner esto en lenguaje más simple, sean $m = 12$ y $t = 30$, lo que nos da la hipoteca más común en USA: una con interés fijo por 30 años (o 360 meses). El VP es el valor monetario de la promesa fiel del cliente de entregar c dólares cada mes, durante 30 años, empezando dentro de un mes a partir del momento presente.

Alguna mañana, cuando no hemos consumido suficiente café, podríamos ser susceptible de escribir algo como

Código de Sage

```
1 var('Valor_Presente c n r m t')
2 show(Valor_Presente == c * (1-(1+r/m)^(m*t)) / r / m)
```

Sage nos mostrará la respuesta

$$Valor_{Presente} = -\frac{c\left(\left(\frac{r}{m} + 1\right)^{mt} - 1\right)}{mr}$$

Esto nos ayudaría a caer en cuenta de que hemos cometido un error. El problema es “ $/r/m$ ”, que debería ser en cambio “ $/(r/m)$ ”. Podemos notar el error porque vemos r y m lado a lado en el denominador de la ecuación mostrada, que no es lo que queríamos. En consecuencia, hacemos la corrección como sigue:

Código de Sage

```
1 var('Valor_Presente c n r m t')
2 show(Valor_Presente == c * (1-(1+r/m)^(m*t)) / (r/m))
```

Sage ahora nos muestra la respuesta

$$Valor_{Presente} = -\frac{cm\left(\left(\frac{r}{m} + 1\right)^{mt} - 1\right)}{r}$$

Aunque esto no es idéntico a lo que teníamos en mente, es fácil ver que sí es matemáticamente equivalente. Por lo tanto, podemos tener la confianza de que hemos escrito la fórmula correctamente.

El comando `show()` también es muy útil para trabajar con matrices. Aquí tenemos un ejemplo sencillo en el que podemos apreciar que este nos da una salida más elegante de lo normal —aunque la salida usual también es perfectamente aceptable—. Si escribimos

Código de Sage

```
1 M = matrix(3, 3, [1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9])
2 print(M)
3 show(M)
```

obtenemos

```
[ 1 1/2 1/3]
[1/4 1/5 1/6]
[1/7 1/8 1/9]
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

Aquellos que estén familiarizados con \LaTeX estarán felices de saber que también podemos escribir la instrucción `latex(M)`, con objeto de obtener el código para la misma composición de la matriz M que nos muestra `show` aquí, pero en el sistema tipográfico \LaTeX . Este tema lo discutiremos más a fondo en la sección 4.15 en la página 159. Recuértese que se pronuncia “la-tec”, de la misma forma como la expresión “la tecnología”; *no se pronuncia* “látex”, como en la expresión “guantes de látex”. También tenemos disponible el paquete `SageTeX`, el cual nos permite incluir resultados obtenidos con Sage directamente en \LaTeX . Por ejemplo, en nuestro documento podemos escribir `\sage{M}`, lo que nos mostrará la matriz directamente, sin necesidad de hacer ninguna transcripción manual. Este tema lo discutiremos más a fondo en la sección 4.25 en la página 204

Si estuviésemos enseñando a alguien acerca de la Fórmula Cúbica de Cardano y las raíces algebraicas exactas de polinomios, entonces el comando `show()` es mucho mejor para el trabajo que `print()`. Como podemos ver en la imagen abajo, la salida usual no es muy legible, pero la salida formateada con `show()` está muy cercana a ser perfecta. Si escribimos

Código de Sage

```
1 soluciones = solve(x^3 + x^2 + x + 2 == 0, x)
2
3 print(soluciones[0])
4 print(soluciones[1])
5 print(soluciones[2])
6
7 show(soluciones[0])
8 show(soluciones[1])
9 show(soluciones[2])
```

Sage nos responderá con

```
x == -1/6*(1/2)^(1/3)*(3*sqrt(83)*sqrt(3) - 47)^(1/3)*(I*sqrt(3) + 1) +
2/3*(1/2)^(2/3)*(-I*sqrt(3) + 1)/(3*sqrt(83)*sqrt(3) - 47)^(1/3) - 1/3

x == -1/6*(1/2)^(1/3)*(3*sqrt(83)*sqrt(3) - 47)^(1/3)*(-I*sqrt(3) + 1) -
2/3*(1/2)^(2/3)*(-I*sqrt(3) - 1)/(3*sqrt(83)*sqrt(3) - 47)^(1/3) - 1/3

x == 1/3*(1/2)^(1/3)*(3*sqrt(83)*sqrt(3) - 47)^(1/3) -
4/3*(1/2)^(2/3)/(3*sqrt(83)*sqrt(3) - 47)^(1/3) - 1/3
```

$$x = \left(\frac{-1}{6}\right) \left(\frac{1}{2}\right)^{\frac{1}{3}} \left(3\sqrt{83}\sqrt{3} - 47\right)^{\frac{1}{3}} (i\sqrt{3} + 1) + \frac{2 \left(\frac{1}{2}\right)^{\frac{2}{3}} (-i\sqrt{3} + 1)}{3 \left(3\sqrt{83}\sqrt{3} - 47\right)^{\frac{1}{3}}} + \frac{-1}{3}$$

$$x = \left(\frac{-1}{6}\right) \left(\frac{1}{2}\right)^{\frac{1}{3}} \left(3\sqrt{83}\sqrt{3} - 47\right)^{\frac{1}{3}} (-i\sqrt{3} + 1) - \frac{2 \left(\frac{1}{2}\right)^{\frac{2}{3}} (-i\sqrt{3} - 1)}{3 \left(3\sqrt{83}\sqrt{3} - 47\right)^{\frac{1}{3}}} + \frac{-1}{3}$$

$$x = \frac{1}{3} \left(\frac{1}{2}\right)^{\frac{1}{3}} \left(3\sqrt{83}\sqrt{3} - 47\right)^{\frac{1}{3}} - \frac{4 \left(\frac{1}{2}\right)^{\frac{2}{3}}}{3 \left(3\sqrt{83}\sqrt{3} - 47\right)^{\frac{1}{3}}} + \frac{-1}{3}$$

Igual que antes, la instrucción `latex(soluciones[2])` nos da el código de \LaTeX para componer la última solución. También podemos escribir en nuestro documento `\sage{soluciones[2]}`, para incluir la solución automáticamente. Determinar el código de \LaTeX para esto sin la ayuda de Sage sería una tarea muy poco placentera para un humano.

Finalmente, pero no menos importante, si el lector conoce lo que son las fracciones continuas, estará feliz de saber que `show()` las puede mostrar elegantemente. (Aprenderemos más acerca de las fracciones continuas en la sección 4.20 en la página 178.) De hecho, el comando `show` hace que el uso de las fracciones continuas en Sage sea mucho más comprensible, especialmente para los principiantes. Aquí tenemos un ejemplo:

Código de Sage

```
1 fc = continued_fraction(79/141)
2 show(fc)
```

Esto produce la siguiente salida:

$$0 + \frac{1}{1 + \frac{1}{1 + \frac{1}{3 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{5}}}}}}}$$

Nuevamente, podemos obtener el código de \LaTeX para esta fracción continua al escribir `latex(fc)` en Sage o podemos incluirla directamente con la instrucción `\sage{fc}` en nuestro documento de \LaTeX .

1.15 Un detalle técnico sobre funciones en Sage

Antes de continuar, vale la pena hacer una pausa para discutir un tecnicismo que está presente cuando trabajamos con funciones matemáticas en Sage. Si el lector es aún principiante y encuentra esta subsección confusa, no debe preocuparse; puede regresar a este punto más adelante cuando tenga más experiencia con Sage.

Consideremos por ejemplo,

Código de Sage

```
1 f(x) = x^2 - 5*x + 6
```

Sage interpreta f y $f(x)$ de la misma forma en que un matemático lo haría: f es la *función* en sentido formal, mientras que $f(x)$ es el *valor* de f en el punto x , así como $f(2)$ es el valor de la función en el punto $x = 2$. Esto tiene consecuencias muy interesantes. Por ejemplo, si escribimos

Código de Sage

```
1 print(f)
```

obtenemos la respuesta

$$x \mapsto x^2 - 5x + 6$$

mientras que, si escribimos

Código de Sage

```
1 print(f(x))
```

obtenemos en cambio

$$x^2 - 5x + 6$$

La primera notación nos dice que f es una *función* o *mapeo* tal que, dado un número para x , Sage reemplazará x en la fórmula mostrada ahí con ese número y nos devolverá la respuesta que resulta. El símbolo \mapsto se lee “evalúa a” o “mapea a”. Este resultado que se muestra es exactamente lo mismo que la notación matemática $f: x \mapsto f(x)$ en general, o $f: x \mapsto x^2 - 5x + 6$ en particular, lo cual nos da una descripción completa de la función misma. La segunda notación nos muestra el *valor* de la función f en un punto x , es decir $f(x)$. No es muy diferente a decir que el valor $f(5)$ es $5^2 - 5 \cdot 5 + 6 = 6$, o que $f(y) = y^2 - 5y + 6$. Esto es exactamente lo mismo que la notación matemática $f(x) = x^2 - 5x + 6$.

En general, cuando operamos con funciones formales, obtenemos una función formal, pero si trabajamos con valores de funciones, obtenemos el valor de una función. El siguiente código ilustrará mejor esta distinción:

Código de Sage

```
1 f(x) = x^2 - 5*x + 6
2 g(x) = x^2 - 8*x + 15
3
4 print('Funciones formales:')
5 print(f + g)
6 print(f.factor())
7 print(gcd(f, g))
8
9 print('Valores de funciones:')
10 print(f(x) + g(x))
11 print(f(x).factor())
12 print(gcd(f(x), g(x)))
```

Este produce la siguiente salida:

```
Funciones formales:
x |--> 2*x^2 - 13*x + 21
x |--> (x - 2)*(x - 3)
x |--> x - 3
Valores de funciones:
2*x^2 - 13*x + 21
(x - 2)*(x - 3)
x - 3
```

(Recordemos que podemos reemplazar `f.factor()` con la forma equivalente `factor(f)`, así como también `f(x).factor()` puede reemplazarse con `factor(f(x))`.)

Algo que las funciones f y sus valores $f(x)$ tienen en común es que pueden ser evaluados en los puntos que deseemos, usando la notación estándar de evaluación.

- Por ejemplo, $f(5)$ y $f(x)(x=5)$ dan el mismo resultado 6. La primera forma indica evaluar la función f en el punto $x = 5$; la segunda indica que en la fórmula que define el valor $f(x)$ se reemplace cada ocurrencia de x por 5.
- Si queremos obtener el gcd de f y g , y luego evaluarlo en $x = 2$, podemos escribir $\text{gcd}(f, g)(x=2)$ o también $\text{gcd}(f(x), g(x))(x=2)$. En cualquier caso, el resultado es -1 . Puede resultar más bondadoso para cualquiera que deba leer nuestro código que escribamos $h(x) = \text{gcd}(f(x), g(x))$ y entonces $h(2)$ en la siguiente línea.
- De manera similar, si quisiéramos factorizar la función f y evaluarla en $x = -3$, podemos escribir $f.\text{factor}()(x=-3)$ o también $f(x).\text{factor}()(x=-3)$; en ambos casos, obtenemos 30. Sin embargo, nótese que factorizar una función antes de evaluarla no tiene ningún efecto, aparte de desperdiciar ciclos computacionales.

Por supuesto, este comportamiento no se restringe a polinomios, como en estos ejemplos. En la sección 1.11 vimos cómo trabajar con derivadas. Ahí definimos la función $g(x) = e^{-10x}$. Nuevamente, podemos manejar funciones en sentido formal, simplemente eliminando la parte “ (x) ” de “ $g(x)$ ”. Por ejemplo, podemos escribir $\text{diff}(g, x)$ y $g.\text{derivative}()$ como alternativas a $\text{diff}(g(x), x)$ y $g(x).\text{derivative}()$, respectivamente. En esos casos obtendremos los mismos resultados, pero con la añadidura $x |-->$. Es decir,

Código de Sage

```
1 g(x) = e^(-10*x)
2 diff(g(x), x)
3 diff(g, x)
4 g(x).derivative()
5 g.derivative()
```

nos dará los resultados

```
-10*e^(-10*x)
x |--> -10*e^(-10*x)
-10*e^(-10*x)
x |--> -10*e^(-10*x)
```

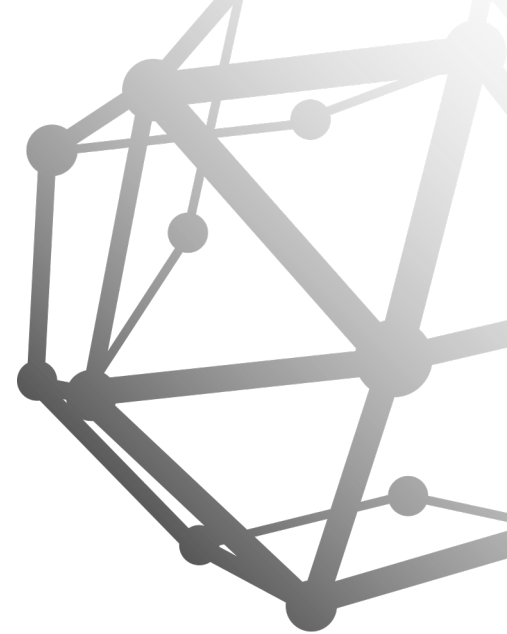
Aquí tenemos otro caso en el que podemos trabajar con funciones formales: en la sección 1.12, definimos la función $f(x) = (x^3 - x) / (x^2 + 5x + 6)$ para descomponerla en fracciones parciales y obtener su integral. Si en lugar de escribir $f(x).\text{partial_fraction}()$ (como hicimos entonces), escribimos $f.\text{partial_fraction}()$, obtendremos

```
x |--> x^2 - 5*x + 6
```

También podríamos escribir $\text{integral}(f, x)$ para obtener la integral:

```
x |--> 1/3*x^3 - 5/2*x^2 + 6*x
```

En este libro trabajaremos más con valores $f(x)$ que con funciones f , pero el lector puede usar la notación que más le plazca. En particular, los estudiantes de matemáticas querrán usar f . A partir de este punto, la palabra “función” se referirá tanto a f como a $f(x)$, según corresponda.



2 Proyectos divertidos usando Sage

A continuación presentamos algunos ejemplos extendidos que usan Sage para explorar problemas no triviales en áreas fuera del álgebra computacional. Presentamos un ejemplo para cada una de varias disciplinas. La idea es que cada uno de estos proyectos típicamente tomaría a un estudiante un fin de semana en completar. Algunos de estos pueden ser explorados usando lo que aprendimos en el capítulo 1, y no requieren ninguna familiaridad adicional con Sage; otros exigirán aprender algo más, lo que se indica claramente en los primeros párrafos de la descripción del proyecto.

Microeconomía: Modelaremos el efecto que tiene el precio de venta en el consumo de helado, en términos de una función de demanda, una de ingresos, una de costos y una de utilidades.

Biología: Modelaremos arterias obstruidas y la Ley de Poiseuille, así como los efectos catastróficos que tienen incluso pequeños porcentajes de obstrucción en el flujo de sangre. No se requiere ningún conocimiento previo de biología.

Optimización industrial: Resolveremos un sistema de inecuaciones lineales para enrutar Taconita de manera óptima desde 7 minas hasta 4 puertos para su embarque, mientras minimizamos costos de envío y respetamos capacidades.

Química: Estudiaremos un método para balancear reacciones químicas bastante complejas, usando la RREF de las matrices.

Física: Resolveremos problemas de movimiento de proyectiles balísticos, incluyendo fuego contra-batería.

Criptología: Usaremos el Método $p - 1$ de Pollard para factorizar enteros, con el objetivo de quebrar el criptosistema RSA.

Matemáticas puras: Usaremos cálculo diferencial para analizar las características de los polinomios de grado cinco, tratando de hallar la ventana óptima de graficación.

Ingeniería eléctrica: Desarrollaremos un miniproyecto sobre la creación de gráficas vectoriales para un campo eléctrico generado por cuatro partículas cargadas en el plano coordenado.

El autor se encuentra actualmente escribiendo un libro para un curso comúnmente conocido como *Matemáticas Finitas* en los Estados Unidos y *Métodos Cuantitativos* en el Reino Unido. Este es un curso para estudiantes que desean especializarse en negocios, contabilidad, finanzas, economía, mercadotecnia, administración, etc. Un tema clásico en ese curso es el método de Análisis de Entrada-Salida de Leontief, que modela cómo los varios sectores de la economía interactúan. Este sería un tema de macroeconomía para complementar el de

microeconomía presentado aquí. Aunque requiere un poco de álgebra matricial, el Análisis de Leontief es adecuado para Sage y relativamente fácil de comprender. Se puede encontrar una versión de ese proyecto en la página web del autor, <https://www.gregorybard.com/>

2.1 Microeconomía: Determinando el precio de venta

Al autor se le dijo en cierta ocasión que el objetivo de un primer curso de microeconomía debería ser doble: primero, enseñar el concepto de *oferta y demanda*; segundo, enseñar que las tres acciones de *maximizar ganancias*, versus *maximizar ingresos*, versus *minimizar costos*, son tres cuestiones totalmente distintas. Esta es una sobresimplificación, por supuesto, y probablemente pretendía ser una broma; sin embargo, resalta la suprema importancia de estos dos temas. Este proyecto consiste en mirar de cerca la cuestión de maximizar utilidades, versus maximizar ingresos, versus minimizar costos. Es un estudio extendido sobre la microeconomía de la venta de helado.

Imaginemos un vendedor con su carrito de helado en una intersección muy transitada en Central Park en Nueva York. El vendedor observa que si cobra \$ 3 por cono de helado, solo vende 120 conos por día; por otro lado, cuando cobra \$ 2 por cono, vende 200 conos diarios. En nuestro ejemplo, dado que hace mucho se han pagado los gastos de arranque del negocio, los costos son de solo 50 centavos por cono, más \$ 100 diarios de cuota por el permiso para vender comida en Central Park. Como podemos imaginar, si su precio de venta es muy alto, venderá pocos helados y no producirá muchas ganancias; si su precio es muy bajo, venderá mucho helados, pero aun así no producirá mucha ganancia.

Nuestro vendedor asumirá que la demanda varía linealmente con respecto al precio. Esto no siempre es correcto, pero frecuentemente constituye una buena aproximación. Existen muchas maneras de determinar que la demanda n y el precio p se relacionan por

$$n = 360 - 80p$$

o, equivalentemente,

$$p = 4,50 - \frac{n}{80}.$$

Hagamos una pausa por un momento y veamos si esto tiene sentido. Primero, introduciendo $n = 200$ y $n = 120$ en la anterior ecuación, obtenemos los precios de \$ 2,00 y \$ 3,00 que esperábamos. Si reemplazamos $p = 2,50$, entonces obtenemos 160, tal como anticiparíamos (ese es el punto medio).

La forma sencilla de obtener estas ecuaciones es usar la ecuación punto-punto de la recta con $(200, 2)$ y $(120, 3)$. (También se puede hallar primero la pendiente determinada por esos puntos, y entonces usar la ecuación punto-pendiente de la recta.) Por supuesto, la forma de la ecuación dependerá de si se asigna p o n para hacer el papel de x . Se encuentran ambos usos en los textos —no existe un “estándar de la industria”—. En este caso, hemos elegido que n haga el papel de x .

Ahora observemos que a un precio de \$ 4,50, no se logrará vender nada del helado. Esto debe resultar creíble, pues si los compradores usualmente esperan pagar alrededor de \$ 2, entonces no podría cobrarse más del doble de esa cantidad. El precio para el cual la demanda es nula es llamado el “máximo precio factible”. Por otro lado, si el precio de un cono es \$ 0, entonces se venderán 360 conos cada día —eso es llamado el “punto de saturación”, y es lo máximo que uno podría esperar vender, la demanda para un precio de \$ 0—. El modelo colapsa para precios por encima de \$ 4,50, pues predice una venta de una cantidad negativa de helados, lo cual carece de sentido. Similarmente, si establecemos una demanda por encima de los 360 conos diarios (el punto de saturación), entonces un precio negativo es requerido, lo que claramente tampoco tiene sentido. Así que nuestro modelo es válido para $0 \leq p \leq 4,50$ dólares o, equivalentemente, $0 \leq n \leq 360$ conos de helado.

Ahora nos gustaría construir las funciones de ingresos, costos y utilidades en términos del número de conos vendidos, n . Claramente, si uno vende n conos a un precio p cada uno, los ingresos son np . Por lo tanto, tenemos

$$I(n) = np = n\left(4,50 - \frac{n}{80}\right) = 4,50n - \frac{n^2}{80}$$

como función de ingresos. Por otro lado, los costos son de 50 centavos por cono, más \$ 100 de cuota por el permiso de venta, haciendo un total de

$$C(n) = 0,50n + 100.$$

Y esto nos da la función de ganancias o utilidad,

$$U(n) = I(n) - C(n) = \frac{-n^2}{80} + 4n - 100,$$

la cual es una parábola.

Tiene sentido que obtengamos algún tipo de gráfica curvada, pues si hacemos el precio muy bajo, la ganancia será muy poca, y si lo hacemos muy alto, la ganancia será nula o, peor, tendríamos pérdidas. En algún punto en medio se encuentra el precio ideal, donde la ganancia es máxima.

Construyamos una tabla y veamos si estas funciones tienen sentido y corresponden con nuestros datos:

Precio [\$/cono]	Venta [conos]	Ingresos [\$]	Costos [\$]	Ganancia [\$]
0,00	360	0,00	280,00	-280,00
0,50	320	160,00	260,00	-100,00
1,00	280	280,00	240,00	40,00
1,50	240	360,00	220,00	140,00
2,00	200	400,00	200,00	200,00
2,50	160	400,00	180,00	220,00
3,00	120	360,00	160,00	200,00
3,50	80	280,00	140,00	140,00
4,00	40	160,00	120,00	40,00
4,50	0	0,00	100,00	-100,00

Es recomendable que el lector elija dos o tres valores de la tabla y verifique las columnas correspondientes antes de continuar adelante.

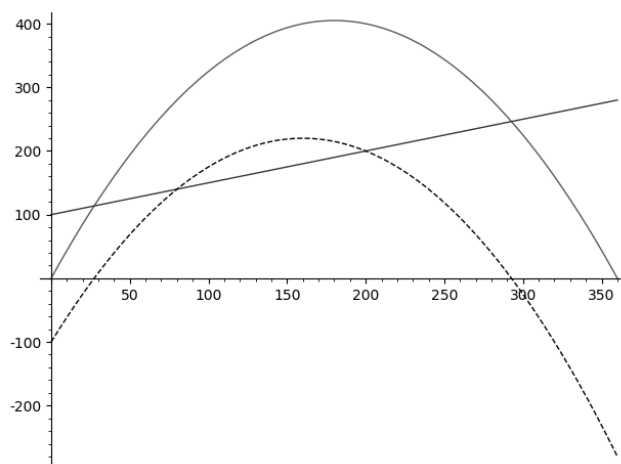


Figura 2.1 La gráfica para el proyecto de microeconomía

En cualquier caso, las gráficas de estas tres funciones están dadas en la figura 2.1. Como podemos ver, la función utilidad es la parábola graficada con línea punteada (en color negro), la función de costo es la recta (en rojo, si esta es la versión a colores del libro) y la función de ingresos es la parábola sólida (en verde, si esta es la versión a colores del libro). Hay muchos hechos que pueden ser deducidos de este gráfica:

- Cuando las funciones de costo e ingreso se intersectan, ese es un punto de equilibrio. En otras palabras, cuando el costo y el ingreso son iguales, no existen ni pérdidas ni ganancias. Aquí tenemos dos puntos de equilibrio, uno justo antes de 30 y otro justo antes de 300. Escribimos

Código de Sage

```
1 find_root((-x^2)/80 + 4.5*x == 0.5*x + 100, 10, 50)
```

y también

Código de Sage

```
1 find_root((-x^2)/80 + 4.5*x == 0.5*x + 100, 250, 350)
```

para encontrar que con —aproximadamente— 27,34 y 292,66 conos, tenemos equilibrio. Entre estos valores hay una ganancia, mientras que por fuera hay pérdidas. Esto parece corresponder con los datos en la tabla.

- La otra definición del punto de equilibrio es que en él la utilidad es cero. Así que podríamos tratar de encontrar las raíces de la función de utilidad, como un medio alternativo para hallar los puntos de equilibrio. Podemos hacerlo escribiendo

Código de Sage

```
1 find_root((-x^2)/80 + 4*x - 100, 10, 50)
```

y también

Código de Sage

```
1 find_root((-x^2)/80 + 4*x - 100, 250, 350)
```

lo que efectivamente nos da los mismos valores que acabamos de obtener (27,34 y 292,66, respectivamente). Evidentemente la curva negra/punteada cruza el eje x ahí.

- La función de costo nunca es cero. Esto se debe a que el vendedor de helado debe pagar \$ 100 diarios por el permiso de venta. Como podemos ver, la recta no cruza el eje x para ningún valor positivo de n .
- La función de ingreso es cero en dos puntos. Primero, cuando no se logra vender helado —esto tiene sentido—. De seguro todos estaremos de acuerdo que cuando no se vende helado, el ingreso es nulo. Dedujimos antes que esto ocurre a un precio de \$ 4,50. Segundo, el ingreso también es nulo cuando el helado es vendido a \$ 0. Claramente, si no se cobra nada por helado, entonces no hay ingresos. Esto ocurre en el punto de saturación, de 360 conos. Para encontrar estos dos puntos en Sage, podemos escribir

Código de Sage

```
1 find_root(4.50*x - x^2/80, 0, 50)
```

y también

Código de Sage

```
1 find_root(4.50*x - x^2/80, 300, 400)
```

lo que efectivamente nos da los valores de 0 y 360.

- Ahora hemos determinado cuándo la utilidad es cero y cuándo el ingreso es cero, y hemos determinado que el costo nunca es nulo. El siguiente paso es minimizar o maximizar estas funciones. Si el lector ya estudió cálculo, entonces conoce cómo optimizar un polinomio. Si no, recordemos el hecho que la parábola $y = ax^2 + bx + c$, con $a \neq 0$, tiene su óptimo en el punto $x = -b/2a$.
- Para la función de ingreso, vemos que el máximo ocurre en 180 conos, a un precio de \$ 2,25 cada uno, y un ingreso de \$ 405,00. Los costos ascenderán a \$ 190,00, con una ganancia de \$ 215,00.

- Para la función de utilidad o ganancia, vemos que el máximo ocurre en 160 conos, a un precio de \$ 2,50 cada uno, y un ingreso de \$ 400,00. Los costos ascenderán a \$ 180,00, con una ganancia de \$ 220,00.
- El punto de mínimos costos sería 0 conos de helado, a un precio de \$ 4,50 cada uno. Sin embargo, en ese caso, el ingreso será de \$ 0,00 y la ganancia será negativa, equivalente a una pérdida de \$ 100,00. Definitivamente este no es buen plan de negocios (aun cuando los costos en este caso sean más bajos que de los anteriores dos casos).
- Notemos que el punto de máxima utilidad tiene un ingreso más bajo (\$ 400 vs. \$ 405), pero una ganancia más alta (\$ 220 vs. \$ 215) que en el punto de máximo ingreso. Notemos también que el punto de máximo ingreso tiene una ganancia más baja (\$ 215 vs. \$ 220), pero un ingreso más alto (\$ 405 vs. \$ 400) que en el punto de máxima ganancia.

En conclusión, vemos que “maximizar ingresos”, “minimizar costos” y “maximizar ganancias” son tres objetivos completamente diferentes. Más aun, deberíamos sugerir que el vendedor venda su helado a \$ 2,50 cada uno —pues maximizar la utilidades es la única manera inteligente de proceder—.

A propósito, el código que produce la imagen con las gráficas de las tres funciones de este proyecto es como sigue:

Código de Sage

```

1  r(x) = 4.50*x - x^2/80
2  c(x) = 0.5*x + 100
3  p(x) = r(x) - c(x)
4
5  P1 = plot(r(x), 0, 360, color='green')
6  P2 = plot(c(x), 0, 360, color='red')
7  P3 = plot(p(x), 0, 360, color='black', linestyle='--')
8  P = P1 + P2 + P3
9
10 P.show()
```

El desafío El desafío para el lector es analizar una nueva situación de negocios. Un libro particular es prevendido por un publicador en dos mercados de prueba por \$ 49,95 y por \$ 39,95. Se venden 80 y 140 copias, respectivamente. En el pasado, las ventas generales de un año fueron 100 veces las ventas de los mercados de prueba. (Para aclarar, esto significa que si el precio se fija en \$ 49,95, los expertos en mercadeo prevén que 8000 copias serán vendidas, y de manera similar, 14 000 copias se venderán a \$ 39,95.)

Se requiere pagar \$ 7 por la impresión de cada libro, más un solo pago global de \$ 4000 por los costos de revisión. El publicador obtiene 50 % de los ingresos de venta. (En caso de curiosidad, usualmente el 20 % de los ingresos de venta va a las librerías locales, 20 % va al distribuidor y el 10 % va a manos del autor.) Se puede asumir que el precio de venta y el número de copias vendidas se relacionan linealmente. Los ítems que deben presentarse para este proyecto son

- (1) una ecuación que relacione el precio de venta y el número de copias vendidas;
- (2) la función de ingresos;
- (3) la función de costos;
- (4) la función de utilidad o ganancia;
- (5) la imagen que muestre las tres funciones anteriores graficadas simultáneamente;
- (6) el (los) punto(s) de equilibrio;
- (7) la n y el p que resultan en ingresos máximos, y
- (8) la n y el p que resultan en ganancias máximas.

2.2 Biología: Arterias obstruidas y la Ley de Poiseuille

Cuando un ser humano muere, la probabilidad que sea debido a una enfermedad cardiovascular es del 31,4 %, y que sea por una de varias formas de cáncer es del 15,8 %.¹ Considerando que las enfermedades cardíacas matan a casi una tercera parte de los habitantes del planeta, las realidades de las arterias obstruidas y el colesterol deben preocuparnos a todos.

El flujo de sangre a través de una arteria está gobernado por la misma ecuación que el flujo de agua en una cañería o el flujo de aceite a través de una manguera. Esto aplica a todos los flujos en movimiento a través de un cilindro, excepto por los flujos turbulentos (donde el fluido se agita y se revuelve sobre sí mismo) y los flujos compresibles (tal como el flujo de aire en una manguera neumática). La ecuación es llamada la Ley de Poiseuille, en honor a Jean-Leonard Marie Poiseuille (1797–1869), cuya disertación² consistió en aplicar este método a la aorta humana.

En este proyecto exploraremos la relación entre algunas de las variables en la Ley de Poiseuille. Exhibiremos tales interdependencias a través de tablas. Para poder completar este proyecto es necesario leer la sección 5.1, que describe cómo crear tablas en Sage.

Los fundamentos La fórmula misma es como sigue:

$$\dot{V} = (\Delta P) \frac{\pi r^4}{8\mu L}$$

y, lastimosamente, no contamos con el tiempo para deducirla aquí. Sin embargo, es altamente recomendable que el lector curioso se refiera al artículo “Blood Vessel Branching: Beyond the Standard Calculus Problem”, por el Prof. John Adam, publicado en la revista *Mathematics Magazine*, Vol. 84, de la Mathematical Association of America, en 2011.

Tomémonos un momento para identificar el significado de cada una de las variables involucradas.

- Representamos por \dot{V} la razón o velocidad del flujo de volumen, medida en unidades tales como centímetros cúbicos por segundo. En una arteria mayor, un flujo típico sería 100 [cm³/segundo].
- El radio de la arteria es r , medido en unidades tales como centímetros. Una arteria mayor suele tener 0,3 [cm] o, aproximadamente, 1/8 de una pulgada de diámetro.
- La longitud de la arteria es L , nuevamente en unidades como centímetros. Una arteria mayor puede tener 20 [cm] u 8 pulgadas de longitud.

Nota: La unidad de medida de presión puede variar. La unidad estándar en el sistema americano de medida es libras por pulgada cuadrada, pues las fuerzas se miden en libras. De manera análoga, en el sistema métrico internacional, dado que las fuerzas se miden en Newtons (N), las unidades de presión son entonces de Newtons por metro cuadrado. Usar estas últimas es curioso porque las arterias no tienen muchos metros cuadrados de área. Otra unidad de presión frecuentemente usada es “atmósferas”, donde la presión atmosférica a nivel del mar es de 1 atmósfera. Eso es 14,6959 libras por pulgada cuadrada o 101,325 [N/m²]. Dado que “Newtons por metro cuadrado” tiene ocho sílabas, y para honrar el trabajo de Blaise Pascal en presión hidrostática, la unidad métrica de 1 [Pascal] es definida como 1 [N/m²]. Por lo tanto, 1 atmósfera= 101,325 kiloPascuales, que puede ser escrito como 101,325 [kPa].

- La Medicina ha estado presente por mucho tiempo. Por razones históricas, es usual medir la presión sanguínea en una unidad arcaica llamada “milímetros de mercurio”. Cuando acudimos al doctor y nos dice que nuestra presión es de “120 sobre 80”, el primer número significa “120 milímetros de mercurio”, escrito 120 [mmHg]. Simplemente lo aceptaremos como una unidad de medida y recordaremos que un valor de 120 es considerado “normal”. Con un valor de 145 [mmHg] para el primer número (llamado presión sistólica), uno sería diagnosticado con “hipertensión”. Un valor de 180 [mmHg] para la sistólica podría resultar en daño de órganos, y es declarado como “emergencia hipertensiva”.

¹Con base en datos de 2016 de la Organización Mundial de la Salud (https://www.who.int/healthinfo/global_burden_disease/GHE2016_Deaths_Global_2000_2016.xls).

²Poiseuille, J-L. M., *Recherches sur la force du coeur aortique*, D.Sc., École Polytechnique, Paris, France. 1828.

- La viscosidad es un concepto físico del que puede o no que hayamos escuchado antes, y es denotada por μ en esta sección, pero en ocasiones η puede usarse en su lugar. Representa que tan “pegajoso” o “grueso” es un fluido. La unidad más común tiene tres nombres:

$$1 \text{ centiPoise} = 1 \text{ miliPascal segundo} = 0,001 \text{ Newton segundo} / \text{metro}^2$$

Por supuesto, entenderemos mejor la viscosidad si discutimos algunos ejemplos específicos de fluidos que conocemos de la vida diaria. Estos están listados abajo, y están medidos a temperatura ambiente ($20^\circ\text{C} = 68^\circ\text{F}$).

- La viscosidad del agua es 1,002 centiPoise.
- La viscosidad de la leche es aproximadamente 3 centiPoise. (Este fluido es más grueso y algo más pegajoso, comparado con el agua.)
- La viscosidad del aceite de maíz es aproximadamente 60 centiPoise. (Este fluido es mucho más grueso y pegajoso que el agua o la leche.)
- La viscosidad del aceite de oliva es aproximadamente 74 centiPoise. (Este fluido es ligeramente más grueso y pegajoso que el aceite de maíz.)
- La viscosidad de la miel varía, estando en el rango 20–100 Poise, que significa 2000 a 10 000 centiPoise. (La miel es extremadamente gruesa y pegajosa..)
- La viscosidad de la salsa ketchup a menudo se cita como 50 Poise, que significa 50 000 centiPoise. (Cualquiera que haya derramado ketchup de una botella sabe que esta fluye bastante lentamente y es mucho más gruesa que el agua, la leche o el aceite.)

Muchas publicaciones médicas citan la viscosidad de la sangre entera en el rango 3–4 centiPoise dentro del cuerpo humano ($37^\circ\text{C} = 98,6^\circ\text{F}$). Con esto en mente, usaremos 3,5 centiPoise. (Esto significa que la sangre en el cuerpo humano es más gruesa que la leche, pero no tanto como el aceite, la miel o la salsa ketchup.) No podemos usar el centiPoise como nuestra unidad para la viscosidad, pues estamos usando una medida anticuada de presión. Para nosotros,

$$\mu = 2,62521 \dots \times 10^{-5}$$

es solo una constante en la fórmula. La unidad de viscosidad resulta ser “segundo mmHg”, lo que es extremadamente extraño, pero es lo que hará que las unidades correspondan en la fórmula.

El desafío Para este proyecto, vamos a crear algunas tablas que pueden ser usadas para ayudar a un médico a entender las realidades numéricas de la relación de grado cuatro en la Ley de Poiseuille, entre el radio arterial y la razón o velocidad de flujo de sangre.

Las tablas permitirán cambiar algunas variables, mientras que la mayoría de ellas permanecerán fijas en “valores estándares”, que han sido escogidos para ser bastante típicos. Estos valores estándares para nuestras variables en este proyecto son $\mu = 2,62521 \times 10^{-5}$ [mmHg·seg] para la viscosidad, $L = 20$ [cm] para la longitud de la arteria, $r = 1,2$ [cm] para el radio arterial, $\dot{V} = 100$ [cm³/segundo] y finalmente ΔP es 0,0645 [mmHg].

- (1) La primera tabla debe mantener los valores estándares para μ , L y ΔP , y calcular \dot{V} con base en el valor de r , usando la Ley de Poiseuille. El valor de r debe ser determinado sobre el porcentaje de obstrucción, desde 0 % hasta 50 %, en incrementos de 2 %. La primera columna de la tabla mostrará el porcentaje de obstrucción, la segunda debe mostrar el diámetro efectivo en [cm] y la tercera debe ser la razón de flujo sanguíneo \dot{V} , en [cc/seg].
- (2) La segunda tabla debe mantener los valores estándares de μ , L y \dot{V} . Nuevamente, r será determinado con base en el porcentaje de obstrucción, desde 0 % hasta 50 %, con incrementos de 2 %. En esta ocasión, se calculará el ΔP requerido para alcanzar una velocidad de flujo de 100 [cm³/segundo] para el bloqueo dado (el ΔP para \dot{V} dado), usando la Ley de Poiseuille. La primera columna debe mostrar el porcentaje de obstrucción, la segunda debe ser el ΔP correspondiente y la tercera columna debe representar ΔP como el “porcentaje de normalidad”. (En otras palabras, en la tercera columna, 3/2 el ΔP normal debe reportarse como un 150 %.)

- (3) La tercera tabla debe mantener los valores estándares para μ , L y ΔP . La primera columna mostrará el “porcentaje de flujo normal”, desde un 100 % hasta un 25 %, con decrementos de 3 %. La segunda columna debe ser la velocidad de flujo, en centímetros cúbicos por segundo, identificado con esos porcentajes. La tercera columna debe ser el radio que implica la Ley de Poiseuille. La cuarta columna mostrará el porcentaje de obstrucción. (En otras palabras, 0,15 [cm] de radio es un 50 % de bloqueo en la cuarta columna, pues $0,15/0,3 = 0,5$.)
- (4) En todas las columnas de las tablas, se deben tener cuatro dígitos de precisión. **Sugerencia:** Si el lector ha olvidado cómo hacer eso, vea la discusión acerca de la opción `digits` del comando `N()` bajo el título “Las constantes especiales π y e ” en la página 4.

Podemos revisar nuestro trabajo introduciendo los valores necesarios en la ecuación original, pero también puede ser útil comparar las tablas obtenidas con los siguientes resultados seleccionados.

Tabla 1 Resultados seleccionados para la primera parte del proyecto biología

Bloqueo [%]	Diámetro efectivo [cm]	Razón de flujo [cc/seg]
10,00	2,160	65,63
22,00	1,872	37,03
48,00	1,248	7,314

Tabla 2 Resultados seleccionados para la segunda parte del proyecto biología

Bloqueo [%]	ΔP [mmHg]	Normalidad de ΔP [%]
4,000	0,075 91	117,7
18,00	0,1426	221,1
40,00	0,4975	771,3

Tabla 3 Resultados seleccionados para la tercera parte del proyecto biología

Flujo [%]	Velocidad de flujo [cc/seg]	Radio [cm]	Bloqueo [%]
94,00	94,00	1,181	1,544
37,00	37,00	0,9358	22,01
25,00	25,00	0,8485	29,30

2.3 Optimización industrial: Transportando taconita

En este proyecto resolveremos un muy simple pero realista problema de rutas de envío, usando Programación Lineal. Este proyecto asume que se ha leído la sección 4.21 en la página 179, acerca de cómo trabajar con programas lineales en Sage.

En particular, un conglomerado minero tiene siete minas alrededor de Minnesota (MN) y Wisconsin (WI), y tienen instalaciones para embarque en Two Harbors, Green Bay, Minneapolis, más un complejo de embarque en los puertos gemelos de Duluth, MN, y Superior, WI. Debemos determinar cuánta taconita debe ser enviada desde cada mina a cada una de las cuatro instalaciones de embarque. Sin embargo, tenemos varias restricciones que debemos cumplir.

Primero, debe cumplirse que la cantidad total de taconita que parte de cada mina (hacia los cuatro destinos) debe ser exactamente la cantidad producida esa semana. No debe ser mayor —sino estaríamos ordenando a las minas más taconita de la que pueden producir—; no debe ser menor —sino, tendríamos acumulación de taconita que no puede ser almacenada en el lugar—. La producción total de cada mina se muestra en la última columna de la tabla 4.

	Costos de envío				
	Duluth, MN Superior, WI	Two Harbors, Minnesota	Green Bay, Wisconsin	Minneapolis, Minnesota	Producción semanal
Mina 1	\$ 18/ton	\$ 30/ton	\$ 74/ton	\$ 35/ton	8000 tons
Mina 2	\$ 73/ton	\$ 21/ton	\$ 50/ton	\$ 65/ton	13 000 tons
Mina 3	\$ 37/ton	\$ 93/ton	\$ 67/ton	\$ 61/ton	7000 tons
Mina 4	\$ 24/ton	\$ 95/ton	\$ 63/ton	\$ 35/ton	16 000 tons
Mina 5	\$ 38/ton	\$ 77/ton	\$ 80/ton	\$ 71/ton	13 000 tons
Mina 6	\$ 73/ton	\$ 15/ton	\$ 39/ton	\$ 68/ton	8000 tons
Mina 7	\$ 48/ton	\$ 51/ton	\$ 69/ton	\$ 14/ton	9000 tons

Tabla 4 Costos de envío y producción minera

Segundo, la demanda y la capacidad de cada estación receptora deben ser respetadas. La instalación de Two Harbors, WI, puede recibir y embarcar hasta 18 000 toneladas; la instalación de Green Bay puede recibir y embarcar hasta 25 000 toneladas; Duluth/Superior puede recibir y embarcar entre 15 000 y 45 000 toneladas, donde el mínimo asegura que existan suficientes ingresos para pagar las cuentas de esta costosa ubicación; finalmente, Minneapolis puede recibir y embarcar entre 10 000 y 30 000 toneladas.

Tercero, el cronograma de envío debe cumplir la tarea de transportar la taconita hasta las estaciones de embarque por un costo mínimo. El precio del transporte desde cada mina hasta cada estación receptora está dado en la tabla 4.

El objetivo es modelar el problema con un programa lineal, transcribirlo a código de Sage y resolverlo. Como ayuda, podemos ofrecer una la siguiente sugerencia: Sean d_1, d_2, \dots, d_7 las cantidades enviadas a Duluth/Superior desde las siete minas; sean t_1, t_2, \dots, t_7 las cantidades enviadas a Two Harbors; g_1, g_2, \dots, g_7 las cantidades enviadas a Green Bay, y m_1, m_2, \dots, m_7 las cantidades enviadas a Minneapolis.

Como el problema requiere una gran cantidad de trabajo, no hay perjuicio en dar la respuesta final. Esta es la salida del programa del autor:

```
El costo mínimo de envío es
1883000,00
Las cantidades enviadas a Duluth/Superior desde las siete minas son
{1: 8000.0, 2: 0.0, 3: 7000.0, 4: 15000.0, 5: 13000.0, 6: 0.0, 7: 0.0}
Las cantidades enviadas a Two Harbors desde las siete minas son
{1: 0.0, 2: 13000.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 5000.0, 7: 0.0}
Las cantidades enviadas a Green Bay desde las siete minas son
{1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 3000.0, 7: 0.0}
Las cantidades enviadas a Minneapolis desde las siete minas son
{1: 0.0, 2: 0.0, 3: 0.0, 4: 1000.0, 5: 0.0, 6: 0.0, 7: 9000.0}
```

A propósito, si el lector piensa que este es un problema complicado, por favor tenga en cuenta lo siguiente. Un problema realista puede incluir muchos puntos intermedios de recolección, muchas docenas de minas, y capacidades restringidas en todos los caminos entre estos puntos. Es fácil imaginar un problema con 30 minas, 12 puntos intermedios de recolección y 5 puertos, haciendo un total de $360 + 60 = 420$ rutas. Cada ruta tiene un costo asociado y una restricción de capacidad. Cada una de estas últimas es, en sí misma, una desigualdad. Por lo tanto, este tipo de problemas “del mundo real” pueden llegar a ser gigantescos.

2.4 Química: Balanceando reacciones con matrices

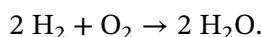
En la clase de química hay muchas cosas divertidas para hacer y aprender. Una de estas es balancear una ecuación química, lo cual es frecuentemente fácil —sin embargo, algunas reacciones son bastante difíciles (para los que no somos químicos)—. Aquí aprenderemos a balancear rápida y fácilmente hasta los casos más desagradables usando álgebra matricial.

Todos los ejemplos a lo largo de este proyecto, así como la idea del proyecto mismo, se deben enteramente al Prof. Gergely Sirokman, del Wentworth Institute of Technology en Boston, Massachusetts. El autor está agradecido con él por el tiempo dedicado a diseñar estos ejemplos para el presente libro.

Este proyecto requiere conocimiento de cómo trabajar sistemas de ecuaciones lineales con infinitas soluciones. El apéndice D provee cobertura completa de ese tema, pero algunos lectores ya conocen sobre ello y no necesitarán leer esa referencia.

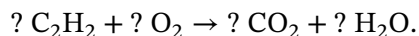
2.4.1 Los fundamentos

Iniciemos con un ejemplo energético. Tiempo atrás, en la época de los dirigibles, las versiones norteamericanas de este transporte eran frecuentemente llenadas de helio. Sin embargo, Alemania no tenía acceso a grandes suministros de este gas, así que usaban hidrógeno en su lugar. Esto es desafortunado porque el hidrógeno puede explotar fácilmente en presencia de una chispa (y oxígeno). El Hindenburg fue un dirigible construido por la compañía Zeppelin, que explotó en una bola de fuego en 1937 cerca de Lakehurst, New Jersey. El hidrógeno se combina con el oxígeno del aire y forma vapor de agua. Aquí está la fórmula química para tal reacción:



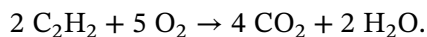
Recordemos lo que significa que una ecuación química esté balanceada. Si tenemos dos moléculas de H_2 y una molécula de O_2 antes de la reacción, entonces contamos con 4 átomos de hidrógeno y 2 átomos de oxígeno. Por lo tanto, si después de la reacción tenemos dos moléculas de H_2O , entonces también contamos con 4 átomos de hidrógeno y 2 átomos de oxígeno. Ningún átomo de hidrógeno ni de oxígeno fue creado o destruido durante la reacción química, pues tenemos las mismas cantidades antes y después. Por otro lado, dado que los átomos nunca son creados o destruidos en reacciones químicas, tendríamos que rechazar una ecuación como desbalanceada si las cantidades antes y después no coincidieran. Estas deben coincidir para todos y cada uno de los átomos involucrados en la reacción —si tan solo un par antes-después no coincide, de seguro cometimos un error de cálculo—.

La quema de acetileno en un soplete es una reacción química en la cual el acetileno (C_2H_2) y el oxígeno (O_2), cada uno almacenado en un cilindro de gas comprimido separado, se combinan para formar dióxido de carbono (CO_2) y agua (H_2O). Con base en estas fórmulas químicas, debemos encontrar los valores enteros positivos para los siguientes cuatro signos de interrogación, de manera que la ecuación esté balanceada:



Tomémonos un momento para intentarlo ahora, con un poco de ingenio, y algo de prueba y error a la antigua. Intente el lector encontrar valores adecuados para los símbolos “?”. (Lo decimos en serio, hay que intentarlo; es un ejercicio revelador que ayudará a entender el mecanismo de este proyecto.)

Ahora bien, hasta este punto, mediante un intento serio, seguramente se encontraron los valores para balancear la ecuación:



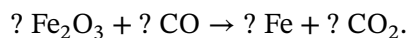
De seguro que esto fue un poco difícil (al menos para el autor lo fue); tal vez fue más fácil para el lector. Con ecuaciones químicas más complejas, el proceso de prueba y error puede tardar mucho más.

Desconocido para muchos estudiantes de química, existe un método para balancear ecuaciones químicas usando matrices, que no involucra adivinar valores.

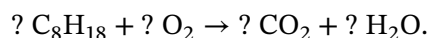
2.4.2 Cinco ejemplos interesantes

A continuación tenemos cinco reacciones químicas interesantes que nos servirán como ejemplos.

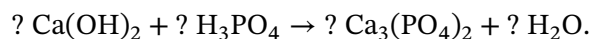
- (1) Esta reacción aparece en la fundición de óxido ferroso (el mineral de hierro) para extraer el hierro.



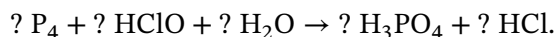
- (2) En la gasolina para auto, el octano es uno de los químicos importantes. Esta es la reacción cuando un auto quema el octano en abundancia de oxígeno. Si aumentamos demasiado las revoluciones, pueden ocurrir otras reacciones químicas que resultan en CO o incluso en C.



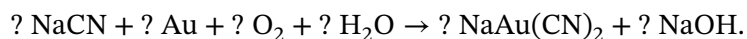
- (3) Aquí tenemos un ejemplo de una reacción ácido-base, un tipo muy importante de reacción en química. El hidróxido de calcio ($\text{Ca}(\text{OH})_2$) tiene numerosos usos, incluyendo el tratamiento de aguas residuales. El ácido fosfórico (H_3PO_4) puede estar presente en las aguas residuales humanas de fuentes tan comunes como las bebidas gaseosas. Aquí vemos que el hidróxido de calcio reaccionará con el ácido fosfórico y lo convertirá en agua ordinaria y fosfato tricálcico. Resulta que este último es mucho más benigno, pues ocurre naturalmente en los huesos de animales y en la leche vacuna.



- (4) En el ejemplo anterior, indicamos que el ácido fosfórico (H_3PO_4) está presente en las bebidas gaseosas. ¿De dónde proviene? Puede ser fácilmente fabricado de fósforo elemental puro y ácido hipocloroso (HClO) en presencia de agua. El ácido clorhídrico se forma como un producto residual. Aquí está la reacción:

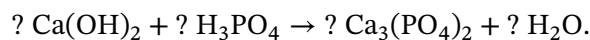


- (5) Este es el método preferido para la extracción del oro de muestras diluidas de rocas. Es tremendamente tóxico (NaCN es cianuro de sodio...¡y es tan malo como suena!).



2.4.3 La forma lenta: derivando el método

Ahora derivaremos lentamente el método matricial para balancear una ecuación química. Un enfoque abreviado será presentado en la siguiente subsección, pero no tendrá sentido a menos que se lea muy cuidadosamente esta deducción. Tomaremos como base el tercer ejemplo de la lista anterior:



Si tuviésemos x_1 moléculas de $\text{Ca}(\text{OH})_2$, x_2 moléculas de H_3PO_4 , x_3 moléculas de $\text{Ca}_3(\text{PO}_4)_2$ y x_4 moléculas de H_2O , entonces tendríamos

- x_1 átomos de calcio a la izquierda y $3x_3$ a la derecha;
- $2x_1 + 4x_2$ átomos de oxígeno a la izquierda y $8x_3 + x_4$ a la derecha;
- $2x_1 + 3x_2$ átomos de hidrógeno a la izquierda y $2x_4$ a la derecha, y
- x_2 átomos de fósforo a la izquierda y $2x_3$ a la derecha.

Dado que, para cada elemento, requerimos el mismo número de átomos a la izquierda y a la derecha, entonces tenemos las siguientes cuatro ecuaciones con cuatro variables:

$$\begin{aligned} x_1 &= 3x_3 \\ 2x_1 + 4x_2 &= 8x_3 + x_4 \\ 2x_1 + 3x_2 &= 2x_4 \\ x_2 &= 2x_3 \end{aligned}$$

El equivalente matricial de este sistema es

$$A = \left[\begin{array}{cccc|c} 1 & 0 & -3 & 0 & 0 \\ 2 & 4 & -8 & -1 & 0 \\ 2 & 3 & 0 & -2 & 0 \\ 0 & 1 & -2 & 0 & 0 \end{array} \right].$$

Usando Sage, calculamos la RREF de esta matriz, lo que resulta en

$$\begin{array}{l} [\quad 1 \quad 0 \quad 0 \quad -1/2 \quad 0] \\ [\quad 0 \quad 1 \quad 0 \quad -1/3 \quad 0] \\ [\quad 0 \quad 0 \quad 1 \quad -1/6 \quad 0] \\ [\quad 0 \quad 0 \quad 0 \quad 0 \quad 0] \end{array}$$

Como podemos ver, tenemos una fila nula (todas sus entradas son ceros). Hay un pivote para x_1 , x_2 y x_3 , así que esas variables están determinadas. Pero no hay pivote para x_4 , así que esta es libre. La “forma estándar” del conjunto infinito de soluciones se escribe como sigue:

$$\begin{aligned} x_1 &= (1/2)x_4 \\ x_2 &= (1/3)x_4 \\ x_3 &= (1/6)x_4 \\ x_4 &= x_4 \end{aligned}$$

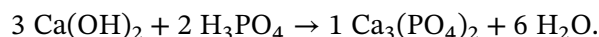
Sin embargo, es costumbre solo considerar valores enteros para los coeficientes de la ecuación de una reacción química. ¿Qué hacemos cuando deseamos solamente este tipo de soluciones? Es un truco muy poco común, que no se enseña frecuentemente, pero es fácil de aprender. Primero tomamos el mínimo común múltiplo (que denotaremos como lcm^3) de todos los denominadores de las tres fracciones. En esta caso, el lcm de $\{2, 3, 6\}$ es 6. Segundo, reemplazamos $x_4 = 6n$, donde n es una nueva variable. Ahora tenemos

$$\begin{aligned} x_1 &= (1/2)6n = 3n \\ x_2 &= (1/3)6n = 2n \\ x_3 &= (1/6)6n = n \\ x_4 &= 6n \end{aligned}$$

Para cualquier n entero los cuatro valores que obtendríamos forman una solución entera para las ecuaciones dadas. Esto incluye valores carentes de sentido en el contexto actual, como $n = 0$ o $n < 0$. (O para ser precisos, los casos donde $n \leq 0$ son soluciones matemáticamente válidas para el sistema, pero carecen de significado químico.) Queremos la respuesta más sencilla de valores estrictamente positivos, así que reemplazamos $n = 1$. Ahora tenemos

$$x_1 = 3, \quad x_2 = 2, \quad x_3 = 1, \quad x_4 = 6,$$

dándonos la reacción química final



Pero debemos hacer notar que los químicos usualmente no escriben “1”, como hicimos con la tercera molécula.

Ahora deberíamos revisar nuestro trabajo. Como podemos ver de esta última ecuación química,

- tenemos $3(1) + 0 = 3$ átomos de calcio a la izquierda y $1(3) + 0 = 3$ a la derecha;
- tenemos $3(2) + 2(4) = 14$ átomos de oxígeno a la izquierda y $1(8) + 6(1) = 14$ a la derecha;
- tenemos $3(2) + 2(3) = 12$ átomos de hidrógeno a la izquierda y $0 + 6(2) = 12$ a la derecha;
- tenemos $0 + 2(1) = 2$ átomos de fósforo a la izquierda y $1(2) + 0 = 2$ a la derecha.

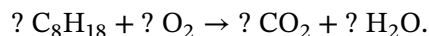
¡Todos los valores coinciden, así que declaramos victoria!

³Least common divisor. Sage tiene una función llamada `lcm` para este propósito.

2.4.4 Balanceando de la manera rápida

Este fue un excelente, pero muy lento ejercicio. Ahora queremos balancear ecuaciones químicas más rápido. La clave es recordar que “los átomos son filas y las moléculas son columnas”. Un truco mnemotécnico para no olvidarlo es observar que la palabra “átomo” es más corta que “molécula”, y similarmente la palabra “fila” es más corta que “columna”. Los átomos pueden codificarse en cualquier orden, pero es recomendable mantener el orden de las moléculas como se observa en la fórmula de la reacción.

Intentemos con el segundo de nuestros ejemplos, la combustión del octano:



Procedamos un átomo a la vez, preguntando para cada molécula cuántos átomos de cada tipo particular tiene. Para los reactivos (al lado izquierdo de la flecha), usaremos enteros positivos; para los productos (al lado derecho de la flecha), usaremos enteros negativos.

Carbono: Las cuatro moléculas tiene 8, 0, 1 y 0 átomos de carbono, respectivamente, así que escribimos 8, 0, -1, 0.

Hidrógeno: Las cuatro moléculas tienen 18, 0, 0 y 2 átomos de hidrógeno, respectivamente, así que escribimos 18, 0, 0, -2.

Oxígeno: Las cuatro moléculas tienen 0, 2, 2 y 1 átomos de oxígeno, respectivamente, así que escribimos 0, 2, -2, -1.

Ahora definimos la matriz asociada y pedimos su RREF, en la notación que aprendimos en la página 23. No olvidemos que añadimos una columna de ceros a la derecha de la matriz para representar la parte “= 0” de cada ecuación.

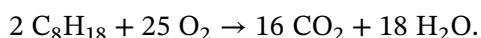
Código de Sage

```
1 B = matrix([[8,0,-1,0,0], [18,0,0,-2,0], [0,2,-2,-1,0]])
2
3 print('Problema:')
4 print(B)
5 print('Respuesta:')
6 print(B.rref())
```

Así tenemos la salida

```
Problema:
[ 8  0 -1  0  0]
[18  0  0 -2  0]
[ 0  2 -2 -1  0]
Respuesta:
[  1  0  0  -1/9  0]
[  0  1  0 -25/18 0]
[  0  0  1  -8/9  0]
```

Rápidamente verificamos que la matriz que ingresamos efectivamente era la que deseábamos ingresar. A continuación, tomamos el lcm de los denominadores en la columna interesante (la cuarta en este caso). El lcm de {9, 18, 9} es claramente 18. Para obtener los primeros tres coeficientes de la ecuación química, multiplicamos la columna interesante por el lcm negativo (es decir, por -18) y obtenemos 2, 25 y 16. El cuarto coeficiente es simplemente el lcm mismo, o sea 18. Terminamos con



Pero aún debemos verificarlo:

- Tenemos $2(8) + 0 = 16$ átomos de carbono a la izquierda y $16(1) + 0 = 16$ a la derecha.
- Tenemos $2(18) + 0 = 36$ átomos de hidrógeno a la izquierda y $0 + 18(2) = 36$ a la derecha.
- Tenemos $0 + 25(2) = 50$ átomos de oxígeno a la izquierda y $16(2) + 18 = 50$ a la derecha.

¡Todos los valores coinciden, así que declaramos victoria!

Por último, pero no menos importante, sería bueno notar que los químicos cuentan con atajos que usan para simplificar la cuestión significativamente, pero que requieren intuición química. Por ejemplo, PO_4 y OH son iones poliatómicos, lo que significa que pueden “fingir” ser átomos individuales. Esto permite remover una variable del sistema de ecuaciones lineales, lo que es significativo.

El desafío Ahora, intente el lector el primer, cuarto y quinto ejemplos por sí solo. ¡Buena suerte y no olvide revisar su trabajo!

2.5 Física: Projectiles balísticos

Los problemas de movimiento de proyectiles balísticos son el “pan de cada día” en clases de cálculo y física. Aquí asumimos que el lector tiene conocimientos de Cálculo I y Física I, o los equivalentes de colegio. Vamos a explorar dos problemas relativamente sencillos —para los cuales no se requiere usar Sage— y otro adicional, que es un poco más complicado —para el cual se requiere de Sage u otra herramienta computacional—.

Para un tipo amplio de problemas simples e intermedios, la siguiente técnica de siete pasos resuelve completamente la cuestión planteada.

- (1) Leemos el problema muy cuidadosamente y escribimos una ecuación de suma de fuerzas, frecuentemente con la ayuda de un diagrama de cuerpo libre. Esto, a su vez, produce una función de aceleración, usando la ecuación $\sum F = ma$.
- (2) Integramos la función de aceleración para obtener la función de velocidad, la que desafortunadamente tendrá una $+C$.
- (3) Usamos información del problema para calcular el valor de C .
- (4) Integramos la función de velocidad para obtener la función de altitud, la que desafortunadamente tendrá una $+C$ (muy probablemente diferente a la anterior).
- (5) Nuevamente, usamos información del problema para calcular el valor de la nueva C .
- (6) Verificamos que las tres funciones son consistentes con todos los datos dados (reemplazando valores para las variables) y unas con otras (calculando derivadas).

Nota: En este punto tenemos las tres funciones de manera explícita y podemos responder casi cualquier cuestión sobre el proyectil.

- (7) Volvemos a leer la pregunta para saber qué es lo que se pide específicamente.

A lo largo de este proyecto, $y(t)$ representará la altitud de un proyectil, $y'(t) = v(t)$ su velocidad y $y''(t) = v'(t) = a(t)$ su aceleración. Ignoraremos la resistencia del aire en todos los casos. Debido a los riesgos del error de redondeo, es desafortunadamente necesario que usemos $g = 9,80665 \text{ [m/s}^2\text{]}$ a lo largo del proyecto —realmente necesitamos las seis cifras significativas—.

Antes de empezar, debemos mencionar que existen tres enfoques para lidiar con los cálculos en este tipo de problemas. Típicamente, los matemáticos aplicados usan varias cifras de precisión durante la resolución, y solo redondean al final, reflejando la incertidumbre en los datos de entrada. Sin embargo, ellos usualmente no manejan las unidades, ligadas a los coeficientes y constantes, dentro de las ecuaciones. En cambio, los físicos sí manejan las unidades dentro de las ecuaciones (!). Por otro lado, en los cursos universitarios de primer año, algunos textos redondean los números en cada paso del procedimiento, reflejando la precisión limitada de los datos de entrada. Dado que el autor está acostumbrado a la técnica de los matemáticos aplicados, la usaremos aquí.

Podemos realizar los siguientes dos ejemplos de calentamiento con papel y lápiz, y una calculadora científica. Luego procederemos a resolver un problema que requiere el uso de Sage.

2.5.1 Un primer ejemplo de trayectorias balísticas

Supongamos que una pieza de artillería se encuentra en una colina, de tal manera que la boca del cañón está a 25 [m] por encima del campo. Este dispara sus proyectiles con una velocidad inicial de 300 [m/s], a un ángulo de 60° sobre la horizontal. ¿Qué tan lejos puede ir el proyectil?

Primer paso. Debemos construir una función de aceleración. La única fuerza actuando en el proyectil (en pleno vuelo) es su peso debido a la gravedad de la Tierra. Tenemos entonces,

$$\begin{aligned}\sum F &= ma(t) \\ -mg &= ma(t) \\ -g &= a(t) \\ a(t) &= -9,80665.\end{aligned}$$

Segundo paso. Ahora integramos para encontrar $v(t)$ con una $+C$:

$$\begin{aligned}\int a(t) dt &= \int a(t) dt \\ \int v'(t) dt &= \int -9,80665 dt \\ v(t) &= -9,80665t + C.\end{aligned}$$

Tercer paso. Sabemos que $v(0)$ es la componente vertical de la velocidad de disparo. Esa la podemos calcular como

$$(300)(\sin 60^\circ) = (300)(0,866025) = 259,807$$

y, por lo tanto,

$$\begin{aligned}v(t) &= -9,80665t + C \\ v(0) &= -9,80665(0) + C \\ 259,807 &= C.\end{aligned}$$

Ahora sabemos que $v(t) = -9,80665t + 259,807$, al menos durante el vuelo.

Cuarto paso. Integramos para encontrar $y(t)$ con una $+C$:

$$\begin{aligned}\int v(t) dt &= \int v(t) dt \\ \int y'(t) dt &= \int (-9,80665t + 259,807) dt \\ y(t) &= -\frac{1}{2}(9,80665)t^2 + 259,807t + C \\ y(t) &= -4,90332t^2 + 259,807t + C.\end{aligned}$$

Quinto paso. Usamos el hecho que $y(0)$ es la altitud de la colina, o 25 [m]:

$$\begin{aligned}y(t) &= -4,90332t^2 + 259,807t + C \\ y(0) &= -4,90332(0)^2 + 259,807(0) + C \\ 25 &= C\end{aligned}$$

Sexto paso. Finalmente tenemos

$$y(t) = -4,90332t^2 + 259,807t + 25$$

y podemos empezar la parte más importante del problema: revisar nuestro trabajo.

- Podemos ver que $y'(t) = -9,80664t + 259,807$, lo que coincide con nuestra $v(t)$, excepto tal vez por error de redondeo.
- Podemos ver que $y''(t) = -9,80664$, lo que coincide con nuestra $a(t)$, excepto tal vez por error de redondeo.
- Tenemos que $y'(0) = v(0) = 259,807$, como se deseaba. (Esta es la componente vertical de la velocidad inicial.)
- Tenemos que $y(0) = 25$, como se deseaba. (Esta es la altitud de la boca del cañón.)

Séptimo paso. Ahora podemos determinar lo que el problema de hecho nos solicita. Pide calcular el rango del proyectil (qué tan lejos viaja), de manera que hallamos el tiempo de vuelo y lo multiplicamos por la velocidad horizontal.

Para encontrar el tiempo de vuelo, simplemente debemos saber en qué momento el proyectil golpea el suelo (cuándo termina el vuelo), pues el vuelo inicia cuando $t = 0$. Por supuesto, al golpear el suelo, la altitud es cero. Por lo tanto reemplazamos $y(t) = 0$ y usamos la fórmula cuadrática.

$$\begin{aligned} 0 &= y(t) \\ 0 &= -4,90332t^2 + 259,807t + 25 \\ t &= \frac{-259,807 \pm \sqrt{(259,807)^2 - 4(-4,90332)(25)}}{2(-4,90332)} \\ t &= -0,0960512 \text{ o } 53,0820 \end{aligned}$$

Podemos verificarlo calculando $y(53,0820) = 0,0174218$ [m]; una diferencia de 17 [mm], lo que no constituye un problema, sino que obviamente es un mero error de redondeo.

A continuación, necesitamos la velocidad horizontal

$$(300)(\cos 60^\circ) = (300)(1/2) = 150,$$

y finalmente, tenemos el rango del proyectil

$$(150)(53,0820) = 7962,30.$$

Un rango de 7962,30 [m], o casi ocho kilómetros, parece plausible. Después de todo, el propósito de la artillería es atacar objetivos a varios kilómetros de distancia. Debemos notar, sin embargo, que la altitud de nuestra colina de 25 [m] es una medida probablemente precisa al metro o medio metro más próximo, y no a seis cifras significativas; la velocidad de disparo probablemente es solo conocida al 1 % más próximo. Por lo tanto, lo mejor es dar una respuesta a dos cifras significativas y reportar que el rango del proyectil es 7900–8000 metros o 7,9–8,0 kilómetros.

2.5.2 Un segundo ejemplo de trayectorias balísticas

Ahora exploraremos un problema de defensa aérea. Vamos a simular la computadora de un sistema de radar de defensa aérea que está rastreando misiles entrantes. Este no es un problema irreal. Durante la Primera Guerra del Golfo (1991), Iraq disparó misiles SS-1 “Scud” a lo largo de grandes distancias a Israel y Arabia Saudita. Algunos pudieron ser interceptados durante vuelo por misiles tierra-aire, pero había muchos Scuds entrantes. Es útil calcular el punto de impacto —después de todo, si un misil se dirige hacia el desierto vacío, entonces no es problema, pero si se dirige hacia un centro comercial, esa es una emergencia—. Al momento en que la versión en inglés de este libro era enviado al publicador en julio de 2014, cálculos similares estaban siendo realizados por el sistema de defensa aéreo israelí “Iron Dome” (“Domo de Hierro”), así que podemos estar seguros que estas técnicas son bastante relevantes.

Para mantener simples los números y el modelo, trabajaremos con un problema de proyectiles de artillería en lugar de misiles balísticos regionales. Un proyectil ha sido detectado en la vecindad de una batería de defensa aérea sobre el cual nuestra computadora tiene responsabilidad. Esta ha determinado su sistema de coordenadas, de manera que la ubicación del radar es $(0, 0, 0)$. Se ha tomado una instantánea del proyectil entrante en pantalla, indicando su posición y velocidad. La dirección y es directamente hacia arriba, la dirección x es hacia el este y la dirección z es hacia el sur. El radar ha reportado que el proyectil está en la posición $(12\,000, 5000, 7000)$ metros, con una velocidad de $(-60, -240, 80)$ [m/s]. La localización del impacto debe ser calculada primero. Por simplicidad, sea $t = 0$ el momento de la instantánea del radar, en lugar del lanzamiento.

Primero, usaremos el método de los siete pasos para encontrar $y(t)$, la función de altitud. Dejaremos este ejercicio al lector, de manera que pueda practicar. Sin embargo, a continuación mostramos las respuestas que debería obtener después de cada paso, para que pueda revisar su trabajo.

- (1) Paso uno: $a(t) = -9,80665$.
- (2) Paso dos: $v(t) = -9,80665t + C$.
- (3) Paso tres: $v(t) = -9,80665t - 240$.
- (4) Paso cuatro: $y(t) = -4,90332t^2 - 240t + C$.
- (5) Paso cinco: $y(t) = -4,90332t^2 - 240t + 5000$.
- (6) Paso seis:
 - La primera derivada es $y'(t) = -9,80664t - 240$, coincidiendo con $v(t)$.
 - La segunda derivada es $y''(t) = -9,80664$, coincidiendo con $a(t)$.
 - La altitud para $t = 0$ es $y(0) = 5000$, como se deseaba.
 - La componente vertical de la velocidad para $t = 0$ es $y'(0) = -240$, como se deseaba.
- (7) Paso siete: Necesitamos determinar el tiempo de impacto, tal como hicimos en el problema anterior. (Sin embargo, este no siempre es el caso con este tipo de problemas.) De cualquier manera, el tiempo de impacto es muy pronto, a los $t = 15,7593$ segundos.
- (8) Paso siete, continuación: A una velocidad de $\dot{x} = -60$ [m/s], en 15,7593 segundos, el proyectil se habrá movido $-945,558$ en dirección x ; a una velocidad de $\dot{z} = +80$ [m/s], el proyectil se habrá movido 1260,74 en la dirección z . Por supuesto, al impacto, $y = 0$. Por lo tanto, el punto de impacto es (11 054,4; 0; 8260,74).

Nota: Los símbolos \dot{x} y \dot{z} son solo abreviaciones para dx/dt y dz/dt , respectivamente.

2.5.3 Fuego contra-batería

En la vida real, una de las cosas divertidas que se pueden hacer es rastrear matemáticamente una parábola de vuelo en sentido contrario, para detectar el punto de lanzamiento. Una vez que este es conocido, claramente hay una pieza de artillería ahí, y por lo tanto sería prudente bombardear fuertemente esa ubicación para destruir la batería enemiga. Esta técnica, llamada “fuego contra-batería”, fue desarrollada en la Primera Guerra Mundial y ha sido un pilar fundamental en el combate moderno desde aquella época.

Otra razón por la cual esta técnica es divertida es que debemos usar “el valor equivocado” de t cuando resolvemos $y(t) = 0$. Es decir, usamos la raíz tal que $t < 0$, algo que casi nunca ocurre en problemas de movimiento de proyectiles. El momento del lanzamiento es cuando $y(t) = 0$ con $t < 0$, mientras que el de impacto es cuando $y(t) = 0$ con $t > 0$. (De hecho, con esto estamos asumiendo implícitamente que la altura del lanzador, del observador y del punto de impacto son iguales, o al menos muy similares. Sin embargo, si no fuese este el caso, los ajustes correspondientes serían bastante directos.)

La ecuación cuadrática del ejemplo anterior nos revela que $t = -64,7057$ segundos es el momento de disparo. Esto parece razonable, pues hace que el tiempo total de vuelo sea igual a 80,4650 segundos, es decir un poco más de un minuto. Entonces, las coordenadas del obús que hizo el disparo pueden ser obtenidas calculando

$$\begin{aligned} (-60)(-64,7057) + 12000 &= 15\,882,3, \\ (+80)(-64,7057) + 7000 &= 1823,5. \end{aligned}$$

Por lo tanto, reportamos que la batería enemiga se encuentra en las siguientes coordenadas:

$$(15\,882,3; 0; 1823,5).$$

La verdad acerca de la resistencia del aire Por supuesto, hemos despreciado el viento y la resistencia del aire; en la vida real, sería mejor incluirlos. Probablemente el lector tenga curiosidad de cómo se hace esto. Lo que ocurre es que el tiempo es dividido en intervalos diminutos, alrededor de 10^{-4} o 10^{-5} segundos, y durante esas porciones de tiempo, uno asume que la resistencia del aire y el viento es una constante o una función lineal. Cerca de 10^6 o 10^7 porciones de tiempo representan el vuelo entero del proyectil. Usando técnicas que estudiaremos en las primeras partes del capítulo 5, es extremadamente sencillo escribir un programa de computadora que avanza repetidamente en el tiempo por cada uno de los intervalos.

La modificación por una constante o una función lineal solo elevará ligeramente la complejidad de los cálculos en cada intervalo de tiempo. Sin embargo, dado que estos son tan diminutos, la precisión del proceso completo puede ser muy alta.

El concepto general que estamos discutiendo aquí se llama “simulación numérica”.

2.5.4 El desafío: Un cohete multietapa

Ahora que ya hemos practicado las técnicas para estudiar el movimiento de proyectiles, consideremos el vuelo de prueba de un cohete. En este caso, el cohete falla y se estrella, y vamos a modelar su trayectoria de vuelo.

Durante operaciones normales, el motor provee una fuerza contante de empuje de 1,27 meganewtons. Sin embargo, once segundos en pleno vuelo, la computadora que controla la bomba de combustible falla y el flujo al motor se reduce hasta cero. Por supuesto, el cohete se estrella. Un buen modelo del empuje del motor después de la falla sería

$$f(t) = 1,27e^{-(t-11)/2} \text{ para todo } t > 11,$$

nuevamente medido en meganewtons. Será útil saber que la masa del cohete al momento del lanzamiento era de 107 000 kilogramos.

Haremos dos suposiciones para simplificar el problema. Primero, aunque el combustible claramente está siendo quemado durante los primeros once segundos, podemos suponer que la masa es contante durante el vuelo completo. (Eso es porque solo una diminuta fracción de combustible fue consumida antes del fallo.) Segundo, el cohete no llega muy alto y por lo tanto podemos asumir que $g = 9,806\,65 \text{ [m/s}^2\text{]}$ es una constante. (En realidad, el valor de g sí cambia conforme se alcanzan alturas cada vez mayores, pero un proyectil debe llegar relativamente alto antes que el cambio sea importante.)

Comencemos. Primero determínese la función de altitud cuando el motor funcionaba normalmente —es decir, para $0 < t < 11$ —. (Se puede asumir que el cohete estaba estático en la plataforma de lanzamiento, en $y = 0$, durante un largo tiempo antes del lanzamiento.) Una vez calculado esto, determínese la altura, velocidad y aceleración a los 11 segundos después del lanzamiento —al momento en que ocurrió la falla—. A continuación, usando la función $f(t)$ de arriba, calcúlense nuevas funciones de aceleración, velocidad y posición, para uso después del fallo. Ello permitirá determinar el tiempo de impacto.

Hay diez ítems que deben completarse para este proyecto:

- Primero, segundo y tercero, determinar las funciones de aceleración, velocidad y posición antes del fallo.
- Cuarto, quinto y sexto, determinar las funciones de aceleración, velocidad y posición después del fallo.
- Séptimo, calcular el tiempo del impacto.
- Octavo, construir una sola gráfica que muestre la altitud para el vuelo entero, de lanzamiento a impacto.
- Noveno, construir una sola gráfica de la velocidad durante el vuelo entero, de lanzamiento a impacto.
- Décimo, calcular la altura máxima alcanzada.

Para ayudar al lector a verificar si resolvió el problema correctamente, las dos gráficas se muestran en la figura 2.2.

2.6 Criptología: El ataque $p - 1$ de Pollard contra RSA

El Criptosistema RSA es fácilmente el más famoso sistema criptográfico en uso hoy en día. Casi cualquier transacción de E-comercio en la internet usa TLS (Transport Layer Security) o SSL (Secure Socket Layer) y, por lo tanto, usa RSA para intercambiar claves de cifrado por bloques. La seguridad de RSA está basada en la dificultad de factorizar el producto de dos (enormes) números primos.

Este proyecto es una de las muchas formas de ataque contra este criptosistema. Asumimos familiaridad con RSA y algo de teoría de números. Si el lector no está familiarizado con RSA, este proyecto no podrá ser completado. Sin embargo, incentivamos a investigarlo un poco. Es relativamente fácil de aprender, al menos

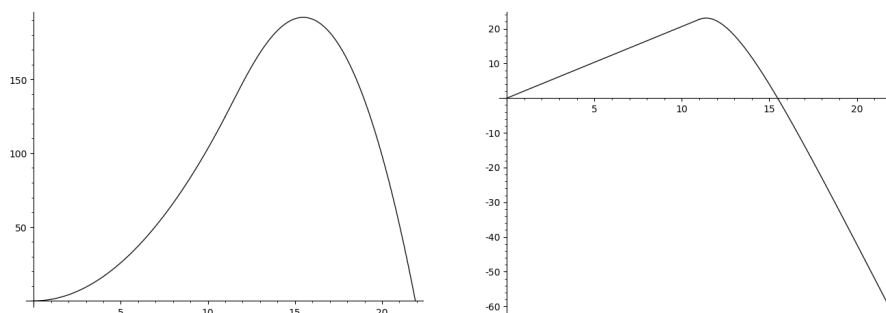


Figura 2.2 Las gráficas para el proyecto de física

en comparación con otros conceptos matemáticos, y provee una ventana a las aplicaciones de lo que, de otra manera, son áreas “puras”, como los números primos y la aritmética modular.

Este proyecto sigue de cerca la notación dada en *Cryptography and Coding Theory*, segunda edición, de Wade Trappe y Lawrence Washington, publicado por Pearson en 2005. Véase la sección 6.4 de ese texto. El método original fue descubierto por John Pollard en 1974.

Alternativamente, Minh Van Nguyen ha escrito un tutorial de criptografía para Sage que usa RSA como ejemplo. El título es “Number Theory and the RSA Public Key Cryptosystem”, y puede ser encontrado en

https://doc.sagemath.org/html/en/thematic_tutorials/numtheory_rsa.html

Finalmente, debemos indicar que este proyecto requiere conocimiento de los bucles `for`, que son discutidos en el capítulo 5 de este libro.

2.6.1 Los fundamentos: Números B -lisos y $B!$

Quienes estudian teoría de números suelen hablar del hecho de que ciertos números son “lisos”. Este es un término vago que significa que un número no tiene factores primos grandes. Más precisamente, para cualquier entero par positivo k , decimos que un número es k -liso si todos sus factores primos son menores que k . Por ejemplo, 15, 10, 100 y 2^{5000} son todos 6-lisos, pero 14 no lo es, pues $14 = 2 \times 7$ y 7 es mayor que 6. Como otro ejemplo, 14 y 1400 son 8-lisos, pero 11 no lo es. Este concepto puede ser de interés en varias ramas de la Teoría de Números, pero es de especial interés en criptografía. Esta es la forma estándar de definir lo que significa que un número sea “liso”.

Como llega a ocurrir, una clave pública RSA puede ser fácilmente violada (factorizando su $N = pq$, donde p y q , de ahora en adelante, son primos grandes), si $p - 1$ o $q - 1$ es liso. Esta es una afirmación extraña, por supuesto. Primero que nada, p y q no son información pública. Sin embargo, eso no constituye una barrera para este ataque, pues solamente es requerido que $p - 1$ o $q - 1$ sea liso —no necesitamos conocer ni $p - 1$ ni $q - 1$, o incluso cuál de los dos primos es “1 sumado con un número liso”—.

Otra cosa extraña es que vamos a usar una definición no estándar de lisura. Vamos a elegir un número grande B y preguntarnos si, cualquiera, p o q divide $B!$. Dentro de poco usaremos $B = 10\,000$, pero por el momento usemos $B = 100$. Exploremos cómo este uso de $100!$ se relaciona con que un número sea 100-liso.

La única manera en que un número z falle en ser 100-liso es si z tiene un factor primo que es mayor a 100, tal como podrían ser 101 o 103. Dado que estos son mayores a 100, sabemos que ni 101 ni 103 dividirán ninguno de los números de 1 a 100. Por lo tanto, por un famoso lema de teoría de números, que dice que $p|ab$ si y solo si $p|a$ o $p|b$, podemos concluir que ni 101 ni 103 dividen $100!$. Similarmente, ningún múltiplo de 101 o 103 tampoco dividirá $100!$. Lo mismo es válido para cualquier primo mayor, tal como 107 y otros. Por lo tanto, concluimos que cualquier número que tenga un factor primo mayor a 100 (es decir, que no sea 100-liso), definitivamente no dividirá $100!$.

Sin embargo, podríamos estar interesados en lo opuesto: si un número y no divide $100!$, ¿será siempre el caso que y no es 100-liso? Aquí encontraremos algunas excepciones. Usando la instrucción

Código de Sage

```
1 factor(factorial(100))
```

vemos que

$$100! = 2^{97} \cdot 3^{48} \cdot 5^{24} \cdot 7^{16} \cdot 11^9 \cdot 13^7 \cdot 17^5 \cdot 19^5 \cdot 23^4 \cdot 29^3 \cdot 31^3 \cdot 37^2 \cdot 41^2 \cdot 43^2 \cdot 47^2 \cdot 53 \cdot 59 \cdot 61 \cdot 67 \cdot 71 \cdot 73 \cdot 79 \cdot 83 \cdot 89 \cdot 97$$

Por lo tanto, podemos ver que 2^{98} , 3^{49} y 5^{25} no dividirán $100!$. Sin embargo, estos tres números son 6-lisos, así que de seguro que son 100-lisos. Entonces, hemos encontrado tres ejemplos de números 100-lisos que no dividen $100!$.

Con todo esto, hemos mostrado que cualquier divisor de $100!$ es 100-liso, pero que no todos los números 100-lisos son divisores de $100!$. Así, de hecho, tenemos que el requerimiento de que un número sea divisor de $100!$ es *un estándar más estricto* que el requerimiento que sea 100-liso. En general, el conjunto de números que dividen $B!$ es un subconjunto de los B -lisos. Aunque en general, si consideramos números menores que $B!$, las excepciones son pocas y distantes entre sí, y los conceptos son más o menos equivalentes.

2.6.2 La teoría detrás del ataque

Requerimos un teorema más de Teoría de Números. Para cualesquiera número primo p y número entero $a \not\equiv 0 \pmod p$, el Pequeño Teorema de Fermat (PTF) afirma que

$$a^p \equiv a \pmod p \quad \text{o, equivalentemente,} \quad a^{p-1} \equiv 1 \pmod p.$$

Supongamos ahora que $p-1$ divide $B!$ (y por lo tanto es B -liso). Eso significa que existe algún k tal que $(k)(p-1) = B!$. Entonces, por el PTF, podemos concluir que

$$2^{B!} = 2^{(k)(p-1)} = (2^{p-1})^k \equiv 1^k \equiv 1 \pmod p.$$

Como $2^{B!} \equiv 1 \pmod p$, debemos tener que

$$(2^{B!} - 1) \equiv 0 \pmod p$$

o, más claramente, que $c = 2^{B!} - 1$ es un múltiplo de p . Este hecho clave, que c es un múltiplo de p , es el corazón de la idea entera.

De hecho, pensándolo mejor, debemos tener que c es un múltiplo de p en los enteros ordinarios, y la imagen de $c \pmod N$ es un múltiplo de p en “los enteros $\pmod N$ ”. La mayoría de los lectores pueden tener fe este punto un tanto técnico y saltar directamente al comienzo del siguiente párrafo. Para aquellos que deseen rigor absoluto, observemos que si c es un múltiplo de p , entonces $c = jp$, para algún entero j . No tenemos razón para creer que j es divisible por q , pero hagamos la división de todas maneras, obteniendo un cociente j_1 y un residuo j_2 . Es decir, $j = qj_1 + j_2$, y así

$$c = jp = (qj_1 + j_2)p = pqj_1 + pj_2 = Nj_1 + pj_2 \equiv 0 + pj_2 \pmod N,$$

que nos permite concluir que la imagen de $c \pmod N$ es justo pj_2 , claramente un múltiplo de p .

¿Qué hemos ganado con todo esto? Pues bien, dado que $N = pq$, sabemos que este también es un múltiplo de p . Como c y N son ambos múltiplos de p , entonces $\gcd(c, N)$ también lo es. Este gcd de hecho puede ser solo uno de dos valores: o es p o es N .

Por lo tanto, para quebrar RSA, primero calculamos $c = (2^{B!} - 1) \pmod N$, lo que no es tarea fácil, y entonces calculamos el valor de $\gcd(c, N)$. Si obtenemos algo entre 1 y N , sabemos que debe ser p , y la división ordinaria nos proveerá de $q = N/p$. Hemos factorizado N y hemos violado la clave pública de esta persona. Por otro lado, si obtenemos que el gcd es N , no tuvimos suerte y hemos fallado.⁴ El gcd será N (resultando en un ataque fallido) si y solo si q también divide c . Eso esencialmente solo ocurrirá si $p-1$ y $q-1$ ambos dividen $B!$.

⁴Si alguna vez nos ocurre esto en la práctica, podemos tratar nuevamente con un B más pequeño. Si el primo más grande en la factorización de $p-1$ es f_1 y el primo más grande en la factorización de $q-1$ es f_2 , entonces elegir un B entre f_1 y f_2 —probablemente por prueba y error—, resultará en éxito, es decir, $\gcd(c, N) = p$.

Como suele ocurrir, tener que *cualquiera*, $p - 1$ o $q - 1$, divida $B!$ es bastante raro. Sin embargo, tener que *ambos*, $p - 1$ y $q - 1$, dividan $B!$ es extremadamente raro. Más precisamente, p y q se suponen generados aleatoriamente, y por lo tanto, la probabilidad que $p - 1$ divida $B!$ es independiente de la probabilidad que $q - 1$ divida $B!$. La probabilidad que ambos dividan $B!$ es por lo tanto *el cuadrado de la probabilidad* que uno de ellos lo divida. De acuerdo a esto, esperamos plenamente que el ataque funcionará para cualquier N , siempre que $p - 1$ o $q - 1$ divida $B!$

2.6.3 Calculando $2^{B!}$ mód N

Reflexionemos sobre el echo que calcular

$$2^{10\,000!} \text{ mód } N$$

no es una tarea que se nos pida muy frecuentemente. Sería imprudente sumergirse en el problema sin pensarlo un poco primero. Podríamos preguntarnos: ¿exactamente cuán grande será $10\,000!$? o, más precisamente, ¿cuántos dígitos tendrá?

Resulta que esta pregunta es fácil de responder si recordamos la aproximación de Stirling:

$$B! \approx B^B e^{-B} \sqrt{2\pi B}$$

y procedemos tomando el logaritmo común:

$$\begin{aligned} \log_{10}(B!) &\approx \log_{10}(B^B)(e^{-B})(\sqrt{2\pi B}) \\ &\approx (\log_{10} B^B) + (\log_{10} e^{-B}) + (\log_{10} \sqrt{2\pi B}) \\ &\approx (B \log_{10} B) - (B \log_{10} e) + \frac{1}{2} (\log_{10} 2\pi B) \\ &\approx B (\log_{10} B - \log_{10} e) + \frac{1}{2} (\log_{10} 2\pi) + \frac{1}{2} (\log_{10} B) \\ &\approx B (\log_{10} B - 0,434\,294\,481 \dots) + 0,399\,089\,934 \dots + \frac{1}{2} \log_{10} B \end{aligned}$$

...o, en nuestro caso de $B = 10\,000$ y $\log_{10} B = 4,...$

$$\begin{aligned} &\approx 10\,000 (4 - 0,434\,294\,481 \dots) + 0,399\,089\,934 \dots + \frac{1}{2}(4) \\ &\approx 35\,657,0551 \dots + 2,399\,089\,93 \dots \\ &\approx 35\,659,4542 \dots \end{aligned}$$

Al caer en cuenta que $10\,000!$ es un número que tiene 35 660 dígitos (aproximadamente), probablemente no deberíamos calcularlo para entonces determinar $2^{10\,000!}$ y finalmente $2^{10\,000!}$ mód N . Ese cálculo puede tomar un tiempo muy largo. Debemos encontrar un enfoque alternativo.

Consideremos calcular

$$\begin{aligned} c_1 &= 2^1 \text{ mód } N \\ c_2 &= (c_1)^2 \text{ mód } N \\ c_3 &= (c_2)^3 \text{ mód } N \\ c_4 &= (c_3)^4 \text{ mód } N \\ c_5 &= (c_4)^5 \text{ mód } N \\ c_6 &= (c_5)^6 \text{ mód } N \\ &\vdots \end{aligned}$$

Entonces tendríamos que

$$c_6 \equiv (c_5)^6 \equiv ((c_4)^5)^6 \equiv (((c_3)^4)^5)^6 \equiv (((((c_2)^3)^4)^5)^6) \equiv 2^{(1)(2)(3)(4)(5)(6)}.$$

Pero observemos que eso es justamente lo que queríamos calcular, es decir,

$$2^{(1)(2)(3)(4)(5)(6)} \equiv 2^{6!}.$$

De manera similar, podemos ver que $c_B \equiv 2^{B!} \pmod{N}$.

2.6.4 El desafío: Hacer que todo esto ocurra en Sage

Nuestra tarea es fácil. Usando la estrategia de arriba, calculamos

$$c_B = 2^{10\,000!} \pmod{N},$$

y entonces tenemos $c = c_B - 1 = (2^{10\,000!} - 1) \pmod{N}$. A continuación, hallamos $p = \gcd(c, N)$, seguido de $q = N/p$ y verificamos el trabajo multiplicando p y q de nuevo. Aquí tenemos siete casos de estudio para el lector:

```
357 177 030 895 805 008 781 319 266 876 641 103 581 839 678 151 495 135 129 133 067 010 127
5 000 272 002 251 811 540 446 579 586 816 039 346 308 785 565 641 862 331 905 860 763 123 733
111 293 094 034 769 304 884 167 051 458 174 320 813 595 510 448 138 274 866 152 002 067 365 927
189 114 989 429 752 082 926 285 457 551 369 642 787 381 790 039 260 802 307 452 110 490 304 547
250 733 654 482 468 568 921 620 042 866 859 718 852 920 028 026 076 547 177 974 758 239 109 177
201 557 389 900 540 095 613 559 219 541 299 540 522 405 259 329 399 736 824 858 252 876 376 521 311 053 006 710 577 163 057 234 093
862 021 547 643 631 582 396 998 212 208 722 914 288 258 644 234 791 307 950 582 916 442 747 222 039 795 609 417 741 932 278 317 121
```

A propósito, se pueden copiar y pegar estos números en Sage del archivo PDF de este libro, disponible en la página web www.sage-para-estudiantes.com/. ¡No es recomendable transcribirlos manualmente!

2.6.5 Protecciones contra el ataque factorial de Pollard

Como podemos ver, las claves públicas RSA pueden ser violadas (lo que significa que N puede ser factorizado) si cualquiera, $p - 1$ o $q - 1$, es liso. Un número liso (como uno 6-liso o 100-liso) puede tener un número bastante grande de factores primos, cada uno de los cuales contribuye una pequeña magnitud al producto final. Por ejemplo,

$$10^{100} = 2^{100} 5^{100}$$

es un número muy liso, con 200 primos en su factorización (contando multiplicidades) y ninguno contribuye más que un mísero 5 al enorme producto final. ¿Cuál es el extremo opuesto de esto?

El otro extremo sería un número primo. Hay un único factor primo, y hace toda la contribución a la magnitud por sí mismo. Por supuesto, p y q son impares y, por lo tanto, ni $p - 1$ ni $q - 1$ pueden ser primos. Sin embargo, podrían ser 2 veces un número primo. Los números primos p tales que $(p - 1)/2$ es también primo son llamados “primos seguros”. En una noción casi idéntica, cuando p y $2p + 1$ son primos, estos son llamados “primos de Sophie Germain”, nombrados por Marie-Sophie Germain (1776–1831).

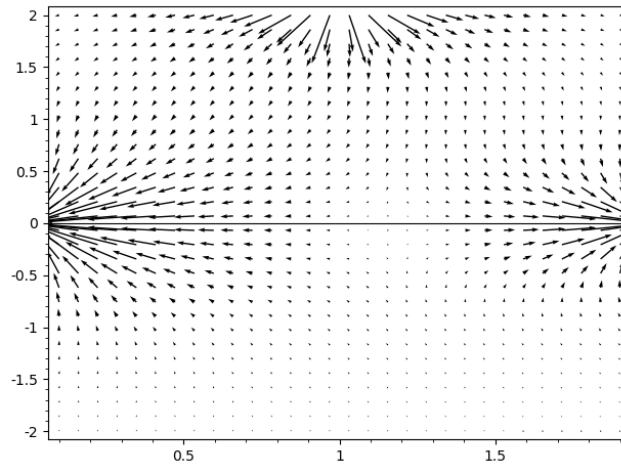
Para tornar el ataque de Pollard totalmente inviable, muchas implementaciones de RSA requieren que ambos primos sean “seguros”. Para estos números, B debería ser $(p - 1)/2$ o mayor. Incluso si p tiene solo 200 bits de longitud, lo que es extremadamente pequeño, entonces $(p - 1)/2$ tendría 199 bits de longitud, y eso haría computacionalmente inviable calcular $2^{B!}$. El Sol consumiría todo su combustible, tornándose en una gigante roja,⁵ mucho antes que el cálculo pudiese terminar.

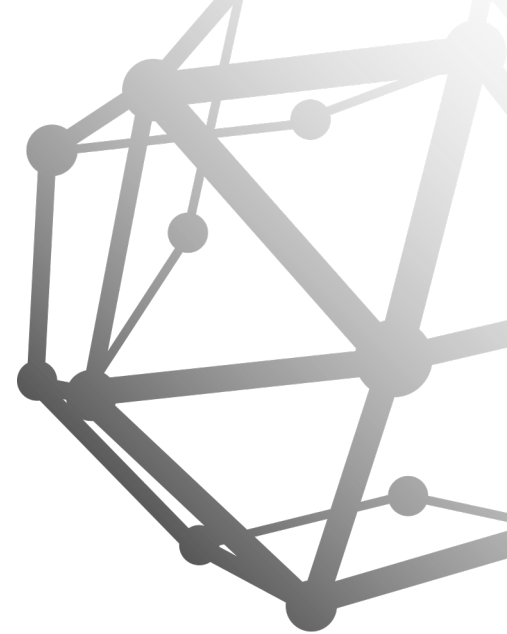
⁵Esto pasará en más o menos 5 mil millones de años a partir de hoy

2.7 Miniproyecto sobre gráficas vectoriales de un campo eléctrico

Tómese el lector un momento para leer la sección 3.7, “gráficas de campos vectoriales”, empezando en la página 110, si no la ha leído aún. En este miniproyecto se pide modificar el modelo usado por el autor en la subsección “una aplicación de la física” para generar la gráfica del campo vectorial de tres cargas eléctricas: añadiremos una cuarta carga, con $q_3 = -1$, la cual está localizada en el punto $(1; 2,25)$.

El desafío es modificar el código del autor para acomodar la cuarta carga y entonces producir la gráfica del campo vectorial resultante. Se debería obtener la siguiente imagen:





3

Técnicas avanzadas de graficación

Uno de los usos más gratificantes de Sage es la producción de gráficos de alta calidad, ya sea para la enseñanza o para publicación. Sage tiene una enorme variedad de utilidades para componer gráficos e imágenes. Los elementos básicos de graficación fueron presentados en la sección 1.4 en la página 9. Si el lector no ha leído aún esa parte, es recomendable que lo haga ahora. Casi todo lo demás que uno quisiera graficar es cubierto en este capítulo. La excepción ciertamente son los gráficos coloridos y las gráficas 3D, que solo pueden apreciarse en una pantalla de computadora y no en un libro impreso. Estos serán discutidos en el apéndice únicamente electrónico en línea de este libro, “Graficando en color, en 3D, y animaciones”, disponible en la página web www.sage-para-estudiantes.com, para descarga gratuita. Más aun, dado que los gráficos de dispersión son esencialmente solo usados al hacer regresiones, la técnica correspondiente es discutida en la sección 4.9 en la página 151, por simplicidad. Similarmente, los gráficos de campos de pendientes son solamente usados (según conoce el autor) en el estudio de ecuaciones diferenciales de primer orden, por lo que serán discutidos en la subsección 4.22.3 en la página 188, para mayor continuidad de la exposición.

3.1 Anotando gráficos por claridad

En esta sección tenemos una colección de técnicas para hacer que las gráficas 2D se vean más atractivas. Estas incluyen etiquetar los ejes, añadir cuadriculados, incluir un marco, resaltar un punto haciéndolo grande, dibujar flechas, insertar texto, sombrear gráficas (para integrales), y usar líneas punteadas y quebradas.

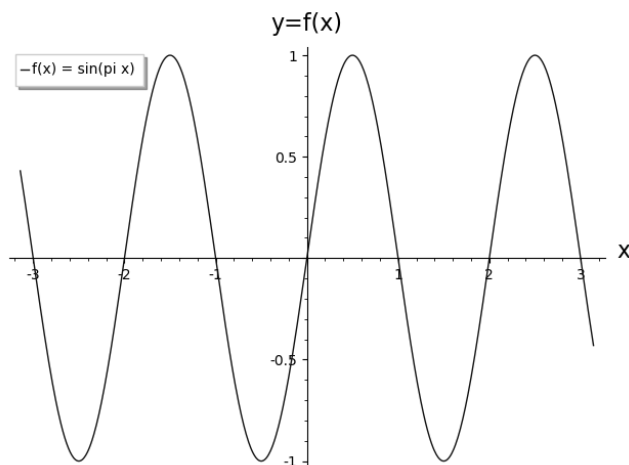
3.1.1 Etiquetando los ejes de los gráficos

En ocasiones es muy útil etiquetar los ejes directamente en el gráfico. Esto es particularmente práctico cuando existe el riesgo de malinterpretar las unidades involucradas, o en aplicaciones científicas donde se manejan varias funciones. En cualquier caso, las siguientes instrucciones cumplen ese objetivo:

Código de Sage

```
1 p = plot(sin(pi*x), -pi, pi, legend_label='f(x) = sin(pi x)')
2 p.axes_labels(['x', 'y=f(x)'])
3 p.show()
```

Este código, proporcionado por Martin Albrecht y Keith Wojciechowski, produce la siguiente imagen:



Aquí tenemos otro ejemplo, de la economía. La curva de demanda

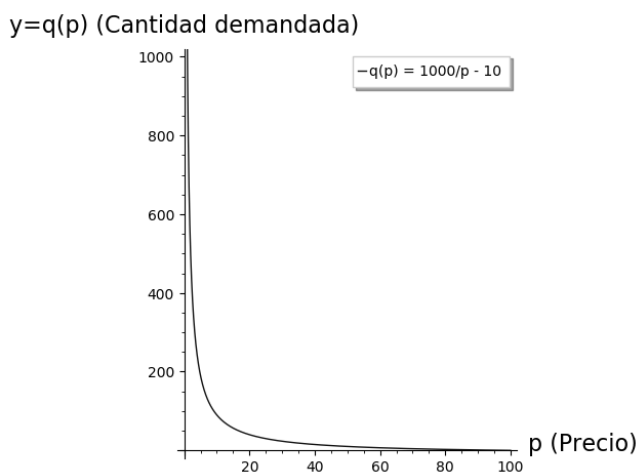
$$q(p) = \frac{1000}{p} - 10$$

relaciona la cantidad demandada por los consumidores de algún producto con su precio. El siguiente código grafica esta curva, junto con etiquetas informativas.

Código de Sage

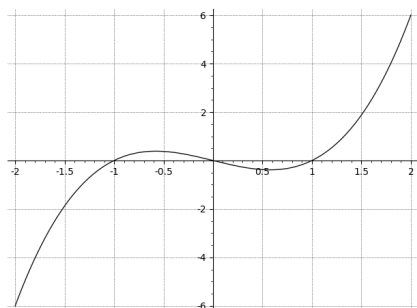
```
1 p = plot(1_000/x - 10, 0, 100, ymax=1_000, legend_label='q(p) = 1000/p -
  ↳ 10')
2 p.axes_labels(['p (Precio)', 'y=q(p) (Cantidad demandada)'])
3 p.show()
```

La gráfica producida es

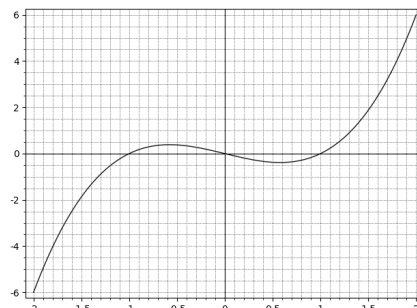


3.1.2 Cuadriculados y gráficas estilo calculadora

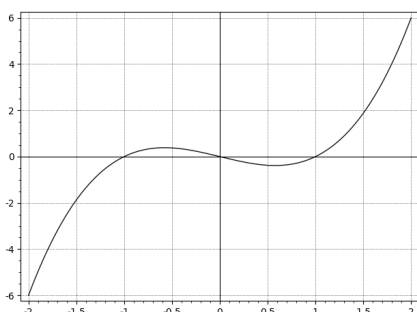
En ocasiones es agradable a la vista mostrar un cuadriculado en una gráfica, como si esta hubiese sido hecha en papel de dibujo. Existen dos opciones de comando para lograr esto: una es `frame`, que puede tomar los valores `True` o `False`, y la otra es `gridlines`, que puede tomar los valores `True`, `False` o `'minor'`. Por lo tanto, en realidad existen seis elecciones posibles de diseño. Estas se muestran en la siguiente colección de gráficas.



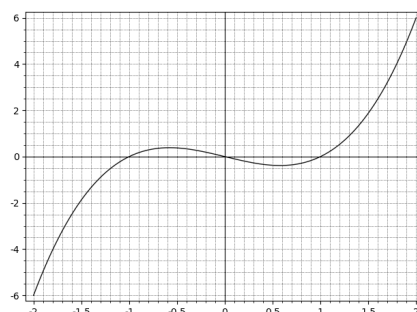
Gráfica 1



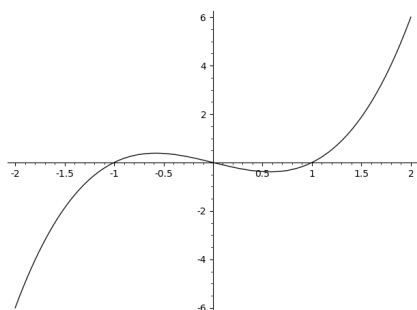
Gráfica 2



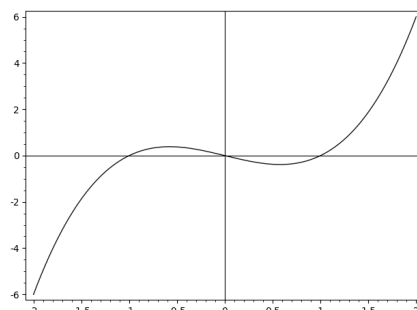
Gráfica 3



Gráfica 4



Gráfica 5



Gráfica 6

Aquí tenemos el código que generó cada una de las imágenes anteriores.

Gráfica 1:

Código de Sage

```
1 plot(x^3-x, -2, 2, gridlines=True, frame=False)
```

Gráfica 2:

Código de Sage

```
1 plot(x^3-x, -2, 2, gridlines='minor', frame=True)
```

Gráfica 3:

Código de Sage

```
1 plot(x^3-x, -2, 2, gridlines=True, frame=True)
```

Gráfica 4:

Código de Sage

```
1 plot(x^3-x, -2, 2, gridlines='minor', frame=True)
```

Gráfica 5:

Código de Sage

```
1 plot(x^3-x, -2, 2, gridlines=False, frame=False)
```

Gráfica 6:

Código de Sage

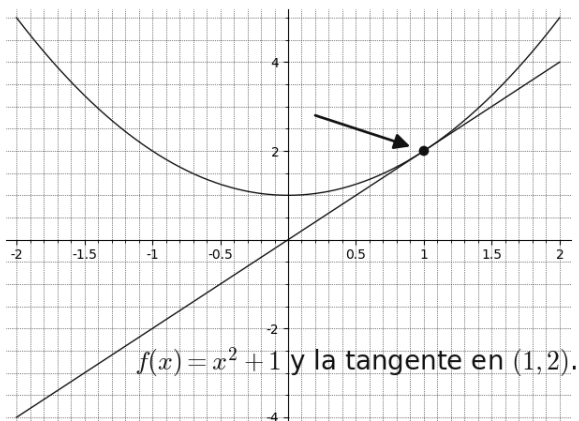
```
1 plot(x^3-x, -2, 2, gridlines=False, frame=True)
```

3.1.3 Añadiendo flechas y texto

Sage tiene unas técnicas muy atractivas para hacer los gráficos más fáciles de entender. Ya hemos visto cómo el comando `point` puede ser usado para dibujar un gran punto en una gráfica, a manera de atraer la atención. Cuandoquiera que resulte necesario, el comando `arrow` puede ser usado para graficar una grande y poco sutil flecha. Esta está definida por dos puntos: su cola y su cabeza, con las coordenadas de la cola dadas primero y las de la cabeza a continuación. La sintaxis se muestra abajo. También el comando `text` puede ser usado para añadir texto directamente en la gráfica. Tiene dos parámetros: el mensaje a ser escrito y el centro de este.

Código de Sage

```
1 recta_tangente = plot(2*x, -2, 2, gridlines='minor')
2 curva = plot(x^2+1, -2, 2)
3 gran_punto = point((1,2), size=60)
4 la_flecha = arrow((0.2, 2.8), (0.9, 2.1))
5 las_palabras = text('$f(x) = x^2 + 1$ y la tangente en $(1,2)$.', (0.5,
6   ↪ -2.75), fontsize=20)
7 imagen_total = recta_tangente + curva + gran_punto + la_flecha +
8   ↪ las_palabras
9 imagen_total.show()
```



Por último, pero no menos importante, vale la pena mencionar que el comando `text` reconoce el lenguaje \LaTeX de redacción científica, como puede apreciarse en el anterior trozo de código. Si el lector no conoce este lenguaje, entonces puede escribir también

Código de Sage

```
1 las_palabras = text('f(x) = x^2 + 1 y la tangente en (1,2)', (0.5, -2.75),
  ↪ fontsize=20)
```

y obtendrá casi el mismo efecto.

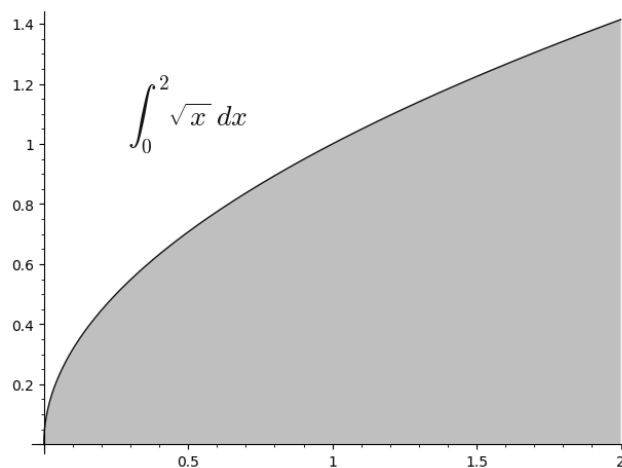
3.1.4 Graficando una integral

Aquí tenemos otro ejemplo del poder de \LaTeX , junto con la graficación de un sombreado de la misma forma que cuando se enseña integrales.

Código de Sage

```
1 P1 = text('\int_0^2 \sqrt{x} \; dx$', (0.5, 1.1), fontsize=20)
2 P2 = plot(sqrt(x), 0, 2, fill=True)
3 P = P1 + P2
4 P.show()
```

La siguiente imagen es el resultado:

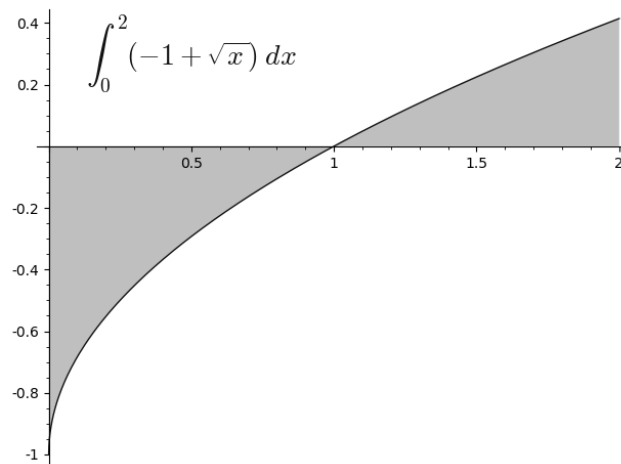


La opción `fill` es excelente para trabajar con integrales, incluso de funciones que cruzan el eje x :

Código de Sage

```
1 P1 = text('\int_0^2 (-1+\sqrt{x}) \; dx$', (0.5, 0.3), fontsize=20)
2 P2 = plot(-1+sqrt(x), 0, 2, fill=True)
3 P = P1 + P2
4 P.show()
```

Como podemos ver en la imagen abajo, Sage sabe cómo sombrear la gráfica, dependiendo de si la función es positiva o negativa.



Sin embargo, para otros tipos de gráficos sombreados, tales como al graficar sistemas de inecuaciones, otras técnicas son mucho más flexibles y adecuadas. Ese tema lo cubriremos en el apéndice únicamente electrónico en línea de este libro “Graficando en color, en 3D, y animaciones”, disponible en la página web del libro, en la dirección www.sage-para-estudiantes.com, para descarga gratuita.

3.1.5 Líneas punteadas y quebradas

Algunas veces, cuando tres o más gráficas son dibujadas juntas en la misma área, es bueno mostrar cuál es cuál mediante algún método visual. La mejor manera de hacer esto es usando colores distintos, pero otra forma consiste en usar líneas punteadas y quebradas. Sage ofrece cuatro variaciones en adición al trazo sólido usual. El siguiente fragmento de código:

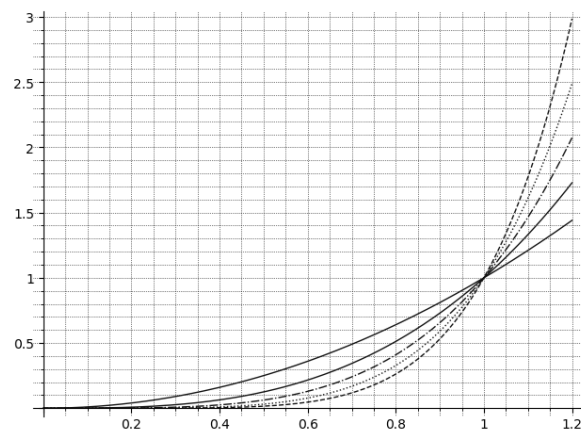
Código de Sage

```

1 P1 = plot(x^2, 0, 1.2, gridlines='minor')
2 P2 = plot(x^3, 0, 1.2, linestyle = '-')
3 P3 = plot(x^4, 0, 1.2, linestyle = '-.')
4 P4 = plot(x^5, 0, 1.2, linestyle = ':')
5 P5 = plot(x^6, 0, 1.2, linestyle = '--')
6 P = P1 + P2 + P3 + P4 + P5
7 P.show()

```

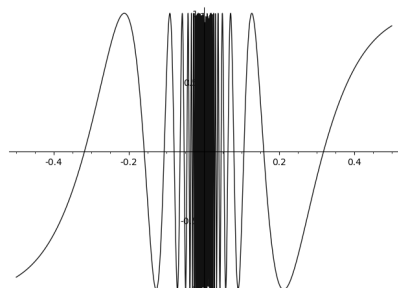
mostrará cada una de las cuatro alternativas simultáneamente, como en el siguiente gráfico:



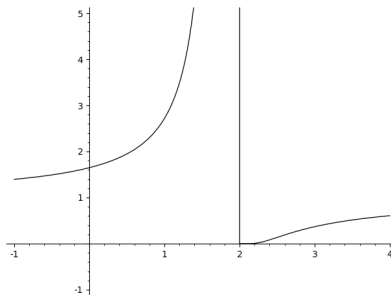
Si el lector desea saber cómo diferenciar las gráficas usando colores en lugar de diferentes trazos, puede consultar el apéndice únicamente electrónico de este libro, “Graficando en color, 3D, y animaciones”, disponible en la página web www.sage-para-estudiantes.com, para descarga gratuita.

3.2 Gráficas de algunas funciones hiperactivas

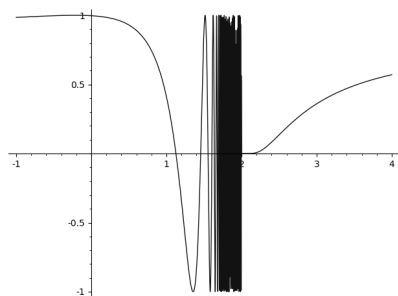
La función $\sin(1/x)$ se “sacude como loca” cerca de $x = 0$, pero Sage detecta esto y grafica muchos puntos adicionales en esa región para compensar. Hay muchas otras funciones hiperactivas en matemáticas. Aquí mostramos algunos ejemplos, junto con sus gráficas generadas por Sage. Podemos ver que Sage no es perfecto, pero cumple con el objetivo.



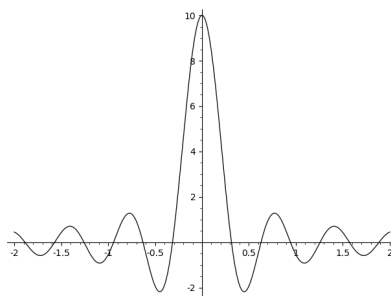
Gráfica 1



Gráfica 2



Gráfica 3



Gráfica 4

Gráfica 1: La función $f(x) = \sin(1/x)$ oscila de forma extrema en la vecindad de $x = 0$. Si observamos cuidadosamente cerca de los puntos $(0, 1)$ y $(0, -1)$, veremos que la “mancha azul” en la gráfica no es uniforme. Eso se debe a cómo Sage grafica funciones. Este toma el intervalo del eje x que fue solicitado y lo divide en —digamos— 200 subintervalos. Un valor x aleatorio es elegido en cada subintervalo. Bajo ciertas condiciones, Sage puede detectar con esta información que la función es “complicada” y empezar a usar más puntos, cada vez más cercanos entre sí. Sin embargo, siempre usará un número finito de puntos y así, en regiones “infinitamente complicadas”, puede no haber uniformidad. Más aun, esto significa que a veces la gráfica se verá diferente cuando se grafique dos veces seguidas.

Código de Sage

```
1 plot(sin(1/x), -0.5, 0.5)
```


Gráfica 2: La función $f(x) = \exp\left(\frac{1}{2-x}\right)$ tiene un límite interesante cuando x se aproxima a 2. Si lo hace por la izquierda (usando valores de x ligeramente menores que 2), el límite es infinito; si lo hace por la derecha (usando valores ligeramente mayores que 2), el límite es cero. Por lo tanto, el límite cuando x tiende a 2 en general *no existe*. Este es un ejemplo interesante pues el autor encuentra que los estudiantes en *Cálculo I* frecuentemente creen que los límites complicados son invenciones artificiales y es poco probable tan siquiera toparse con ellas.

Código de Sage

```
1 plot(exp(1/(2-x)), -1, 4, ymin=-1, ymax=5)
```

Gráfica 3: La función $f(x) = \sin\left(\exp\left(\frac{1}{2-x}\right)\right)$ es solamente el seno de la anterior. A la derecha de $x = 2$, esta es similar al $\sin(0)$ y, por lo tanto, casi no tiene oscilaciones; a la izquierda de $x = 2$, esta es similar a una onda sinusoidal cuya frecuencia de aproxima a infinito. Es interesante estudiar la gráfica de esta función a varias escalas.

Código de Sage

```
1 plot(sin(exp(1/(2-x))), -1, 4)
```

Gráfica 4: La función $f(x) = \frac{\sin(10x)}{10x}$ es interesante por varias razones. Primero, es visualmente atractiva. Segundo, esta se presenta en el estudio de las *wavelets*, que es un área entera por derecho propio. Tercero, muchos estudiantes cometen el error de simplificar la x y trabajan con $f(x) = \sin(10)/10$ en su lugar, o tal vez incluso simplifican los 10s, dejando solo $f(x) = \sin$. Cuarto, esta función aparece en el Análisis Matemático (así como el Procesamiento de Señales) tan frecuentemente que tiene su propio nombre:

$$f(x) = \text{sinc}(10x) = \frac{\sin(10x)}{10x}.$$

Código de Sage

```
1 plot(sin(10*x)/x, -2, 2)
```

3.3 Gráficas polares

La idea general detrás de las coordenadas polares es que, en lugar de tener una coordenada x y una coordenada y , uno tiene un radio r y un ángulo θ . Para cualquier punto, r es la distancia al origen, y θ está definida de manera que 0 es dirección “este” u horizontal a la derecha y los ángulos positivos representan movimiento contra las manecillas del reloj. Así, 90° o $\pi/2$ radianes sería directamente hacia arriba o “norte”, mientras que 180° o π radianes sería hacia la izquierda u “oeste”. De manera análoga, los ángulos negativos representan movimiento en dirección de las manecillas del reloj. Así, -90° o $-\pi/2$ radianes sería directamente hacia abajo o “sud”.

Cada dirección de movimiento a partir del “este” tiene, por lo tanto, varios nombres. Por ejemplo,

$$\theta \in \left\{ \dots, -\frac{22\pi}{4}, -\frac{15\pi}{4}, -\frac{7\pi}{4}, \frac{\pi}{4}, \frac{9\pi}{4}, \frac{17\pi}{4}, \frac{25\pi}{4}, \dots \right\}$$

son todos valores de θ que representan “noreste”. Teniendo esto en cuenta, graficar una función en coordenadas polares a mano es un poco desafiante, pues desconocemos cuántas “vueltas” o “revoluciones” alrededor del origen son requeridas antes de volver a partes de la curva que ya han sido graficadas. Por lo tanto, resulta algo no trivial deducir cómo elegir los valores de θ que necesitamos para graficar.

Una función en coordenadas polares toma θ como argumento y devuelve r como imagen. Consideremos, por ejemplo,

$$r(\theta) = 2 \cos(5\theta),$$

que produce una gráfica llamada rosa o rosetón de 5 pétalos. Para graficarla en Sage, escribimos

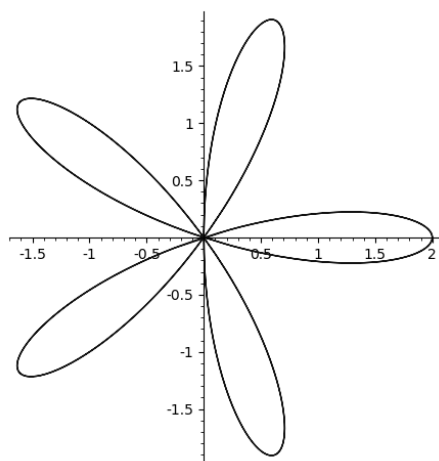
Código de Sage

```

1 var('theta')
2 polar_plot(2*cos(5*theta), (theta, 0, 2*pi))

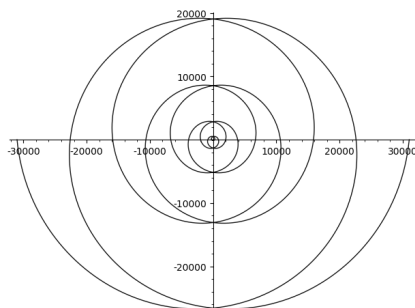
```

que resulta en la siguiente imagen:

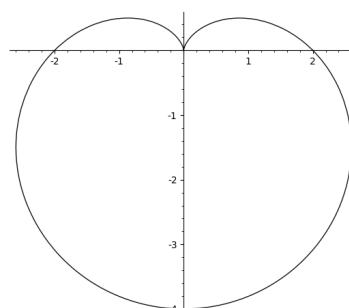


3.3.1 Ejemplos de gráficas polares

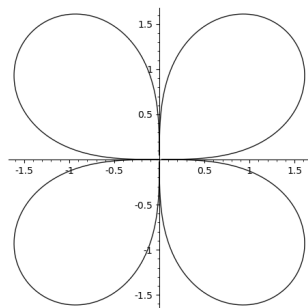
Aquí tenemos cuatro interesantes gráficas polares adicionales.



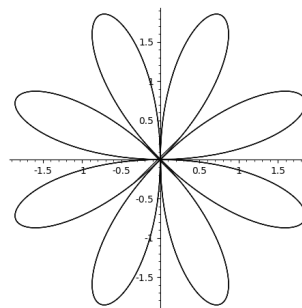
Gráfica 1



Gráfica 2



Gráfica 3



Gráfica 4

Gráfica 1: En ocasiones, funciones simples tienen gráficas polares complicadas. Un ejemplo es $r(\theta) = \theta^3$ para $-10\pi < \theta < 10\pi$.

Código de Sage

```
1 var('theta')
2 polar_plot(theta^3, (theta, -10*pi, 10*pi))
```

Gráfica 2: Esta forma es llamada “cardioide”, y está dada por

$$r(\theta) = 2 - 2 \sin(\theta)$$

Código de Sage

```
1 var('theta')
2 polar_plot(2-2*sin(theta), (theta, 0, 2*pi))
```

Gráfica 3: Aquí tenemos la forma de un trébol. Nótese que los pétalos tienen una forma diferente que en el rosetón. La función es $r(\theta) = 2\sqrt{|\sin(2\theta)|}$.

Código de Sage

```
1 var('theta')
2 polar_plot(2*sqrt(abs(sin(2*theta))), (theta, -pi, pi))
```

Gráfica 4: Esta es una rosa de 8 pétalos, dada por $r(\theta) = 2 \sin(4\theta)$.

Código de Sage

```
1 var('theta')
2 polar_plot(2*sin(4*theta), (theta, -2*pi, 2*pi))
```

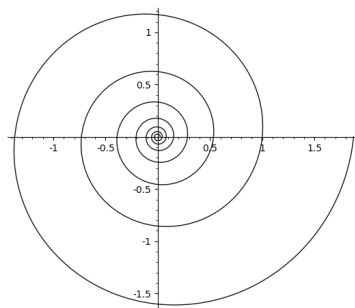
3.3.2 Problemas que pueden ocurrir ocasionalmente

La siguiente función produce una agradable espiral exponencial:

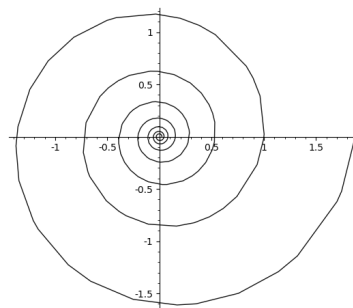
Código de Sage

```
1 polar_plot(exp(theta/10), (theta, -12*pi, 2*pi), plot_points=10_000)
```

Esta curva requiere que especifiquemos una gran cantidad de puntos —en este caso, 10 000 puntos— para graficar. Para mostrar por qué, consideremos las siguientes dos imágenes. Para la de la izquierda, la opción `plot_points` ha sido usada como arriba, y para la de la derecha, ha sido removida —usando 200 puntos por defecto—.



Graficado con 10000 puntos



Graficado con 200 puntos

Como podemos apreciar, obtenemos una calidad muy inferior cuando usamos solamente 200 puntos en lugar de 10 000. Una forma de entender esto es notar que el dominio de la función es $-12\pi < \theta < 2\pi$, de manera que 200 puntos implica solo

$$200/14\pi \approx 4,54728 \dots \text{ puntos por radián,}$$

lo que es un problema porque un radián es cerca de 57 grados. En contraste, si usamos 10 000 puntos, tenemos $227,364 \dots$ puntos por radián, que es 50 veces más denso. Si observamos de cerca la espiral exponencial de la derecha, vemos que Sage ha graficado una curva lineal por trozos —una sucesión de segmentos de recta conectando los 200 puntos—.

Aquí tenemos otro ejemplo con una razón diferente para requerir una cantidad grande de puntos. Esta curva es llamada *lemniscata*. El código que la produce es

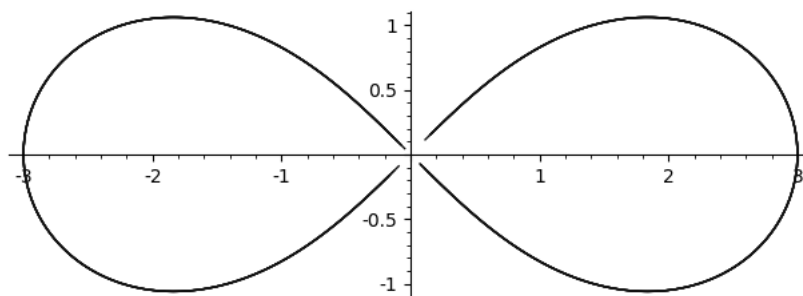
Código de Sage

```
1 var('theta')
2 polar_plot(3*sqrt(cos(2*theta)), (theta, -6*pi, 6*pi), plot_points=10_000)
```

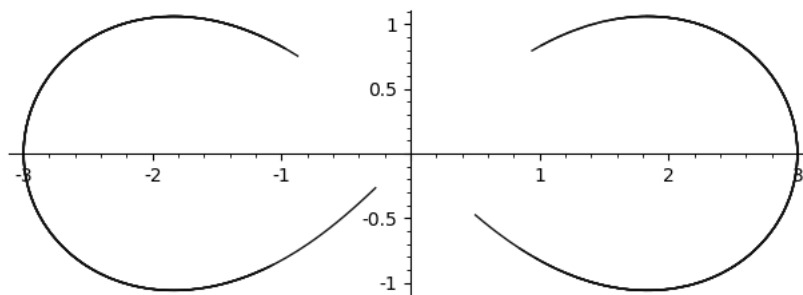
Al ejecutar estas instrucciones, el lector recibirá un mensaje similar al siguiente:

```
verbose 0 (3791: plot.py, generate_plot_points) WARNING: When plotting,
failed to evaluate function at 4999 points.
verbose 0 (3791: plot.py, generate_plot_points) Last error message: 'math
domain error'
```

Esta es la forma que tiene Sage de indicarnos que está teniendo problemas en determinar algunos puntos de la gráfica. Este mensaje puede ser ignorado. En la figura 3.1, podemos ver las gráficas producidas con 10 000 puntos (arriba) y con 200 puntos (abajo). En parte, el fenómeno es exagerado porque estamos usando $-6\pi < \theta < 6\pi$ como dominio. Podríamos haber elegido $-\pi < \theta < \pi$ en su lugar para tener más puntos por radián.



Una lemniscata graficada con 10 000 puntos.



Una lemniscata graficada con 200 puntos.

Figura 3.1 Una gráfica que requiere una gran cantidad de puntos

Nos preguntamos si existe alguna condición mediante la cual podemos saber, por adelantado, si este problema ocurrirá o no. De hecho, existe tal condición. La derivada $dr/d\theta$ no existirá en cualesquiera puntos P donde la curva sea paralela a la recta que conecta P con el origen. Llamamos a tales P s “puntos radialmente tangentes”. Ahí, la computadora debe graficar varios pixeles para un único valor de θ . Es por eso que el algoritmo no funciona bien bajo tales condiciones. La analogía en coordenadas rectangulares sería cuando la recta tangente es vertical. En tales puntos, dy/dx no existe y los gráficos frecuentemente se desvían ahí —sin importar si usamos una calculadora barata o una pieza sofisticada de software—. Vimos ejemplos de este fenómeno en la página 15, donde graficamos algunas funciones racionales.

3.4 Graficando una función implícita

En ocasiones escribimos las ecuaciones para curvas en un formato tal como

$$(x - 1)^2 + (y - 2)^2 = 9$$

en lugar de un formato como $y = f(x)$. La primera es llamada “función implícita” y la segunda es llamada “función explícita”. Este es un buen ejemplo, pues necesitaríamos escribir dos funciones explícitas:

$$f(x) = 2 + \sqrt{9 - (x - 1)^2}$$

$$g(x) = 2 - \sqrt{9 - (x - 1)^2}$$

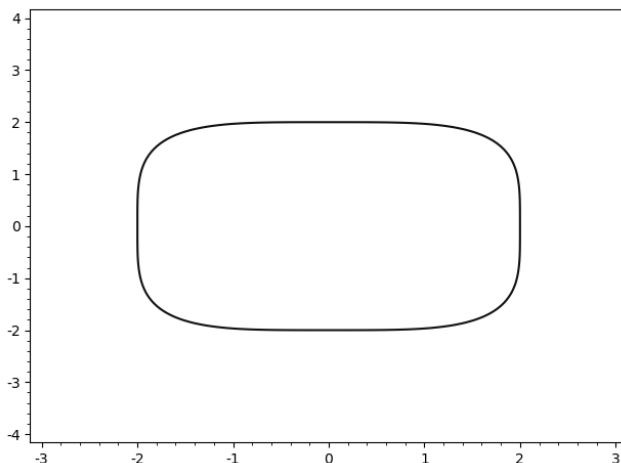
para representar la curva correspondiente. Por lo tanto, Sage nos da la comodidad de graficar funciones definidas en forma implícita.

Por ejemplo,

Código de Sage

```
1 g(x,y) = x^4 + y^4 - 16
2
3 implicit_plot(g, (x,-3,3), (y,-4,4))
```

es una curva de grado cuatro favorita entre los diseñadores de mobiliario para construir mesas para salas de conferencias. Como podemos ver, su fórmula matemática es implícita, $x^4 + y^4 = 16$. La gráfica (un poco alargada aquí) es:



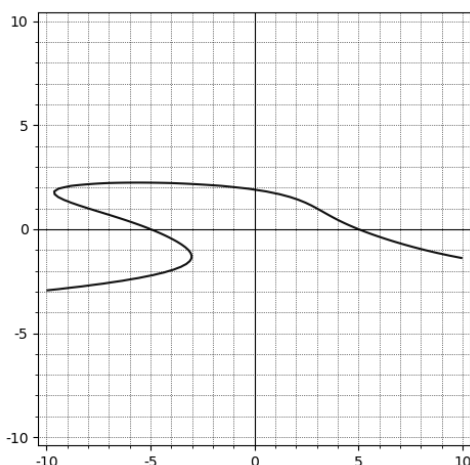
Para otras curvas, tales como

$$x^2 + y^5 + 5xy = 25,$$

la forma explícita sería incluso más tediosa de obtener, si no imposible. La gráfica, abajo, está dada por los comandos

Código de Sage

```
1 var('y')
2 implicit_plot( x^2 + y^5 + 5*x*y == 25, (x,-10,10), (y,-10,10),
  ↪ gridlines="minor", axes=True)
```



Como podemos apreciar, esta imagen muestra un agradable fondo similar al del papel gráfico, usando la opción `gridlines` que fue primero vista en la sección 1.4 en la página 9. La opción `axes=True` es necesaria para mostrar los ejes con el comando `implicit_plot`.

Tómese nota de que las funciones implícitas a tres variables (x, y, z) son discutidas en el apéndice únicamente electrónico en línea de este libro, “Graficando en color, en 3D, y animaciones”, disponible en la página web www.sage-para-estudiantes.com, para descarga gratuita.

Compatibilidad retrógrada La siguiente sintaxis hace que el comando `plot` luzca similar a la sintaxis de `implicit_plot`:

Código de Sage

```
1 plot(x^2, (x,-2,2))
```

donde usualmente hubiésemos escrito

Código de Sage

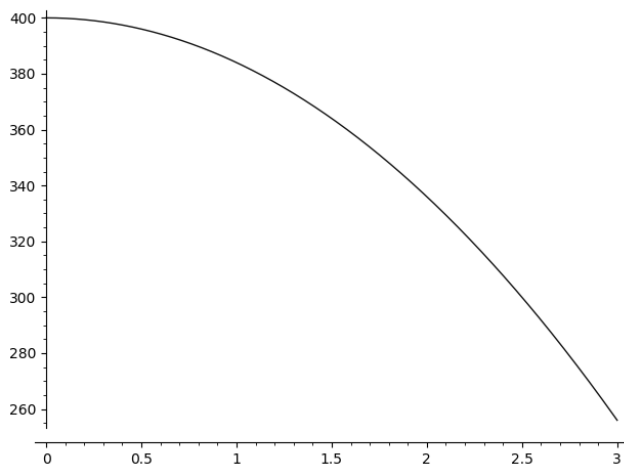
```
1 plot(x^2, -2, 2)
```

como en la sección original de graficación (sección 1.4 de la página 9). Esta notación es útil cuando se quiere graficar en términos de, digamos, t en lugar de x . Por ejemplo,

Código de Sage

```
1 var('t')
2 plot(-16*t^2+400, (t,0,3))
```

es la altura de una bola de boliche soltada desde un edificio de 400 pies de altura,¹ t segundos después de ser liberada. Aquí tenemos la gráfica:



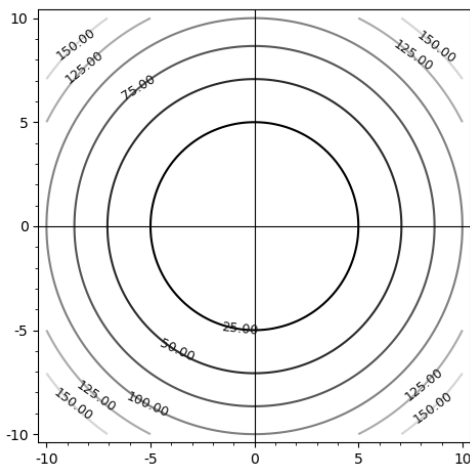
Esta misma notación, usando $(t, 0, 3)$ o $(x, -2, 2)$ para indicar $0 < t < 3$ y $-2 < x < 2$, respectivamente, será usada no solo para `plot` e `implicit_plot`, sino para muchos otros comandos de graficación que estudiaremos ahora.

3.5 Gráficas de contorno y conjuntos de nivel

Una forma clásica de visualizar una función a dos variables $f(x, y)$ es a través de un gráfico de contornos, el cual muestra una colección de “conjuntos de nivel” para la función. Un conjunto de nivel es el conjunto de puntos (x, y) tales que $f(x, y) = z$ para algún z fijo. Por ejemplo, si

$$f(x, y) = x^2 + y^2,$$

entonces la siguiente gráfica muestra los conjuntos de nivel para $z \in \{25, 50, 75, 100, 125, 150\}$.



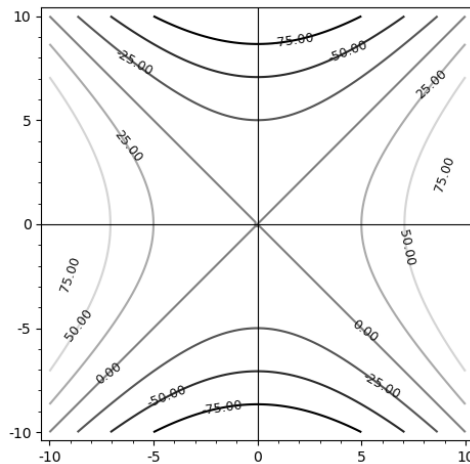
¹Como la altura está en pies, la aceleración gravitacional en las unidades correspondientes es 32 [pies/segundo²].

El código para producir esta imagen es

Código de Sage

```
1 var('y')
2 contour_plot(x^2+y^2, (x,-10,10), (y,-10,10), fill=False, axes=True,
  ↪ labels=True)
```

Aquí tenemos otra gráfica de contornos, para $g(x, y) = x^2 - y^2$. Para generar esta imagen, solo debemos cambiar el + por un - entre x^2 y y^2 en el código anterior.

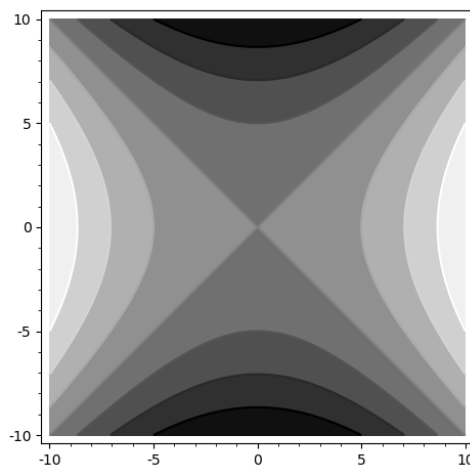


Estas son las gráficas de contorno anticuadas que podían encontrarse en textos durante el siglo XX. Las gráficas de contorno modernas usan sombreado, que las suelen hacer más fáciles de leer, pero a un costo de tinta extra. El siguiente código:

Código de Sage

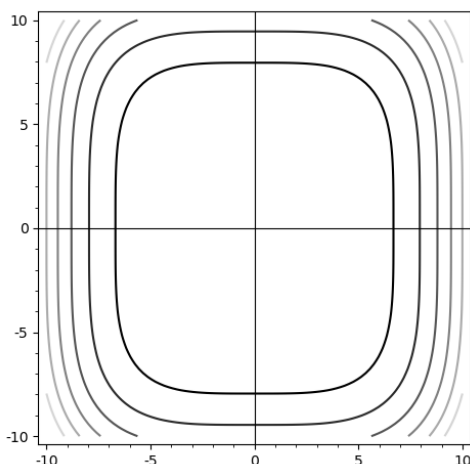
```
1 var('y')
2 contour_plot(x^2-y^2, (x,-10,10), (y,-10,10), plot_points=200)
```

produce una gráfica de la función previa, $g(x, y) = x^2 - y^2$, pero con un estilo sombreado.



En estos gráficos de contorno sombreados, las regiones más oscuras representan valores más pequeños de la función f , mientras que las regiones más claras indican valores más grandes de f . A propósito, la opción `plot_points` ayuda a Sage a deducir cuán cuidadosamente graficados se quieren los contornos. El autor suele usar valores de 200 o 300, por considerar que el valor por defecto de 100 es demasiado pequeño.

Para ser justos, debemos notar que en ocasiones el sombreado es más dañino que beneficioso. El siguiente diagrama de contorno es perfectamente legible sin sombreado:



El código que lo produjo es

Código de Sage

```
1 var('y')
2 contour_plot(x^4+y^4/2, (x,-10,10), (y,-10,10), fill=False, axes=True)
```

Por otro lado, si nos echamos un poco para atrás cambiando los valores de “fill=False” y “axes=True”, el sombreado resultante tendrá una gran región negra ocupando la parte central, dentro del óvalo más interno. Esto hace que la gráfica sea casi ilegible —el lector puede hacer la prueba si lo desea—.

Para poder explorar las relaciones entre algunos objetos tridimensionales y sus diagramas de contorno, el autor ha creado una página web interactiva (a veces llamada simplemente un “interactivo” o “applet”), que muestra varias formas complejas en 3D y sus gráficas de contorno simultáneamente. Esta puede encontrarse en la página web www.sage-para-estudiantes.com

3.5.1 Una aplicación a la termodinámica

El siguiente ejemplo está tomado del libro *Elementary Differential Equations and Boundary Value Problems*, de William Boyce y Richard DiPrima, 9na edición, publicado por Wiley en 2009. Es el ejemplo 1 de la sección 10.5, “Separación de variables: Conducción del calor en una barra”. Este ejemplo extendido requerirá algunas técnicas de las secciones 4.1 y 4.19. Es el deseo sincero del autor que este proyecto sea comprensible sin la necesidad de estudiar esas secciones a mucho detalle, pero si el lector lo encuentra confuso, puede ser beneficioso leerlas. Antes de iniciar, es necesario que el autor se disculpe por el hecho de que la tipografía aquí no le permitió incluir unidades con las constantes en todos los casos, como suele ser práctica común entre los físicos.

La función $u(x, t)$ abajo nos indica la temperatura u en el tiempo t y en la posición x dentro de una barra. Esta tiene 50 [cm] de longitud, está aislada en el exterior y se encuentra inicialmente a una temperatura de 20°C o 68°F. Los extremos son abruptamente expuestos a una temperatura de 0°C o 32°F en $t = 0$ y mantenidos así para $t > 0$.

Como podemos ver, la función está definida por una serie:

$$u(x, t) = \frac{80}{\pi} \sum_{n=1,3,5,\dots}^{\infty} \frac{1}{n} e^{-n^2 \pi^2 \alpha^2 t / 2500} \sin \frac{n \pi x}{50},$$

pero la variable n de esta serie varía sobre los enteros impares positivos. En su lugar, definamos $n = 2j + 1$. Así, conforme j toma los valores 0, 1, 2, 3, ..., entonces n tomará los valores 1, 3, 5, 7, ..., produciendo todos los enteros impares positivos. Esto nos permitirá usar el comando `sum` de Sage. Más aun, dejaremos que j varíe

desde 0 hasta 100. Después de todo, cuando $j = 31$, tendremos $n = 63$, y ese término contiene $e^{-63^2\pi^2}$, que hace que sea totalmente despreciable. En realidad, podríamos detenernos en $j = 10$ o $j = 20$ y probablemente producir la misma gráfica, pero a las computadoras no les importa hacer muchos cálculos tediosos, así que nos comprometemos en $j = 100$. Boyce y DiPrima usan $\alpha = 1$ y nosotros haremos lo mismo. Ahora tenemos

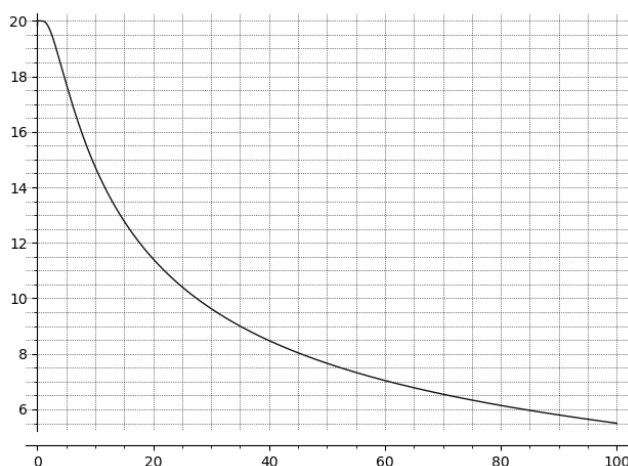
$$u(x, t) = \frac{80}{\pi} \sum_{j=0}^{n=\infty} \frac{1}{2j+1} e^{-(2j+1)^2\pi^2 t/2500} \sin \frac{(2j+1)\pi x}{50}.$$

Dado que esta es una función a dos variables, una forma en la que podríamos visualizarla es usando “media evaluación”. Graficaríamos $u(5, t)$ para $0 < t < 100$ con objeto de tener una idea de cómo se comporta la temperatura en el tiempo, a 5 [cm] desde cualquier extremo de la barra. Para lograr esto, escribimos

Código de Sage

```
1 var('j t')
2 u(x, t) = N(80/pi) * sum(1/(2*j+1) * exp(-(2*j+1)^2*N(pi^2/2_500)*t) *
   ↪ sin((2*j+1)*N(pi/50)*x), j, 0, 100)
3 plot(u(5,t), (t, 0, 100), gridlines='minor')
```

Esto produce la siguiente gráfica:



Unos cuantos comentarios rápidos están a la orden. Hemos usado la función $N()$ dondequiera que tenemos la constante π para forzar a Sage a evaluar la expresión numéricamente, y no de forma exacta² —de lo contrario, la expresión algebraica exacta es demasiado compleja y la computadora puede tomarse mucho tiempo para hacer el cálculo, o fallar en hacerlo—. Por otro lado, $u(5, t)$ es un ejemplo de una media evaluación (véase la página 123). Finalmente, la sintaxis para sumar una serie está dada en la página 174.

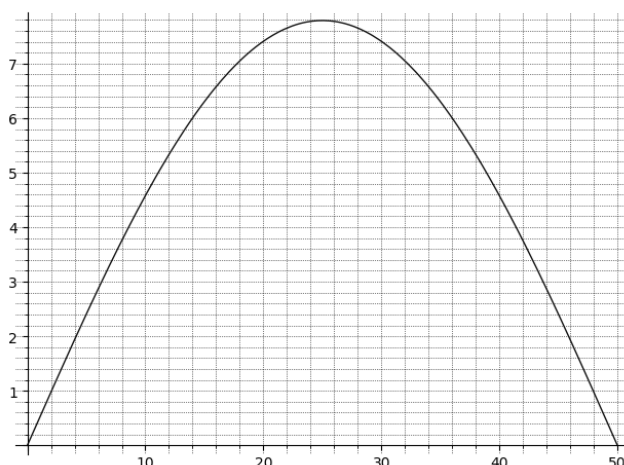
Alternativamente, podríamos graficar $u(x, 300)$ para tener una idea de cómo será la temperatura a través de la barra a los 300 segundos, o 5 minutos, después que el enfriado ha empezado. Podríamos reemplazar la última línea del código anterior con

Código de Sage

```
1 plot(u(x, 300), (x, 0, 50), gridlines='minor')
```

²Como vimos en muchas ocasiones, Sage se especializa en realizar cálculos exactos. Una manera de forzar aproximaciones decimales es usar la función $N()$, la cual estudiamos en la página 4; otra manera es introducir un decimal en algún lugar, como al escribir $\sqrt{8.0}$ en lugar de $\sqrt{8}$ (véase también la página 4). Sin embargo, cuando trabajamos con π , esta última técnica no funciona. En efecto, $80.0/\pi$ devuelve $80.0000000000000/\pi$ en lugar de $25,464\,790\,894\,703\,3$. Esto se debe a un fenómeno conocido como *coerción*, que no estudiaremos en este libro. Si el lector no desea usar $N(\pi)$, puede usar $RR.\pi()$ en su lugar.

Esto produce la imagen

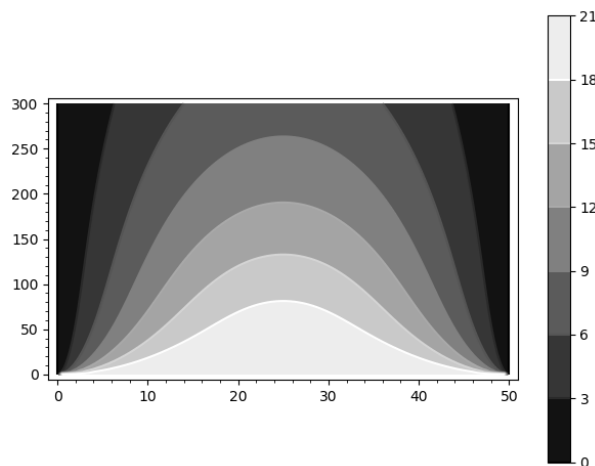


Sin embargo, una gráfica de contornos es mucho más poderosa que cualquiera de estas dos alternativas. Si tenemos una gráfica de contornos y graficamos una línea vertical, observando a lo largo de esta veremos cómo varía la temperatura con respecto al tiempo en un punto específico de la barra. Si graficamos una línea horizontal, observando a lo largo de esta veremos cómo varía la temperatura a través de la barra, en un tiempo específico. Para obtener la gráfica de contornos, reemplazamos la última línea del código anterior con

Código de Sage

```
1 contour_plot(u(x, t), (x, 0, 50), (t, 0, 300), aspect_ratio=0.1,
  ↪ colorbar=True)
```

Esto resulta en la siguiente imagen:



Como podemos ver, una “barra de color” o leyenda de escala de grises ha sido añadida a la derecha de la gráfica para ayudarnos a identificar qué indica cada tono de gris, en términos de grados Celsius. Esto resulta bastante útil aquí en termodinámica, pero no hubiese sido muy relevante en los ejemplos de matemáticas puras que vimos antes en esta sección.

Por último, pero no menos importante, la opción `aspect_ratio` requiere una explicación. Cuando graficamos objetos geométricos, tales como las funciones a dos variables que vimos momentos atrás, es importante que la misma escala sea usada en los ejes x y y , caso contrario, los ángulos estarían desviados y otras distorsiones ocurrirían. Si removiésemos la opción `aspect_ratio=0.1` en este caso, Sage observaría que estamos graficando 300 unidades en una variable y 50 en la otra. Por lo tanto, la gráfica sería seis veces más estrecha de

lo que es alta, para mantener la escala. La mejor forma de apreciar esto es borrar el `aspect_ratio=0.1` y presionar “Evaluate”. En cambio, con esta opción, el 50 en el eje x se interpreta como 500 y obtenemos una gráfica que es 1,6 veces más ancha de lo que es alta. Esta es una gráfica mucho más fácil de leer. No debe preocuparnos hacer este cambio, pues 50 [cm] y 300 [s] son unidades totalmente diferentes, así que la noción de “la misma escala” carece de sentido.

3.5.2 Aplicación a la microeconomía (ecuaciones de Cobb-Douglas)

El siguiente es un ejemplo de una función de producción de Cobb-Douglas,

$$f(K, L) = 1400K^{0.51}L^{0.32},$$

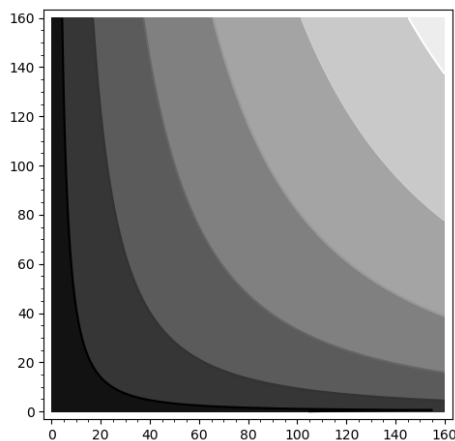
que fue derivada a partir de datos hipotéticos de un pequeño negocio que produce tubería de concreto. El modelo fue encontrado en *Linear and Non-Linear Programming with Maple: An Interactive, Applications-Based Approach*, por Paul Fishback, publicado por CRC Press en 2010. La función aparece en varios lugares a lo largo del libro, pero es derivada a partir de datos como el ejemplo principal de la sección 7.3.5, “Non-Linear Regression”.

Usando el código

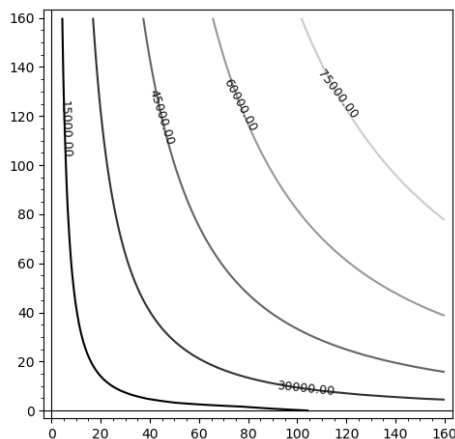
Código de Sage

```
1 var('y')
2 contour_plot(1_400*x^0.51*y^0.32, (x, 0, 160), (y, 0, 160),
  ↪ plot_points=300)
```

obtenemos la siguiente gráfica de contornos sombreados, que es moderadamente informativa.



Sin embargo, un gráfico más legible sin sombreado, como el siguiente:



puede ser producido por el código:

Código de Sage

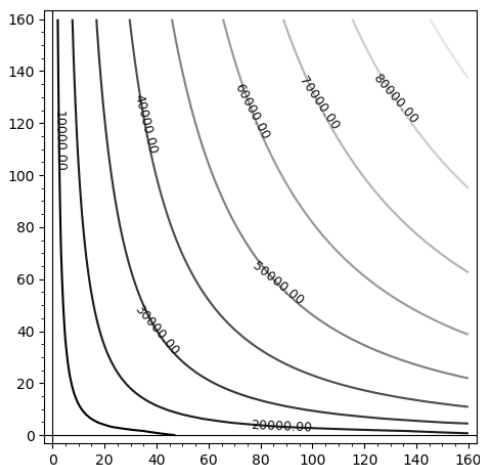
```
1 var('y')
2 contour_plot(1_400*x^0.51*y^0.32, (x, 0, 160), (y, 0, 160), fill=False,
  ↪ axes=True, labels=True)
```

Forzando valores específicos En ocasiones es útil forzar que Sage elija líneas de contorno para valores particulares. Por ejemplo, si ciertos valores de una función son muy importantes o son requeridos para algún trabajo, entonces uno querrá indicar a Sage estos zs específicos que deben ser usados para los conjuntos de nivel. Aquí tenemos un ejemplo.

Código de Sage

```
1 var('y')
2 contour_plot(1_400*x^0.51*y^0.32, (x, 0, 160), (y, 0, 160), fill=False,
  ↪ axes=True, labels=True, contours=[10_000, 20_000, 30_000, 40_000,
  ↪ 50_000, 60_000, 70_000, 80_000, 90_000, 100_000])
```

Esto produce la siguiente imagen:



3.6 Gráficas paramétricas 2D

Los matemáticos usan el término “función paramétrica” para referirse a situaciones en las que la coordenada x de un punto está dada como $f(t)$, mientras que la coordenada y está dada como $g(t)$. Por lo tanto, para cualquier punto en un tiempo t , conocemos ambas coordenadas. Esto es muy útil para describir tipos más avanzados de movimiento en física, incluyendo mecánica celeste —el movimiento de planetas, asteroides y cometas—.

Un ejemplo realmente atractivo se refiere a las Curvas de Lissajous, nombradas en honor a Jules Antoine Lissajous (1822–1880), aunque en realidad fueron descubiertas por Nathaniel Bowditch (1773–1838). Sin embargo, dado que Lissajous realizó un estudio detallado de estas, fueron nombradas en su honor. Estas curvas son de la forma $x = \sin(at + b)$ y $y = \sin(ct + d)$, con a y c usualmente restringidas a los enteros, aunque algunos textos permiten una constante multiplicativa en frente de cada seno, o valores racionales para a y c . En cualquier caso, si $a = c$, entonces no tenemos más que una elipse o circunferencia. Sin embargo, $a \neq c$ produce figuras mucho más interesantes. Aquí tenemos el código para el caso $a = 2$ y $c = 3$:

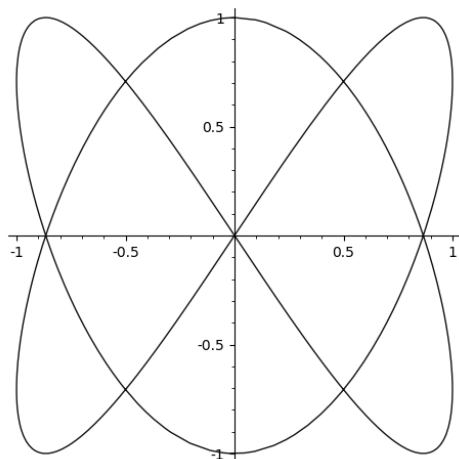
Código de Sage

```

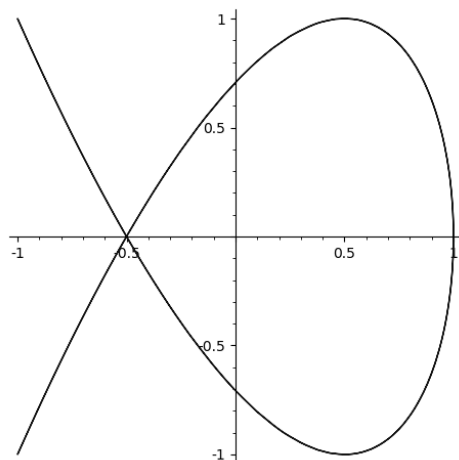
1  var('y t')
2
3  f(t) = sin(2*t)
4  g(t) = sin(3*t)
5
6  parametric_plot((f(t), g(t)), (t, 0, 2*pi))

```

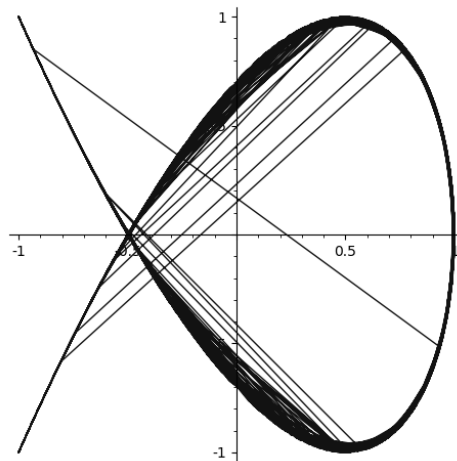
Este produce la siguiente imagen:



Las curvas de Lissajous son muy importantes para afinar osciloscopios, así como sistemas de sonido. Si cambiamos $f(t)$ por $\cos(2t)$, obtenemos una figura diferente:



El fenómeno que vimos con coordenadas polares en la página 99, donde teníamos muy pocos puntos por radián, también puede ocurrir con las ecuaciones paramétricas. Consideremos cambiar el dominio a $-100 < t < 100$ en el ejemplo anterior, es decir, cambiemos $(t, 0, 2\pi)$ por $(t, -100, 100)$. La salida resultante tiene una calidad muy pobre, pues los puntos están muy separados. Dado que el dominio tiene longitud 200 en unidades de t y la cantidad por defecto de puntos para la curva es 200, cada punto usado para graficar está a una unidad de t de distancia. Cuando nuestro intervalo tenía longitud 2π , los puntos estaban $\pi/100 \approx 0,0314159$ unidades de t aparte, una considerable diferencia.



Es un excelente ejercicio para los estudiantes tratar de convertir la forma paramétrica de una función en una función implícita —si es posible—; sin embargo, para algunos casos, no es posible (usando solamente funciones elementales). También, el cálculo de curvas paramétricas es muy directo, pero significativamente distinto al de funciones explícitas. La mayoría de los textos de cálculo cubren el tema de funciones paramétricas, pero algunos profesores lo omiten —lo que puede ser un problema para estudiantes de ingenierías o física, que deben lidiar con este tópico en cursos superiores—. Si el lector desea aprender más, recomendamos las secciones 9.2 y 9.3 de *Calculus with Early Transcendentals*, por Soo Tan, 1ra edición, publicado por Cengage en 2011.

3.7 Gráficas de campos vectoriales

Antes que nada, sería buena idea recordar cómo luce una gráfica de un campo vectorial. Aquí tenemos un ejemplo, donde estamos graficando la siguiente función con valores vectoriales:

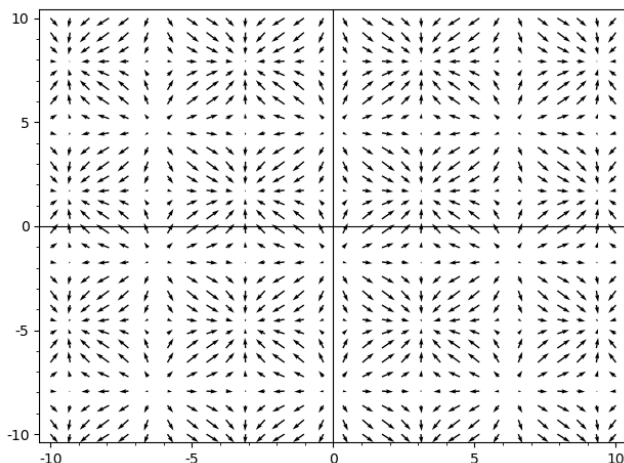
$$\vec{f}(x, y) = \begin{bmatrix} \sin(x) \\ \cos(y) \end{bmatrix}.$$

Para construir la gráfica correspondiente, escribimos

Código de Sage

```
1 var('y')
2 plot_vector_field((sin(x), cos(y)), (x, -10, 10), (y, -10, 10),
  ↪ plot_points=30)
```

Notemos que este es otro ejemplo de cómo Sage sabe que x es una variable, pero debemos especificarle cualquier otra, declarándola con el comando `var`. Aquí estamos indicando que el valor x de nuestra función es $\sin(x)$ y que el valor y es $\cos(y)$. La notación $(x, -10, 10)$ significa que el rango en x que deseamos es $-10 < x < 10$, y naturalmente, $(y, -10, 10)$ indica $-10 < y < 10$. El parámetro `plot_points` le dice a Sage cuántos vectores dibujar en cada fila y columna del gráfico. De seguro que estaremos de acuerdo que construir este gráfico a mano, con 900 flechas independientes, sería excepcionalmente tedioso.



De la misma forma en que $\vec{f}(x, y)$ devuelve un vector por cada (x, y) , tenemos (en un puñado de puntos del plano coordenado) una flecha, representando el vector. La dirección es fielmente reproducida y la longitud de la flecha es una representación aproximada de la magnitud del vector.

¿Cómo podemos interpretar tal gráfico? Una forma es pensar que si un pequeño objeto estuviese localizado en el plano coordenado, en cada instante del tiempo se movería en la dirección apuntada por la flecha bajo el objeto. Las únicas suposiciones son que su masa y radio son diminutos. Otra interpretación involucra gradientes, que exploraremos ahora.

3.7.1 Gradientes y gráficos de campos vectoriales

Una razón para hacer una gráfica de un campo vectorial es para estudiar el gradiente de una función. Podemos ver que si

$$g(x, y) = \sin(y) - \cos(x),$$

entonces las derivadas parciales son

$$\frac{\partial g}{\partial x} = \sin(x) \quad \text{y} \quad \frac{\partial g}{\partial y} = \cos(y),$$

lo que haría el gradiente

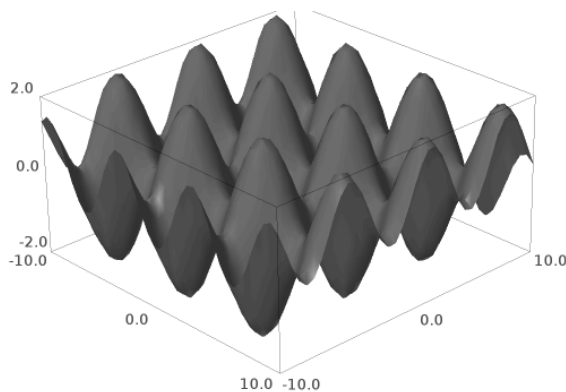
$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial x} \\ \frac{\partial g}{\partial y} \end{bmatrix} = \begin{bmatrix} \sin(x) \\ \cos(y) \end{bmatrix} = \vec{f}(x, y).$$

Como podemos ver, esta es la misma función con valores vectoriales que recién graficamos.

Si deseamos visualizar $g(x, y)$ como la coordenada z para (x, y) , podemos producir una gráfica 3D usando

Código de Sage

```
1 g(x,y) = sin(y) - cos(x)
2
3 plot3d(g, (x, -10, 10), (y, -10, 10))
```

Obtenemos un objeto con un patrón muy parecido a un maple de huevos.

Ahora veamos el campo vectorial nuevamente. Hay zonas en las que un puñado de flechas están apuntado lejos de un punto particular. En una interpretación intuitiva, si pusiésemos una canica en la vecindad de uno de esos puntos, esta sería atraída hacia él y se quedaría ahí. Estos son los puntos donde ∇g es el vector cero, y corresponden a mínimos locales del maple. También hay zonas donde todas las flechas apuntan hacia un mismo punto. Si pusiésemos una canica en la vecindad de uno de esos puntos, esta sería repelida lejos de él y nunca volvería. Esos puntos también hacen que ∇g sea cero y corresponden a máximos locales del maple. Un matemático o físico aplicado diría que estos son puntos de equilibrio atractivos y repulsivos, para los casos de los mínimos y máximos locales, respectivamente.

¿Cómo podemos distinguir entre un mínimo y un máximo local? Uno podría hacer que Sage genere una gráfica de campo vectorial y entonces realizar la identificación visualmente. O, podría considerarse la matriz Hessiana (en este caso, una de orden 2×2 formada por las derivadas parciales) y usar el Criterio de Sylvester. Claramente, una tarea es considerablemente más difícil que la otra.

3.7.2 Una aplicación de la física

Imaginemos tres cargas eléctricas en el plano cartesiano. La Carga 1 se encuentra fija en el origen y la Carga 2 está fija en el punto $(2, 0)$. Una tercera carga es móvil, en el punto (x, y) . Tal vez nos gustaría conocer la fuerza total ejercida por el campo eléctrico en la tercera. Para hacer las cosas más interesantes, imaginemos que las cargas son $+2$, $+1$ y -1 , respectivamente. El hecho que sean desiguales hará más interesante el problema. Una forma fantástica de visualizar este escenario es con una gráfica de campo vectorial.

Antes de continuar, recordemos brevemente la Ley de Coulomb. La fuerza \vec{F} (en Newtons) de una carga de q_0 Coulombs sobre una de q_1 Coulombs está dada por

$$\vec{F} = -\left(\frac{1}{4\pi\epsilon_0}\right)\left(\frac{q_1q_0}{\|\vec{r}\|^3}\right)\vec{r},$$

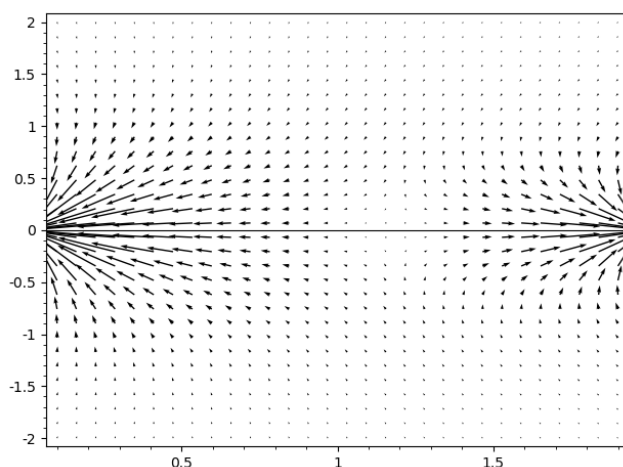
donde \vec{r} es el vector desde q_1 hasta q_0 , ϵ_0 es una constante (llamada “permitividad del vacío”) y π tiene el significado usual. Algunos textos definen \vec{r} como el vector desde q_0 hasta q_1 , en cuyo caso eliminaríamos el signo menos en frente del lado derecho. (A propósito, la notación $\|\vec{r}\|$ significa la magnitud, longitud o norma del vector \vec{r} .)

Algunos lectores estarán sorprendidos de ver el cubo en el denominador. Este se debe a que estamos trabajando con vectores; si quisiésemos trabajar con escalares solamente, es decir los números reales que expresan la magnitud de la fuerza, tendríamos en su lugar

$$\|\vec{F}\| = \left|-\frac{1}{4\pi\epsilon_0}\right|\left(\frac{|q_1||q_0|}{\|\vec{r}\|^3}\right)\|\vec{r}\| = \frac{1}{4\pi\epsilon_0}\left(\frac{|q_1||q_0|}{\|\vec{r}\|^2}\right),$$

dándonos la familiar ley del cuadrado inverso.

Ahora estamos listos para analizar el código en la figura 3.2, que fue usado para generar el siguiente gráfico.



Código de Sage

```

1  var('y')
2
3  x1 = 0
4  y1 = 0
5  x2 = 2
6  y2 = 0
7
8  r_vec_1 = vector([x1-x, y1-y])
9  r_vec_2 = vector([x2-x, y2-y])
10
11  q0 = -1
12  q1 = 2
13  q2 = 1
14
15  f1 = -q0*q1*r_vec_1 / r_vec_1.norm()^3
16  f2 = -q0*q2*r_vec_2 / r_vec_2.norm()^3
17
18  f_neta = f1 + f2
19
20  plot_vector_field(f_neta, (x, 0.1, 1.9), (y, -2, 2), plot_points=30)

```

Figura 3.2 El código para la aplicación de física a gráficas de campos vectoriales

Primero, definimos x_1 y y_1 como las coordenadas de la Carga 1. Segundo, definimos x_2 y y_2 como las coordenadas de la Carga 2. Resulta útil tener estos valores aislados al principio del programa (en lugar de sustituirlos directamente en las fórmulas más adelante), pues esto nos permitirá experimentar con el modelo al modificar ligeramente los valores.

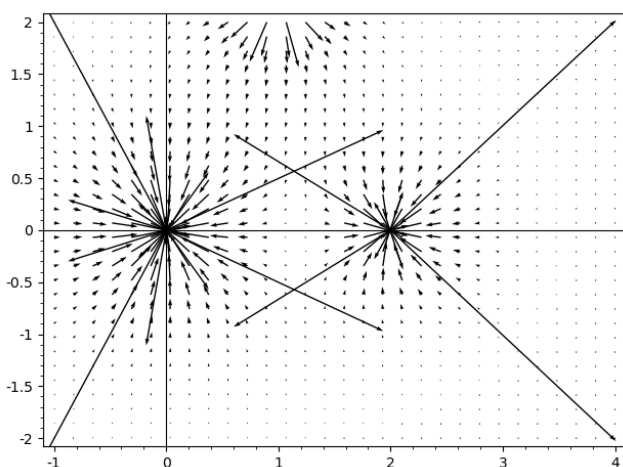
Tercero, definimos dos vectores: r_vec_1 va de (x, y) a (x_1, y_1) y r_vec_2 va de (x, y) a (x_2, y_2) . Cuandoquiera que tengamos que definir un vector, ya sea en Sage o cualquier otra situación matemática, es conveniente recordar la regla “cabeza menos cola”: si visualizamos r_vec_1 como una flecha, entonces su cabeza estaría en (x_1, y_1) y su cola en (x, y) .

Cuarto, definimos q_0 , q_1 y q_2 como las cargas de la que es móvil, la número 1 y la 2, respectivamente. Quinto, aplicamos la Ley de Coulomb. Para la fuerza entre la Carga 1 y la que es móvil, definimos f_1 , y hacemos lo equivalente con f_2 . Como podemos ver, hemos despreciado todas las constantes, pues estas no afectarán las direcciones de los vectores.³

Sexto, f_{neta} es la suma de las fuerzas, dándonos la fuerza total sobre la carga móvil. Nótese que no queremos calcular la fuerza de la Carga 1 sobre la Carga 2, o viceversa, pues estas no son relevantes para la que es móvil.

Finalmente, usamos el comando `plot_vector_field` para graficar el campo vectorial para $0,1 \leq x \leq 1,9$ y $-2 \leq y \leq 2$.

Como Ícaro, ¡no nos acerquemos demasiado! Aunque las cargas están ubicadas en $(0, 0)$ y $(2, 0)$, el lector observará que deliberadamente hemos ajustado los valores x para excluir $x = 0$ y $x = 2$, así como sus vecindades inmediatas. Si cambiamos $(x, 0.1, 1.9)$ a $(x, -1, 4)$ en la última línea del código de la figura 3.2, veremos que aparecen unos vectores gigantesco extraños que parecen indicar conclusiones disparatadas. En particular, parece como si algunos de estos apuntaran en la dirección equivocada.



Lo que en realidad ocurre aquí es que, cuando (x, y) se aproxima demasiado a una de las cargas fijas, la distancia se hace muy pequeña. Esto significa que el denominador, que está elevado al cubo, se hace extremadamente diminuto, haciendo que la fuerza sea enorme. El vector entonces es tan largo que, aunque empieza a un lado de la carga, termina ocupando mucho más espacio de lo que debería, pasando por encima de esta y más allá hasta el otro lado, y eso crea el resultado visualmente engañoso. En una gráfica de campo vectorial de Sage, la cola de la flecha representa el punto sobre el cual actúa. En este caso, la cola de estas flechas problemáticas se encuentra más allá de la carga, en dirección contraria al vector, pero los ojos parecen engañarnos haciéndonos pensar que parten de la carga misma. Normalmente, esto no ocurre muy seguido.

Matemáticamente hablando, $(0, 0)$ y $(2, 0)$ son los polos de la función de fuerza total, pues hay división por cero ahí. En general, obtendremos una gráfica de campo vectorial mucho mejor si excluimos los polos de la región siendo dibujada.

Un desafío El lector ahora está listo para abordar el miniproyecto sobre graficación de campos eléctricos, que está planteado en la página 87.

³Es por esta misma razón que ni siquiera consideramos las unidades en este problema —no afectarán las direcciones de los vectores—.

3.7.3 Gradientes vs. gráficas de contorno

El siguiente ejemplo muestra elegantemente la relación entre gráficas de contorno y gráficas de campo vectorial del gradiente. Este fue creado por Augustine O'Keefe. Consideremos la función

$$f(x, y) = \cos(x) - 2 \sin(y)$$

y su gradiente, que es

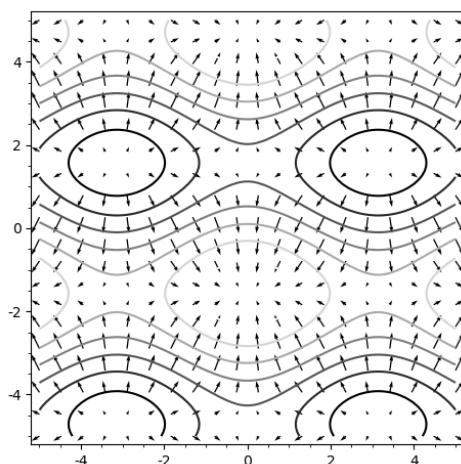
$$\nabla f(x, y) = \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right\rangle = \langle -\sin(x), -2 \cos(y) \rangle.$$

Graficaremos los conjuntos de nivel de $f(x, y)$, así como el campo vectorial del gradiente, $\nabla f(x, y)$, pero superpuestos en la misma imagen. El código que usamos es el siguiente:

Código de Sage

```
1 f(x, y) = cos(x) - 2*sin(y)
2 gradient = derivative(f)
3
4 P1 = plot_vector_field(gradient, (x, -5, 5), (y, -5, 5))
5
6 P2 = contour_plot(f, (x, -5, 5), (y, -5, 5), fill=False)
7
8 show(P1 + P2)
```

Este produce la siguiente salida:



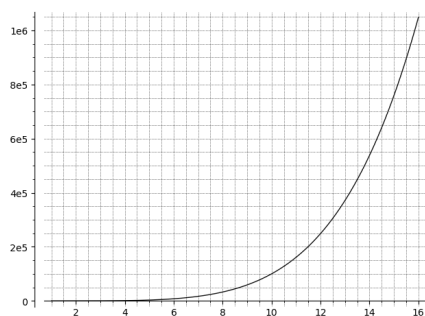
Aquí podemos apreciar cómo lucen las colinas y valles (la gráfica de contornos), casi del mismo estilo que los mapas de contorno que los geólogos suelen usar. Las flechas, representando el gradiente, indican la dirección de más rápido ascenso. Si multiplicamos el vector por -1 , lo que invierte la dirección (intercambiando la cabeza y la cola de la flecha), entonces eso representaría la dirección de más rápido descenso — la dirección que una gota de lluvia tomaría, moviéndose desde terreno alto hasta más bajo. Podemos ver claramente dónde ocurriría la inundación en caso de una tormenta masiva de lluvia. Por otro lado, definitivamente podemos ver que el vector gradiente siempre es perpendicular al conjunto de nivel correspondiente.

3.8 Gráficas log-log

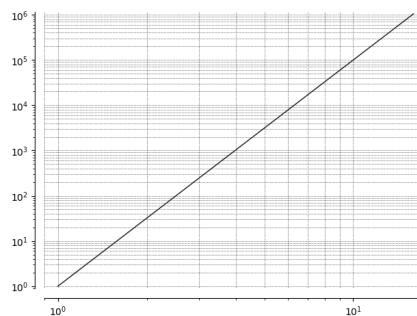
A mediados del siglo XX era muy común el “papel log-log”, disponible para compra en las librerías universitarias. Este es papel para graficar en el que las líneas del cuadrículado estaban espaciadas de una forma peculiar: cada línea horizontal y vertical es una escala logarítmica. Por ejemplo, en lugar que un centímetro particular representara $x = 1$ a $x = 2$ dividido en pasos tal vez $1/10$ (es decir, $1; 1,1; 1,2; 1,3; \dots; 2$), este podía representar $x = 10$ a $x = 100$. Las subdivisiones indicarían $10^{1,1}$, $10^{1,2}$, $10^{1,3}$, ..., 10^2 . El siguiente centímetro representaría $x = 100$ a $x = 1000$, mientras que el previo mostraría $x = 1$ a $x = 10$.

La primera pregunta que se nos ocurre es “¿para qué se hacía esto?”. El propósito de una gráfica log-log es permitir los cálculos de leyes de potencia por medio de regresión, o tal vez simplemente exhibir una ley de potencia. Por ejemplo, el volumen de la esfera es proporcional al cubo de su radio, la distancia de frenado de un vehículo es proporcional al cuadrado de su velocidad, la energía de retorno de una señal de radar es inversamente proporcional a la cuarta potencia de la distancia al objetivo; muchos otros ejemplos existen. Aunque c y n pueden variar según el caso, todas estas relaciones son de la forma $y = cx^n$.

En la era de las calculadoras graficadoras y los paquetes de álgebra computacional, el papel log-log es extremadamente difícil de encontrar. Sin embargo, Sage puede fácilmente crear gráficas al estilo log-log. Primero, ilustremos el efecto al considerar $y = x^5$, que es claramente una ley de potencia con $c = 1$ y $n = 5$.



Coordenadas ordinarias



Coordenadas log-log

El gráfico de la izquierda fue hecho con

Código de Sage

```
1 plot(x^5, (x, 1, 16), gridlines='minor')
```

y el de la derecha con

Código de Sage

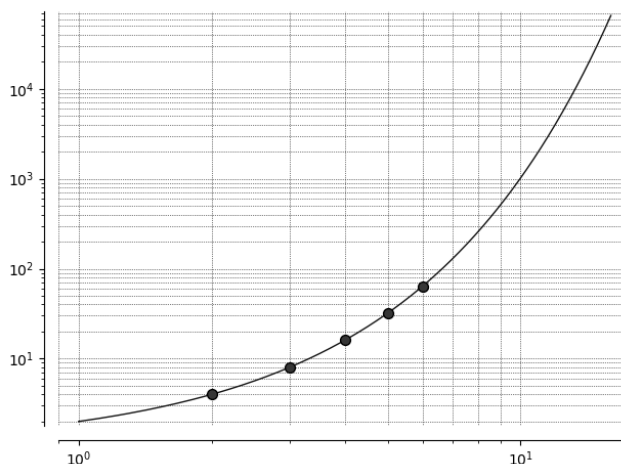
```
1 plot_loglog(x^5, (x, 1, 16), gridlines='minor')
```

El siguiente código grafica la función $y = 2^x$ en escala log-log, junto con los cinco puntos de datos del conjunto $\{(2, 4), (3, 8), (4, 16), (5, 32), (6, 64)\}$:

Código de Sage

```
1 plot_loglog(2^x, (x, 1, 16), gridlines='minor') + scatter_plot([(2,4),
↪ (3,8), (4,16), (5,32), (6,64)], facecolor='red')
```

El resultado obtenido es



Para más información acerca del comando `scatter_plot`, véase la sección 4.9 en la página 151.

Dado que esta transformación del espacio bidimensional ordinario (es decir \mathbb{R}^2) al plano log-log ha sido tan útil, uno puede preguntarse existe algo análogo para el espacio tridimensional (es decir \mathbb{R}^3). Y la hay; es llamado el espacio log-log-log. Este suele hacer presencia en la derivación de las fórmulas tipo Cobb-Douglas en macroeconomía. En general, esas ecuaciones son de la forma

$$P = cK^aL^b,$$

donde P es la producción, K es el capital invertido y L es la mano de obra invertida. Los coeficientes a , b y c se encuentran mediante un plano de mejor ajuste (el equivalente tridimensional de una línea de mejor ajuste), pero en el espacio log-log-log en lugar del espacio tridimensional ordinario. Véase la subsección 3.5.2 en la 107 para un ejemplo de una función de Cobb-Douglas derivada a partir de datos.

3.9 Situaciones raras

Aquí presentamos dos situaciones raras que puede que surjan o no en nuestro trabajo diario. La primera es un defecto necesario en la forma en que Sage maneja las raíces impares de números negativos. La otra tiene que ver con funciones que tienen un valor x faltante en sus dominios.

Fundamentos sobre raíces cúbicas Cuando decimos que $\sqrt[3]{64} = 4$, ¿qué es lo que realmente queremos expresar? Por supuesto, esta es solo una forma de escribir el hecho que $4^3 = 64$. Sin embargo, cuando los números complejos hacen su aparición en escena, las cosas se ponen un poco más complicadas.

Si preguntamos por la raíz cúbica de 8, hay tres números z en el plano complejo tales que $z^3 = 8$. Específicamente, estos son

$$z = 2 \cos(0^\circ) + i2 \sin(0^\circ) = 2 + 0i = 2,$$

$$z = 2 \cos(120^\circ) + i2 \sin(120^\circ) = -1 + i\sqrt{3} \text{ y}$$

$$z = 2 \cos(240^\circ) + i2 \sin(240^\circ) = -1 - i\sqrt{3}$$

Tomémonos un momento para verificar este último resultado elevándolo cuidadosamente al cubo:

$$\begin{aligned}
 (-1 - i\sqrt{3})^3 &= (-1 - i\sqrt{3})(-1 - i\sqrt{3})^2 \\
 &= (-1 - i\sqrt{3})(1 + 2i\sqrt{3} + i^2 3) \\
 &= (-1 - i\sqrt{3})(1 + 2i\sqrt{3} - 3) \\
 &= (-1 - i\sqrt{3})(-2 + 2i\sqrt{3}) \\
 &= (-1)(-2) + (-1)(2i\sqrt{3}) + (-i\sqrt{3})(-2) + (-i\sqrt{3})(2i\sqrt{3}) \\
 &= 2 - 2i\sqrt{3} + 2i\sqrt{3} - 2i^2(3) \\
 &= 2 - (2)(-1)(3) \\
 &= 2 + 6 = 8.
 \end{aligned}$$

De manera similar, podemos verificar el segundo resultado. Por lo tanto, el conjunto de números complejos cuyo cubo es 8 resulta ser

$$\{2, -1 + i\sqrt{3}, -1 - i\sqrt{3}\}.$$

Sin embargo, de estos tres, solamente el primero es real —usualmente ese es “el que queremos”—.

De manera similar, si preguntamos por la raíz cúbica de -8 , hay tres números z en el plano complejo tales que $z^3 = -8$. Específicamente, estos son

$$z = -2 \cos(0^\circ) - i2 \sin(0^\circ) = -2 - 0i = -2,$$

$$z = -2 \cos(120^\circ) - i2 \sin(120^\circ) = 1 - i\sqrt{3} \text{ y}$$

$$z = -2 \cos(240^\circ) - i2 \sin(240^\circ) = 1 + i\sqrt{3}$$

Por lo tanto, el conjunto de los números complejos cuyo cubo es -8 resulta ser

$$\{-2, 1 - i\sqrt{3}, 1 + i\sqrt{3}\}$$

y, como podemos ver, este es solo el conjunto previo con sus elementos multiplicados por -1 . Nuevamente, de estas tres soluciones, solo la primera es real, y esa es la que usualmente buscamos.

Es notable que podamos obtener las tres soluciones del primer caso con el siguiente comando:

Código de Sage

```
1 solve(x^3 == 8, x)
```

que, por supuesto, también funciona para el segundo caso con $== -8$.

El problema es que en ciertos temas extremadamente importantes, incluyendo campos de números algebraicos y campos finitos (ramas del álgebra abstracta), es necesario que Sage retorne por lo menos una de las dos raíces complejas cuando por ejemplo escribimos

Código de Sage

```
1 print(N((-8)^(1/3)))
```

aun cuando usualmente deseamos la raíz real. Esto significa que si escribimos

Código de Sage

```
1 plot(x^(1/3), -5, 5)
```

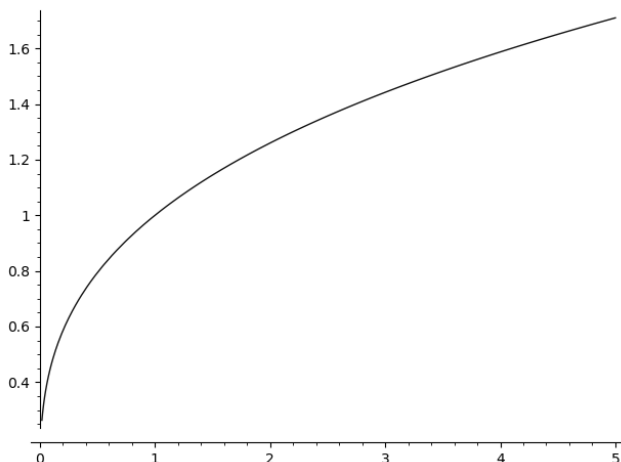
obtendremos solamente la gráfica para $0 < x < 2$, junto con el mensaje de advertencia

```

verbose 0 (3791: plot.py, generate_plot_points) WARNING: When plotting,
failed to evaluate function at 100 points.
verbose 0 (3791: plot.py, generate_plot_points) Last error message: 'can't
convert complex to float'

```

pues Sage está devolviendo raíces complejas (que no son visibles en el plano real) para valores de $x < 0$. Como podemos ver, Sage nos está reprimiendo explícitamente por elevar un número negativo a una potencia fraccional. La gráfica está dada abajo:



Ahora veremos cómo evitar este problema.

Graficando raíces impares de números negativos Para poder graficar la raíz cúbica real negativa, debemos hacer lo siguiente:

Código de Sage

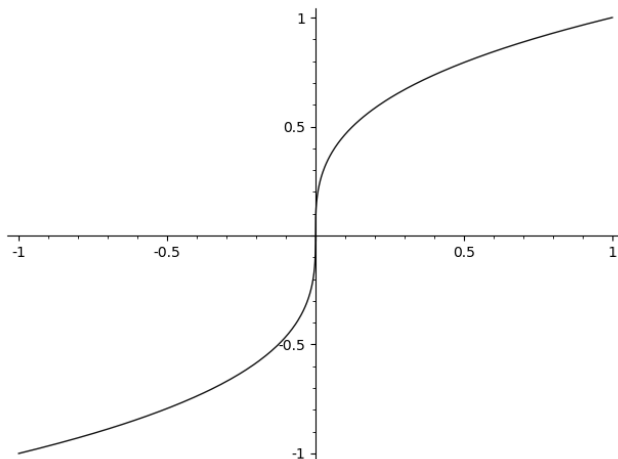
```
1 plot(lambda x: RR(x).nth_root(3), (x, -1, 1))
```

en lugar de

Código de Sage

```
1 plot(x^(1/3), -1, 1)
```

El código con `lambda` produce la agradable y suave gráfica de la raíz cúbica que esperábamos ver:

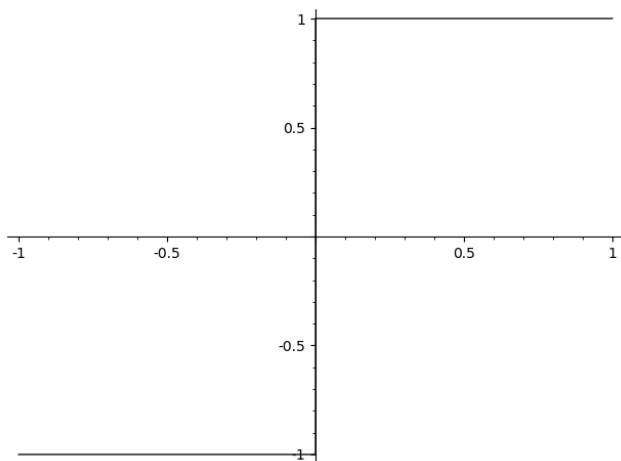


Una alternativa es usar

Código de Sage

```
1 plot(sign(x) * abs(x)^(1/3), -1, 1)
```

que es mucho más fácil de explicar. El valor absoluto, calculado con `abs`, garantizará que la entrada para la raíz cúbica es positiva. Sin embargo, valores negativos de x deberían tener raíces cúbicas negativas, mientras que los valores positivos de x deberían tener raíces cúbicas positivas. Por lo tanto, multiplicamos por `sign(x)`. Esta función devolverá -1 para cualquier x negativo, $+1$ para x positivo y 0 para $x = 0$. Al multiplicar por `sign(x)` garantizamos que la raíz cúbica tenga el signo correcto. Podemos ver la gráfica de la función `sign(x)` abajo:



Restringir los valores x de una gráfica Esta corta discusión no se relaciona con las raíces cúbicas, sino con las raíces cuadradas y cualquier otra función que tenga un dominio diferente a la recta real completa. Cuando trabajamos con funciones que tienen dominios limitados, muy de vez en cuando necesitamos restringir los valores x de la gráfica.

Considérese la función

$$f(x) = \sqrt{5x + 8} + \log(3 - 9x) + \sqrt{1 - 4x}$$

que claramente tiene el dominio $-8/5 \leq x \leq 1/4$. Si escribimos

Código de Sage

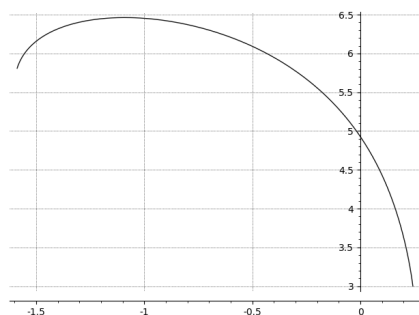
```
1 plot(sqrt(5*x+8) + log(3-9*x) + sqrt(1-4*x), -2, 2, gridlines=True)
```

aun cuando hayamos especificado el intervalo $-2 \leq x \leq 2$, solo obtenemos $-8/5 \leq x \leq 1/4$. Este resultado también viene acompañado de un mensaje de advertencia. En cambio, debemos hacer

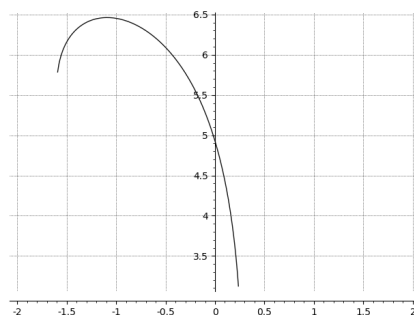
Código de Sage

```
1 plot(sqrt(5*x+8) + log(3-9*x) + sqrt(1-4*x), -2, 2,
  ↪ gridlines=True).show(xmin=-2, xmax=2)
```

donde `xmin` y `xmax` explícitamente fuerzan que las coordenadas x sean exactamente las que deseamos



Gráfica sin usar show

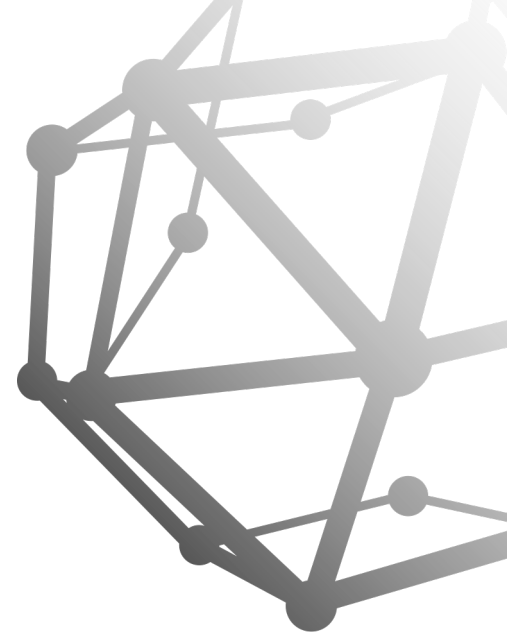


Gráfica usando show

Por supuesto, como el lector se imaginará, esta última instrucción es equivalente al pedazo de código más largo, aunque algo más claro, a continuación:

Código de Sage

```
1 p = plot(sqrt(5*x+8) + log(3-9*x) + sqrt(1-4*x), -2, 2, gridlines=True)
2 p.show(xmin=-2, xmax=2)
```

4

Características avanzadas de Sage

En este capítulo encontraremos comandos y ejemplos para temas seleccionados de las matemáticas que no son tan elementales como para estar incluidos en el capítulo 1. A excepción de algunas partes (claramente indicadas), las secciones son totalmente independientes unas de otras, y ciertamente no es necesario leerlas en orden. El lector simplemente puede sumergirse en las partes de este capítulo, conforme vaya requiriendo un pedazo de matemáticas avanzadas u otro.

4.1 Usando Sage con ecuaciones y funciones multivariadas

Consideremos la siguiente función:

$$f(x, y) = (x^4 - 5x^2 + 4 + y^2)^2$$

y veamos qué es lo que Sage puede hacer con ella.

Tal como vimos en la subsección 1.8.1 en la página 36, cuando usamos variables distintas de x , a menos que sean constantes como $k = 355/113$, necesitamos declararlas como variables usando el comando `var`. Por ejemplo,

Código de Sage

```
1 var('y')
2 f(x, y) = (x^4 - 5*x^2 + 4 + y^2)^2
3 f(1, 2)
```

El código anterior define la función $f(x, y)$ como se deseaba y correctamente imprime el valor $f(1, 2) = 16$. A partir de esto, podemos ver que las funciones multivariadas se trabajan de la misma forma que las funciones univariadas en cuanto a definición y evaluación. Sin embargo, también podemos emplear media evaluación y obtener una función con menos variables.

De la misma manera que las funciones a una variable se grafican en papel o en pantallas de computadora (espacio bidimensional), podemos graficar funciones a dos variables en el espacio tridimensional. Este tema es cubierto en el apéndice únicamente electrónico en línea del libro, “Graficando en color, en 3D, y animaciones”, que puede encontrarse en la página web www.sage-para-estudiantes.com, para descarga gratuita.

Ya vimos las funciones paramétricas en la sección 3.6 en la página 108. Algo que podríamos querer hacer es evaluar $f(x, y)$, como está definida arriba, pero con $x = 2t + 5$ y $y = 3t - 1$, o algo similar. Podemos escribir

Código de Sage

```
1 var('y t')
2 f(x, y) = (x^4 - 5*x^2 + 4 + y^2)^2
3 f(x=2*t+5, y=3*t-1)
```

y recibimos la salida

$$((2*t + 5)^4 + (3*t - 1)^2 - 5*(2*t + 5)^2 + 4)^2$$

que evidentemente es correcta, pero no muy legible. Alternativamente, si reemplazamos la última línea con

Código de Sage

```
1 f(x=2*t+5, y=3*t-1).expand()
```

entonces obtendremos la salida más comprensible

$$256*t^8 + 5120*t^7 + 44448*t^6 + 217088*t^5 + 649161*t^4 + 1214732*t^3 + 1394126*t^2 + 902940*t + 255025$$

Evaluando a medias una función Consideremos el siguiente código:

Código de Sage

```
1 var('y')
2 f(x,y) = (x^4 - 5*x^2 + 4 + y^2)^2
3 print(f(x,y))
4 print(f(x=1))
5 print(f(y=2))
```

Este produce la siguiente salida:

$$\begin{aligned} &(x^4 - 5*x^2 + y^2 + 4)^2 \\ &y^4 \\ &(x^4 - 5*x^2 + 8)^2 \end{aligned}$$

La tercera línea del código simplemente muestra la función $f(x, y)$ como la habíamos definido antes. La cuarta línea muestra lo que ocurre si x se fija en $x = 1$. En ese caso, obtenemos una función de y solamente, y es una bastante sencilla. La quinta línea muestra lo que pasa si y se fija en $y = 2$. Como podemos imaginar, obtenemos una función de x solamente. Este proceso de fijar una o más variables de una función, mientras el resto se dejan libres, es llamado “media evaluación” o “evaluación a medias”.

Trabajando con parámetros Un ejemplo más extenso de media evaluación es tomar una función con cuatro, cinco o seis variables, e introducir valores para varias de ellas (tales como la masa, impulso, $g = 9,82$ u otros coeficientes) para obtener una ecuación con menos variables. Veremos un ejemplo de esto en la subsección 4.22.4 en la página 189.

4.2 Trabajando con fórmulas grandes en Sage

Aquí exploraremos el uso de dos fórmulas grandes en Sage —una de finanzas y otra de física—. El ejemplo de finanzas es una hipoteca personal de casa. El ejemplo de física es la Ley de Gravitación de Newton. En ambos casos vamos a trabajar con algunas fórmulas y modelos complicados, escribiendo un pedazo de código que eventualmente crece hasta 4–6 líneas de longitud. Asumimos que el lector ha leído la sección 4.1 anterior, empezando en la página 123.

4.2.1 Finanzas personales: hipotecas

Este ejemplo asume que el lector sabe hacer cálculos de hipotecas. Si no es así, tal vez quiera leer un ejemplo diferente —o tal vez le gustaría aprender más sobre el tema, en cuyo caso debería seguir leyendo—.

La siguiente notación, cariñosamente llamada “la sopa de letras” por los estudiantes de áreas empresariales, es bastante estándar en los textos de finanzas, aunque con algunas variaciones.

- La duración del préstamo (o inversión) es t años.
- La tasa nominal de interés por año es r . (Nótese, 5 % se designa como $r = 0,05$.)
- El número de veces por año que el interés se compone es m . Por ejemplo, trimestralmente es $m = 4$ y semanalmente es $m = 52$, mientras que mensualmente es $m = 12$ y anualmente es $m = 1$.
- El número de composiciones a lo largo de la duración del préstamo (o inversión) es $n = mt$.
- La tasa de interés por periodo de composición (en lugar de por año) es $i = r/m$. Por ejemplo, 8 % compuesto trimestralmente hace $i = 0,02$ mientras que 6 % compuesto mensualmente hace $i = 0,005$.
- Para préstamos (o inversiones) con pagos regularmente espaciados, el monto pagado es c .
- Para préstamos (o inversiones) con un único pago inicial, el monto es llamado *capital* y denotado por P .

Algunos problemas para calentar Si depositamos \$ 1000 una sola vez y lo dejamos crecer por tres años con interés 4,8 % compuesto anualmente, entonces $r = 0,048$, $m = 12$, $i = r/m = 0,004$, $t = 3$, $n = 36$ y $P = 1000$. La fórmula para el monto que tenemos al final es simplemente la fórmula del interés compuesto (que diferentes libros la escriben ligeramente diferente):

$$A = P \left(1 + \frac{r}{m} \right)^{mt} = P(1 + i)^n.$$

La respuesta puede ser obtenida en Sage escribiendo

Código de Sage

```
1 1_000 * (1 + 0.048/12)^(12*3)
```

que nos da \$ 1154,55.

Si estamos ahorrando para nuestro retiro y queremos depositar \$ 100 mediante un cheque quincenal por 40 años, e invertimos a razón de 9,1 % compuesto bimestral, dado que hay 52 semanas en un año, el cheque quincenal se convierte en 26 depósitos por año o $m = 26$, y entonces tenemos $t = 40$, $r = 0,091$, $i = 0,0035$, $n = 1040$ y $c = 100$. La fórmula correcta en este caso es la de “valor futuro de una anualidad” o “anualidad creciente”, dada como

$$FV = c \frac{(1 + i)^n - 1}{i} = c \frac{(1 + r/m)^{mt} - 1}{r/m},$$

que puede ser evaluada con

Código de Sage

```
1 100 * ((1 + 0.003_5)^1_040 - 1) / 0.003_5
```

obteniendo la hermosa suma de \$ 1 052 872,11 para nuestro retiro.

Finalmente, si deseamos obtener una hipoteca para una casa por 30 años con un interés de 5,4 % compuesto mensualmente, haciendo pagos de \$ 900 por mes, entonces tendríamos $t = 30$, $m = 12$, $n = 360$, $r = 0,054$, $i = r/m = 0,054/12 = 0,0045$ y $c = 900$. La fórmula correcta para este problema es la de “valor presente de una anualidad” o “anualidad decreciente”, que está dada como

$$PV = c \frac{1 - (1 + i)^{-n}}{i} = c \frac{1 - (1 + r/m)^{-mt}}{r/m}$$

la cual podemos evaluar con

Código de Sage

```
1 900 * (1 - (1 + 0.004_5)^(-360)) / 0.004_5
```

obteniendo el resultado de \$ 160 276,16. Dependiendo de dónde vivamos, esta puede ser una muy buena casa. Por ejemplo, si encontramos un lugar agradable con un precio solicitado de \$ 178 084,62, entonces una reducción del 10 % implicaría un pago inicial de \$ 17 808,46, dejando \$ 160 276,16 para que proporcione el banco a través de esta hipoteca.

Creando un tabulador de Costo por Mil Los tres problemas de calentamiento anteriores pueden ser sencillos para el lector, pues de seguro está muy avanzado en matemáticas. Muchos empleados de banco no son graduados de universidad y los cálculos anteriores están fuera de su alcance. Para permitir a tales empleados poder reclutar clientes y ofrecerles hipotecas, la técnica de *costo por mil* es usada.

Una *hoja de tarifas* es publicada por el banco, ya sea diaria o semanalmente. Estas hojas muestran un *costo por mil* o “CPT” para cada uno de los préstamos disponibles del banco. Por ejemplo, debería haber CPT para una hipoteca de casa con interés fijo por 15 años y por 30 años, y probablemente tres CPTs de préstamos para autos (de 36, 48 y 60 meses de duración), seguidos de otros préstamos (para botes, educación, etc.).

Continuando con el ejemplo de arriba, con hipotecas de 30 años y 5,4 % compuesto mensualmente, el CPT resulta ser 5,615 307 918 7. (Explicaremos cómo hacer ese cálculo en breve.) Dado que el lector entiende matemáticas, si un cliente se le acercara para preguntar cuánto sería el pago mensual por una casa de \$ 180 000, podría introducir este monto en la ecuación de *PV* arriba, reemplazar $n = 360$ e $i = 0,0045$, y simplemente resolver para c . Sin embargo, el estrato inferior de empleados de un banco no siempre son capaces de hacer esto. En su lugar, estos observarán el CPT en la hoja de tarifas, encontrarán que corresponde 5,615 307 918 7 y calcularán

$$(180)(5,615\,307\,918\,7) = 1010,75\dots$$

permitiéndoles indicar al cliente que \$ 1010,76 será su pago mensual por la casa con esa hipoteca.

Ahora bien, la hoja de tarifas debe provenir de algún lugar. Un empleado de nivel medio en el banco calcularía el CPT mientras genera la hoja de tarifas tomando

$$PV = c \frac{1 - (1 + i)^{-n}}{i},$$

resolviendo para c y obteniendo

$$c = \frac{(PV)(i)}{1 - (1 + i)^{-n}},$$

donde uno puede introducir los valores de i y n . Naturalmente, $PV = 1000$. Calculemos esto ahora con Sage.

Como $PV = 1000$ no ha de cambiar, podemos poner como primera línea

Código de Sage

```
1 PV = 1_000
```

y entonces definimos nuestra función propia escribiendo

Código de Sage

```
2 costoPorMil(i, n) = (PV * i) / (1 - (1 + i)^(-n))
```

como la segunda línea. Observemos que hacer $PV = 1000$ aquí no es diferente a cómo establecimos $k = 355/113$ en la página 31.

Manteniendo esas dos líneas fijas, podemos usar la función repetidamente como nuestra tercera línea. Por ejemplo, para una hipoteca de 30 años con interés de 6,5 % compuesto mensualmente, podemos escribir

Código de Sage

```
3 costoPorMil(0.065/12, 30*12)
```

y, si el interés aumenta a 7 %, podemos reemplazar esta línea con

Código de Sage

```
3 costoPorMil(0.07/12, 30*12)
```

y así sucesivamente, conforme requiramos.

Notemos que el nombre de la función debe ser una sola palabra, pero variando el uso de mayúsculas y minúsculas como hemos hecho (lo que los programadores llaman “tipografía de camello”), pudimos expresar una idea que abarca más de una palabra. De seguro que es mucho más fácil leer `costoPorMil` que `costopormil` o `COSTOPORMIL`.

Aunque esta función es útil (por ejemplo, nos libera del sufrimiento de pensar dónde colocar paréntesis en cada ocasión), también es imperfecta. Cuando hacemos estos cálculos, nuestros datos estarán dados en términos del número de años del préstamo t , no el número de pagos $n = mt = 12t$. Similarmente, el interés está dado como la razón nominal publicada r , no la razón por periodo $i = r/m = r/12$. En cierta ocasión, un estudiante describió al autor este hecho diciendo que n e i son amistosos para las matemáticas, mientras que t y r son amistosos para los humanos. Esto puede solucionarse al definir una nueva función. Para evitar cualquier confusión, nuestro pedazo entero de código en el servidor Sage Cell debería verse como sigue:

Código de Sage

```
1 PV = 1_000
2 costoPorMil(i, n) = (PV * i) / (1 - (1 + i)^(-n))
3 costoPorMil1(r, t) = costoPorMil(i=r/12, n=12*t)
```

Ahora hemos creado una nueva fórmula, la cual usa el interés en términos de r , como queríamos, y el número de años t , no el número de periodos de composición. Entonces, para una hipoteca de 30 años al 6,5 % compuesto mensualmente, podemos escribir

Código de Sage

```
4 costoPorMil1(0.065, 30)
```

y tener una entrada y una salida más sencillas para el usuario.

Un último recordatorio: Si estamos acostumbrados a la notación decimal inglesa, cuando trabajemos con cantidades grandes, tales como el costo de una casa, debemos tener cuidado de no usar comas como separador de factores de mil. Todos los dígitos deben ser consecutivos, como aprendimos en la página 2.

Los símbolos de agrupación Algunos textos matemáticos antiguos acostumbraban usar corchetes (“[” y “]”) como una especie de “superparéntesis”. Esto se hacía para evitar tener conjuntos de paréntesis anidados dentro de otros paréntesis. Alguien educado con tales libros puede sentirse impulsado a escribir

$$f(i, n) = (PV \cdot i) \div [1 - (1 + i)^{-n}]$$

en lugar del más moderno

$$f(i, n) = (PV \cdot i) \div (1 - (1 + i)^{-n}),$$

que es idéntico al anterior, excepto por “[” y “]”, que están reemplazados por “(” y “)”, respectivamente.

Los matemáticos pueden escribir como gusten en papel, pero en código de Sage hay menos flexibilidad. Esto se debe mayormente a que Sage está construido sobre el lenguaje de programación Python, por lo que las convenciones de Python deben ser respetadas. Por lo tanto, uno no puede escribir

Código de Sage

```
1 costoPorMil(i, n) = (PV * i) / [1 - (1 + i)^(-n)]
```

sino que debe escribir

Código de Sage

```
1 costoPorMil(i, n) = (PV * i) / (1 - (1 + i)^(-n))
```

Recuérdese que en Sage, los únicos operadores de agrupación son los paréntesis. Detalles adicionales pueden ser encontrados en la página 2.

4.2.2 Física: Gravitación y satélites

Supongamos que estamos analizando un satélite en órbita alrededor de la Tierra. La fórmula de Newton para la gravedad es

$$F = \frac{GM_e M_s}{r_s^2},$$

donde $G = 6,77 \times 10^{-11}$ [N m²/kg²] es la constante de gravitación universal; $M_e = 5,9742 \times 10^{24}$ [kg] es la masa de la Tierra; r_s es la distancia del satélite al centro del planeta, medida en metros, y M_s es la masa del satélite en kilogramos.

De seguro que G y M_e no cambiarán, así que podemos fijarlas como constantes en las primeras dos líneas de nuestro código escribiendo

Código de Sage

```
1 G = 6.77 * 10^(-11)
2 Me = 5.974_2 * 10^(24)
```

de la misma forma en que definimos $k = 355/113$ en la página 31.

A diferencia de la masa de la Tierra o la constante de gravitación universal, la distancia al satélite puede cambiar más o menos frecuentemente, y su masa podría cambiar también (particularmente si tiene combustible a bordo, que será quemado en las maniobras). Debemos considerar estas dos cantidades como no constantes. Eso significa que nuestra función de fuerza dependerá de dos variables. Debemos escribir como nuestra tercera línea:

Código de Sage

```
3 fuerza(Ms, rs) = (G * Me * Ms) / (rs^2)
```

Usando estas tres líneas, podemos ingresar (como cuarta instrucción) un código tal como

Código de Sage

```
4 fuerza(1_000, 10*10^6)
```

para obtener valores de la fuerza. En este caso, estamos indicando que la masa del satélite es 1000 [kg] y que la distancia al centro de la Tierra es 10^7 metros; la respuesta devuelta debería ser 4044,53 Newtons. Esta función es muy útil si tenemos un gran número de valores r o M_s en los que estamos interesados, y queremos ahorrarnos mucho tecleo. Solo necesitamos cambiar un número, o tal vez dos, y presionar “Evaluate”.

Supongamos ahora que en realidad queríamos una función para trabajar en términos de altitud del satélite, medida desde la superficie, no desde el centro de la Tierra. Es fácil darse cuenta que solo debemos sumar el radio del planeta a la altitud para obtener la distancia al centro. Primero, ponemos el radio de la Tierra, que es 6378,1 [km], pero desde luego que debemos reescribirlo en metros para evitar el conflicto de unidades. Para evitar confusiones, nuestro trozo entero de código debería verse así:

Código de Sage

```

1  G = 6.77 * 10^(-11)
2  Me = 5.974_2 * 10^(24)
3  re = 6_378_100
4  fuerza(Ms, rs) = (G * Me * Ms) / (rs^2)
5  fuerza1(Ms, alt) = fuerza(Ms, rs = alt+re)

```

La quinta línea anterior es muy interesante. Hemos redefinido r_s como la suma de la altitud y el radio de la Tierra. Ahora podemos continuar nuestro estudio, usando `fuerza1`. Por ejemplo, como nuestra sexta línea de código, podemos escribir

Código de Sage

```

6  fuerza1(1_000, 3_621_900)

```

que nos devuelve 4044,53 [N], exactamente lo que obtuvimos antes. Esto se debe a que una altitud de 3621,9 [km] es equivalente a 10^7 metros desde el centro de la tierra. Esto es cierto, a su vez, porque

$$3621,9 \text{ [km]} + 6378,1 \text{ [km]} = 10\,000 \text{ [km]}$$

¿Qué función debemos usar, `fuerza` o `fuerza1`? Pues bien, depende de si el problema dado está en términos de altitud o en términos de r (la distancia al centro del planeta).

Esta idea de construir unas funciones sobre otras es muy poderosa. El autor frecuentemente ha escrito grandes juegos de funciones de esta manera —una definida sobre otra, definida sobre la anterior— para modelar fenómenos complejos en Sage. De esta forma, el cerebro humano solamente requiere considerar una pequeña cuestión a la vez antes de moverse al siguiente aspecto pequeño, y por lo tanto, sistemas grandes y complicados se vuelven accesibles y digeribles.

4.3 Derivadas y gradientes en cálculo multivariado

En la sección 1.11 de la página 46 aprendimos a calcular derivadas de funciones de una variable. Ahora aprenderemos acerca de las derivadas multivariadas.

4.3.1 Derivadas parciales

Probablemente no sea sorpresa enterarse que Sage puede calcular derivadas parciales muy fácilmente. Por ejemplo, si tenemos

$$g(x, y) = xy + \sin(x^2) + e^{-x}$$

podemos escribir

Código de Sage

```

1  g(x,y) = x*y + sin(x^2) + e^(-x)
2  diff(g(x,y), x)

```

y obtener

$$2*x*\cos(x^2) + y - e^{-x}$$

que es $\partial g / \partial x$. De manera similar, podemos escribir

Código de Sage

```

1 g(x,y) = x*y + sin(x^2) + e^(-x)
2 derivative(g(x,y), y)

```

para ver que $\partial g/\partial y$ es simplemente x . Notemos que esto realmente muestra cómo `diff` y `derivative` son sinónimos.

Incluso podemos encontrar

$$\frac{\partial^2}{\partial x \partial y} f$$

al escribir `derivative(g(x,y), x, y)`. La respuesta, por supuesto, no es más que 1.

4.3.2 Gradientes

La siguiente cuestión que nos puede interesar es encontrar el gradiente de una función multivariada. Si deseamos ∇g , entonces escribimos

Código de Sage

```

1 g(x,y) = x*y + sin(x^2) + e^(-x)
2 g.derivative()

```

y Sage calculará correctamente el gradiente. Alternativamente, podemos escribir `derivative(g), g.diff()` o incluso `diff(g)` en lugar de `g.derivative()`. En cualquier caso, Sage responde

```
(x, y) |--> (2*x*cos(x^2) + y - e^(-x), x)
```

Ya habíamos visto esta notación antes, en la sección 1.15 en la página 61, que es la forma de representar funciones formales. En este caso, Sage nos indica que el gradiente es un mapeo de un punto (x, y) a un par de números reales, y no una función ordinaria. Si el lector no gusta de esta notación, puede escribir en cambio `g.derivative()(x,y)`, `derivative(g)(x,y)`, `g.diff()(x,y)` o `diff(g)(x,y)` para obtener una forma equivalente:

```
(2*x*cos(x^2) + y - e^(-x), x)
```

Es interesante que el gradiente de una función univariada tal como $f(x) = 2e^{-x}$, calculada de esta manera, es solo la primera derivada ordinaria —lo que está totalmente de acuerdo con las reglas del cálculo—.

4.4 Matrices en Sage, parte dos

A diferencia de la sección sobre matrices en el capítulo 1 (véase la página 19), esta sección sí asume familiaridad con el primer semestre de *Álgebra Matricial* (a veces llamada *Álgebra Lineal*). Sin embargo, se ha procurado hacerla tan accesible e intuitiva como sea posible. Aquellos lectores que no están familiarizados con el álgebra matricial y que deseen adquirir pericia en este tema, harían bien en consultar *A First Course in Linear Algebra*, de Robert Beezer, publicado de manera virtual en 2007. No solo el texto es completamente gratis, sino que está escrito con Sage en mente y es de lectura fácil.

4.4.1 Definiendo algunos ejemplos

Vamos a estudiar los siguientes ejemplos en esta sección. Primero, consideremos la muy humilde matriz de orden 3×3

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & -1 \end{bmatrix},$$

y también consideremos $\vec{b} = \langle 24, 63, 52 \rangle$, así como $\vec{c} = \langle 10, 14, 8 \rangle$.

Nótese que algunos libros prefieren escribir $\vec{b} = \langle 24, 63, 52 \rangle$ para un vector y $B = (24, 63, 52)$ para un punto en el espacio. Esta es una excelente distinción —útil tanto para estudiantes como para profesores— y los libros de matemáticas deberían usarla. Sin embargo, esta notación también es una innovación reciente y puede resultar extraña para muchos lectores que están acostumbrados a usar paréntesis tanto para vectores como para puntos.

Dado que Sage ya usa los símbolos $< y >$ para comparaciones, es decir, “menor que” y “mayor que”, respectivamente, una consecuencia desafortunada es que Sage (como todos los textos, exceptos los más recientes) no usa esta nueva notación. Es inevitable. Sin embargo, el comando `vector` evita cualquier confusión, como veremos muy pronto.

En este punto sabemos cómo definir A con

Código de Sage

```
1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1])
```

como aprendimos en la página 21, pero para \vec{b} y \vec{c} tenemos opciones. Podemos considerar \vec{b} como una matriz con tres filas y una columna. En ese caso escribiríamos

Código de Sage

```
1 b = matrix(3, 1, [24, 63, 52])
```

Pero, como ya mencionamos, también existe un comando `vector`, que tiene la siguiente sintaxis:

Código de Sage

```
1 c = vector([10, 14, 8])
```

el cual requiere menos tecleo y es más legible para un humano.

Tomémonos un momento en un servidor Sage Cell para definir estos tres objetos, es decir A , \vec{b} y \vec{c} . A lo largo de lo que queda de esta sección, mantengamos esas tres líneas en Sage Cell. De esta forma, no tendremos que reescribirlas y podremos simplemente cambiar la última línea de la celda conforme aprendemos varias operaciones.

4.4.2 Multiplicación y exponenciación matricial

El símbolo “*” o asterisco es usado para realizar la multiplicación matricial, de la misma forma en que realiza el producto de dos números. Por ejemplo, uno puede escribir $A*b$ o $A*A$ para obtener la respuesta deseada. De manera similar, la suma y la resta de matrices es realizada con los usuales símbolos de adición y sustracción, como si estuviésemos realizando sumas y restas de números.

Es útil, en ocasiones, tener la matriz identidad lista. La podemos definir con

Código de Sage

```
1 Id = matrix(3, 3, [1, 0, 0, 0, 1, 0, 0, 0, 1])
```

si lo deseamos. Sin embargo, esto puede ser tedioso, especialmente con órdenes mayores. En particular, el autor frecuentemente tiene problemas colocando la cantidad correcta de ceros en los lugares adecuados. En vez de esto, podemos definir la matriz identidad con

Código de Sage

```
1 Id = identity_matrix(3)
```

que es mucho más corto. El autor tiene la costumbre de llamar `Id` a la matriz identidad, en lugar de `I`. Esto es para no sobrescribir la constante $I = \sqrt{-1}$, predefinida en Sage. También es por mera legibilidad, pues la “I” mayúscula puede confundirse fácilmente con una “i” minúscula, o incluso con el número uno. Sin embargo, uno puede llamar a la matriz identidad como desee. Es útil verificar que $Id*A$ y $A*Id$ producen la respuesta correcta, es decir A .

También podemos elevar matrices a varias potencias. El siguiente código:

Código de Sage

```
1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1])
2 print(A*A*A)
3 print()
4 print(A^3)
```

nos muestra que Sage puede realizar multiplicaciones matriciales mucho más rápido que un humano. También muestra cómo los exponentes de matrices usan la misma notación que el exponente de un número.

Con objeto de preservar la propiedad $(A^n)(A^m) = A^{n+m}$ del mundo de los números reales, los matemáticos definen los exponentes negativos con $A^{-n} = (A^{-1})^n$, suponiendo que A sea invertible. También A^0 es la matriz identidad. Incluso existe una noción de la “raíz cuadrada” de una matriz, pero se usa muy raramente y no la detallaremos aquí.

4.4.3 Resolviendo sistemas por izquierda y por derecha

Usemos ahora los objetos definidos previamente en esta sección para resolver algunos problemas. Para hallar el vector \vec{x} tal que $A\vec{x} = \vec{b}$, escribimos `A.solve_right(b)` y vemos que la solución es $\vec{x} = \langle 7, 1, 5 \rangle$. También podemos verificar esto escribiendo

Código de Sage

```
1 A * vector([7, 1, 5])
```

con lo que obtenemos nuevamente el vector $\vec{b} = \langle 24, 63, 52 \rangle$, como se esperaba. Para encontrar el vector \vec{y} tal que $\vec{y}A = \vec{c}$, podemos escribir `A.solve_left(c)` y ver que la solución es $\vec{y} = \langle 3, 0, 1 \rangle$. Como antes, verificamos esto escribiendo

Código de Sage

```
1 vector([3, 0, 1]) * A
```

con lo que obtenemos nuevamente el vector $\vec{c} = \langle 10, 14, 8 \rangle$, como queríamos.

Hay un pequeño punto técnico aquí. El vector $\langle 7, 1, 5 \rangle$ es un vector columna, o una matriz de tres filas y una columna; el vector $\langle 3, 0, 1 \rangle$ es un vector fila, o una matriz de una fila y tres columnas. Sage siempre tratará de deducir si pretendíamos usar un vector fila o un vector columna cuando usemos el comando `vector`. Sin embargo, algunos profesores serán menos indulgentes que Sage y requerirán la notación $\vec{y}^T A$ en lugar de $\vec{y}A$.

El operador de quebrado invertido “\” puede usarse para abreviar `solve_right`. No debe confundirse con el símbolo de quebrado “/”. Podemos escribir

Código de Sage

```
1 A \ b
```

como abreviación de `A.solve_right(b)`. Esta notación extremadamente concisa es un retroceso al muy antiguo lenguaje MATLAB, que aún se usa en la industria y que tiene el mismo operador.

También podemos escribir `A.augment(b, subdivide=True)` para obtener la matriz

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 24 \\ 4 & 5 & 6 & 63 \\ 7 & 8 & -1 & 52 \end{array} \right]$$

en preparación para usar el comando `rref` que aprendimos en la página 22. La opción `subdivide` simplemente indica a Sage que debe colocar una línea vertical que separe la matriz original y la columna recién añadida,

como vemos arriba. Algunos textos ignoran la línea vertical de separación. Para aplicar la estrategia *aumentar y hallar la RREF*, escribimos algo como

Código de Sage

```

1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1])
2 b = matrix(3, 1, [24, 63, 52])
3 M = A.augment(b, subdivide=True)
4
5 print('Antes:')
6 print(M)
7 print('')
8 print('Después:')
9 print(M.rref())

```

Por supuesto, los comandos `print` no son realmente necesarios, pero son muy informativos para el usuario. Algunos matemáticos usarán el operador quebrado invertido o el comando `solve_right`. Sin embargo, esta estrategia de aumentar y hallar la RREF tiene sus ventajas.

Consideremos cambiar la última entrada de A , es decir A_{33} , a 9 en lugar de -1 . Entonces ejecutamos nuestro código nuevamente. Vemos la siguiente salida:

```

Antes:
[ 1  2  3|24]
[ 4  5  6|63]
[ 7  8  9|52]
Después:
[ 1  0 -1| 0]
[ 0  1  2| 0]
[ 0  0  0| 1]

```

Esto es extremadamente informativo. De la fila de ceros seguidos de un 1, vemos que el sistema de ecuaciones original no tiene solución. También podemos apreciar que la primera y segunda columnas son vectores efectivos o elementales, mientras que la tercera es un vector defectivo o no elemental. Si el lector conoce el significado de *rango* en álgebra lineal, podemos ver que esta matriz tiene rango 2. Si el lector desconoce el concepto de rango, no es necesario preocuparse de ello en este momento.

Veamos qué pasaría si hubiésemos tratado el mismo problema con el operador quebrado invertido. Si hubiésemos escrito

Código de Sage

```

1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9])
2 b = matrix(3, 1, [24, 63, 52])
3
4 A \ b

```

obtendríamos la respuesta

```
ValueError: matrix equation has no solutions
```

que es innegablemente correcta, pero no tan informativa como la que nos da la estrategia aumentar y hallar la RREF.

¿Qué ocurre si usamos el quebrado invertido cuando hay infinitas soluciones? ¡Intentémoslo y veamos! Los comandos

Código de Sage

```
1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9])
2 d = vector([6, 15, 24])
3
4 A \ d
```

producen el vector $\langle 0, 3, 0 \rangle$, que es una solución perfectamente válida. Basándonos en esta respuesta, estaríamos justificados al asumir que esta es la única solución a ese problema. Sin embargo, podemos ver que $\langle 1, 1, 1 \rangle$ también es una solución. Cuando hay infinitas soluciones, el operador quebrado invertido simplemente elige una. En particular, elegirá la solución formada al hacer todas las variables libres iguales a cero.

Podríamos escribir en cambio

Código de Sage

```
1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9])
2 d = vector([6, 15, 24])
3 B = A.augment(d, subdivide=True)
4
5 B.rref()
```

para obtener la salida

```
[1 0 0|0]
[0 1 0|3]
[0 0 1|0]
```

que, si sabemos cómo leerla, nos dice mucho más. La circunstancia de tener una cantidad infinita de soluciones para un sistema lineal (en otras palabras, tener variables libres) es discutida en la apéndice D en la página 289.

Finalmente, recordemos que `A.rref()` solicita la “forma escalonada reducida por filas”. Sin embargo, también existe una “forma escalonada” o REF¹. Esta versión a medias de la RREF es extremadamente útil si debemos trabajar con matrices a mano. Pero en la era de la computadora, su rol es secundario en el mejor caso. El comando para calcular la REF es simplemente

Código de Sage

```
1 A.echelon_form()
```

4.4.4 Matrices inversas

Algunas matrices tienen inversas y otras no. Podemos hallar una matriz inversa con `A.inverse()`. Por ejemplo,

Código de Sage

```
1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1])
2 B = A.inverse()
3 print(B)
```

produce la salida

```
[-53/30  13/15  -1/10]
[ 23/15 -11/15   1/5]
[ -1/10   1/5  -1/10]
```

¹Row-echelon form. Siguiendo la costumbre ya establecida desde el principio de este libro, usamos las siglas en inglés.

Podemos verificar que la inversa es correcta solicitando el producto $A * B$ o, alternativamente, $B * A$, los cuales deben devolver la matriz identidad. También podemos reemplazar `A.inverse()` con A^{-1} , como abajo:

Código de Sage

```
1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1])
2 B = A^(-1)
3 print(B)
```

Puede que nos estemos preguntando qué ocurre si tratamos de invertir una matriz singular. Podemos simplemente cambiar A_{33} de -1 a 9 , y entonces la matriz es singular. Si tratamos de hallar su inversa, obtenemos un mensaje de error con mucha palabrería, terminando con

```
ZeroDivisionError: matrix must be nonsingular
```

A veces es útil calcular la inversa de una matriz “de la forma larga”. Para lograr esto, acoplamos la matriz identidad del orden adecuado, a la derecha de la matriz original. Por ejemplo, el código

Código de Sage

```
1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9])
2 Id = identity_matrix(3)
3 C = A.augment(Id, subdivide=True)
4
5 print('Antes:')
6 print(C)
7 print('')
8 print('Después:')
9 print(C.rref())
```

produce la salida de abajo:

```
Antes:
[1 2 3|1 0 0]
[4 5 6|0 1 0]
[7 8 9|0 0 1]
Después:
[ 1 0 -1| 0 -8/3 5/3]
[ 0 1 2| 0 7/3 -4/3]
[ 0 0 0| 1 -2 1]
```

Como podemos ver, la matriz no es invertible. El proceso de la RREF se “estancó” en la tercera columna. Esto nos dice mucho más que el mensaje de error que se obtiene simplemente tratando de invertir A con `A.inverse()`.

4.4.5 Calculando el núcleo de una matriz

Para cualquier matriz A , en varias ocasiones es útil conocer el conjunto de vectores \vec{v} tales que $\vec{v}A = \vec{0}$. De la misma manera, es útil conocer el conjunto de vectores \vec{v} tales que $A\vec{v} = \vec{0}$. Estos dos conjuntos son llamados “núcleo izquierdo de A ” y “núcleo derecho de A ”, respectivamente. Exploreemos un ejemplo:

Código de Sage

```

1  A = matrix(2, 2, [-2, 7, 0, 0])
2
3  print(A.left_kernel())
4
5  print('Test:')
6  print(vector([0, 1]) * A)
7  print(A * vector([0, 1]))

```

Esto produce la siguiente salida:

```

Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1]
Test:
(0, 0)
(7, 0)

```

Es notable que nuestro código verifique que $\vec{v}A = \vec{0}$, como se quería, pero que $A\vec{v} \neq \vec{0}$. Esto es para enfatizar que el núcleo izquierdo y el derecho son conjuntos distintos.

Expliquemos ahora las tres primeras líneas de la salida anterior. Como veremos a lo largo de estos ejemplos, la parte de “degree 2” (“grado 2”) se refiere a la dimensión del espacio que contiene al núcleo, o alternatively, el número de elementos de los vectores del núcleo; la parte de “rank 1” (“rango 1”) indica la dimensión del núcleo (es decir, la dimensión del espacio que se envía a cero) —esto usualmente se llama la *nulidad* de A , pero debemos tener cuidado de diferenciar entre la *nulidad izquierda* y la *nulidad derecha*—. Es importante enfatizar que los rangos del núcleo izquierdo, núcleo derecho y la matriz A misma, pueden ser todos distintos. La segunda y tercera líneas de la salida anterior simplemente nos indican una base del núcleo, en este caso $\{(0, 1)\}$.

La matriz

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 0 \end{bmatrix}$$

tiene rango 2, pero su núcleo izquierdo tiene rango 0 y su núcleo derecho tiene rango 3. Podemos hallar estos tres números inmediatamente con las funciones `A.right_nullity()`, `A.left_nullity()` y `A.rank()`, para cualquier matriz A .

Ahora bien, el código equivalente usando el comando `right_kernel` sería como sigue:

Código de Sage

```

1  A = matrix(2, 2, [-2, 7, 0, 0])
2
3  print(A.right_kernel())
4
5  print('Test:')
6  print(vector([7, 2]) * A)
7  print(A * vector([7, 2]))

```

lo que produce la salida

```

Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[7 2]
Test:
(-14, 49)
(0, 0)

```

Nuevamente vemos que $A\vec{v} = \vec{0}$, como se deseaba, pero que $\vec{v}A \neq \vec{0}$. De seguro, el núcleo izquierdo y el derecho son objetos matemáticos distintos. En ocasiones, el núcleo derecho es llama “espacio nulo” de A . Por esta razón, cuando se menciona *el* núcleo de A en matemáticas, usualmente se refiere al derecho. Sin embargo, en Sage, `A.kernel()` se refiere al núcleo izquierdo. El programador prudente será explícito usando `A.right_kernel()` y `A.left_kernel()`, de manera que no exista posibilidad de confusión.

A propósito, en cierta ocasión el autor discutió esto con el Prof. William Stein, y este específicamente solicitó mencionar en este libro que *los programadores reales siempre revisan la documentación*. Realmente nadie recuerda detalles finos como la distinción entre núcleo izquierdo y derecho. Incluso si uno está seguro de recordar cuál es cuál, de todas formas es recomendable verificarlo (usando las técnicas de la sección 1.10 en la página 44) para evitar la posibilidad de errores.

Para mostrar de forma más poderosa cómo funciona el concepto, el autor ha escrito un programa para que lo estudie el lector. Véase la figura 4.1 en la página 138. Ahí vemos una matriz más grande y calculamos los núcleos izquierdo y derecho muy cuidadosamente. También podemos ver que si \vec{a} y \vec{b} están en el núcleo, entonces $r_1\vec{a} + r_2\vec{b}$ también estará en el núcleo, para cualesquiera números reales r_1 y r_2 que elijamos.

Aplicaciones del cálculo de núcleos Hemos pasado por mucho para aprender a calcular los núcleos izquierdo y derecho de una matriz. Tal vez el lector tenga curiosidad sobre qué utilidad tiene todo esto. Existen varios usos en matemáticas puras. En matemáticas aplicadas, una aplicación tiene que ver con sistemas de ecuaciones lineales con infinitas soluciones.

Otra aplicación de los núcleos, en la criptografía, es el algoritmo de la Criba Cuadrática para factorizar el producto de dos números primos grandes. Este es un método de ataque contra el criptosistema RSA. Resulta que el autor discute la Criba Cuadrática detalladamente en el capítulo 10 de otro libro que ha escrito, *Algebraic Cryptanalysis*, publicado por Springer en 2009. Aunque la mayor parte del libro requiere conocimientos de un curso previo de criptografía, el capítulo en cuestión es algo independiente y sería entendible para cualquiera que ha tomado un curso de *Teoría de Números*, o tal vez en cambio, dos semestres de *Álgebra Abstracta*, en ocasiones llamada *Álgebra Moderna*.

4.4.6 Determinantes

Volvamos brevemente a la matriz con la que empezamos esta lección. Si quisiéramos conocer su determinante, todo lo que necesitamos hacer es escribir lo siguiente:

Código de Sage

```
1 A = matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, -1])
2 det(A)
```

Entonces veremos que el determinante es 30. Si cambiamos A_{33} de -1 a 9 , entonces el determinante se hará 0 . Esto tiene sentido, pues vimos que A producía una solución única cuando resolvimos el sistema lineal con el vector $\langle 24, 63, 52 \rangle$ y con el -1 aún presente en A_{33} . Por otro lado, vimos que $A \setminus \mathbf{b}$ no tenía soluciones cuando intentamos resolver el sistema con el mismo vector, pero cambiando el -1 por 9 .

Los casos de “infinitas soluciones” y “sin soluciones” pueden y ocurrirán solamente cuando el determinante sea cero. El caso de una única solución es lo que ocurre cuando el determinante es no nulo.

4.5 Operaciones vectoriales

Los vectores están presentes en Álgebra Lineal y Cálculo Multivariado, así como Física I y sus sucesores, entre otros. Sage es excelente trabajando con vectores. Aquí exploraremos las operaciones básicas de suma, resta, multiplicación por un escalar, el producto punto, el producto cruz y la norma de un vector. Entonces las combinaremos explorando cómo calcular el ángulo entre dos vectores. Usaremos los siguientes dos como ejemplo:

$$\vec{a} = \langle 1, 2, 3 \rangle \quad \text{y} \quad \vec{b} = \langle 4, 5, 6 \rangle.$$

Código para el núcleo izquierdo

Código de Sage

```

1  A = matrix(4, 4, [1, 2, 3, 0, 4, 5, 6,
    ↪ -1, 7, 8, 9, -2, 12, 15, 18, -3])
2
3  print('Matriz de entrada:')
4  print(A)
5
6  print('Espacio nulo:')
7  print(A.left_kernel())
8  print()
9
10 xvec = vector([1, 1, 1, -1])
11 yvec = vector([0, 3, 0, -1])
12
13 print('Tests:')
14 print(xvec, 'mapea a', xvec*A)
15 print(yvec, 'mapea a', yvec*A)
16 print(xvec+yvec, 'mapea a',
    ↪ (xvec+yvec)*A)
17 print(xvec-2*yvec, 'mapea a',
    ↪ (xvec-2*yvec)*A)
18 print(17*xvec + 47*yvec, 'mapea a',
    ↪ (17*xvec + 47*yvec)*A)
19 print(71*xvec + 888*yvec, 'mapea a',
    ↪ (71*xvec + 888*yvec)*A)

```

Salida para el núcleo izquierdo

```

Matriz de entrada:
[ 1  2  3  0]
[ 4  5  6 -1]
[ 7  8  9 -2]
[12 15 18 -3]
Espacio nulo:
Free module of degree 4 and rank 2 over
Integer Ring
Echelon basis matrix:
[ 1  1  1 -1]
[ 0  3  0 -1]

Tests:
(1, 1, 1, -1) mapea a (0, 0, 0, 0)
(0, 3, 0, -1) mapea a (0, 0, 0, 0)
(1, 4, 1, -2) mapea a (0, 0, 0, 0)
(1, -5, 1, 1) mapea a (0, 0, 0, 0)
(17, 158, 17, -64) mapea a (0, 0, 0, 0)
(71, 2735, 71, -959) mapea a (0, 0, 0, 0)

```

Código para el núcleo derecho

Código de Sage

```

1  A = matrix(4, 4, [1, 2, 3, 0, 4, 5, 6,
    ↪ -1, 7, 8, 9, -2, 12, 15, 18, -3])
2
3  print('Matriz de entrada:')
4  print(A)
5
6  print('Espacio nulo:')
7  print(A.right_kernel())
8  print()
9
10 xvec = vector( [1, 1, -1, 3] )
11 yvec = vector( [0, 3, -2, 3] )
12
13 print('Tests:')
14 print(xvec, 'mapea a', xvec*A)
15 print(yvec, 'mapea a', yvec*A)
16 print(xvec+yvec, 'mapea a',
    ↪ (xvec+yvec)*A)
17 print(xvec-2*yvec, 'mapea a',
    ↪ (xvec-2*yvec)*A)
18 print(17*xvec + 47*yvec, 'mapea a',
    ↪ (17*xvec + 47*yvec)*A)
19 print(71*xvec + 888*yvec, 'mapea a',
    ↪ (71*xvec + 888*yvec)*A)

```

Salida para el núcleo derecho

```

Matriz de entrada:
[ 1  2  3  0]
[ 4  5  6 -1]
[ 7  8  9 -2]
[12 15 18 -3]
Espacio nulo:
Free module of degree 4 and rank 2 over
Integer Ring
Echelon basis matrix:
[ 1  1 -1  3]
[ 0  3 -2  3]

Tests:
(1, 1, -1, 3) mapea a (34, 44, 54, -8)
(0, 3, -2, 3) mapea a (34, 44, 54, -8)
(1, 4, -3, 6) mapea a (68, 88, 108, -16)
(1, -5, 3, -3) mapea a (-34, -44, -54, 8)
(17, 158, -111, 192) mapea a (2176, 2816,
3456, -512)
(71, 2735, -1847, 2877) mapea a (32606,
42196, 51786, -7672)

```

Figura 4.1 El ejemplo muy extenso de los comandos `left_kernel` y `right_kernel`.

Se pretende que cada uno de los comandos en la tabla 1 esté precedido por la definición de estos vectores, es decir, escribiendo primero:

Código de Sage

```
1 a = vector([1, 2, 3])
2 b = vector([4, 5, 6])
```

Tabla 1 Las operaciones vectoriales básicas en Sage

Operación	Notación	Sintaxis de Sage	Resultado esperado
Adición	$\vec{a} + \vec{b}$	<code>a + b</code>	$\langle 5, 7, 9 \rangle$
Sustracción	$\vec{a} - \vec{b}$	<code>a - b</code>	$\langle -3, -3, -3 \rangle$
Producto por un escalar	$3\vec{a}$	<code>3 * a</code>	$\langle 3, 6, 9 \rangle$
Producto punto	$\vec{a} \cdot \vec{b}$	<code>a.dot_product(b)</code>	32
Producto cruz	$\vec{a} \times \vec{b}$	<code>a.cross_product(b)</code>	$\langle -3, 6, -3 \rangle$
Norma o longitud	$\ \vec{a}\ $	<code>norm(a)</code>	$\sqrt{14}$

Para un ejemplo de cómo combinar varias de estas operaciones para realizar una tarea útil, encontremos el ángulo entre \vec{a} y \vec{b} . Debemos recordar que la fórmula es

$$\cos \theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|}$$

y entonces podemos escribir

Código de Sage

```
1 a = vector([1, 2, 3])
2 b = vector([4, 5, 6])
3
4 producto_punto = a.dot_product(b)
5 mi_fraccion = producto_punto / (norm(a) * norm(b))
6 theta = arccos(mi_fraccion)
7
8 print(N(theta), 'radianes')
9 print(N(theta*180/pi), 'grados')
```

Esto resultará en la siguiente salida:

```
0.225726128552734 radianes
12.9331544918991 grados
```

Nótese que la función `print` es capaz de recibir varios argumentos a la vez, separados por comas. Estos se convierten en cadenas de caracteres y entonces son concatenados en una sola línea de texto, usando un espacio como eslabón (donde antes estaban las comas que separaban los argumentos). El lector seguramente notó que usamos esta característica en la figura 4.1 para hacer el código más legible.

4.6 Trabajando con enteros y Teoría de Números

El estudio de problemas que tratan con los enteros es llamado “Teoría de Números”. Fue un área famosamente pura por siglos, pero en los últimos cincuenta años algunas aplicaciones fantásticas han ido apareciendo, incluyendo la criptografía y la programación entera.

Esta área de estudio puede empezarse a muy temprana edad (por ejemplo, “¿qué es un número primo?”), pero también puede llegar a ser muy avanzada. Muchas disertaciones para el grado de Ph.D. son escritas cada año con nuevos resultados en esta área. En su mayoría, la teoría de números trata de escribir pruebas, y uno puede imaginar que Sage no es muy útil para ello. De hecho, Sage es muy bueno trabajando con enteros para ayudar a desarrollar ejemplos específicos sobre el efecto que los instrumentos de teoría de números tienen sobre enteros específicos. Esto puede ayudar a entender nuevos conceptos más íntimamente y puede formar un puente entre la lectura de un tema y el momento en el que uno realmente está listo para escribir pruebas sobre ese tema.

Por ejemplo, podemos escribir

Código de Sage

```
1 factor(-2_007)
```

para encontrar la factorización de ese entero, que es $-1 \cdot 3^2 \cdot 223$ y, similarmente,

Código de Sage

```
1 factor(2_008)
```

produce la respuesta $2^3 \cdot 251$. Para encontrar el siguiente número primo después de un millón, podemos usar

Código de Sage

```
1 next_prime(10^6)
```

para descubrir que la respuesta es 1 000 003. También podemos testar un número por primalidad. Consideremos `is_prime(101)`, o alternativamente `is_prime(102)`. Por supuesto, Sage responde True para 101 y False para 102. Finalmente pero no menos importante, podemos encontrar todos los primos en un cierto intervalo. Por ejemplo,

Código de Sage

```
1 prime_range(1_000, 1_100)
```

nos devolverá todos los números primos entre 1000 y 1100.

4.6.1 El máximo común divisor y el mínimo común múltiplo

Para encontrar el “máximo común divisor” o \gcd^2 de dos números, digamos 120 y 64, solo escribimos

Código de Sage

```
1 gcd(120, 64)
```

y veremos que la respuesta es 8. Esto tiene sentido, pues $120/8 = 15$ mientras que $64/8 = 8$, y como podemos ver, estos dos últimos números no comparten factores. Más explícitamente, podemos observar la factorización prima de $15 = 5 \times 3$ y $8 = 2 \times 2 \times 2$. También, otra forma de ver esto es

$$120 = 2^3 \times 3 \times 5 \quad \text{mientras que} \quad 64 = 2^6,$$

de donde claramente deducimos que lo único que 120 y 64 tienen en común es 2^3 . Por lo tanto, el \gcd es 8.

²Greatest common divisor, en inglés.

De forma similar, podemos encontrar el “mínimo común múltiplo” o lcm^3 . Esto lo hacemos escribiendo

Código de Sage

```
1 lcm(120, 64)
```

de lo que vemos que el lcm es 960. Esto tiene sentido debido a que $960/120 = 8$ y $960/64 = 15$, y a que nuevamente 8 no comparte factores con 15. Un matemático diría que “8 es coprimo de 15”; otros dirían que son “mutuamente primos”. Otra manera de ver esto es que el lcm debe ser

$$2^6 \times 3^1 \times 5^1 = 960$$

porque 2 aparece 6 veces en 64 y 3 veces en 120 (así que un máximo de 6), mientras que 3 y 5 aparecen ambos 0 veces en 64 y 1 vez en 120 (así que un máximo de 1 cada uno). Elevando cada uno de esos primos (2, 3 y 5) a esas potencias (6, 1 y 1), obtenemos el lcm.

Antes de continuar, un ejemplo más será provechoso. Trabajemos con números ligeramente más grandes, tal vez 3600 y 1000. Primero, hacemos las factorizaciones primas de

$$3600 = 2^4 \times 3^2 \times 5^2 \quad \text{y} \quad 1000 = 2^3 \times 5^3,$$

y entonces podemos calcular el lcm tomando el máximo exponente en cada ocasión:

$$2^4 \times 3^2 \times 5^3 = 18\,000,$$

mientras que el gcd lo obtenemos tomando los exponentes mínimos en cada caso:

$$2^3 \times 3^0 \times 5^2 = 200.$$

Tomémonos un momento para verificar que 1000 y 3600 dividen 18 000, pero que no existe un número menor para el cual se cumpla esto mismo. A continuación, nos tomamos otro momento para verificar que 200 divide 1000 y 3600, así como el hecho que ningún número mayor lo hará. Y, por supuesto, el producto de los números originales dividido por el gcd es el lcm, y el producto de los números originales dividido por el lcm es el gcd. Una idea clave aquí es que la factorización prima comunica una enorme cantidad de información sobre un número.

Si deseamos determinar el gcd de varios números a la vez (por ejemplo, 120, 55, 25 y 35), No hay necesidad de hacer

Código de Sage

```
1 gcd(120, gcd(55, gcd(25, 35)))
```

porque podemos escribir en cambio

Código de Sage

```
1 gcd([120, 55, 25, 35])
```

que nos devuelve 5. Similarmente,

Código de Sage

```
1 lcm([120, 55, 25, 35])
```

nos da 46 200. Este es otro ejemplo de una lista en Sage. Podemos delimitar cualesquiera datos con `[y]`, y separar las entradas con comas, para crear una lista. Esta es notación que Sage heredó del lenguaje de programación Python.

³Least common multiple, en inglés.

4.6.2 Más acerca de los números primos

Un buen truco es que si quisiéramos encontrar el 54 321^{er} número primo, podríamos escribir

Código de Sage

```
1 nth_prime(54_321)
```

y descubrir que es 670 177. Podemos hacer una verificación con

Código de Sage

```
1 factor(670_177)
```

que devuelve el mismo número, confirmando que es un primo. Mejor aun, podemos escribir

Código de Sage

```
1 is_prime(670_177)
```

que nos devolverá True.

Uno de los temas más fascinantes en las matemáticas es la distribución de los números primos. Por ejemplo, quisiéramos ver cómo crece la función `nth_prime` conforme x crece. En español simple, ¿cuán grande es el millonésimo primo?, ¿el billonésimo primo?, ¿el trillonésimo?. Resulta que una buena aproximación para el tamaño del x^{mo} primo es

$$f(x) = 1,13 \times \log_e(x),$$

aunque también existen mejores aproximaciones.

Podemos testar esto con

Código de Sage

```
1 f(x) = 1.13 * x * log(x)
2
3 print(f(10^6))
4 print(N(nth_prime(10^6)))
5
6 print(f(10^7))
7 print(N(nth_prime(10^7)))
```

Esto produce la salida

```
1.56115269304996e7
15485863
1.82134480855829e8
179424673
```

Después de hacer los ajustes de la notación científica, podemos ver que el estimado y la realidad están dentro de cerca de 1 % entre sí.

4.6.3 Acerca de la función phi de Euler

En ocasiones es útil saber cuántos enteros son coprimos con x , en el rango desde 1 hasta x . Por ejemplo, ¿cuántos números desde 1 hasta 10 comparten un factor primo con 10 y cuántos son coprimos de 10? Los divisores primos de 10 son 2 y 5, y por lo tanto cualquier número cuya factorización prima contiene 2s y 5s compartirá un factor con 10. Esto incluye

$$\{2, 4, 5, 6, 8, 10\},$$

y por lo tanto, los demás números,

$$\{1, 3, 7, 9\},$$

que no tienen 2s ni 5s en sus factorizaciones son coprimos de 10. En particular, si el lector conoce aritmética modular, entonces estos son los números que tiene *inversos multiplicativos* o *recíprocos* módulo 10. Dado que tenemos 4 coprimos de 10, escribiremos $\phi(10) = 4$.

En general, hay $\phi(n)$ números z , con $1 \leq z \leq n$, tales que z es coprimo de n . Esta es una forma más o menos invertida de definir una función matemática, pero resulta que ϕ es extremadamente útil cuando se trabaja con aritmética modular, y especialmente en criptografía.

Incluso si el lector no conoce la aritmética modular, podemos considerar el 7. Cada uno de los números

$$\{1, 2, 3, 4, 5, 6\}$$

es coprimo de 7, pues el único número primo que divide 7 es él mismo, y este no divide a ninguno de los que están en ese conjunto. Así que, seis de seis números son coprimos de 7, y entonces escribimos $\phi(7) = 6$.

De hecho, siempre será el caso, para un número primo p , que $\phi(p) = p - 1$. Por otro lado, consideremos 12. Entonces entre

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

vemos que

- 2 y 12 son divisibles por 2;
- 3 y 12 son divisibles por 3;
- 4 y 12 son divisibles por 2;
- 6 y 12 son divisibles por 2 y 3;
- 8 y 12 son divisibles por 2;
- 9 y 12 son divisibles por 3;
- 10 y 12 son divisibles por 2.

Por estas razones, excluimos 2, 3, 4, 6, 8, 9 y 10 de nuestra lista. Finalmente, esto significa que solamente 1, 5, 7 y 11 son coprimos de 12, apenas cuatro enteros, y así escribimos $\phi(12) = 4$.

Esta estrategia de hacer una lista de números y verificar uno a la vez, es buena para números pequeños, pero no sería posible para números de 100 dígitos, como cuando trabajamos en criptografía. Sin embargo, Sage puede calcular $\phi(n)$ con un solo comando. Todo lo que necesitamos hacer es escribir

Código de Sage

```
1 euler_phi(12)
```

para ver que la respuesta es 4. Esta función se llama `euler_phi` porque fue descubierta por Leonhard Euler (1707–1783), quien también descubrió el número

$$e \approx 2,718281828 \dots$$

y muchos otros conceptos matemáticos. Él aún mantiene el récord para el mayor número de artículos publicados por un matemático, aun cuando lleva muerto casi de 240 años.

Un reto divertido Como resulta ser, si p y q son números primos distintos, el valor de $\phi(pq)$ es muy fácil de calcular. Sin embargo, no daremos la fórmula aquí. Será muy interesante para el lector intentar varios números de la forma pq y ver si puede descubrir la fórmula por sí mismo —no es muy difícil, de hecho—. Solo hay que tener el cuidado que $p \neq q$.

4.6.4 Los divisores de un número

Por otro lado, consideremos el conjunto de los divisores de un número. Si queremos conocer el conjunto de divisores de 250, es decir los enteros positivos que dividen 250, escribiríamos

Código de Sage

```
1 divisors(250)
```

que resulta en la siguiente lista:

[1, 2, 5, 10, 25, 50, 125, 250]

De manera similar, si queremos conocer el conjunto de divisores de 312 500, entonces escribimos

Código de Sage

```
1 divisors(312_500)
```

que nos da la salida

[1, 2, 4, 5, 10, 20, 25, 50, 100, 125, 250, 500, 625, 1250, 2500, 3125, 6250, 12500, 15625, 31250, 62500, 78125, 156250, 312500]

Así como para ϕ , hay dos funciones interesantes de teoría de números que surgen con los divisores. El *tamaño* del conjunto de divisores de n es denotado por $\tau(n)$, mientras que la *suma* del conjunto de los divisores de n es⁴ denotado por $\sigma(n)$. El símbolo τ es la letra griega “tau” y σ es la letra “sigma”. Por ejemplo, acabamos de ver la lista de los divisores de 250. Hay 8 de estos, así que $\tau(250) = 8$. Por otro lado,

$$1 + 2 + 5 + 10 + 25 + 50 + 125 + 250 = 468$$

por lo que $\sigma(250) = 468$. Podemos calcular σ en Sage con `sigma(250)`. Una regla mnemotécnica para evitar que estos datos se mezclen en la mente es que sigma es la letra griega equivalente a nuestra “s”, y la palabra “suma” empieza con esa letra; sin embargo, obviamente no existe “s” ni en “tau” ni en “contar”, pero ambas tienen “t”. Mantengamos en la mente que $\sigma(n)$ suma divisores, mientras $\tau(n)$ cuenta divisores.

En ocasiones, en teoría de números queremos calcular la suma de los cuadrados o cubos de los divisores. Consideremos que 45 tiene como divisores

$$\text{divisores}(45) = \{1, 3, 5, 9, 45, 15\},$$

así que sigma totaliza 78, pues

$$1 + 3 + 5 + 9 + 15 + 45 = 78.$$

En consecuencia, la suma de los cuadrados sería

$$1^2 + 3^2 + 5^2 + 9^2 + 15^2 + 45^2 = 2366$$

y de los cubos,

$$1^3 + 3^3 + 5^3 + 9^3 + 15^3 + 45^3 = 95\,382.$$

En Sage, los comando para esto son `sigma(45, 2)` y `sigma(45, 3)`, respectivamente. De manera análoga, `sigma(45, 1)` es lo mismo que `sigma`.

Hay dos maneras de calcular τ en Sage. Una está dada por `sigma(250, 0)`, que simplemente suma un 1 para cada divisor. En otras palabras, cada divisor es elevado a la potencia 0, y así se convierte en 1. Entonces, al sumar todos esos 1s, tenemos la cantidad de divisores que hay. Otra forma de calcular $\tau(250)$ sería escribiendo

Código de Sage

```
1 len(divisors(250))
```

que nos devuelve la longitud de la lista de divisores de 250 —y, bueno, esa es exactamente la definición de τ —. La palabra clave `len` en Python devuelve la longitud de cualquier lista, sin importar el tipo de problema en cuestión.

⁴Esté prevenido el lector, algunos estadísticos se encuentran ofendidos con los teóricos de números por haber secuestrado el símbolo σ para este propósito, pues a través de la ciencia, ingeniería y estadística, σ es el símbolo para la desviación estándar.

Pares de números amigos Relacionada a este tema hay una tradición desde el periodo de Edad Media.⁵ Considérese dos números particulares, 220 y 248. Los divisores *proprios*⁶ de 220 son

$$\text{divisores}(220) = \{1, 2, 4, 5, 10, 11, 44, 110, 20, 22, 55\},$$

y estos sumarían

$$1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284.$$

Por otro lado, los divisores *proprios* de 284 son

$$\text{divisores}(284) = \{1, 2, 4, 71, 142\},$$

que suman

$$1 + 2 + 4 + 71 + 142 = 220.$$

Por lo tanto 220 y 284 están íntimamente relacionados. El término técnico es que 220 y 284 forman un “par de amigos” enteros. Los árabes encontraron esto tan conmovedor que grababan estos números en joyería que daban a sus esposas, representando la relación íntima. Estos son llamados pares amigables —220 representando a la joven dama y 284 representando al joven caballero—.

Obsérvese que omitimos n mismo cuando sumamos los divisores propios de n . Podemos hacer esto en Sage vía

Código de Sage

```
1 sigma(220) - 220
```

y

Código de Sage

```
1 sigma(284) - 284
```

Otros dos ejemplos de pares de números amigos son 1184 y 1210, así como 2620 y 2924.

4.6.5 Otro uso para tau

Como sabemos, algunas fracciones como $1/4$ y $1/2$, así como $1/25$, pueden ser expresadas como decimales finitos (que terminan). Otros, como $1/3$ o $1/9$ se repetirán infinitamente. Es un teorema muy interesante que una fracción puede expresarse como un decimal exacto que termina con n posiciones decimales (o menos) si y solo si el denominador divide 10^n . Tomémonos un momento para convencernos de esto, o alternativamente, consideremos el siguiente ejemplo.

Si queremos saber qué denominadores resultan en decimales exactos que terminan en dos posiciones después de la coma, solo necesitamos ver qué enteros positivos dividen 100. Esos serían denominadores permisibles. En Sage, escribimos

Código de Sage

```
1 divisors(100)
```

y entonces obtenemos

$$[1, 2, 4, 5, 10, 20, 25, 50, 100]$$

que efectivamente son los únicos denominadores con esta propiedad.

En el sistema binario, la regla análoga sería considerar 2^n . Es obvio que los únicos enteros positivos que dividen 2^n son los números $1, 2, 4, 8, \dots, 2^n$. En general, para números “base b ”, los denominadores admisibles para fracciones que usan dos o menos dígitos después de la coma decimal, serían los divisores de b^2 . Así que, para el sistema binario, $2^2 = 4$, y los denominadores son $\{1, 2, 4\}$.

⁵Aunque los números amigos aparecieron en los escritos de los griegos antiguos, el primer matemático en escribir teoremas acerca de ellos fue Al-Sabi Thabit ibn Qurra al Harrani (826–901), usualmente llamado Thebit or Thebith en los libros de la Europa occidental, quien fue más conocido por su trabajo astronómico y geográfico.

⁶Los divisores propios de un número n son todos sus divisores, excepto n mismo.

Resulta claro que podemos usar este criterio para ver qué bases son convenientes para fracciones y cuáles no lo son. Podemos convertirnos en jueces, comparando los sistemas binario, decimal, Maya (base 20) y Babilonio (base 60).

Primero consideremos el sistema de numeración babilonio, que usa base 60. Con el comando de Sage `divisors(3_600)` vemos que dos símbolos (o menos) serían suficientes para describir cualquier fracción con los siguientes denominadores:

```
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 30, 36, 40, 45,
48, 50, 60, 72, 75, 80, 90, 100, 120, 144, 150, 180, 200, 225, 240, 300,
360, 400, 450, 600, 720, 900, 1200, 1800, 3600]
```

Para el sistema maya, que usa base 20, tenemos una lista más corta, obtenida con `divisors(400)`:

```
[1, 2, 4, 5, 8, 10, 16, 20, 25, 40, 50, 80, 100, 200, 400]
```

Como podemos ver, para cada base ahora conocemos cuántos denominadores admisibles existen:

- Para la base 2, hay 3 denominadores. (Excluyendo el 1, hay solo 2.)
- Para la base 10, hay 9 denominadores. (Excluyendo el 1, hay solo 8.)
- Para la base 20, hay 15 denominadores. (Excluyendo el 1, hay 14.)
- Para la base 60, hay 45 denominadores. (Excluyendo el 1, hay 44.)

Nota: Lo que realmente estamos haciendo es calcular $\tau(n^2)$.

Evidentemente, el sistema babilonio es el más conveniente para divisiones simples de productos, ganado y dinero durante transacciones. No solo hay 45 formas de escribir fracciones de manera exacta usando dos símbolos después de la coma o punto decimal, sino que la lista empieza con

$$\{1, 2, 3, 4, 5, 6, 8, 9, 10, 12, \dots\},$$

es decir, cada posible denominador hasta e incluyendo 12, con la excepción de 7 y 11. Tal vez no resulte sorprendente que usemos base 60 para los ángulos en geometría y trigonometría como consecuencia, así como dividimos las horas en 60 minutos y estos en 60 segundos.

4.6.6 Aritmética modular

Para averiguar qué es 1939 módulo 37, podemos escribir

Código de Sage

```
1 1_939 % 37
```

Pero, a diferencia de cómo se usa en notación matemática, el operador `%` actúa solo sobre el número más cercano, a menos que usen paréntesis para indicar lo contrario. Esto significa que para evaluar

$$(3^2)4 + 2 \pmod{5}$$

no debemos escribir

Código de Sage

```
1 3^2*4 + 2 % 5
```

que evalúa a 38, porque `%` solo está actuando sobre el 2, sino

Código de Sage

```
1 (3^2*4 + 2) % 5
```

que nos da 3, la respuesta correcta. Para evitar estas confusiones, también se puede usar la función `mod`, que es equivalente al operador `%`, pero con una notación más clara. Por ejemplo,

Código de Sage

```
1 mod(3^2*4 + 2, 5)
```

es lo mismo que $(3^2 \cdot 4 + 2) \% 5$, mientras que

Código de Sage

```
1 3^2*4 + mod(2, 5)
```

es lo mismo que $3^2 \cdot 4 + 2 \% 5$. Como podemos ver, esta notación, no deja ninguna duda sobre qué números están involucrados.

El opuesto del operador “mod” es el cociente entero. Para encontrar el cociente entero de un número usamos

Código de Sage

```
1 25 // 4
```

para obtener 6. Este es el resultado $25/4 = 6,25$ redondeado hacia cero. De manera más precisa, $x//y = z$ significa que z es el mayor entero tal que $zy \leq x$. Como $(7)(4) = 28$, vemos que 6 es el mayor entero z tal que $(z)(4) \leq 25$. Nótese que, para cualquier entero x , $x = y * x//y + x\%y$.

4.6.7 Lectura adicional sobre teoría de números

Para el lector que gustaría de explorar sobre Teoría de Números, el autor tiene dos libros que recomendar. El primero es ligeramente más fácil: *An Introduction to Number Theory with Cryptography*, de James S. Kraft y Lawrence C. Washington, publicado en 2013 por Chapman & Hall. El segundo es más enfocado en Sage, y ha sido escrito por el creador de Sage, el Prof. William Stein. El título es *Elementary Number Theory: Primes, Congruences, and Secrets: A Computational Approach*, publicado por Springer bajo la serie “Undergraduate Texts in Mathematics” en 2008.

4.7 Algunos comandos menores de Sage

Aquí tenemos una lista de algunos comandos que aparecen de tiempo en tiempo. Pueden resultar muy útiles en las situaciones en que se requieren.

<code>min(x, y)</code>	Devuelve el mínimo entre x y y .
<code>max(x, y)</code>	Devuelve el máximo entre x y y .
<code>floor(x)</code>	Redondea x hacia abajo, es decir $\lfloor x \rfloor$
<code>ceil(x)</code>	Redondea x arriba, es decir $\lceil x \rceil$
<code>factorial(n)</code>	Este es $n!$, también llamado “ n factorial.”
<code>binomial(x, m)</code>	Este es escrito $\binom{x}{m}$, $\frac{x!}{m!(x-m)!}$ o ${}_xC_m$
<code>binomial(x, m) * m!</code>	Usualmente escrito $\frac{x!}{m!}$ o ${}_xP_m$

4.7.1 Redondeo, pisos y techos

Como podemos ver, el comando `floor` (“piso”) redondea un número hacia abajo, mientras que el comando `ceil` (abreviación de “ceiling” o “techo”) redondea hacia arriba.

En libros antiguos, la función “piso” a veces era llamada “la función máximo entero”. Esto es porque puede ser definida por “el mayor entero que es menor o igual que x ”. Algunos libros de matemáticas más antiguos escriben $\lfloor x \rfloor$ en lugar de $\lfloor x \rfloor$. Por otro lado, podemos definir la función techo, también llamada “función menor entero”, como “el menor entero que es mayor o igual a x ”.

4.7.2 Combinaciones y permutaciones

En mucho textos, particularmente para *Matemáticas Finitas* o *Teoría Cuantitativa*, así como *Probabilidad & Estadística*, se ocupa mucho tiempo hablando de combinaciones y permutaciones. Por ejemplo, si tenemos 52 cartas distintas en un mazo, podríamos preguntarnos cuántas manos de 5 cartas podemos construir con ellas.

Si el orden no importa, tal como en el Póker, entonces tendríamos

$${}_{52}C_5 = \binom{52}{5} = \frac{52!}{5! 47!} = \frac{52 \times 51 \times 50 \times 49 \times 48}{5 \times 4 \times 3 \times 2 \times 1} = 2\,598\,960$$

posibilidades. En Sage calculamos esto con

Código de Sage

```
1 binomial(52, 5)
```

Este es un ejemplo de la fórmula de “combinaciones”.

Sin embargo, si el orden es importante, que no es el caso para el Póker, tendríamos

$${}_{52}P_5 = \binom{52}{5} 5! = \frac{52!}{47!} = 52 \times 51 \times 50 \times 49 \times 48 = 311\,875\,200$$

posibilidades. En Sage podemos escribir

Código de Sage

```
1 factorial(52) / factorial(52 - 5)
```

para ello, o podemos escribir

Código de Sage

```
1 binomial(52, 5) * factorial(5)
```

siendo que ambos devuelven la misma respuesta. Este es un ejemplo de la fórmula de “permutaciones”.

Uno puede preguntarse por qué Sage no cuenta con un comando para permutaciones, cuando sí tiene uno para combinaciones. Pues bien, existe un comando para permutaciones, pero hace algo un poco diferente al de arriba. En el ejemplo anterior calculamos *cuántas* permutaciones hay, pero en ocasiones es preferible saber cuáles son estas. Para eso, uno puede escribir

Código de Sage

```
1 S = Permutations(['as', 'rey', 'reina', 'bufón'])
2 print('Esperamos', len(S.list()), 'ítems abajo.')
3 S.list()
```

lo que obedientemente mostrará todas las 24 posibilidades:

Esperamos 24 ítems abajo.

```
[[ 'as', 'rey', 'reina', 'bufón'], [ 'as', 'rey', 'bufón', 'reina'],
[ 'as', 'reina', 'rey', 'bufón'], [ 'as', 'reina', 'bufón', 'rey'],
[ 'as', 'bufón', 'rey', 'reina'], [ 'as', 'bufón', 'reina', 'rey'],
[ 'rey', 'as', 'reina', 'bufón'], [ 'rey', 'as', 'bufón', 'reina'],
[ 'rey', 'reina', 'as', 'bufón'], [ 'rey', 'reina', 'bufón', 'as'],
[ 'rey', 'bufón', 'as', 'reina'], [ 'rey', 'bufón', 'reina', 'as'],
[ 'reina', 'as', 'rey', 'bufón'], [ 'reina', 'as', 'bufón', 'rey'],
[ 'reina', 'rey', 'as', 'bufón'], [ 'reina', 'rey', 'bufón', 'as'],
[ 'reina', 'bufón', 'as', 'rey'], [ 'reina', 'bufón', 'rey', 'as'],
[ 'bufón', 'as', 'rey', 'reina'], [ 'bufón', 'as', 'reina', 'rey'],
[ 'bufón', 'rey', 'as', 'reina'], [ 'bufón', 'rey', 'reina', 'as'],
[ 'bufón', 'reina', 'as', 'rey'], [ 'bufón', 'reina', 'rey', 'as']]
```

El lector también puede intentar

Código de Sage

```
1 S = Permutations(['as', 'rey', 'reina', 'bufón', 'diez', 'nueve'])
2 print('Esperamos', len(S.list()), 'ítems abajo.')
3 S.list()
```

si tiene curiosidad.

De hecho, este es uno de los muchos usos de la infraestructura “Permutations” en Sage, pero no exploraremos otros aquí por ser un tópico muy avanzado.

4.7.3 Las funciones trigonométricas hiperbólicas

Si el lector ha aprendido las funciones trigonométricas hiperbólicas, esa es una buena noticia. Personalmente, al autor nunca le fueron enseñadas en un curso (a pesar de tener un PhD), pero sí sabe que resultan útiles en ocasiones. Estas aparecen en física, principalmente en el estudio de la curva catenaria, que es la forma de un cable al que se le permite colgar bajo su propio peso, pero sujetado por los extremos. También, existen algunas integrales cuya solución resulta más compacta si se usan funciones hiperbólicas. Pero a final de cuentas, uno no debe preocuparse si nunca las aprendió.

Hay doce comandos para estas funciones, los cuales se detallan en la tabla 2.

Término matemático	Funciones de Sage	
Seno hiperbólico	$\sinh(x)$	
Coseno hiperbólico	$\cosh(x)$	
Tangente hiperbólica	$\tanh(x)$	
Cotangente hiperbólica	$\coth(x)$	
Secante hiperbólica	$\operatorname{sech}(x)$	
Cosecante hiperbólica	$\operatorname{csch}(x)$	
Seno hiperbólico inverso	$\operatorname{arcsinh}(x)$	$\operatorname{asinh}(x)$
Coseno hiperbólico inverso	$\operatorname{arccosh}(x)$	$\operatorname{acosh}(x)$
Tangente hiperbólica inversa	$\operatorname{arctanh}(x)$	$\operatorname{atanh}(x)$
Cotangente hiperbólica inversa	$\operatorname{arccoth}(x)$	$\operatorname{acoth}(x)$
Secante hiperbólica inversa	$\operatorname{arcsech}(x)$	$\operatorname{asech}(x)$
Cosecante hiperbólica inversa	$\operatorname{arccsch}(x)$	$\operatorname{acsch}(x)$

Tabla 2 Las funciones trigonométricas hiperbólicas y sus inversas

4.8 Calculando límites expresamente

Si queremos calcular

$$\lim_{x \rightarrow 0} \frac{x - 2}{x - 3}$$

podemos escribir

Código de Sage

```
1 limit((x - 2) / (x - 3), x=0)
```

con lo que veríamos que la respuesta es $2/3$, lo que puede resultar trivial. Tal vez menos obvio es el límite

$$\lim_{x \rightarrow 1} \frac{x^2 - 4x + 3}{x^2 - 3x + 2},$$

para el cual escribimos

Código de Sage

```
1 limit((x^2 - 4*x + 3) / (x^2 - 3*x + 2), x=1)
```

obteniendo la respuesta 2. Si el lector desconoce cómo hacer este cálculo sin una computadora o calculadora (es decir, a mano), entonces sugerimos intentar factorizar el numerador y el denominador como primer paso.

Si quisiéramos verificar el extraño, pero fascinante hecho que

$$\lim_{x \rightarrow 0} (\cos x)^{1/x^2} = \frac{1}{\sqrt{e}},$$

entonces podríamos escribir

Código de Sage

```
1 limit(cos(x)^(1/x^2), x=0)
```

que nos devuelve la respuesta deseada (aunque aún sorprendente) de

$$e^{(-1/2)}$$

Para ver que este último límite es efectivamente cierto, uno podría intentar reemplazar valores como $x = 0,001$ en cualquier calculadora científica o con Sage. Pero la demostración formal de que este límite es cierto es bastante complicada —requiere dos aplicaciones de la Regla de L'Hôpital o la Serie de Taylor (pero no ambas), según conoce el autor—.

Límites infinitos En ocasiones los límites pueden ir al infinito. Por ejemplo,

$$\lim_{x \rightarrow 0^+} \frac{1}{x^2} = \lim_{x \rightarrow 0^-} \frac{1}{x^2} = \lim_{x \rightarrow 0} \frac{1}{x^2} = \infty$$

es el clásico ejemplo en la mayoría de textos de cálculo. Sin embargo, si colocamos $1/x^3$ en lugar de $1/x^2$ en estos límites, entonces tendríamos un problema. Si nos aproximamos a 0 desde los números positivos, el límite para $1/x^3$ es $+\infty$, pero si nos aproximamos a 0 desde los números negativos, el límite de $1/x^3$ es $-\infty$. Por lo tanto, el límite no existe cuando x se aproxima a 0 en general.

Para ver cómo maneja Sage esta situación, consideremos el siguiente código:

Código de Sage

```
1 print(limit(1/x^2, x=0))
2 print(limit(1/x^3, x=0))
3 print(limit(1/x^4, x=0))
4 print(limit(-1/x^4, x=0))
```

Este genera la siguiente salida:

```
+Infinity
Infinity
+Infinity
-Infinity
```

Es claro que Sage reportará fielmente $+\text{Infinity}$ cuando el límite va a $+\infty$, y reportará fielmente $-\text{Infinity}$ cuando el límite vaya a $-\infty$. El “Infinity” sin signo puede resultar un poco sorprendente. Podemos clarificar su significado escribiendo

Código de Sage

```

1 print(limit(1/x^3, x=0, dir='right'))
2 print(limit(1/x^3, x=0, dir='left'))
3 print(limit(1/x^3, x=0))

```

Este código produce la siguiente salida:

```

+Infinity
-Infinity
Infinity

```

Esencialmente, Sage indica Infinity como el límite de $f(x)$ cuandoquiera que

$$\lim_{x \rightarrow 0} |f(x)| = +\infty,$$

suponiendo que las condiciones para reportar +Infinity o -Infinity para $f(x)$ no se satisfagan. Muchos profesores de cálculo podrían molestarse con esta convención, sin embargo. Tradicionalmente, se dice que el límite de $1/x^3$ cuando x tiende a cero *no existe*.

A propósito de esto, `dir='right'` también puede especificarse como `dir='+'` o `dir='plus'`. Análogamente, `dir='left'` es equivalente a `dir='-'` y `dir='minus'`. Diferentes mentes encuentran uno u otro esquema de nombrado más o menos intuitivo. Sin embargo, lo importante es recordar que los límites desde el lado negativo y el lado positivo son en ocasiones diferentes. Esta es una trampa común en los exámenes de cálculo.

4.9 Gráficas de dispersión en Sage

Supongamos que estamos analizando alguna clase de datos expresados por los siguientes puntos:

$$\{(0; 7,1), (1; 5,2), (2; 2,9), (3; 1,05), (4; -0,9)\}.$$

Primero, debemos indicar a Sage acerca de los datos usando el siguiente comando, que es otro ejemplo de una lista en Sage:

Código de Sage

```

1 puntos_datos = [(0,7.1), (1,5.2), (2,2.9), (3,1.05), (4,-0.9)]

```

Ahora podemos crear fácilmente un gráfico de dispersión, simplemente escribiendo

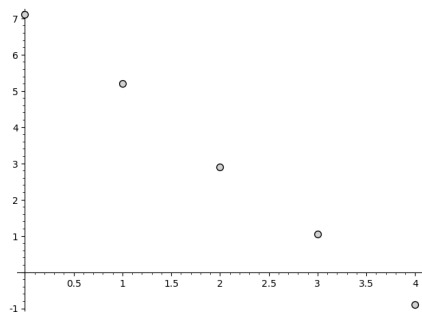
Código de Sage

```

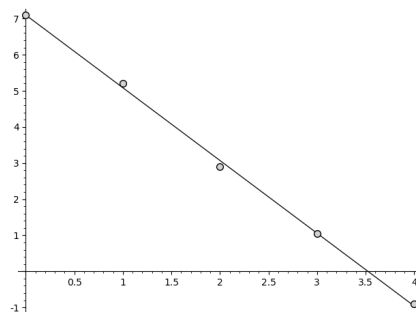
2 scatter_plot(puntos_datos)

```

que producirá la imagen mostrada a continuación a la izquierda.



Sin recta de mejor ajuste



Con recta de mejor ajuste

Probablemente el lector tenga la costumbre de calcular “rectas de mejor ajuste” en una herramienta de hoja de cálculo o con algún paquete estadístico. Si es así, eso es excelente; si no, Sage es muy capaz de hacer esto por nosotros. Explicaremos cómo en la subsección 4.16.4. Mientras tanto, la recta de mejor ajuste para este ejemplo es $y = 7.1 - 2.015x$, y el gráfico de dispersión de la derecha en la figura anterior, incluye esta línea. Como podemos apreciar, es un ajuste bastante bueno, aunque uno imperfecto. Esta figura fue creada con los comandos

Código de Sage

```
1 scatter_plot(datapoints) + plot(7.1 - 2.015*x, 0, 4)
```

Ahora consideremos otro conjunto de datos. Este ejemplo nos mostrará por qué siquiera nos molestamos en hacer un gráfico, cuando hay herramientas, incluyendo Sage mismo, que pueden hallar la recta de mejor ajuste sin necesidad de graficarla. Primero, ingresemos los datos

Código de Sage

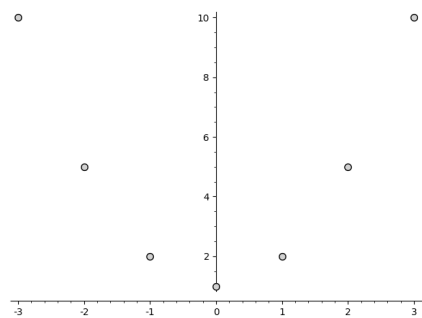
```
1 otros_datos = [(-3,10), (-2,5), (-1,2), (0,1), (1,2), (2,5), (3,10)]
```

Imaginemos que estos son datos de algún experimento científico, y que nuestro compañero de laboratorio los ha copiado y ha encontrado la recta de mejor ajuste, $y = 0x + 5$, usando algún paquete estadístico, pero no ha creado la gráfica de dispersión. Nosotros, por supuesto, somos conscientes de las mejores prácticas científicas, y por lo tanto escribimos

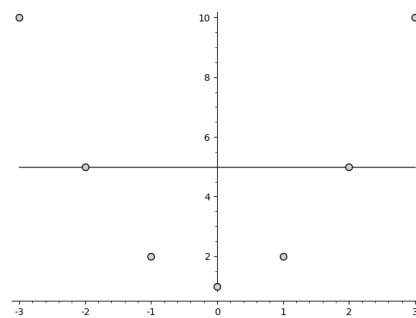
Código de Sage

```
1 scatter_plot(otros_datos)
```

para obtener la imagen de abajo a la izquierda. Entonces podemos añadir la recta $y = 0x + 5$ para obtener la gráfica abajo a la derecha



Sin recta de mejor ajuste



Con recta de mejor ajuste

Podemos ver que esto es absurdo. Los datos claramente forman una parábola. Una línea es una terrible elección para aproximar estos datos. Aun cuando, de todas las rectas que podrían ser trazadas (todas las cuales tendrán un error inaceptablemente alto), la que tiene el menor error (cuadrático) posible para este conjunto es $y = 0x + 5$, sigue siendo manifiestamente claro que es una horrible aproximación, como se ve en la figura anterior. Es por esta razón que hacer un gráfico de dispersión con la recta de mejor aproximación es considerado un “procedimiento operativo estándar” en la mayoría de los laboratorios. Esta figura fue producida con el comando:

Código de Sage

```
1 scatter_plot(otros_datos) + plot(0*x + 5, -3, 3).
```

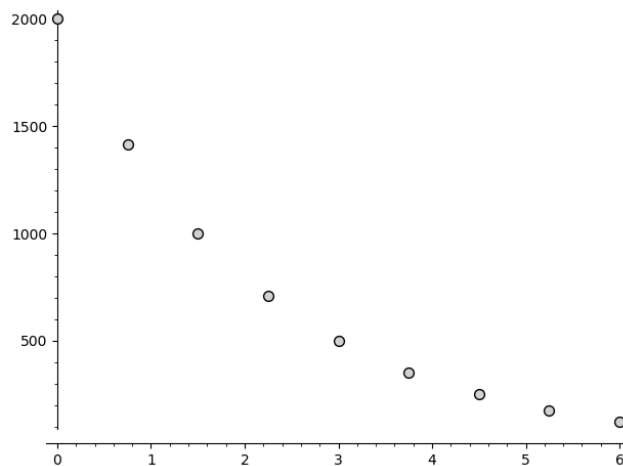
Ahora es un buen momento para explicar que el objeto que los científicos llaman la “recta de mejor ajuste” es de hecho mejor conocido entre los matemáticos como “regresión lineal”. Con esto queremos indicar que se analiza un conjunto de datos para encontrar la recta que mejor los aproxime o ajuste. En cualquier caso, aquí debíamos haber hecho una “regresión cuadrática”, que nos daría la parábola que mejor ajusta los datos. En unas cuantas páginas veremos cómo hacer esto.

Un ejemplo de química Tal vez el lector piense que el ejemplo de la parábola es algo artificial. Si es así, está en lo correcto: el autor lo diseñó solamente para probar un punto. Sin embargo, los siguientes datos ilustran cómo una muestra de 2 kilogramos de Cobalto-62 podría decaer. Aquí, la coordenada x es el tiempo en minutos y la coordenada y es el peso en gramos.

Código de Sage

```
1 datos_cobalto = [(0,2_000), (0.75,1_414), (1.5,1_000), (2.25,707),
↪ (3.00,500), (3.75,353), (4.50,250), (5.25,177), (6.00,125)]
```

Podemos escribir `scatter_plot(datos_cobalto)` para obtener el gráfico



Un estudiante imprudente podría verse tentado a hacer una regresión lineal, especialmente si unos cuantos reportes anteriores lo requirieran. La regresión lineal de estos datos resulta ser $y = -290,333x + 1596,11$. En ese caso, uno obtiene la gráfica mostrada a la izquierda de la figura 4.2. Sin embargo, un estudiante juicioso debería recordar que el decaimiento radiactivo sigue una curva exponencial. En lugar de pedir $y = ax + b$, que es una recta, pediría $y = ae^{bx}$, que es una exponencial. Esta es parte de la belleza de las herramientas de regresión de Sage, que una regresión lineal es tan fácil como una cuadrática, exponencial o logarítmica. Veremos estas herramientas en la siguiente sección. Mientras tanto, la regresión exponencial resulta ser $y = 1999,96e^{-0,462162x}$, lo que nos da la curva a la derecha de la figura 4.2.

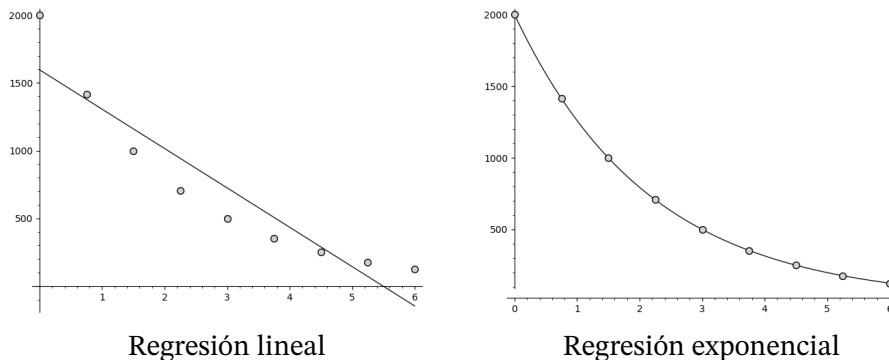


Figura 4.2 Dos gráficas que ajustan el conjunto de datos para el cobalto

La moraleja de la historia es que, aunque es interesante calcular regresiones de datos, no debemos simplemente asumir que una regresión lineal es lo que queremos. Dividir el mundo de las funciones en funciones lineales y no lineales es como dividir el mundo de los seres vivos del planeta Tierra en armadillos y no armadillos. La mayoría de las funciones son no lineales y por lo tanto no debemos asumir que nuestros datos están mejor servidos por un modelo lineal.

4.10 Haciendo nuestras propias regresiones en Sage

Esta lección asume que el lector ha leído la sección anterior sobre gráficos de dispersión. En particular, usaremos los datos sobre el cobalto, que hemos llamado `datos_cobalto`. Por las próximas páginas, necesitaremos esos datos como primera línea en el servidor Sage Cell. Si se está usando CoCalc, estos solo necesitan estar en la misma página, antes que los comandos discutidos a continuación.

Primero intentaremos ajustar los datos con una regresión lineal. Esto significa que le pediremos a Sage que nos dé una función $y = ax + b$ que modele, como mejor sea posible, estos nueve puntos. Los comandos para esto son

Código de Sage

```
2 var('a b')
3 modelo(x) = a*x + b
4 find_fit(datos_cobalto, modelo)
```

donde el comando `var` es uno que hemos estudiado antes (véase la página 36), pero los otros dos restantes son nuevos para nosotros. La respuesta que obtenemos es

```
[a == -290.33333269040816, b == 1596.1111098318909]
```

que es la forma que tiene Sage de decirnos que la regresión lineal es

$$y = -290,333x + 1596,11.$$

A continuación, podemos graficar esta función con

Código de Sage

```
5 scatter_plot(datos_cobalto) + plot(-290.333*x+1_596.11, 0, 6)
```

La imagen resultante puede observarse a la izquierda en la figura 4.2.

Como podemos ver, este no es ajuste muy bueno. Por supuesto, la química básica nos dice que el decaimiento radiactivo sigue una curva exponencial. Para hacer una regresión exponencial escribimos

Código de Sage

```
2 var('a b')
3 modelo(x) = a * exp(b*x)
4 find_fit(datos_cobalto, modelo)
```

Solo hemos cambiado la línea del medio de los anteriores comandos. Es decir, el modelo ha cambiado de $y = ax + b$ a $y = ae^{bx}$. Ahora la respuesta de Sage es

```
[a == 1999.9607020042693, b == -0.462162180693913]
```

que significa que nuestro modelo es

$$y = 1999,96e^{-0,462162x}.$$

Inmediatamente podemos ver que hay un error de redondeo. El primer coeficiente 1999,96 realmente debería ser 2000, pues, si recordamos los datos del cobalto, empezamos con 2000 gramos. Esto es parte de los pros y contras de los métodos numéricos: podemos hacer casi cualquier cosa con métodos numéricos, pero muy rara vez podemos ser exactos. En cualquier caso, apreciaremos la calidad de esta regresión con el comando

Código de Sage

```
5 scatter_plot(datos_cobalto) + plot(1_999.96*exp(-0.462_162*x), 0, 6)
```

El gráfico obtenido se puede apreciar a la derecha en la figura 4.2.

Ahora bien, por otro lado, uno puede considerar una regresión cuadrática, es decir, algo de la forma $y = ax^2 + bx + c$. Los comandos para ello serían

Código de Sage

```
2 var('a b c')
3 modelo(x) = a*x^2 + b*x + c
4 find_fit(datos_cobalto, modelo)
```

donde el único cambio real de nuestro código anterior es nuevamente la línea media, aunque también hemos declarado c como una variable mediante el comando `var`. La respuesta de Sage es

```
[a == 62.755170737557854, b == -666.8643577216476, c ==
1925.5757574133333]
```

que nos indica que el modelo es

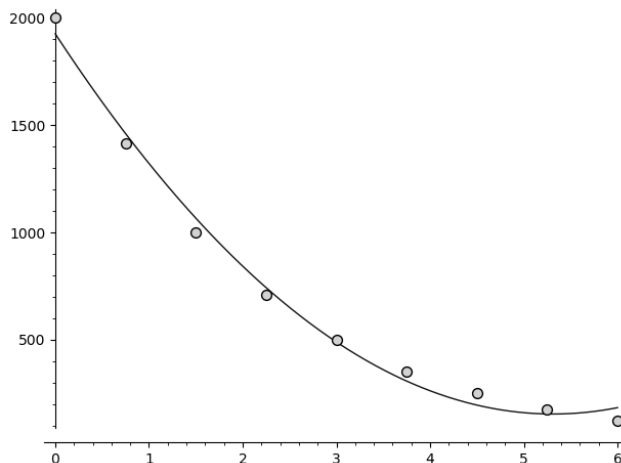
$$y = 62,7551x^2 - 666,864x + 1925,57$$

A continuación, graficamos esta función con

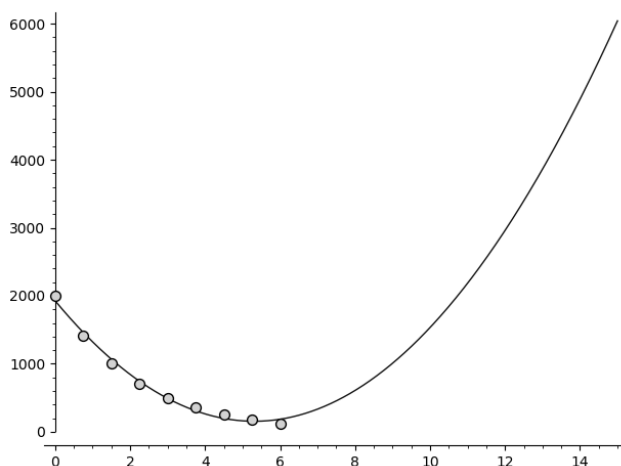
Código de Sage

```
5 scatter_plot(datos_cobalto) + plot(62.755_1*x^2-666.864*x+1_925.57, 0, 6)
```

que produce la imagen



Aunque esta claramente no es una muy mala aproximación, es poco prudente modelar el decaimiento radiactivo con una función cuadrática. Esto no solo se debe a que “fallamos” algunos puntos. El problema puede apreciarse más fácilmente permitiendo que el tiempo tenga el rango 0 a 15, en lugar de 0 a 6. Solo cambiemos lo necesario en la última línea del código anterior. Obtenemos la siguiente imagen



que claramente nos muestra que este *no* es buen modelo para decaimiento radiactivo.

4.11 Calculando en octal, binario y hexadecimal

Para encontrar el valor decimal del hexadecimal 71, escribimos `0x71` y obtenemos 113. Esto es porque $7 \times 16 + 1 = 113$. Notemos que es un cero antes de x, no la letra “o”. Alternativamente, para el hexadecimal 1F escribiríamos `0x1F` y obtendríamos 31, pues $1 \times 16 + 15 = 31$.

El sistema octal es considerablemente menos común, pero aparece de tiempo en tiempo. Para el octal 11 escribimos `0o11` (primero un cero y luego una “o”). La respuesta es 9, pues $1 \times 8 + 1 = 9$.

El sistema binario es similar al hexadecimal, excepto con una b en lugar de x. Así, para encontrar qué es el binario 1101, debemos escribir `0b1101` para ver que la respuesta es 13. Esto es porque

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13.$$

Para ir en reversa, podemos escribir `395.hex()` para obtener la expansión hexadecimal 18b, pues

$$18b_{\text{hex}} = 1 \times 16^2 + 8 \times 16 + 11 = 395.$$

Similarmente, para la expansión binaria escribimos `63.binary()` y obtendremos 111 111, pues

$$111111_{\text{bin}} = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 = 63.$$

La expansión octal la obtenemos con `31.oct()`, que nos da la respuesta 37, pues

$$37_{\text{oct}} = 3 \times 8 + 7 = 24 + 7 = 31.$$

4.12 ¿Puede Sage resolver el Sudoku?

Para crear un Sudoku de 9×9 , definido por una matriz de orden 9×9 , podemos ingresar por ejemplo

Código de Sage

```
1 A = matrix(9, 9, [5,0,0, 0,8,0, 0,4,9, 0,0,0, 5,0,0, 0,3,0, 0,6,7, 3,0,0,
  ↪ 0,0,1, 1,5,0, 0,0,0, 0,0,0, 0,0,0, 2,0,8, 0,0,0, 0,0,0, 0,0,0, 0,1,8,
  ↪ 7,0,0, 0,0,4, 1,5,0, 0,3,0, 0,0,2, 0,0,0, 4,9,0, 0,5,0, 0,0,3])
```

Entonces, si escribimos `print(A)` en una línea propia, obtenemos

```
[5 0 0 0 8 0 0 4 9]
[0 0 0 5 0 0 0 3 0]
[0 6 7 3 0 0 0 0 1]
[1 5 0 0 0 0 0 0 0]
[0 0 0 2 0 8 0 0 0]
[0 0 0 0 0 0 0 1 8]
[7 0 0 0 0 4 1 5 0]
[0 3 0 0 0 2 0 0 0]
[4 9 0 0 5 0 0 0 3]
```

que tiene algunas entradas no nulas, pero la mayoría son ceros para marcar dónde irían los espacios en blanco, en un Sudoku ordinario.

Ahora podemos resolver el juego escribiendo simplemente

Código de Sage

```
1 sudoku(A)
```

que nos muestra

```
[5 1 3 6 8 7 2 4 9]
[8 4 9 5 2 1 6 3 7]
[2 6 7 3 4 9 5 8 1]
[1 5 8 4 6 3 9 7 2]
[9 7 4 2 1 8 3 6 5]
[3 2 6 7 9 5 4 1 8]
[7 8 2 9 3 4 1 5 6]
[6 3 5 1 7 2 8 9 4]
[4 9 1 8 5 6 7 2 3]
```

la cual es una solución válida al rompecabezas, aunque pueden existir otras.

4.13 Midiendo la velocidad de Sage

Se supone que factorizar números grandes es difícil y tal vez querramos saber cuánto tarda Sage en ejecutar el comando

Código de Sage

```
1 factor(2_250_000_063_000_000_041)
```

en cuyo caso escribiríamos

Código de Sage

```
1 timeit('factor(2_250_000_063_000_000_041)')
```

En la computadora en que se compiló este libro, la respuesta a esta instrucción fue

625 loops, best of 3: 39.2 μ s per loop

lo que significa que Sage lo intentó 625 veces, y el mejor estimado para el tiempo de ejecución es 39,2 microsegundos, o aproximadamente $1/25\,523$ segundos.

Por supuesto, el tiempo de respuesta cuando trabajamos con el servidor Sage Cell es mucho más largo que lo que acabamos de obtener. ¿Qué está pasando? La mayoría del tiempo usando Sage Cell es desperdiciado con los comandos viajando por la internet al servidor y de vuelta. También, parte del tiempo se gasta mostrando los resultados. En este caso, probablemente uno pierda dos segundos en tiempo de tránsito, un cuarto de segundo en mostrar los resultados y en la interfaz, y cerca de 0 segundos en el cálculo.

Sin embargo, para ciertos problemas *muy duros*, podemos emplear 10 minutos en cálculo, dos segundos en tránsito y un cuarto de segundo en impresión en pantalla e interfaz. Así que esta característica de Sage está ahí para ayudarnos a medir eso.

4.14 Números gigantescos y Sage

En ocasiones al estudiar combinatoria, uno debe lidiar con números gigantescos. Por ejemplo, pedirle a Sage `factorial(52)` resultará en

80658175170943878571660636856403766975289505440883277824000000000000

Es difícil comprender lo que esto significa; sin embargo, podemos escribir

Código de Sage

```
1 floor(log(factorial(52), 10)) + 1
```

que nos dará 65. Esto nos indica el número de dígitos en el resultado, como explicamos a continuación.

¿Cómo funciona esto? Sabemos que $\log_{10}(10^7) = 7$ y $\log_{10}(10^8) = 8$. De acuerdo a esto, cualquier número x tal que $10^7 \leq x < 10^8$ cumple por lo tanto $7 \leq \log_{10}(x) < 8$. Sin embargo, ¿qué números cumplen $10^7 \leq x < 10^8$? ¡Precisamente los que tiene ocho dígitos, por supuesto! Entonces, el conjunto de todos los números que tienen 8 dígitos es el conjunto de x s tales que $7 \leq \log_{10}(x) < 8$. Al tomar el suelo con `floor`, redondeamos hacia abajo y entonces debemos compensar con un `+1`.

Escribir `factorial(factorial(7))` para calcular $(7!)!$ también será interesante. Obtenemos un número extremadamente grande si hacemos esto. De hecho, usando la técnica anterior, podemos saber que $(7!)!$ tiene 16 474 dígitos. Este número se mostrará en Sage Cell como una línea de texto muy larga, en un campo con una barra deslizadora. Al mover la barra, podremos ver la parte del resultado que quepa en la pantalla. En CoCalc, la salida se mostrará completa, pero en varias líneas de texto, convenientemente cortadas de manera que se ajusten lo mejor posible a nuestra pantalla.

Entonces podríamos ver que `respuestas[2]` es la raíz en la que estamos interesados. Sin embargo, ¿cómo vamos a componer eficientemente el texto de la fórmula

$$x == \frac{1}{3} \cdot 5^{\frac{1}{3}} \cdot (3\sqrt[3]{3}\sqrt[3]{2} - 7)^{\frac{1}{3}} - \frac{1}{3} \cdot 5^{\frac{2}{3}} / (3\sqrt[3]{3}\sqrt[3]{2} - 7)^{\frac{1}{3}} - \frac{2}{3}$$

en \LaTeX para un artículo de conferencia o para una revista científica? Por suerte, Sage hará esto por nosotros.

El siguiente código:

Código de Sage

```
1 respuestas = solve(x^4 + x^3 + x^2 + x - 4, x)
2
3 print(respuestas[2])
4 print()
5 print(latex(respuestas[2]))
```

produce la siguiente salida:

$$x == \frac{1}{3} \cdot 5^{\frac{1}{3}} \cdot (3\sqrt[3]{3}\sqrt[3]{2} - 7)^{\frac{1}{3}} - \frac{1}{3} \cdot 5^{\frac{2}{3}} / (3\sqrt[3]{3}\sqrt[3]{2} - 7)^{\frac{1}{3}} - \frac{2}{3}$$

$$x = \frac{1}{3} \cdot 5^{\frac{1}{3}} \cdot \left(3\sqrt[3]{3}\sqrt[3]{2} - 7\right)^{\frac{1}{3}} - \frac{5^{\frac{2}{3}}}{3\left(3\sqrt[3]{3}\sqrt[3]{2} - 7\right)^{\frac{1}{3}}} + \frac{-2}{3}$$

Como podemos ver, la primera línea está escrita en Sage y la segunda está escrita en \LaTeX . Este último código produce una composición tipográfica muy elegante, aunque aún imperfecta:

$$x = \frac{1}{3} \cdot 5^{\frac{1}{3}} \cdot \left(3\sqrt[3]{3}\sqrt[3]{2} - 7\right)^{\frac{1}{3}} - \frac{5^{\frac{2}{3}}}{3\left(3\sqrt[3]{3}\sqrt[3]{2} - 7\right)^{\frac{1}{3}}} + \frac{-2}{3}$$

Sin embargo, es un buen punto de partida para añadir a una publicación.

4.16 Matrices en Sage, parte tres

Gran parte de Sage usa álgebra lineal para realizar sus tareas, así que no es sorprendente que tenga grandes conjuntos de comandos dedicados a esta rama de las matemáticas. Hay demasiado para tratarlo aquí, pero el autor ha seleccionado tres temas importantes para completar nuestra tercera sección sobre álgebra lineal. Estos son problemas de autovalores/autovectores, factorizaciones de matrices y formas canónicas, y resolución de problemas de mínimos cuadrados.

4.16.1 Introducción a autovectores

Un autovector derecho de la matriz A es cualquier vector $\vec{v} \neq \vec{0}$ tal que existe algún número real o complejo λ con $A\vec{v} = \lambda\vec{v}$. En otras palabras, cuando multiplicamos $A\vec{v}$, obtenemos el doble de \vec{v} , o un tercio de \vec{v} , o un millón de veces \vec{v} , o algún múltiplo constante de \vec{v} . Este valor λ se llama “autovalor”. Demos por hecho que estos vectores son muy importantes en ciertas aplicaciones científicas y enfoquémonos en cómo encontrarlos en su lugar.

Si A es una matriz de $n \times n$, entonces sea I_n la matriz identidad de $n \times n$. Las siguientes formulaciones son equivalentes:

$$\begin{aligned} A\vec{v} &= \lambda\vec{v} \\ A\vec{v} - \lambda\vec{v} &= \vec{0} \\ A\vec{v} - \lambda(I_n\vec{v}) &= \vec{0} \\ (A - \lambda(I_n))\vec{v} &= \vec{0} \end{aligned}$$

De esta última ecuación es claro que para cualquier vector interesante (es decir, no nulo) \vec{v} a encontrar, la matriz $(A - \lambda I_n)$ debe ser singular. Por supuesto esto significa que el determinante de $(A - \lambda I_n)$ es cero. Aún no sabemos qué λ es este, así que llamémoslo x . Estamos esencialmente construyendo una nueva matriz donde todas las entradas fuera de la diagonal coinciden con las de A , pero en la diagonal las entradas de A tienen un “ $-x$ ” adjunto. Podemos pedir a Sage que calcule el determinante de esta matriz, o podemos calcularlo a mano. Como sea, sabemos que si x es un autovalor, entonces este determinante debe ser cero.

Veamos un pequeño ejemplo:

$$A = \begin{bmatrix} 0 & 0 & 0 & -4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 0 \end{bmatrix} \xrightarrow{\text{se convierte en}} A - xI_n = \begin{bmatrix} 0-x & 0 & 0 & -4 \\ 1 & 0-x & 0 & 0 \\ 0 & 1 & 0-x & 5 \\ 0 & 0 & 1 & 0-x \end{bmatrix}$$

Ahora bien, resulta que este determinante es

$$\det(A - xI_n) = x^4 - 5x^2 + 4$$

que nos puede preocupar en un principio, pues es un polinomio de grado 4; sin embargo, es bicuadrático, así que se factoriza fácilmente:

$$x^4 - 5x^2 + 4 = (x+2)(x+1)(x-1)(x-2)$$

Este polinomio que acabamos de hallar es llamado el *polinomio característico de A* . Ahora sabemos que los valores de x que serán autovalores son

$$x \in \{-2, -1, 1, 2\}$$

y ahora podemos proceder a encontrar los autovectores. Solo tomando $\lambda = 2$ por ejemplo, tenemos que resolver

$$(A - \lambda(I_n))\vec{v} = (A - 2I_n)\vec{v} = \vec{0}.$$

Pero este es solo un sistema de ecuaciones lineales. Aprendimos cómo resolver esto en la página 132 usando el operador quebrado invertido o el comando `solve_right`. Debemos hacer esto para cada uno de los cuatro autovalores, lo cual claramente no es un problema difícil.

Hay otras formas de encontrar los autovectores. Lo que fue conveniente aquí es que el polinomio era fácil de manejar. Como no está garantizado que los polinomios de grado 5 o mayor sean resolubles, nuestra técnica previa (de factorizar el polinomio característico) puede ser computacionalmente infactible. Por suerte, Sage se encarga de esto por nosotros. En Sage, el proceso entero anterior puede ser racionalizado como abajo:

Código de Sage

```

1  A = matrix([[0, 0, 0, -4], [1, 0, 0, 0], [0, 1, 0, 5], [0, 0, 1, 0]])
2  Id = identity_matrix(4)
3
4  print('A:')
5  print(A)
6  print()
7  print('Id:')
8  print(Id)
9  print()
10 print(A-Id*x)
11 print()
12 print(det(A-Id*x))
13 print(factor(det(A-Id*x)))

```

Esto produce la salida

```

A:
[ 0  0  0 -4]
[ 1  0  0  0]
[ 0  1  0  5]
[ 0  0  1  0]

Id:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

[-x  0  0 -4]
[ 1 -x  0  0]
[ 0  1 -x  5]
[ 0  0  1 -x]

x^4 - 5*x^2 + 4
(x + 2)*(x + 1)*(x - 1)*(x - 2)

```

A propósito, el polinomio característico guarda una relación fascinante con su matriz. Si escribimos

Código de Sage

```

1  print(A^4 - 5*A^2 + 4)

```

donde hemos tomado la matriz A y la hemos tratado como si fuera x , obtenemos la matriz cero como respuesta —eso ocurrirá siempre—. Algunas veces se expresa esta interesante propiedad diciendo que “una matriz es raíz de su propio polinomio característico”⁷.

⁷Este resultado se llama el *Teorema de Cayley-Hamilton*, en honor a Arthur Cayley (1821–1895) y William Rowan Hamilton (1805–1865).

4.16.2 Encontrando autovalores de manera eficiente en Sage

El proceso de la subsección anterior fue para darnos una idea de lo que realmente significa hallar autovalores y autovectores. Los procedimientos usados pueden ser interesantes tanto en las matemáticas de las que dependen, como en la ciencia computacional algorítmica escogida para hacer que el proceso corra eficientemente. No podemos entrar en esos detalles aquí, pero el autor recomienda el libro de Lloyd Trefethen y David Bau, *Numerical Linear Algebra*, publicado por la Sociedad para la Matemática Industrial y Aplicada en 1997.

Podemos encontrar los autovalores y autovectores directamente con los comandos

Código de Sage

```
1 A.eigenvalues()
2 A.eigenvectors_left()
3 A.eigenvectors_right()
```

Sin embargo, en ocasiones el polinomio característico es valioso por derecho propio. Para obtenerlo, podemos usar cualquiera de los siguientes comandos:

Código de Sage

```
1 A.charpoly()
2 factor(A.charpoly())
```

Para demostrar las ventajas y desventajas de estos métodos estudiaremos la siguiente matriz:

$$B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Consideremos el siguiente código:

Código de Sage

```
1 B = matrix(2, 2, [1, 2, 3, 4])
2
3 print(B.eigenvectors_right())
4 print()
5 b(x) = B.charpoly()
6 print(solve(b(x)==0, x))
```

que produce la salida:

```
[(-0.3722813232690144?, [(1, -0.6861406616345072?)], 1),
 (5.372281323269015?, [(1, 2.186140661634508?)], 1)]

[
x == -1/2*sqrt(33) + 5/2,
x == 1/2*sqrt(33) + 5/2
]
```

Vemos en la parte superior lo que ya conocemos como una lista en Sage o Python. Cada uno de los dos ítems en la lista es una tripleta ordenada, que consiste de un autovalor λ , un autovector asociado \vec{v} , y la multiplicidad del autovalor como raíz del polinomio característico. Por debajo está la salida de `solve`, donde hemos encontrado explícitamente las raíces del polinomio característico; es decir, hemos obtenido la forma algebraica exacta de los autovalores. Esto puede resultar útil, pues de otro modo, no conocemos realmente el significado de los valores numéricos. Sin embargo, en muchas aplicaciones solo se requieren los valores numéricos. Notemos cómo Sage nos recuerda que los autovalores y autovectores obtenidos con `eigenvectors_right` son aproximaciones, usando el símbolo “?” después del dígito menos significativo.

Antes indicamos que si introducimos la matriz A en lugar de la x dentro de su polinomio característico, entonces obtenemos la matriz cero. El polinomio característico es usualmente, pero no siempre, el polinomio de menor grado que cumple con esta propiedad. Sin embargo, si el polinomio característico tiene raíces repetidas, existe otro, que es de menor grado, que también enviará A a la matriz cero. Este es el *polinomio mínimo* de A , y se puede ser hallado con el comando `A.minpoly()`.

4.16.3 Factorizaciones matricial

Existen muchas formas canónicas y factorizaciones de matrices en el mundo de las matemáticas. Sage tiene catorce de ellas predefinidas, según conoce el autor (aunque pueden haber otras más que no conoce). Si el lector tiene una aplicación particular en mente que frecuentemente usa alguna descomposición, factorización o forma canónica particular, esperamos que la encuentre abajo.

<code>A.as_sum_of_permutations()</code>	<code>A.LU()</code>
<code>A.cholesky()</code>	<code>A.QR()</code>
<code>A.frobenius()</code>	<code>A.rational_form()</code>
<code>A.gram_schmidt()</code>	<code>A.smith_form()</code>
<code>A.hessenberg_form()</code>	<code>A.symplectic_form()</code>
<code>A.hermite_form()</code>	<code>A.weak_popov_form()</code>
<code>A.jordan_form()</code>	<code>A.zigzag_form()</code>

Sin embargo, la factorización matricial más conocida de todas es la “factorización LU”. Usualmente, cuando alguien habla de esta, en realidad se refiere a la factorización LUP, donde $A = LUP$, tal que L es alguna matriz triangular inferior, U es alguna matriz triangular superior y P es alguna matriz de permutación. Esta última, frecuentemente pero no siempre, es la matriz identidad. Esto tiende a hacerla “opcional” en el sentido que la mayoría de las matrices A pueden ser escritas como $A = LU$ sin la necesidad de $P \neq I_n$. En Sage, sin embargo, `A.LU()` es una factorización PLU. La matriz de permutación está a la izquierda, no a la derecha, lo que es inusual. ¡El lector está advertido!

De hecho, esto le causó vergüenza al autor en cierta ocasión, durante el semestre de primavera de 2014, en una clase de *Análisis Numérico II*, cuando la factorización de $A = PLU$ no producía la matriz original A . Esto era porque estaba usando $A = LUP$.

4.16.4 Resolviendo sistemas lineales de forma aproximada con mínimos cuadrados

¿Qué pasaría si fuéramos a resolver un sistema de cinco ecuaciones con dos incógnitas? Incluso suena extraño al oído humano —¿por qué hay cinco ecuaciones si solo hay dos incógnitas?—. Tal vez queremos resolver el siguiente sistema de ecuaciones:

$$\begin{aligned} 0m + b &= 7,1 \\ 1m + b &= 5,2 \\ 2m + b &= 2,9 \\ 3m + b &= 1,05 \\ 4m + b &= -0,9 \end{aligned}$$

Como este sistema es lineal, sabemos que podemos convertirlo en un problema matricial:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} 7,1 \\ 5,2 \\ 2,9 \\ 1,05 \\ -0,9 \end{bmatrix}$$

Por lo tanto, escribiríamos en Sage

Código de Sage

```

1  A = matrix(5, 2, [0, 1, 1, 1, 2, 1, 3, 1, 4, 1])
2  print(A)
3  print()
4
5  b = matrix(5, 1, [7.1, 5.2, 2.9, 1.05, -0.9])
6  print(b)
7
8  A \ b

```

No es sorpresa que Sage responda que no existe solución. Hay muy pocas variables para satisfacer todas las restricciones o ecuaciones simultáneamente. ¿Qué es lo que podemos hacer entonces?

¿Qué tal si una solución aproximada es “aceptable”? Resulta que podemos obtener una como sigue:

$$A\vec{x} = \vec{b} \quad \text{se convierte en} \quad A^T A \vec{x} = A^T \vec{b},$$

donde A^T es la transpuesta de A . Nótese que mientras que A es una matriz de 5×2 , como A^T es una de 2×5 , tenemos que $A^T A$ una matriz de orden 2×2 . Similarmente, $A^T \vec{b}$ es un vector 2-dimensional. Si el lector no está familiarizado con la operación de transposición, note que es solo un intercambio de filas y columnas de la matriz. Esto significa que las antiguas filas ahora son columnas y viceversa. En nuestro caso, tenemos

$$A^T = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Por lo tanto, debemos escribir en Sage el siguiente código:

Código de Sage

```

1  A = matrix(5, 2, [0, 1, 1, 1, 2, 1, 3, 1, 4, 1])
2  print(A)
3  print()
4
5  b = matrix(5, 1, [7.1, 5.2, 2.9, 1.05, -0.9])
6  print(b)
7  print()
8
9  M = A.transpose() * A
10 d = A.transpose() * b
11
12 M \ d

```

La salida indica que tenemos $m = -403/200 = -2.015$ y $b = 71/10 = 7.1$ como nuestra solución aproximada al problema original. ¿Acaso estos números resultan familiares? Resulta que este es precisa y exactamente el problema de regresión lineal que teníamos en la página 151. Cuandoquiera que estemos calculando una regresión lineal (es decir, la recta de mejor ajuste) con n puntos, entonces estamos resolviendo n ecuaciones lineales con dos incógnitas usando este método notable, llamado “Mínimos Cuadrados”. Este fue inventado por Carl Friedrich Gauss (1777–1855), o por Adrien-Marie Legendre (1752–1833), dependiendo de cuál libro de historia de las matemáticas uno consulte.

Si pensamos en los puntos de datos como

$$(x_1, y_1), (x_2, y_2), \dots, (x_5, y_5),$$

entonces podemos definir el error en el i -ésimo punto como

$$e_i = y_i - \hat{y}_i = y_i - mx_i - b,$$

donde $\hat{y}_i = mx_i + b$ es el valor estimado de y_i , predicho por la recta de mejor ajuste. Este procedimiento de los mínimos cuadrados garantiza tres resultados asombrosos:

- La suma de los e_i s es cero.
- La longitud de los \vec{e}_i , o equivalentemente, la suma $e_1^2 + e_2^2 + \dots + e_n^2$ es tan pequeña como es posible. Nótese que $\|\vec{e}\|^2$ es igual a esa suma, donde $\vec{e} = \langle e_1, e_2, \dots, e_n \rangle$.
- El punto (\bar{x}, \bar{y}) , donde \bar{x} es el promedio de los valores de los x_i s y \bar{y} es el promedio de los valores de los y_i s, está garantizado encontrarse sobre la recta $y = mx + b$.

Por último, las entradas de $A^T A$ y $A^T b$ tienen un significado. Estas son

$$A^T A = \begin{bmatrix} 30 & 10 \\ 10 & 5 \end{bmatrix} \quad \text{y} \quad A^T b = \begin{bmatrix} 10,55 \\ 15,35 \end{bmatrix}.$$

Podemos ver que el 30 es la suma de los cuadrados de los x_i s, los dos 10s son la suma de los x_i s y el 5 es la cantidad de puntos. El 15,35 es la suma de los y_i s y el 10,55 es la suma

$$x_1 y_1 + x_2 y_2 + \dots + x_5 y_5.$$

Debemos notar que existen muchas situaciones en las que convertir $A\vec{x} = \vec{b}$ en $A^T A\vec{x} = A^T \vec{b}$ tiene sentido. Calcular regresiones lineales es solo una aplicación. El sistema lineal de ecuaciones más pequeño que se obtiene con este truco es llamado el sistema de *ecuaciones normales*.

4.17 Calculando los polinomios de Taylor o de MacLaurin

Los polinomios de Taylor hacen excelentes aproximaciones a funciones y naturalmente aparecen en muchas aplicaciones del cálculo a las ciencias —especialmente la física—. Desafortunadamente, calcular un polinomio de Taylor a mano puede ser una tarea difícil. Frecuentemente uno debe calcular ocho términos para obtener apenas cuatro que no sean nulos. Esto, a su vez, requiere calcular hasta la séptima derivada. Afortunadamente, Sage puede hacer esto por nosotros.

Primero, vale la pena repasar la definición de un polinomio de Taylor. Recordemos que la recta $\ell(x) = mx + b$ es llamada la recta tangente de alguna función $f(x)$ en el punto $x = x_0$, si esta “toca” $f(x)$ en x_0 y tiene “la misma pendiente que” $f(x)$ en x_0 . En términos precisos, esto significa que la recta tiene $f'(x_0)$ como su pendiente y que $f(x_0) = \ell(x_0)$. De manera similar, podríamos definir que una parábola $P(x)$ sea tangente en x_0 . Debería cumplir que $P(x_0) = f(x_0)$ y $P'(x_0) = f'(x_0)$, así como que $P''(x_0) = f''(x_0)$.

Análogo a la recta y la parábola tangentes, el polinomio de Taylor de n -ésimo grado en x_0 es el único polinomio $p(x)$ de grado n tal que $p(x_0) = f(x_0)$ y $p'(x_0) = f'(x_0)$, así como que $p''(x_0) = f''(x_0)$ y así sucesivamente, hasta la n -ésima derivada. La definición no restringe la derivada $n + 1$ ni ninguna de orden superior —pueden ser diferentes o pueden ser iguales—.

Curiosamente, algunos profesores de matemáticas insisten que si $x_0 = 0$, el polinomio sea llamado *Polinomio de MacLaurin*, mientras que si $x_0 \neq 0$, sea llamado *Polinomio de Taylor*. Esta regla es muy extraña al parecer del autor. Los matemáticos involucrados son Brook Taylor (1685–1731) y Colin Maclaurin (1698–1746), así como James Gregory (1638–1675). Tristemente, cada uno tuvo una vida inusualmente corta, incluso para los estándares de sus propias épocas.

4.17.1 Ejemplos de Polinomios de Taylor

Primero calcularemos el Polinomio de Taylor de quinto grado de $f(x) = \cos x$ en el punto $x = 0$. El código de abajo también graficará la función coseno (como una curva punteada) y el Polinomio de Taylor (como una curva sólida).

Código de Sage

```

1  f(x) = cos(x)
2  p(x) = taylor(f(x), x, 0, 5)
3
4  print('Polinomio:')
5  print(p(x))
6
7  P1 = plot(p(x), -3*pi, 3*pi, ymax=3, ymin=-3)
8  P2 = plot(f(x), -3*pi, 3*pi, linestyle='--')
9  show(P1 + P2)

```

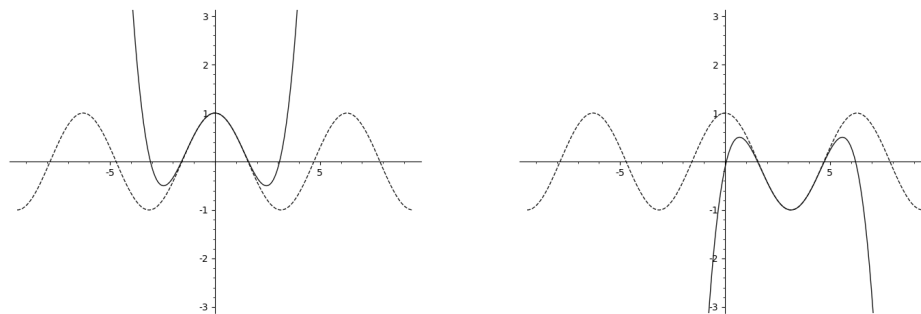
La salida producida es el polinomio mismo:

```

Polinomio:
1/24*x^4 - 1/2*x^2 + 1

```

así como la gráfica mostrada a la izquierda de la figura 4.3. Por otro lado, podemos cambiar el 0 en la segunda



Polinomio de Taylor alrededor de $x = 0$ Polinomio de Taylor alrededor de $x = \pi$

Figura 4.3 Dos Polinomios de Taylor de $f(x) = \cos(x)$.

línea de nuestro código por π . Entonces obtenemos el Polinomio de Taylor de quinto grado en el punto $x = \pi$. Esa es la gráfica a la derecha de la figura 4.3.

En resumen, el comando `taylor` espera cuatro entradas: primero, la función; segundo, la variable; tercero, el punto alrededor del cual será hecha la aproximación, y cuarto, el grado requerido de la aproximación. Aquí, solicitamos un polinomio de grado 5, pero obtuvimos uno de 4. Esto se debe a que en coeficiente de quinto grado resulta ser cero. Dado que normalmente no escribimos $0x^5$, ese término es completamente invisible.

A propósito, dado que esta es una función de Sage con cuatro parámetros, el autor casi siempre olvida el orden en que deben ser especificados. *Los programadores reales siempre revisan la documentación.* Nadie recuerda realmente los detalles finos, como el orden particular de las entradas. Incluso si uno está seguro de recordar cuál es cuál, de todas formas es recomendable verificar (usando las técnicas de la sección 1.10 en la página 44) para evitar la posibilidad de errores.

4.17.2 Una aplicación: Entendiendo cómo cambia g

En Física I, cuando trabajamos con problemas que se desenvuelven en la Tierra (y cerca de su superficie), simplemente tomamos $g = 9,82 \text{ [m/s}^2\text{]}$, y entonces podemos calcular la fuerza de la gravedad (es decir, el peso) de cualquier masa por medio de $w = mg$ o $w = -mg$, dependiendo de cuál convención de signos prefiramos. Cuando trabajamos en problemas que involucran satélites no trivialmente lejos de la superficie de la Tierra, aprendemos la fórmula de Newton para la fuerza de la gravedad:

$$F = \frac{Gm_e m_s}{r^2},$$

donde G es la constante universal de gravitación, m_e es la masa de la Tierra, m_s es la masa del objeto (usualmente un satélite) y r es la distancia del objeto al centro de la Tierra.

Todo eso es muy útil, pero ¿qué hacemos para distancias que están en medio? Si estamos a un metro sobre la superficie de la Tierra, entonces sabemos que debemos usar $g = 9,82 \text{ [m/s}^2\text{]}$. Si estamos a 10^6 metros por encima, entonces de seguro deberíamos usar la fórmula de Newton. Sin embargo, sería interesante saber qué tan errados estamos si usamos $g = 9,82 \text{ [m/s}^2\text{]}$ en un problema, por ejemplo, en un avión, o sobre el Monte Everest. Ahora exploraremos esta cuestión al hacer una representación de la fórmula de Newton, mediante un Polinomio de Taylor. Los tamaños de los coeficientes nos darán mucha información.

Si definimos $F = w = m_s g$, entonces estamos encontrando qué valor toma “ g minúscula” para este r particular. Veamos los siguientes cálculos:

$$\begin{aligned} F &= \frac{Gm_e m_s}{r^2} \\ m_s g &= \frac{Gm_e m_s}{r^2} \\ g &= \frac{Gm_e}{r^2} \\ g &= \frac{Gm_e}{(r_e + x)^2} \end{aligned}$$

Como podemos apreciar, el término m_s de la izquierda cancela el m_s de la derecha. Esto no debe ser sorpresa, pues así como $g = 9,82 \text{ [m/s}^2\text{]}$ para todos los objetos en la superficie de la Tierra, sin importar su masa, el valor local de g lejos de la superficie no depende de m_s . Por otro lado, podemos ver que hemos sustituido $r = r_e + x$ en el último paso. Este r_e representa el radio de la Tierra y la x es la distancia de superficie de la Tierra al objeto —usualmente llamado la *altitud*—. La razón por la que hacemos esto es que resulta fácil medir la altitud de cualquier objeto con un altímetro —pero es muy difícil cortar la Tierra entera por la mitad simplemente por el beneficio de colocar un extremo de una regla enorme en su centro—.

Ahora tomaremos esta función

$$g(x) = \frac{Gm_e}{(r_e + x)^2},$$

es decir la aceleración gravitacional en términos de la altitud, y calcularemos su Polinomio de Taylor en $x = 0$, que es la superficie de la Tierra. ¿Deberíamos usar un polinomio de primer grado?, ¿uno de segundo grado?, ¿tal vez uno de tercer grado? En este punto del problema, esto no resulta obvio. La estrategia del autor (de sus días de ingeniería) es hacer una aproximación de grado superior y luego descartar la mayoría de los coeficientes. Las magnitudes de los coeficientes mismos nos dirán cuáles descartar. Por lo tanto, vamos a pedir a Sage que produzca una aproximación de sexto grado. El código se ve abajo:

Código de Sage

```

1  G = 6.673_84 * 10^(-11)
2  Me = 5.972_19 * 10^24
3  re = 6.371 * 10^6
4  g(x) = G * Me / (re + x)^2
5
6  p(x) = taylor(g(x), x, 0, 6)
7
8  print('Polinomio:')
9  print(p(x))
10
11 print('Solo para confirmar:')
12 print(g(0))
13 print(p(0))

```

Este código produce la siguiente salida:

```

Polinomio:
(1.02788991700924e-39)*x^6 - (5.61315999537077e-33)*x^5 +
(2.98012019420893e-26)*x^4 - (1.51890766058441e-19)*x^3 +
(7.25772052918746e-13)*x^2 - (3.08259583276355e-06)*x + 9.819609025268294
Solo para confirmar:
9.81960902526829
9.819609025268294

```

Las líneas “solo para confirmar” están para recordarnos que en $x = 0$ estamos sobre la superficie de la Tierra y esperamos usar $g = 9,82 \text{ [m/s}^2\text{]}$. Esto parece corresponder. En cuanto al polinomio mismo, los coeficientes de los términos de tercer, cuarto, quinto y sexto grado son excepcionalmente pequeños —y por lo tanto podemos eliminarlos—. Para la mayoría de los problemas, podríamos despreciar también el coeficiente cuadrático, pero si por ejemplo $x = 2 \times 10^6 \text{ [m]}$, entonces $x^2 = 4 \times 10^{12} \text{ [m}^2\text{]}$, y por lo tanto parece valer la pena mantener el $7,25 \times 10^{-13}$ para ese caso.

El coeficiente constante es alentador, recordándonos que $g = 9,82 \text{ [m/s}^2\text{]}$ sobre la superficie de la Tierra. Notemos que el número 9,82 no aparece en ningún lugar de nuestro programa de Sage —no es una entrada—, sino que fue determinado matemáticamente.

El coeficiente lineal es el más importante de todos en este caso. Dado que es $-3,08259 \times 10^{-6}$ sabemos que si $x \approx 1000 \text{ [m]}$, entonces no tenemos nada de que preocuparnos. Podemos usar $9,82 \text{ [m/s}^2\text{]}$ con impunidad. Por otro lado, si $x \approx 10^6 \text{ [m]}$, entonces este valor de g es una aproximación muy pobre. Más aun, por cada 1 [km] que nos alejamos de la superficie de la Tierra, g se reduce por $0,003 \text{ [m/s}^2\text{]}$. Esto significa que incluso a 10^4 [m] de altura, podríamos querer considerar la Fórmula de Newton en lugar de $9,82 \text{ [m/s}^2\text{]}$.

4.18 Minimizaciones y multiplicadores de Lagrange

Encontrar el mínimo o máximo de una función a una variable es tarea común en *Cálculo I*. Para una función multivariada, el problema es un poco más difícil. Sage tiene varios comandos predefinidos para ahorrarnos tiempo en optimización multivariada.

Existen dos categorías: optimización sin restricciones y restringida.

4.18.1 Optimización sin restricciones

Supongamos que debemos encontrar el mínimo de la función

$$f(x, y, z) = 120(y - x^2 + 1)^2 + (2 - x)^2 + 110(z - y^2)^2 + 150(3 - y)^2$$

Una opción —tal vez la más clásica— es tomar el gradiente, ∇f , e igualarlo al vector nulo. La ecuación vectorial $\nabla f = \vec{0}$ tiene tres incógnitas y tres ecuaciones, pero es no lineal. Como el sistema es pequeño, puede ser resuelto mediante resultantes, pero eso es mucho trabajo avanzado.

Frecuentemente, una solución numérica será suficiente, pero ello requiere de una suposición inicial o condición inicial. Tal vez nuestra suposición inicial es $x = 0,1$, $y = 0,3$ y $z = 0,4$. Entonces escribiríamos lo siguiente:

Código de Sage

```
1 var('y z')
2 f(x, y, z) = 120*(y-x^2+1)^2 + (2-x)^2 + 110*(z - y^2)^2 + 150*(3-y)^2
3 minimize(f, [0.1,0.3,0.4])
```

Esto produce la salida

```
(1.9999999999345388, 3.00000000012687, 9.000000000777003)
```

lo cual representa un punto donde hay un mínimo. Sin embargo, generalmente es conveniente una respuesta más detallada, para la que podemos usar la opción `verbose=True`:

Código de Sage

```
1 var('y z')
2 f(x, y, z) = 120*(y-x^2+1)^2 + (2-x)^2 + 110*(z - y^2)^2 + 150*(3-y)^2
3 minimize(f, [0.1,0.3,0.4], verbose=True)
```

Entonces la respuesta que recibiríamos sería

```
Optimization terminated successfully.
Current function value: 0.000 000
Iterations: 15
Function evaluations: 18
Gradient evaluations: 18
(1.9999999999345388, 3.00000000012687, 9.000000000777003)
```

Podemos tener gran confianza en que hay un buen mínimo para $x = 2$, $y = 3$ y $z = 9$. Como podemos ver, 15 iteraciones de descenso por gradiente⁸ fueron ejecutadas. Sin embargo, podemos sentir curiosidad de ver qué tanto depende la respuesta de nuestra condición inicial. Intentemos con una muy mala suposición, tal vez $x = 10$, $y = 30$ y $z = 40$. Obtendremos la misma respuesta, pero costará más iteraciones.

En el caso de una función con muchos mínimos locales, no es obvio cuál mínimo se encontrará. El dominio de la función está dividido “cuencas de atracción”. Estas son los puntos en el dominio para los que el algoritmo convergerá al mismo mínimo local. Sin embargo, las cuencas mismas pueden ser muy complicadas, y en casos extremos, pueden ser fractales. Lo mismo es cierto para las maximizaciones.

Las funciones convexas, sin embargo, tienen solo un mínimo local (o máximo local), que es por lo tanto también global. Más aun, no pueden tener un máximo y un mínimo globales al mismo tiempo, solo uno de ellos. En cálculo de una variable, esas son funciones donde o $f''(x)$ es siempre positiva (un único mínimo), o $f''(x)$ es siempre negativa (un único máximo). Sin embargo, puede no ser muy claro cuál es la noción equivalente para funciones multivariadas.

⁸El descenso por gradiente es uno de los algoritmos favoritos del autor en toda la matemática, pero no contamos con el espacio suficiente para describirlo aquí. Es suficiente decir que fue descubierto por Augustin-Louis Cauchy (1798–1857), quien es mejor conocido por darnos las nociones modernas de convergencia y divergencia de series infinitas, así como otros resultados importantes en *Análisis Real*.

Para que las funciones multivariadas tales como $f(x, y, z)$ tengan un único máximo o mínimo, resulta que la matriz Hessiana H (véase la página 195) debe tener solamente autovalores positivos (único mínimo) o solamente autovalores negativos (único máximo). Algunos lectores pueden no conocer las matrices Hessianas. La matriz H típicamente tendrá diferentes entradas numéricas en varios puntos (x, y) o puntos (x, y, z) , dependiendo del problema. Por lo tanto, una gran cantidad de análisis puede ser requerida para probar que los autovalores son siempre todos positivos, o siempre todos negativos. Si la función original es un polinomio de grado dos (o menor), entonces las entradas de H serán constantes, y eso simplifica enormemente el análisis. Para funciones de solo dos variables, hay un atajo realmente asombroso: $f(x, y)$ tiene un único mínimo (o máximo) si

$$\left(\frac{\partial^2 f}{\partial x \partial x} \right) \left(\frac{\partial^2 f}{\partial y \partial y} \right) - \left(\frac{\partial^2 f}{\partial x \partial y} \right)^2 > 0.$$

Pero nótese que esta no es una relación “si y solo si”.

Estos problemas “de muy buen comportamiento” (es decir, problemas convexos) aparecen en Economía Matemática de manera relativamente frecuente. Tienen la agradable propiedad de que el descenso por gradiente siempre tendrá éxito y siempre convergerá al único punto óptimo.

4.18.2 Optimización restringida por medio de multiplicadores de Lagrange

En general, si tenemos una función $f(x_1, x_2, \dots, x_n)$ y queremos minimizarla sujeta a varias restricciones de la forma $g_i(x_1, x_2, \dots, x_n) = 0$, para g_1, g_2, \dots, g_m , entonces debemos definir una nueva función

$$\mathcal{F}(x_1, x_2, \dots, x_n, \lambda_1, \lambda_2, \dots, \lambda_m) = f(\vec{x}) + \lambda_1 g_1(\vec{x}) + \lambda_2 g_2(\vec{x}) + \dots + \lambda_m g_m(\vec{x}),$$

donde las nuevas variables, $\lambda_1, \lambda_2, \dots, \lambda_m$ son llamadas “Multiplicadores de Lagrange”⁹. El mínimo genuino sin restricciones de \mathcal{F} nos dará a su vez el mínimo de f en el conjunto de puntos que satisfacen las m restricciones simultáneamente —asumiendo que dicho conjunto no sea vacío—. También necesitamos la suposición técnica que $\nabla g_i \neq \vec{0}$ en todos los puntos donde $g_i(\vec{x}) = 0$, para cada g_i . Esta suposición técnica tiende a ser verdad en problemas de aplicación.

4.18.3 Un ejemplo de multiplicadores de Lagrange en Sage

El siguiente ejemplo fue tomado del libro *Calculus*, de James Stewart, 6^{ta} edición, donde es el ejemplo 4 de la sección 15.8, “Lagrange Multipliers”. Debemos encontrar el punto en la esfera $x^2 + y^2 + z^2 = 4$ que esté más cerca del punto $(3, 1, -1)$. Podríamos tratar de minimizar

$$D = \sqrt{(x-3)^2 + (y-1)^2 + (z+1)^2},$$

pero tiene más sentido intentarlo en cambio con

$$D^2 = (x-3)^2 + (y-1)^2 + (z+1)^2,$$

lo cual es legal porque D nunca es negativo. Por ello, las tareas de hallar un mínimo para D y hallar un mínimo para D^2 son exactamente el mismo acto.

Así que queremos minimizar

$$f(x, y, z) = (x-3)^2 + (y-1)^2 + (z+1)^2$$

sujeto a

$$g(x, y, z) = x^2 + y^2 + z^2 - 4 = 0,$$

lo que significa que debemos minimizar

$$\mathcal{F}(x, y, z, \lambda) = (x-3)^2 + (y-1)^2 + (z+1)^2 + \lambda(x^2 + y^2 + z^2 - 4).$$

Así que naturalmente debemos encontrar $\nabla \mathcal{F}$ y hacerlo igual a $\vec{0}$.

⁹Nombrados en honor a Joseph-Louis Lagrange (1736–1813).

Empezamos encontrando las cuatro derivadas parciales:

$$\partial \mathcal{F} / \partial x = 2(x - 3) + 2\lambda x$$

$$\partial \mathcal{F} / \partial y = 2(y - 1) + 2\lambda y$$

$$\partial \mathcal{F} / \partial z = 2(z + 1) + 2\lambda z$$

$$\partial \mathcal{F} / \partial \lambda = x^2 + y^2 + z^2 - 4$$

Esto resulta en cuatro ecuaciones con cuatro incógnitas:

$$2(x - 3) + 2x\lambda = 0$$

$$2(y - 1) + 2y\lambda = 0$$

$$2(z + 1) + 2z\lambda = 0$$

$$x^2 + y^2 + z^2 - 4 = 0$$

Estamos simultáneamente desilusionados (porque las ecuaciones son cuadráticas) y encantados (porque la última ecuación claramente garantiza que cualquier solución está en la esfera en cuestión). Después de mucha álgebra, obtenemos las dos soluciones

$$x = \frac{6}{\sqrt{11}}, y = \frac{2}{\sqrt{11}}, z = \frac{-2}{\sqrt{11}} \quad \text{al igual que} \quad x = \frac{-6}{\sqrt{11}}, y = \frac{-2}{\sqrt{11}}, z = \frac{2}{\sqrt{11}}.$$

Como podemos ver, la primera solución es el punto en la esfera *más cercano* a (3, 1, -1) y la segunda solución es el punto en la esfera *más lejano* a (3, 1, -1).

El ejemplo en Sage Para hacer esto en Sage, tenemos que calcular $\mathcal{F}(x, y, z, \lambda)$ nosotros mismos. Entonces escribiríamos

Código de Sage

```
1 var('y z L')
2 F(x, y, z, L) = (x-3)^2 + (y-1)^2 + (z+1)^2 + L*(x^2 + y^2 + z^2 - 4)
3 minimize(F, [0,0,0,0], verbose=True)
```

La respuesta de Sage es

```
Warning: Desired error not necessarily achieved due to precision loss.
Current function value: 8.927236
Iterations: 1
Function evaluations: 145
Gradient evaluations: 133

(1.5646852718677962, 0.5215617572892653, -0.5215617572892653,
1.0431235145785307)
```

Este mensaje representa a Sage fallando. Recibimos la advertencia “Warning: Desired error not necessarily achieved due to precision loss” (“Advertencia: Error deseado no necesariamente alcanzado debido a pérdida de precisión”), y también notamos que “Current function value” (“Valor actual de la función”) debería ser cero. El autor intentó diferentes condiciones iniciales, una y otra vez. Esto fue realmente frustrante. Estemos seguros que el comando `minimize` de Sage usualmente funciona mejor. Sin embargo, como le pasó al autor, también le puede pasar al lector, así que explicaremos qué debemos hacer para resolver esta situación.

Cuando el comando `minimize` de Sage no minimiza Recordemos que cuando se desea minimizar una función multivariada, una estrategia muy común es calcular el gradiente e igualarlo a cero. Por lo tanto, intentemos pedirle a Sage que calcule el gradiente por nosotros. El código abajo hace eso:

Código de Sage

```

1 var('y z L')
2 F(x, y, z, L) = (x-3)^2 + (y-1)^2 + (z+1)^2 + L*(x^2 + y^2 + z^2 - 4)
3 derivative(F)

```

Sage responde con

```

(x, y, z, L) |--> (2*L*x + 2*x - 6, 2*L*y + 2*y - 2, 2*L*z + 2*z + 2, x^2
+ y^2 + z^2 - 4)

```

Recordemos que en la sección 1.15 en la página 61, vimos que esta es una función formal que puede llevarse a un formato más sencillo —si el lector lo prefiere así— evaluándola en un punto genérico (no específico), al reemplazar la última línea anterior con:

Código de Sage

```

3 derivative(F)(x, y, z, L)

```

que resultará en

```

(2*L*x + 2*x - 6, 2*L*y + 2*y - 2, 2*L*z + 2*z + 2, x^2 + y^2 + z^2 - 4)

```

Cortando y pegando, ya sea en una ventana diferente del navegador ejecutando Sage Cell, o alternatively en CoCalc, podemos definir las siguientes ecuaciones y resolverlas con el comando `solve`.

Código de Sage

```

1 eqn1 = 2*L*x + 2*x - 6 == 0
2 eqn2 = 2*L*y + 2*y - 2 == 0
3 eqn3 = 2*L*z + 2*z + 2 == 0
4 eqn4 = x^2 + y^2 + z^2 - 4 == 0
5
6 solve([eqn1, eqn2, eqn3, eqn4], [x, y, z, L])

```

Como podemos ver, Sage produce los puntos de máximo y mínimo

```

[[x == -6/11*sqrt(11), y == -2/11*sqrt(11), z == 2/11*sqrt(11), L ==
-1/2*sqrt(11) - 1], [x == 6/11*sqrt(11), y == 2/11*sqrt(11), z ==
-2/11*sqrt(11), L == 1/2*sqrt(11) - 1]]

```

Más aun, esta solución tiene la ventaja de ser exacta y no una mera aproximación numérica.

¿Cuál es la moraleja de esta historia? Si los métodos simbólicos funcionan, no hay que molestarse con los métodos numéricos. Estos últimos son sensibles a condiciones iniciales, están sujetos a error de redondeo y frecuentemente uno tiene que ser cuidadoso seleccionando parámetros. Los métodos algebraicos, cuando son factibles, burlan todos estos defectos. Por otro lado, los métodos algebraicos no son aplicables en todos los casos —tal como cuando se buscan las raíces de ecuaciones polinomiales de grado cinco o superior—.

4.18.4 Algunos problemas aplicados

Existen numerosos ejemplos de Multiplicadores de Lagrange, particularmente en Economía Matemática. Recomendamos primero el libro *Principles of Mathematical Economics*, de Shapoor Vali, publicado por Atlantis Press en 2013. En particular, los Multiplicadores de Lagrange aparecen en el capítulo 10.4.2, “Production Function, Least Cost Combination of Resources, and Profit Maximizing Level of Output.”

También recomendamos *Mathematics for Finance: An Introduction to Financial Engineering*, de Marek Capinski y Tomasz Zastawniak, publicado en 2003 por Springer bajo la serie Springer Undergraduate Mathematics. Las secciones 5.1, 5.2 y 5.3 introducen la Estrategia de Portafolio Óptimo de Markowitz, esencialmente escogiendo acciones que son contravariantes para disminuir el riesgo del inversionista. Esa estrategia es básicamente un enorme problema de Multiplicadores de Lagrange.

4.19 Sumas infinitas y series

Empecemos con una suma muy simple:

$$-4 - 1 + 2 + 5 + 8 + 11 + 14 + 17 + 20 + 23 + 26 + 29 + 32 + 35 + \cdots + 131 + 134 + 137$$

Primero identificamos que esta es una serie aritmética, pues la diferencia entre términos adyacentes es exactamente 3 en todos los casos. Un examen más profundo nos dice que esta progresión puede ser concebida como $3k - 7$ para $k \in \{1, 2, 3, \dots, 48\}$ o $3k - 4$ para $k \in \{0, 2, 3, \dots, 47\}$. La primera notación tiene más sentido para el autor. Para representar esto en Sage, debemos escribir

Código de Sage

```
1 var('k')
2 sum(3*k-7, k, 1, 48)
```

con lo que obtenemos la respuesta correcta, es decir 3192.

Tal vez quisiéramos una fórmula más general para la expresión

$$1 + 2 + 3 + 4 + \cdots + (n-2) + (n-1) + n = \sum_{k=1}^{k=n} k,$$

en cuyo caso escribimos

Código de Sage

```
1 var('k m')
2 print(sum(k, k, 1, m))
3 print(factor(sum(k, k, 1, m)))
```

Recibimos la siguiente respuesta:

$$\begin{aligned} &1/2*m^2 + 1/2*m \\ &1/2*(m + 1)*m \end{aligned}$$

Uno de los grandes problemas en matemáticas es la Hipótesis de Riemann¹⁰, que está relacionada con la función Zeta de Riemann, $\zeta(s)$. Una suma de la forma

$$\frac{1}{1} + \frac{1}{2^4} + \frac{1}{3^4} + \frac{1}{4^4} + \frac{1}{5^4} + \frac{1}{6^4} + \cdots = \sum_{k=1}^{k=\infty} \frac{1}{k^4} = \zeta(4)$$

es un ejemplo de evaluación de dicha función en $s = 4$. Más general,

$$\frac{1}{1} + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \frac{1}{5^s} + \frac{1}{6^s} + \cdots = \sum_{k=1}^{k=\infty} \frac{1}{k^s} = \zeta(s),$$

donde s puede ser cualquier número complejo cuya parte real sea mayor que 1.

En Sage, podemos evaluar el caso $s = 4$, por ejemplo, con

Código de Sage

```
1 sum(1/k^4, k, 1, oo)
```

Obtenemos la respuesta $1/90 \pi^4$. Las dos letras “o” consecutivas son el símbolo para infinito en Sage, es decir oo. La idea es que, visto de reojo, oo parece ∞ .

¹⁰Nombrado en honor a Georg Friedrich Bernhard Riemann (1826–1866). Es el Riemann de la Geometría Riemanniana (la curvatura del espacio), la Hipótesis de Riemann, y también hizo varias contribuciones al estudio de los números primos

4.19.1 Verificando identidades de sumatorias

Si el lector ha trabajado con combinaciones, puede que conozca las siguientes identidades:

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n-2} + \binom{n}{n-1} + \binom{n}{n} = 2^n$$

$$\binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \binom{n}{3} + \binom{n}{4} - \binom{n}{5} + \binom{n}{6} - \cdots \pm \binom{n}{n} = 0$$

Podemos verificar esto con el siguiente código de Sage:

Código de Sage

```
1 var('k m')
2 print(sum(binomial(m,k), k, 0, m))
3 print(sum((-1)^k * binomial(m,k), k, 0, m))
```

La respuesta revela que Sage conoce estas identidades:

$$2^m$$

$$0$$

Otro truco interesante es que la suma de los primeros n cubos es igual al cuadrado de la suma de los primeros n números. En otras palabras:

$$1^3 + 2^3 + 3^3 + 4^3 + \cdots + (n-1)^3 + n^3 = (1 + 2 + 3 + 4 + \cdots + n)^2$$

En principio, podemos tratar de verificar esto con el siguiente código:

Código de Sage

```
1 var('k m')
2 sum(k, k, 1, m)^2 - sum(k^3, k, 1, m)
```

sin embargo, recibiremos la siguiente respuesta:

$$-1/4*m^4 - 1/2*m^3 + 1/4*(m^2 + m)^2 - 1/4*m^2$$

Evidentemente, esta expresión se simplifica a 0. Pero podemos evitarnos el trabajo manual con el código de abajo, que hace automáticamente la simplificación por nosotros.

Código de Sage

```
1 var('k m')
2 f(m) = sum(k, k, 1, m)^2 - sum(k^3, k, 1, m)
3 f(m).full_simplify()
```

Alternativamente, podemos también reemplazar la última línea con

Código de Sage

```
3 f(m).expand()
```

En cualquier caso, obtenemos la respuesta “0”.

4.19.2 La serie geométrica

Si quisiéramos estudiar una serie geométrica particular, como

$$25 + 20 + 16 + \frac{64}{5} + \frac{256}{25} + \frac{1024}{125} + \frac{4096}{625} + \dots,$$

empezaríamos dividiendo cada término por el anterior a él, para descubrir que la razón común es $r_c = 4/5$. Tal vez podríamos considerar una serie geométrica más general:

$$a + a(r_c) + a(r_c)^2 + a(r_c)^3 + a(r_c)^4 + a(r_c)^5 + \dots + a(r_c)^m = \sum_{k=0}^m a(r_c)^k,$$

donde a denota el primer elemento y r_c la “razón común”.

Podemos calcular esta suma en Sage, usando el código mostrado abajo:

Código de Sage

```
1 var('a r_c k m')
2 sum(a*(r_c)^k, k, 0, m)
```

Obtenemos la siguiente fórmula, la cual es correcta.

$$(a*r_c^{(m+1)} - a)/(r_c - 1)$$

Para la suma infinita, podríamos simplemente hacer que k corra de 0 a ∞ en lugar de 0 a m ; sin embargo, Sage tiene una objeción sobre el siguiente código:

Código de Sage

```
1 var('a r_c k m')
2 sum(a*(r_c)^k, k, 0, oo)
```

La respuesta que recibimos es un gran mensaje de error, seguido por

```
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(abs(r_c)-1>0)', see 'assume?' for more
details)
Is abs(r_c)-1 positive, negative or zero?
```

La r_c denota la razón común, y de *Cálculo II* sabemos que debería cumplirse $-1 < r_c < 1$ (o, equivalentemente, $|r_c| < 1$) para que la serie converja. Añadimos la suposición como una nueva línea en medio de nuestro código:

Código de Sage

```
1 var('a r_c k m')
2 assume(abs(r_c) < 1)
3 sum(a*(r_c)^k, k, 0, oo)
```

Con esta línea añadida, la respuesta que recibimos es la fórmula convencional para el límite de la serie:

$$-a/(r_c - 1)$$

Aquí podemos ver que Sage ha sido muy sabio al forzarnos a declarar la suposición que $-1 < r_c < 1$, pues la serie geométrica divergerá de otra manera.¹¹ A propósito, el comando `assume` aparece también en integración. Véase las páginas 56 y 192 para ejemplos.

¹¹Con la notable pero absurda excepción en que $a = 0$, en cuyo caso convergerá a 0, sin importar la elección de r_c .

4.19.3 Usando Sage para guiar una demostración con sumatorias

Supongamos que se nos pide determinar a dónde converge la serie

$$\sum_{j=0}^{j=\infty} \frac{x^{2j+1}}{j!} = x + \frac{x^3}{1!} + \frac{x^5}{2!} + \frac{x^7}{3!} + \frac{x^9}{4!} + \frac{x^{11}}{5!} + \dots,$$

y que entonces demostremos ese hecho. Ignoraremos cuestiones de “radio de convergencia” por ahora, que de hecho no aparecen en este problema.

Primeramente, preguntemos a Sage qué es lo que esta suma realmente resulta ser, usando el siguiente código:

Código de Sage

```
1 var('j')
2 sum(x^(2*j+1)/factorial(j), j, 0, oo)
```

La respuesta es

$$x * e^{(x^2)}$$

que será nuestro objetivo en nuestra prueba.

Supongamos que estamos bastante seguros que

$$e^y = 1 + \frac{y}{1} + \frac{y^2}{2!} + \frac{y^3}{3!} + \frac{y^4}{4!} + \frac{y^5}{5!} + \frac{y^6}{6!} + \dots,$$

aunque no completamente seguros. Podemos verificar esto con

Código de Sage

```
1 var('k y')
2 sum(y^k/factorial(k), k, 0, oo)
```

Como podemos ver, Sage nos ha ayudado dos veces: primero al darnos un objetivo para nuestra prueba, y segundo para ayudarnos a confirmar una fórmula básica importante que puede o no que hayamos recordado mal. El resto es bastante directo, después de sustituir $y = x^2$ en la expansión de e^y . Tenemos la siguiente “demostración”, aunque claro está que un estudiante debería añadir palabras a esto.

$$\begin{aligned} e^y &= 1 + \frac{y}{1} + \frac{y^2}{2!} + \frac{y^3}{3!} + \frac{y^4}{4!} + \frac{y^5}{5!} + \frac{y^6}{6!} + \dots \\ e^{x^2} &= 1 + \frac{x^2}{1} + \frac{x^4}{2!} + \frac{x^6}{3!} + \frac{x^8}{4!} + \frac{x^{10}}{5!} + \frac{x^{12}}{6!} + \dots \\ x e^{x^2} &= x + \frac{x^3}{1} + \frac{x^5}{2!} + \frac{x^7}{3!} + \frac{x^9}{4!} + \frac{x^{11}}{5!} + \frac{x^{13}}{6!} + \dots \\ x e^{x^2} &= \sum_{j=0}^{j=\infty} \frac{x^{2j+1}}{j!} \end{aligned}$$

El punto filosófico Por supuesto, el punto de todo esto es que Sage está para ser un conjunto de ruedas de entrenamiento para que el estudiante aprenda este tema famosamente duro: series infinitas. Sage puede ayudar a superar el desconcierto y confusión cuando el estudiante es expuesto por primera vez a este material. El objetivo es que, al menos después de un tiempo, el estudiante consulte Sage menos frecuentemente, hasta que pueda resolver los problemas por sí mismo. Solo entonces se puede tener éxito en un examen riguroso de *Cálculo II*.

4.20 Fracciones continuas en Sage

Esta pequeña sección asume que el lector ya conoce algo acerca de fracciones continuas. Supongamos que quisiéramos saber la expansión en fracción continua de $\sqrt{11}$. Entonces debemos escribir

Código de Sage

```
1 continued_fraction(sqrt(11))
```

lo que nos devuelve

```
[3; 3, 6, 3, 6, 3, 6, 3, 6, 3, 6, 3, 6, 3, 6, 3, 6, 3, ...]
```

Con esto Sage nos dice que

$$\sqrt{11} = 3 + \frac{1}{3 + \frac{1}{6 + \frac{1}{3 + \frac{1}{6 + \frac{1}{3 + \frac{1}{6 + \frac{1}{3 + \frac{1}{6 + \dots}}}}}}}}$$

Sin embargo, en matemáticas aplicadas, usualmente queremos usar fracciones continuas para obtener alguna aproximación racional explícita, pero compacta, para números irracionales. Existen otros usos de las fracciones continuas, pero este uso particular es tan frecuente que se ha creado un atajo en Sage. Podemos escribir

Código de Sage

```
1 c = continued_fraction(sqrt(11))
2 d = c.convergents()
3 print(d)
```

que resulta en la siguiente salida:

```
lazy list [3, 10/3, 63/19, ...]
```

Esto representa una sucesión infinita de números racionales, cada uno aproximándose más y más a $\sqrt{11}$. Podemos acceder a cualquier elemento de esta lista, como siempre, usando un índice. Por ejemplo, $d[0]=3$ es el primer elemento, $d[1]=10/3$ es el segundo, $d[2]=63/19$ es el tercero,

```
d[19]=4993116004999/1505481120300
```

es el 20^{mo} elemento, y así sucesivamente. La única diferencia con una lista normal de Sage es que su longitud es potencialmente infinita, es decir que, en teoría, no existe una cota superior para los índices que podemos usar —aunque desde el punto de vista práctico estemos limitados por la memoria de la computadora—. Sin embargo, debe notarse que Sage calcula una entrada de esta lista solamente después que la hemos solicitado, no antes. Por ello, este tipo de lista es llamado “lazy list” (“lista perezosa”). También por esa misma razón esta no ocupa mucho de la memoria de la computadora, a menos que se trate de acceder a demasiadas de sus entradas.

Las aproximaciones por fracciones continuas son extremadamente eficientes. Por ejemplo, sin ir muy lejos, $d[4]=1257/379$ tiene un error relativo de $6,328\,91e-07$, lo que es bastante impresionante, considerando que tenemos un numerador de 4 dígitos y un denominador de 3. Si cambiamos el código de arriba para obtener aproximaciones por fracciones continuas (que Sage llama “convergentes”) de π , entonces veremos

```
lazy list [3, 22/7, 333/106, ...]
```

Las primeras tres de estas aproximaciones son las más famosas. Muchos de nosotros aprendimos en colegio que $\pi \approx 22/7$ ($=d[1]$). La aproximación $355/113$ ($=d[3]$) fue extensivamente discutida en la página 31. Se dice que la aproximación 3 ($=d[0]$) fue usada en la antigüedad por muchas civilizaciones, e incluso en la Biblia. Véase

el artículo “The Chronology of Pi” por Herman C. Schepler, en *Mathematical Magazine*, Vol 23, No. 3, 1950. Una interesante refutación de esto último puede encontrarse en

<https://www.purplemath.com/modules/bibleval.htm>

4.21 Sistemas de desigualdades y Programación Lineal

Una increíble diversidad de problemas pueden ser resueltos mediante el concepto de programación lineal. Estos incluyen enrutado de embarques, diseño de cronogramas, mezclas de ingredientes para comidas altamente procesadas, balanceo de portafolios, planeamiento de inventarios, ubicación de propagandas y una horda de problemas en manufactura, sin mencionar la búsqueda de puntos de Equilibrio de Nash en juegos de suma cero.

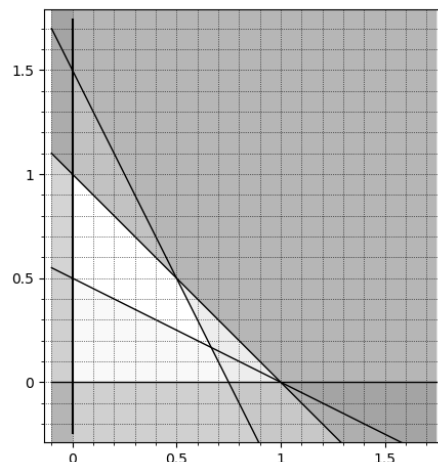
No contamos con el espacio necesario para discutir todas estas interesantes aplicaciones (de hecho, una buena cobertura de este material frecuentemente requiere un libro entero, mucho más grande que este). Sin embargo, veremos cómo usar Sage para resolver programas lineales que son presentados en forma simbólica.

4.21.1 Un ejemplo simple

Aquí tenemos un ejemplo sencillo:

$$\begin{array}{ll} \text{Maximizar} & 5x + 4y \\ \text{sueto a:} & \\ & 1,5 - 2x \geq y \\ & 1 - x \geq y \\ & 1 - x \leq 2y \\ & x \geq 0 \\ & y \geq 0 \end{array}$$

Si el lector siente curiosidad de ver cómo luce este ejemplo gráficamente, este está dado abajo. Las zonas sombreadas son aquellas cuyos puntos no satisfacen todas las desigualdades (es decir, los puntos no factibles), mientras que la región blanca es aquella cuyos puntos sí satisfacen todas desigualdades (es decir, los puntos factibles). A propósito de esto, el código para hacer gráficos de este tipo será incluido en el apéndice únicamente electrónico en línea de este libro, “Graficando a color, 3D, y animaciones”, disponible en la página web www.sage-para-estudiantes.com, para descarga gratuita.



Más aun, este sistema de desigualdades es sujeto de una página web interactiva¹², adecuada incluso para estudiantes en cursos muy elementales.

¹²http://www.sage-para-estudiantes.com/interactivos/region_factible.html

En cualquier caso, para resolver el programa lineal, escribiríamos lo siguiente:

Código de Sage

```

1  LP = MixedIntegerLinearProgram(maximization=True)
2
3  x = LP.new_variable()
4
5  LP.add_constraint(1.5 - 2*x[1] >= x[2])
6  LP.add_constraint(1 - x[1] >= x[2])
7  LP.add_constraint(1 - x[1] <= 2*x[2])
8
9  LP.add_constraint(x[1] >= 0)
10 LP.add_constraint(x[2] >= 0)
11
12 LP.set_objective(5*x[1] + 4*x[2])
13
14 print(LP.solve())
15
16 print(LP.get_values(x))

```

Ahora analizaremos este código pedazo a pedazo.

- La primera línea es la menos intuitiva. La variable LP da un nombre a este programa lineal, y puede ser cualquiera que deseemos darle. La palabra larga `MixedIntegerLinearProgram` siempre debe aparecer, con ese uso exacto de mayúsculas y minúsculas, sin espacios. Entonces indicaremos si el problema es de maximización, como hemos hecho en este caso, o alternativamente, podemos indicar si el problema es minimización, especificando `maximization=False`.
- Entonces declaramos nuestra familia de variables `x`. Esto de hecho define un conjunto infinito de variables, `x[0]`, `x[1]`, `x[2]`, `x[3]`, ... Como podemos ver, en este ejemplo solo necesitamos `x[1]` y `x[2]`. Incluso podemos usar índices no consecutivos. Podríamos haber usado `x[420]` y `x[88]` en lugar de `x[1]` y `x[2]`, de haber tenido una razón para hacerlo.
- Después de esto vienen las restricciones del problema. Como se puede apreciar, estas son bastante directas. Debemos escribir `>=` para \geq y `<=` para \leq . No olvidemos poner un asterisco entre los coeficientes y las variables.
- A continuación debemos proporcionar las restricciones menores —que las variables `x[1]` y `x[2]` nunca deben ser negativas—.
- La línea después de esto declara la función objetivo. Nuevamente, esto es bastante directo en notación de Sage.
- El método `solve()` nos dará el valor óptimo (en este caso, el valor máximo) de la función objetivo.
- La línea siguiente a esa puede ser un poco confusa al principio. Esta imprime los valores de las variables que producen el óptimo. Puede parecer extraño que se requiera una línea separada, después de `LP.solve()`. Sin embargo, cuando tratemos con múltiples nombres de variables abajo, veremos por qué esto es así.

La salida de este código debería ser la siguiente:

```

4.5
{1: 0.5, 2: 0.5}

```

4.21.2 Características convenientes en la práctica

Las siguientes características pueden encontrarse en Sage en relación con programación lineal.

- Si tenemos 20 o 30 variables, tener que escribir repetidamente líneas como

Código de Sage

```
1 LP.add_constraint(x[27] >= 0)
```

sería realmente tedioso. En cambio, podemos escribir

Código de Sage

```
1 x = LP.new_variable(nonnegative=True)
```

en lugar de la línea `x = LP.new_variable()`. Entonces, con una sola línea de código, Sage asumirá que todas las variables de la familia `x` son mayores o iguales a cero. No necesitamos escribir una desigualdad separada para cada variable `x[i]`. Por supuesto, si deseamos variables puramente negativas, podemos especificar `nonnegative=False`. Y en caso de ser irrelevante el signo, simplemente dejamos fuera el argumento `nonnegative`.

- Las restricciones a intervalos aparecen de tiempo en tiempo, y ser capaces de usarlas en Sage puede reducir el número de líneas de código a la mitad en algunos casos. Algunos sistemas de álgebra computacional requieren que se escriba $2 \leq 3x_4 + 7x_8 \leq 5$ como dos restricciones separadas: $2 \leq 3x_4 + 7x_8$ y $3x_4 + 7x_8 \leq 5$. Sin embargo, en Sage podemos hacer esto en una sola línea:

Código de Sage

```
1 LP.add_constraint(2 <= 3*x[4] + 7*x[8] <= 5)
```

- A veces deseamos indicar que $5x_1 + 6x_2 = 300$. Algunos sistemas de álgebra computacional requieren que hagamos esto con dos restricciones: $5x_1 + 6x_2 \leq 300$ y $5x_1 + 6x_2 \geq 300$. Sin embargo, en Sage podemos hacerlo en una sola línea:

Código de Sage

```
1 LP.add_constraint(5*x[1] + 6*x[2] == 300)
```

Nótese que escribimos dos signos de igualdad consecutivos.

- Existen algunas aplicaciones en las que es suficiente hallar una solución factible, y no existe función objetivo. El siguiente truco probablemente solo importa en programas lineales enormes, donde el tiempo de ejecución puede ser extremadamente largo. En lugar de definir la función objetivo como 0, para tener un programa más rápido, uno debería escribir

Código de Sage

```
1 LP.set_objective(None)
```

- Si escribimos `LP.show()`, entonces obtenemos un resumen legible para humanos del programa lineal. Esto puede ser realmente útil para depurar errores.
- Podemos añadir o remover restricciones lineales conforme avanzamos, antes o después del comando `solve`. Existen ciertos casos en los que añadir una restricción, resolver nuevamente, añadir una restricción, resolver nuevamente, y así sucesivamente, es una estrategia muy útil. Cada comando `LP.solve()` conoce las restricciones que se definieron por encima de él en el programa (excepto, por supuesto, las que fueron removidas), pero ignora aquellas que están definidas debajo de él. Estas situaciones aparecen en *Programación por Objetivos* y en el método de *Cortes de Gomory*.

- Para remover una restricción escribimos

Código de Sage

```
1 LP.remove_constraint(17)
```

donde 17 es el número de la restricción en la lista mostrada con el comando `LP.show()`. Recordemos que los científicos computacionales cuentan desde 0, así que la primera restricción es #0, seguida por #1 y entonces #2. De acuerdo a esto, la #17 es la 18^{va} restricción mostrada.

- En ocasiones, tener todas las variables meramente nombradas x es poco informativo y limita una eficiente comprensión del problema. Por ejemplo, en el proyecto descrito en la sección 2.3 en la página 72, podríamos escribir

Código de Sage

```
1 d = LP.new_variable()
2 t = LP.new_variable()
3 g = LP.new_variable()
4 m = LP.new_variable()
```

Esto es simplemente para poder llevar a cabo el siguiente plan: Sean d_1, d_2, \dots, d_7 las cantidades enviadas a Duluth/Superior; sean t_1, t_2, \dots, t_7 las cantidades enviadas a Two Harbors; sean g_1, g_2, \dots, g_7 las cantidades enviadas a Green Bay; sean m_1, m_2, \dots, m_7 las cantidades enviadas a Minneapolis. Evidentemente, esto es mucho más fácil que tener $x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}, \dots, x_{4,5}, x_{4,6}, x_{4,7}$. Sin embargo, una consecuencia de esto es que necesitamos solicitar las variables por separado. A pesar del esfuerzo, esto resulta en una salida mucho más legible que antes:

Código de Sage

```
1 print('Duluth:')
2 print(LP.get_values(d))
3 print('Two Harbors:')
4 print(LP.get_values(t))
```

y así sucesivamente.

- Imaginemos que estamos trabajando en un programa de distribución con 10 fábricas y 300 concesionarias. Sería muy útil poder denotar el número de ítems enviados desde la fábrica i hasta la concesionaria j con $s_{i,j}$. En Sage, esto puede hacerse. Todo lo que necesitamos hacer es definir la familia s con algo como

Código de Sage

```
1 s = LP.new_variable(nonnegative=True)
```

y entonces podemos usar variables tales como $s[2,3]$ o $s[1,5]$, y así sucesivamente, según sea necesario. Nótese que $s[5,7]$ no es lo mismo que $s[7,5]$. Naturalmente, también podemos dejar fuera la opción `nonnegative=True` si deseamos admitir cantidades negativas embarcadas desde las fábricas hasta las concesionarias (por ejemplo, devolviendo los excesos).

- Supongamos que tenemos un par de fábricas en USA y Alemania, respectivamente, y concesionarias en Colombia y Francia. Sería extremadamente útil poder nombrar nuestras variables con nombres como $x_{\text{USA-Colombia}}$, $x_{\text{USA-Francia}}$, $x_{\text{Alemania-Colombia}}$ y $x_{\text{Alemania-Francia}}$, para poder representar las cantidades de producto que viajan de un país a otro. Sage nos permite hacer esto. Podemos usar las variables `x['USA-Colombia']`, `x['USA-Francia']`, `x['Alemania-Colombia']` y `x['Alemania-Francia']` para este caso particular. De hecho, Sage nos da la comodidad de usar cualquier objeto como índice de una variable. De esta manera, incluso `x[1,2,3]`, `x[1, 'Duluth']`, `x[sqrt(-1)]` son todas válidas —aunque tal vez sea poco útil usar la unidad imaginaria $i = \sqrt{-1}$ como índice—.

4.21.3 El poliedro de un programa lineal

El lector probablemente esté familiarizado con la forma de resolver un programa lineal dibujando rectas en el plano. Si no es así, puede consultar la página web interactiva

https://www.sage-para-estudiantes.com/interactivos/region_factible.html

De la misma manera en que un problema a dos variables divide el plano en regiones poligonales (convexas), problemas a tres variables dividen el espacio tridimensional ordinario en poliedros (convexos). Sage puede graficar estos últimos, lo que se describe en el apéndice únicamente electrónico en línea de este libro, “Graficando a color, 3D, y animaciones”, disponible en la página web www.sage-para-estudiantes.com, para descarga gratuita.

4.21.4 Programación lineal entera y variables booleanas

El conjunto de problemas que pueden modelarse por medio de programación lineal se vuelve considerablemente más grande cuando se permite que algunas variables sean designadas como “enteras solamente”. Si solo unas cuantas variables son designadas de esta manera, entonces el problema es soluble como varios programas lineales, usando un método llamado “Cortes de Gomory”, u otro método llamado “Ramificación y poda”. Si muchas variables son designadas de esta manera, entonces el problema puede tardar mucho tiempo en resolverse.

En cualquier caso, Sage tiene esta característica predefinida. Para indicar que las variables $x[i]$ pueden ser enteros no negativos solamente, mientras que las variables $y[i]$ pueden ser números reales no negativos, solo escribimos

Código de Sage

```
1 x = LP.new_variable(nonnegative=True, integer=True)
2 y = LP.new_variable(nonnegative=True)
```

Más aun, podemos escribir

Código de Sage

```
1 y = LP.new_variable(nonnegative=True, integer=False)
```

para realmente resaltar el hecho para el lector humano.

Las variables booleanas también aparecen de tiempo en tiempo. A estas solo se les permite tomar los valores 0 o 1, y nada más. Para indicar que todas las variables $z[k]$ son de esta forma, escribimos

Código de Sage

```
1 z = LP.new_variable(binary=True)
```

4.21.5 Lectura adicional sobre programación lineal

Como en muchos temas, escoger qué libro leer sobre Programación Lineal depende de si uno está buscando información acerca de la teoría o la práctica.

- Para la teoría de cómo se resuelven programas lineales grandes con el método simplex, recomendamos el libro *Linear Programming: An Introduction With Applications*, por Alan Sultan, autopublicado en la Plataforma de Publicación Independiente CreateSpace en 2011. Este libro es adecuado para estudiantes que no han pasado muchos cursos en matemáticas, así como aquellos que sí lo han hecho.
- Para una miriada de aplicaciones, una gran (y notablemente entendible) fuente es *Optimization in Operations Research* por Ronald Rardin, publicado por Prentice Hall en 1997.

4.22 Ecuaciones diferenciales

A muchos de nosotros se nos dijo cuando jóvenes que las matemáticas son la clave para entender el universo, o al menos el universo de la ciencia. Esta atribución ha sido declarada por muchos a través de los siglos, pero su expresión más famosa¹³ se debe a Galileo (1564–1642) en *El Ensayador*,

...en este gran libro —me refiero al universo— que se muestra continuamente abierto a nuestra contemplación, pero no puede ser entendido a menos que uno primero comprenda el lenguaje e interprete los caracteres en los que está escrito. Está escrito en el lenguaje de las matemáticas, y sus caracteres son triángulos, círculos y otras figuras geométricas, sin las cuales es humanamente imposible entender una palabra de ello; sin estos, uno está vagando sin rumbo en un laberinto oscuro.

Sin embargo, no es hasta el curso de *Ecuaciones Diferenciales*, relativamente tarde en la educación matemática de algunos estudiantes, que esta promesa es cumplida. Repentinamente, con los contenidos de ese curso, muchos de los problemas más duros en física, química, ecología y finanzas se hacen transparentes y fáciles —al punto que se convierten en meros ejercicios para la tarea—.

Lastimosamente, los problemas de aplicaciones reales son frecuentemente turbios —los coeficientes tienen comas decimales, las ecuaciones no son directamente solubles (especialmente en problemas en los que uno incluye resistencia del aire), y algunos problemas son simplemente difíciles—. Aquí es donde Sage puede prestar un genuino servicio al estudiante. El estudiante puede concentrarse en modelar fenómenos y plantear correctamente el problema; el tedio de las soluciones numéricas o soluciones con series puede ser mejor delegado a la computadora. Con esta división de la labor entre humano y máquina, problemas considerablemente más difíciles y más realistas pueden ser tratados mejor que con un solo humano trabajando en aislamiento. Este tópico es la joya de la corona de las matemáticas aplicadas y el autor fervientemente recomienda a cualquier estudiante a explorar más allá de los límites del curso de pregrado y penetrar profundamente en tópicos avanzados.

Afortunadamente, un libro entero ha sido escrito para enseñar el curso de *Ecuaciones Diferenciales* usando Sage. El libro es *Introduction to Differential Equations Using Sage*, de David Joyner y Marshall Hampton, publicado por Johns Hopkins University Press en 2012. Este cubre un amplio arreglo de temas relacionados a las ecuaciones diferenciales, y por lo tanto sería redundante reproducir todo eso aquí.

En cambio, mostraremos cómo resolver ecuaciones diferenciales simples, y entonces avanzaremos a problemas con valor inicial y la construcción de campos de pendientes. Finalmente presentaremos un problema crucial, que consiste en modelar la trayectoria de un torpedo, incluyendo la fuerza del arrastre hidrodinámico. Mientras tanto, Sage es extremadamente capaz de soluciones numéricas (son métodos Runge-Kutta), soluciones mediante series, problemas con valor en la frontera, sistemas de ecuaciones diferenciales ordinarias, separación de variables, etc. —el lector puede consultar el libro de Joyner-Hanson para estos procedimientos—.

4.22.1 Algunos ejemplos sencillos

Si quisiéramos resolver la ecuación diferencial

$$\frac{dy}{dx} + y = 7$$

entonces escribiríamos lo siguiente:

¹³**Nota del traductor:** El autor usa la traducción estándar al inglés, realizada por Stillman Drake. Desafortunadamente, no he podido encontrar un equivalente de igual calidad al español, así que me atrevo a traducir recursivamente sobre el texto de este gran historiador de Galileo.

Código de Sage

```

1  y = function('y')(x)
2  h = desolve(diff(y,x) + y == 7, y)
3
4  print('Sin simplificar:')
5  print(h)
6  print('Simplificado:')
7  print(h.expand())

```

La primera línea es algo nueva para nosotros. De la misma manera en que aprendimos a declarar variables desconocidas con el comando `var` en la página 36, podemos también declarar funciones desconocidas con el comando `function`, como se muestra arriba. El único propósito de esta instrucción es indicar a Sage y Python que no sabemos qué es lo que $y(x)$ resultará ser. La parte “(x)” de esa línea simplemente declara que y es una función de la variable x , pero no podemos dejar de usarla. Si no declaramos la variable independiente de esta manera, Sage reconocerá que y es una función, pero no sabrá cómo tratar la instrucción `diff(y,x)`. En efecto, si tuviésemos por ejemplo que y es una función solo de la variable z , es decir si $y = \text{function}('y')(z)$, donde z se declaró previamente con el comando `var`, entonces `diff(y,x) = 0`. En cambio, como hemos declarado que $y = \text{function}('y')(x)$, entonces `diff(y,x)` se entenderá simplemente como dy/dx de manera simbólica. Entonces obtenemos la siguiente respuesta:

```

Sin simplificar:
(_C + 7*e^(-x))*e^(-x)
Simplificado:
_C*e^(-x) + 7

```

Incluso podemos cambiar `diff(y,x)` por `diff(y,x,2)` para tener la segunda derivada, convirtiendo la ecuación diferencial en

$$\frac{d^2y}{dx^2} + y = 7,$$

lo que resulta en la respuesta

```

Sin simplificar:
_K2*cos(x) + _K1*sin(x) + 7
Simplificado:
_K2*cos(x) + _K1*sin(x) + 7

```

Ahora cambiemos la línea que define h con nueva ecuación diferencial, representando

$$y \frac{dy}{dx} + \sin(x) = 0$$

y usando el código

Código de Sage

```

2  h = desolve(y*diff(y,x)+sin(x)==0, y)

```

Obtenemos un tipo muy diferente de respuesta:

```

Sin simplificar:
-1/2*y(x)^2 == _C - 7*x - cos(x)
Simplificado:
-1/2*y(x)^2 == _C - 7*x - cos(x)

```

Aquí, $y(x)$ ha sido definida implícitamente, no explícitamente. Sin embargo, podemos usar un poco de álgebra y ver cuál era realmente la intención:

$$\begin{aligned}(-1/2)(y(x))^2 &= c - \cos(x) \\ (y(x))^2 &= 2 \cos(x) - 2c \\ y(x) &= \pm \sqrt{2 \cos(x) - 2c}\end{aligned}$$

El \pm indica que Sage no hubiera podido darnos una única función como respuesta. Debido a este \pm , como está escrito, $y(x)$ fallaría la “prueba de la línea vertical” y por lo tanto no puede ser considerada una función. En otras palabras, para cualquier función, si escogemos cualquier x específico en su dominio, entonces solo puede existir un único valor y para ese x . En este caso, tenemos dos valores distintos de y , lo cual no está permitido. La mejor manera de pensar en esto es imaginar dos soluciones:

$$\begin{aligned}y_1(x) &= +\sqrt{2 \cos(x) - 2c} \\ y_2(x) &= -\sqrt{2 \cos(x) - 2c}\end{aligned}$$

De hecho, existe un número infinito de funciones que satisfacen la ecuación diferencial, pues podemos escoger cualquier valor de c que queramos. Digamos, por ejemplo, que decidimos incluir el signo menos (es decir, escogiendo $y_2(x)$) y que escogemos $c = 832$. ¿Cómo podemos verificar que esta es de hecho una solución? Podríamos escribir el siguiente código:

Código de Sage

```
1 f(x) = -sqrt(-2*(832-cos(x)))
2
3 print(f(x) * diff(f(x),x) + sin(x))
```

Obtenemos la respuesta “0”, lo que significa que la ecuación diferencial, en efecto, se satisface. El autor tiene el hábito de siempre verificar las soluciones, pues hay muchas oportunidades de escribir incorrectamente una función —tal vez dejando fuera un signo menos o escribiendo mal un número—. Como Sage hace todo el trabajo, sería una lástima no revisar.

4.22.2 Un problema con valor inicial

Ahora vamos a programar Sage para resolver esta ecuación diferencial:

$$\frac{d^2y}{dx^2} y + y = x,$$

sujeta a algunas restricciones de valor inicial.

Si tratamos de resolver el problema de la misma manera que hicimos antes, escribiríamos

Código de Sage

```
1 y = function('y')(x)
2 miecuacion = diff(y,x,2) + y == x
3 desolve(miecuacion, y)
```

Este código produce la salida

```
_K2*cos(x) + _K1*sin(x) + x
```

Esto es muy fácil de verificar:

$$\begin{aligned}
 f(x) &= k_2(\cos x) + k_1(\sin x) + x \\
 f'(x) &= -k_2(\sin x) + k_1(\cos x) + 1 \\
 f''(x) &= -k_2(\cos x) - k_1(\sin x) \\
 f''(x) + f(x) &= (-k_2(\cos x) - k_1(\sin x)) + (k_2(\cos x) + k_1(\sin x) + x) \\
 &= (-k_2 + k_2)(\cos x) + (-k_1 + k_1)(\sin x) + x \\
 &= x.
 \end{aligned}$$

Resulta que si tenemos un problema con valor inicial, tal como $y = 2$ en $x = 10$ y $dy/dx = 1$ en $x = 10$, entonces haremos solo una pequeña modificación: cambiamos la tercera línea por

Código de Sage

```
3 desolve(miecuacion, y, ics=[10, 2, 1])
```

donde ics es abreviación de “initial conditions” (“condiciones iniciales”). La lista $[10, 2, 1]$ significa que $y(10) = 2$ y $y'(10) = 1$. Ahora obtenemos la siguiente respuesta:

$$\begin{aligned}
 &x - 8*\cos(10)*\cos(x)/(\cos(10)^2 + \sin(10)^2) - 8*\sin(10)*\sin(x)/(\cos(10)^2 \\
 &+ \sin(10)^2)
 \end{aligned}$$

Por supuesto, $\sin^2(10) + \cos^2(10) = 1$, así que podemos interpretar esto como

$$f(x) = x - (8 \cos 10)(\cos x) - (8 \sin 10)(\sin x).$$

Si queremos revisar nuestro trabajo, necesitamos un pedazo de código ligeramente más largo:

Código de Sage

```

1 y = function('y')(x)
2 miecuacion = diff(y,x,2) + y == x
3
4 f(x) = desolve(miecuacion, y, [10, 2, 1] )
5 print('f(x) =', f(x))
6
7 fprima(x) = diff(f(x), x)
8 fprimaprima(x) = diff(fprima(x), x)
9
10 print('f(10) =', N(f(10)))
11 print("f'(10) =", fprima(10))
12 print(fprimaprima(x) + f(x) - x)
```

Este código nos dará la salida de abajo:

$$\begin{aligned}
 &f(x) = x - 8*\cos(10)*\cos(x)/(\cos(10)^2 + \sin(10)^2) - \\
 &8*\sin(10)*\sin(x)/(\cos(10)^2 + \sin(10)^2) \\
 &f(10) = 2.000000000000000 \\
 &f'(10) = 1 \\
 &0
 \end{aligned}$$

Como podemos ver, esto verifica nuestros tres requerimientos: primero, que $f(10) = 2$; segundo, que $f'(10) = 1$; tercero, que $f(x)$ satisfaga la ecuación diferencial $f''(x) + f(x) = x$.

4.22.3 Graficando un campo de pendientes

Supongamos que queremos resolver la ecuación diferencial

$$\frac{dy}{dx} + \frac{y}{x} + x = 2$$

con condición inicial $y(2) = 4$. Sin embargo, también nos gustaría saber cómo varían las soluciones bajo otras condiciones iniciales. Primero, resolvamos el problema con valor inicial como hicimos en la subsección previa. Escribimos el siguiente código:

Código de Sage

```
1 y = function('y')(x)
2 miecuacion = diff(y,x) + y/x + x == 2
3 h = desolve(miecuacion, y, [2, 4])
4
5 print(h.expand())
```

Ahora tenemos la respuesta

$$-1/3*x^2 + x + 20/3/x$$

lo que realmente significa

$$f(x) = \frac{-1}{3}x^2 + x + \frac{20/3}{x}$$

y por lo tanto,

$$f'(x) = \frac{-2}{3}x + 1 - \frac{20/3}{x^2},$$

así como

$$\frac{f(x)}{x} = \frac{-1}{3}x + 1 + \frac{20/3}{x^2},$$

permitiéndonos concluir que la solución es correcta al hacer la revisión

$$\begin{aligned} f'(x) + \frac{f(x)}{x} + x &= \left(\frac{-2}{3}x + 1 - \frac{20/3}{x^2}\right) + \left(\frac{-1}{3}x + 1 + \frac{20/3}{x^2}\right) + x \\ &= \left(\frac{-2}{3} + \frac{-1}{3} + 1\right)x + (1 + 1) + \left(\frac{-20/3}{x^2} + \frac{20/3}{x^2}\right) \\ &= 0 + 2 + 0 = 2. \end{aligned}$$

El nuevo paso que vamos a tomar es resolver la ecuación diferencial original para dy/dx . En este caso, obtenemos

$$\frac{dy}{dx} = -\frac{y}{x} - x + 2$$

lo que significa que

$$\frac{dy}{dx} = \left(\frac{1}{3}x - 1 - \frac{20/3}{x^2}\right) - x + 2 = \frac{-2}{3}x - \frac{20/3}{x^2} + 1$$

Nótese que *todas las soluciones* a la ecuación diferencial deben satisfacer esta relación, incluyendo la que corresponde con nuestras condiciones iniciales.

A continuación, construimos la gráfica del campo de pendientes. Para cada posición en el plano coordenado, podemos usar la ecuación anterior para encontrar lo que dy/dx debería ser. Sage seleccionará cerca de 400 puntos en un patrón de 20×20 , formando una rejilla sobre el plano coordenado. Para cada punto, calculará dy/dx en esa posición, y dibujará un vector con esa pendiente, centrado en ese punto. Nuestra gráfica cubrirá el intervalo $0 < x < 6$ en el eje x y el intervalo $-5 < y < 15$ en el eje y , y también graficaremos nuestra $f(x)$ particular de más arriba, superponiéndola en la misma imagen. Aquí tenemos el código:

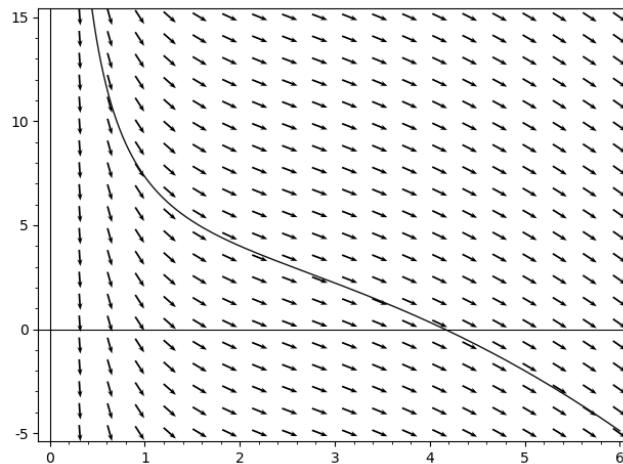
Código de Sage

```

1 var('y')
2 f(x) = -1/3*x^2 + x + (20/3)/x
3
4 P1 = plot_slope_field((-2/3)*x - (20/3)/x^2 + 1, (x, 0, 6), (y, -5, 15),
   ↪ headlength=4, headaxislength=3)
5 P2 = plot(f(x), (x, 0, 6), ymax=15, ymin=-5)
6 P = P1 + P2
7 P.show()

```

Esto produce la siguiente gráfica:



Como podemos apreciar en el gráfico, nuestra solución (la curva sólida) corresponde bien con los vectores que están muy cerca de ella. En cierto sentido, nuestra solución está conectando una sucesión de vectores adyacentes, cabeza con cola, para obtener una curva suave que nos lleva desde $x = 0$ hasta $x = 6$. Similarmente, si tuviésemos una condición inicial distinta, empezaríamos en un punto diferente, pero aún conectaríamos un vector con otro, cabeza con cola, para obtener una curva suave.

Una breve explicación es necesaria acerca de las dos opciones `headlength=4` y `headaxislength=3`. Los expertos en ecuaciones diferenciales prefieren no tener cabezas de flechas en sus vectores cuando grafican un campo de pendientes. En cierta forma, esto tiene sentido, pues podrían sobrepoblar la imagen (especialmente si los vectores están muy juntos). Más aun, la cabeza siempre va a la derecha del segmento de recta que representa la flecha, así que la cabeza en realidad no proporciona ninguna información adicional. Por otro lado, el autor realmente prefiere las cabezas de las flechas en su lugar, pues son visualmente atractivas y porque son útiles para los estudiantes que recién están empezando a explorar las ecuaciones diferenciales. Tal vez el lector quiera echarse un poco para atrás en el código, para ver cómo cambia la imagen cuando estas dos opciones se modifican o se eliminan.

4.22.4 El problema del torpedo: Trabajando con parámetros

Supongamos que deseamos resolver la ecuación diferencial para un torpedo acelerado bajo el agua, desde un submarino en lo profundo hasta la superficie. Por supuesto, la fuerza del arrastre hidrodinámico no será despreciable en este caso, así que debemos incluirla. El peso del torpedo y el empuje del motor son también características importantes. Primero, empezamos con una ecuación general:

$$ma = \sum F = \underbrace{-kv}_{\text{arrastre}} - \underbrace{mg}_{\text{peso}} + \underbrace{T}_{\text{empuje}},$$

donde k es algún coeficiente de arrastre, m es la masa del torpedo, $g = 9,82 \text{ [m/s}^2\text{]}$ es la aceleración debida a la gravedad y T es el empuje del motor del torpedo. Ahora podemos convertir esto en una ecuación diferencial, haciendo la altitud $y(t)$, la velocidad $y'(t)$ y la aceleración $y''(t)$. (A propósito, usamos la convención que “arriba” es positivo y “abajo” es negativo.) Hasta este punto tenemos

$$my''(t) = \underbrace{-ky'(t)}_{\text{arrastre}} - \underbrace{mg}_{\text{peso}} + \underbrace{T}_{\text{empuje}},$$

que ahora introduciremos en Sage. Analizaremos el código de abajo, línea por línea.

Código de Sage

```

1  var('m k T g t')
2  y = function('y')(t)
3
4  ecuacion_general = m*diff(y, t, 2) == -k*diff(y, t) - m*g + T
5  mi_ecuacion = ecuacion_general(m=1_000, k=0.5, T=20_000, g=9.82)
6
7  f(t) = desolve(mi_ecuacion, y, [0, -2_000, 0])
8  print('y(t) = ', f(t))
9
10 plot(f(t), 0, 20, gridlines='minor')
```

La primera línea aquí define cinco variables, como hemos visto a lo largo de este libro. La segunda línea declara que la función $y(t)$, la altitud en el tiempo t , es desconocida por ahora. Entonces tenemos una reproducción fiel de la ecuación diferencial gobernante, asignada a la variable `ecuacion_general`, en la cuarta línea.

La quinta línea sustituye 1000 [kg] para la masa del torpedo, 0,5 para el coeficiente de arrastre, 20 000 [N] para el empuje y el usual 9,82 [m/s²] para g . De manera confusa, las unidades del coeficiente de arrastre son [kg/s], pues así es como podemos obtener Newtons después de multiplicar el coeficiente por la velocidad en [m/s]. Tener las sustituciones de los parámetros arriba en el programa nos permite experimentar con diferentes valores, para estudiar qué impacto tienen en la solución final. En poco añadiremos que el torpedo tiene profundidad inicial de 2000 [m] y velocidad inicial de 0 [m/s].

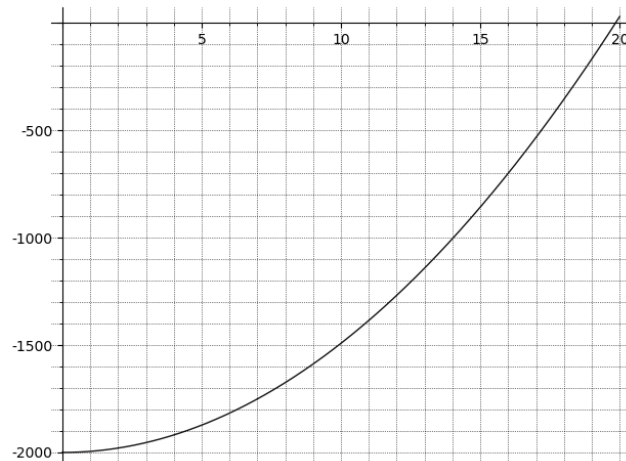
La séptima línea es sin duda familiar ya en este punto. Usamos `desolve` para resolver la ecuación diferencial. Damos la condición inicial que en $t = 0$ la altitud es $y = -2000$ [m] y que la velocidad es $dy/dt = 0$ [m/s]. La respuesta es asignada a $f(t)$.

Para cuando llegamos a la octava línea, $f(t)$ es conocida, pues `desolve` la calculó. La necesidad de dos funciones, $y(t)$ y $f(t)$, confunde a algunos estudiantes. Piensese en $y(t)$ como un espacio reservado donde irá la respuesta final, pero $f(t)$ es en realidad la solución. Por ejemplo, de manera análoga, en $3x + 5 = 5x - 3$, el espacio reservado para la solución final es x , pero la respuesta en realidad es 4.

En cualquier caso, la solución que obtenemos es

$$y(t) = 20360*t + 40720000*e^{(-1/2000*t)} - 40722000$$

y adicionalmente tenemos un gráfico interesante:



Si vemos de cerca la gráfica, notaremos que durante el último 25 % del tiempo, la curva se parece a una recta. También, para valores más grandes de t , $y(t)$ será una recta. Sin la fuerza del arrastre hidrodinámico, sin embargo, nuestra función $y(t)$ sería una parábola. Por lo tanto, podemos ver que la fuerza de arrastre no solo cambia la solución cuantitativamente, sino también cualitativamente. La pendiente de esa recta es llamada “velocidad terminal”, y a esa velocidad, la fuerza de arrastre es igual a la del empuje. Por supuesto, debe notarse que para $y > 0$, el torpedo está por encima del agua y por lo tanto un modelo completamente diferente debe ser usado en esa etapa.

4.23 Transformadas de Laplace

La Transformada de Laplace, nombrada en honor a Pierre-Simon de Laplace (1749–1827), es atractiva en muchas formas. Primero, convierte muchos problemas difíciles en *Ecuaciones Diferenciales* en meros problemas de *Álgebra Universitaria*, aunque frecuentemente bastante complicados. Segundo, es una fascinante analogía entre dos mundos —el mundo del tiempo, t , es decir el nuestro, y el mundo del espacio de las transformadas, s , el cual es un lugar extraño a nuestro entendimiento, sin embargo, uno que resuelve problemas prácticos e importantes—. Tercero, calcular una Transformada de Laplace a mano es un excelente repaso de las técnicas de integración —que a su vez nos ofrece un repaso de *Cálculo II*, típicamente un año o más después de haberlo aprendido—. Veamos ahora cómo calcular Transformadas de Laplace en Sage.

4.23.1 Transformando de $f(t)$ a $L(s)$

Para nuestro ejemplo de apertura, encontraremos el $L(s)$ que coincide con $f(t) = t \cos(t)$ en nuestro mundo. Debemos escribir

Código de Sage

```
1 var('s t')
2 f(t) = t * cos(t)
3 L(s) = laplace(f, t, s)
4 print(L(s))
```

a lo que recibimos la respuesta

$$2*s^2/(s^2 + 1)^2 - 1/(s^2 + 1)$$

Cambiamos $f(t)$ a una nueva función, manteniendo el resto del código sin alterar. Ahora consideramos $f(t) = 5t e^{t-3}$ como nuestro ejemplo. Cambiamos solamente la segunda línea a lo siguiente:

Código de Sage

```
2 f(t) = 5 * t * exp(t-3)
```

lo que resulta en la respuesta

$$5e^{-3}/(s-1)^2$$

Después de todos estos años, aún es sorprendente para el autor cómo una *enorme* colección de funciones, algunas de las cuales son bastante complejas en términos de t , se convierten en meras funciones racionales de s . Por supuesto, ese es el punto: si las funciones no se hicieran vastamente más simple en el mundo de la transformada, entonces ¿no habría razón para transformarlas en primer lugar!

4.23.2 Calculando la Transformada de Laplace de “la forma larga”

Dado que la Transformada de Laplace está definida como

$$L(s) = \int_{0+}^{\infty} f(t)e^{-st} dt,$$

normalmente imaginaríamos que el código para calcularla en nuestro ejemplo anterior sería como abajo:

Código de Sage

```
1 var('s t')
2 f(t) = 5 * t * exp(t-3)
3 L(s) = integral(f(t) * exp(-s*t), t, 0, oo)
4 print(L(s))
```

Sin embargo, Sage nos devuelve un error, que consiste de muchas líneas imposibles de interpretar para nosotros, seguidas del siguiente texto como líneas finales. Recordemos: en Sage, las últimas líneas de un mensaje de error son las que más importan.

```
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(s-1>0)', see 'assume?' for more
details)
Is s-1 positive, negative or zero?
```

Como podemos ver, Maxima, el paquete dentro de Sage que calcula las integrales, está confundido acerca del signo de $s-1$. Por lo tanto, añadimos la línea `assume(s>1)` inmediatamente encima de la integral. La primera vez que vimos el comando `assume` fue en la página 56 y, aunque pueden existir otros usos para este, el autor solo lo ha visto ser usado como ayuda para que los algoritmos de integración y sumatoria lleguen a una conclusión. Es muy poco común. Otro ejemplo, en una sumatoria, puede ser encontrado en la página 176.

Después de hacer esto, obtenemos el resultado

$$5e^{-3}/(s-1)^2$$

que es indudablemente correcto. Sin embargo, solo por curiosidad, tal vez queramos intentar los otros dos casos, es decir `assume(s<1)` y `assume(s==1)`. Pero, por ejemplo con $s=1$, Sage nos devuelve el mensaje de error abajo (después de muchas líneas sin sentido para nosotros).

```
ValueError: Integral is divergent.
```

En resumen, sentimos que Maxima está justificada en demandar esa suposición explícita, pues la integral nos es convergente en otro caso. Más aun, el comando `laplace` hace todo esto por nosotros, lo cual es muy conveniente.

4.23.3 Transformando de $L(s)$ a $f(t)$

Veamos ahora cómo convertir nuestra $L(s)$ del espacio de las transformadas de vuelta a nuestro mundo. El código a escribir es

Código de Sage

```
1 var('s t')
2 L(s) = 5 * e^(-3) / (s - 1)^2
3 print(inverse_laplace(L(s), s, t))
```

Esto produce la respuesta correcta:

$$5 * t * e^{(t - 3)}$$

Alternativamente, podemos intentar desafiar a Sage con un ejercicio más difícil. Imaginemos que resolviendo un problema en *Ingeniería de Sistemas de Control* tenemos $L(s) = 1/(s^3 + 1)$ como respuesta final. Calcular la transformada inversa será bastante difícil hecho a mano, pues debemos realizar una descomposición en fracciones parciales. Sage, por otro lado, puede calcularlo fácilmente. Simplemente cambiamos la línea del medio en el código anterior a

Código de Sage

```
2 L(s) = 1 / (s^3 + 1)
```

Eso producirá la respuesta

$$\frac{1}{3} * (\sqrt{3} * \sin(\frac{1}{2} * \sqrt{3} * t) - \cos(\frac{1}{2} * \sqrt{3} * t)) * e^{(1/2 * t)} + \frac{1}{3} * e^{(-t)}$$

Sin embargo, esa función es muy difícil de comprender. Por ejemplo, si tuviéramos que ponerla en un artículo matemático técnico, entonces tendríamos que codificarla en \LaTeX . Por suerte, Sage puede ayudarnos con esto. Si reemplazamos la última línea de nuestro código con

Código de Sage

```
3 latex(inverse_laplace(L(s), s, t))
```

entonces obtenemos la aterradora respuesta

$$\frac{1}{3} \sqrt{3} \sin\left(\frac{1}{2} \sqrt{3} t\right) - \cos\left(\frac{1}{2} \sqrt{3} t\right) e^{\frac{1}{2} t} + \frac{1}{3} e^{-t}$$

que podemos introducir en un artículo matemático escrito en \LaTeX . Una vez compilado el artículo, el resultado es como sigue:

$$\frac{1}{3} \left(\sqrt{3} \sin\left(\frac{1}{2} \sqrt{3} t\right) - \cos\left(\frac{1}{2} \sqrt{3} t\right) \right) e^{\left(\frac{1}{2} t\right)} + \frac{1}{3} e^{(-t)}$$

lo cual es por supuesto mucho más legible.

Ya habíamos estudiado el comando `latex` y cómo transforma expresiones de Sage en texto de \LaTeX en la sección 4.15 de la página 159.

4.24 Cálculo vectorial en Sage

En esta sección aprenderemos cómo usar Sage para calcular los operadores más importantes¹⁴ en *Cálculo Vectorial*, incluyendo la divergencia, la rotacional, el Laplaciano, el Hessiano y el Jacobiano, al igual que cómo calcular integrales dobles. Puede resultar útil revisar el material sobre gráficas de contornos (sección 3.5 en la página 102) y el material sobre gráficas de campos vectoriales (sección 3.7 en la página 110), pues eso nos permitirá graficar muchos ejemplos a lo largo de esta sección.

El lector no debe sentirse mal si encuentra este contenido confuso. Mientras los docentes de física tienden a conocer este material a detalle, la mayoría de los docentes de matemáticas no tienen estas fórmulas en la punta de los dedos, a menos que hayan enseñado el curso recientemente o que sea su área de investigación. La fenomenalmente famosa guía de estudio para la enseñanza de este material a ingenieros es un libro llamado *Div, Grad, Curl, and All That: an Informal Text on Vector Calculus*, de Harry Moritz Schey, publicado por W. W. Norton and Company. La cuarta edición fue publicada en 2004, pero la primera edición data de 1973. La popularidad longeva de este texto es un testamento a su tono único, locuaz, conversacional, casi sarcástico.

4.24.1 Notación para funciones con valores vectoriales

Antes de empezar, presentamos la notación que usaremos a lo largo de esta sección. Una función matemática f que toma un vector n -dimensional como entrada y devuelve un vector m -dimensional como salida, es escrita $\vec{f}: \mathbb{R}^n \rightarrow \mathbb{R}^m$. Observemos que esta notación es consistente con álgebra lineal. Si $\vec{f}(\vec{x}) = A\vec{x}$ para alguna matriz A de $m \times n$, entonces $\vec{f}: \mathbb{R}^n \rightarrow \mathbb{R}^m$.

Las funciones que vemos en la mayor parte de *Cálculo Multivariado* son aquellas como

$$g(x, y, z) = x^2 + y^3 + z^4 + xyz^2$$

que tienen una entrada multivariada (o vectorial), pero una sola salida. Esto se indica en este caso por $g: \mathbb{R}^3 \rightarrow \mathbb{R}$. En contraste, las funciones que aprendemos durante *Cálculo I* son similares a $b(x) = 2e^{-x}$ y esas son escritas como $b: \mathbb{R} \rightarrow \mathbb{R}$.

El caso menos común, que no es del todo raro, es una función con una entrada y muchas salidas (una salida vectorial). Un ejemplo de esto sería una función que da la posición de un avión en \mathbb{R}^3 en cualquier punto del tiempo, tal como

$$\vec{s}(t) = \langle 200 + 3t, 300 - t, 400 + 5t \rangle,$$

que es denotada por $\vec{s}: \mathbb{R} \rightarrow \mathbb{R}^3$.

En resumen, la dimensión de la entrada va en el primer \mathbb{R} de la notación y la dimensión de la salida va en el segundo \mathbb{R} . Sin embargo, la dimensión “1” nunca se escribe. La flecha sobre \vec{f} y \vec{s} fue incluida porque estas funciones tienen salidas multidimensionales (o vectoriales), mientras que fue excluida en g y b porque estas tienen salidas unidimensionales (o escalares).

¹⁴Por supuesto, el gradiente es el operado más importante en *Cálculo Vectorial* (al menos en matemáticas aplicadas). Usualmente el gradiente es presentado en un curso anterior, tal como *Cálculo Multivariado*. Ya aprendimos cómo calcular gradientes en la subsección 4.3.2 en la página 130, por lo que no lo repetiremos aquí.

4.24.2 Calculando la matriz Hessiana

El Hessiano¹⁵ de una función $f: \mathbb{R}^3 \rightarrow \mathbb{R}$ está dado por

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x \partial x} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y \partial y} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z \partial z} \end{bmatrix}$$

y consiste de todas las posibles segundas derivadas, arregladas en una matriz de 3×3 . Para otras dimensiones, la definición es modificada de la forma obvia.

Seguiremos usando nuestro ejemplo de la sección sobre el gradiente en la página 130. Específicamente, consideramos

$$g(x, y) = xy + \sin(x^2) + e^{-x}.$$

Para este ejemplo, el siguiente código:

Código de Sage

```
1 g(x,y) = x*y + sin(x^2) + e^(-x)
2 diff(g, 2)
```

produce la respuesta

```
[(x, y) |--> -4*x^2*sin(x^2) + 2*cos(x^2) + e^(-x)    (x, y) |--> 1]
[(x, y) |--> 1                                         (x, y) |--> 0]
```

Este extraño formato es para recordarnos que para cada punto en el plano coordenado (x, y) , cada una de las segundas derivadas es un número. Sin embargo, hay cuatro de ellas, así que uno obtiene cuatro números distintos. El arreglo natural es como una matriz de 2×2 .

Recordemos que en la sección 1.15 en la página 61 estudiamos la representación de funciones formales en Sage, como son las entradas de la anterior matriz. Si el lector prefiere un formato más sencillo para las funciones, también vimos que estas se pueden evaluar en un punto genérico (no específico). Por suerte, en este caso, no es necesario hacerlo con cada entrada por separado: Sage nos permite escribir

Código de Sage

```
2 diff(g, 2)(x, y)
```

para obtener

```
[-4*x^2*sin(x^2) + 2*cos(x^2) + e^(-x)    1]
[1                                           0]
```

Otra forma de pedir a Sage que calcule la Matriz Hessiana es con el siguiente código:

¹⁵Nombrado en honor a Ludwig Otto Hesse (1811–1874), quien es conocido por sus resultados en matemáticas y física, y por ser pionero en el estudio de las aplicaciones del determinante, pero también por escribir un popular texto elemental sobre geometría analítica de líneas y círculos.

Código de Sage

```

1 g(x,y) = x*y + sin(x^2) + e^(-x)
2 H = diff(g, 2)
3
4 print(H(0, 3))
5 print()
6 print(H(0, 0))
7 print()
8 print(H(sqrt(pi), 2))

```

Como podemos ver aquí, hemos pedido el Hessiano, pero en un sentido numérico. Hemos pedido la matriz específica en los puntos $(0, 3)$, $(0, 0)$ y $(\sqrt{\pi}, 2)$. Las respuestas son las que esperamos, como se denota bajo:

$$\begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix}, \quad \begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix}, \quad \begin{bmatrix} e^{(-\sqrt{\pi})} - 2 & 1 \\ 1 & 0 \end{bmatrix}.$$

Observemos que tres de cuatro entradas en la matriz tienen valores fijos. La esquina superior derecha varía con x , pero no con y . El lector puede intentar con otros puntos para verificar que esto es cierto. Ahora bien, si observamos el Hessiano simbólico, veremos que en efecto tres de cuatro entradas son constantes, mientras que la superior izquierda es una función de x , pero constante con respecto a y .

En ocasiones solo queremos conocer el determinante del Hessiano. En Economía Matemática, hay situaciones en las que producir ejemplos de funciones $f(x, y)$ con un único mínimo local (y por lo tanto, global) es muy útil. Cuando nos restringimos a funciones de dos variables, $f(x, y)$, hay un atajo muy útil. Resulta que este fenómeno (es decir, que el mínimo o máximo local es único) está garantizado si el determinante del Hessiano es positivo para todos los puntos (x, y) . Este es el resultado análogo de que funciones a una variable $g(x)$ tienen un único mínimo local (y por lo tanto, global) si $g''(x) > 0$ para todo x . Más aun, las funciones univariadas $g(x)$ tienen un único máximo local (y por lo tanto, global) si $g''(x) < 0$ para x . Con eso en mente, podemos escribir

Código de Sage

```

1 g(x,y) = (x + y + 2)*(x + y + 1)*(x + y - 1)*(x + y - 2)
2 h(x,y) = det(diff(g, 2))
3 print(h(x, y))

```

para obtener el determinante del Hessiano rápidamente. En este caso, resulta ser cero en toda partes.

4.24.3 Calculando el Laplaciano

El Laplaciano de una función $g: \mathbb{R}^3 \rightarrow \mathbb{R}$ es denotado ya sea por $\nabla \cdot \nabla g$ o $\nabla^2 g$, dependiendo de cuál texto de matemáticas se está leyendo, y está definido por

$$\nabla \cdot \nabla g = \nabla^2 g = \frac{\partial^2}{\partial x \partial x} g + \frac{\partial^2}{\partial y \partial y} g + \frac{\partial^2}{\partial z \partial z} g,$$

que es simplemente la suma de las entradas de la diagonal principal de la matriz¹⁶ Hessiana de g . El Laplaciano es nombrado en honor a Pierre-Simon de Laplace (1749–1827). Esencialmente, es la suma de las derivadas parciales de segundo orden, *pero excluyendo* las derivadas parciales mixtas. Este operador no está predefinido en Sage, pero es fácil de calcular de todas maneras, usando comandos existentes.

Consideremos la función $g: \mathbb{R}^3 \rightarrow \mathbb{R}$ que vimos en la página 194, es decir,

$$g(x, y, z) = x^2 + y^3 + z^4 + xyz^2.$$

¹⁶Esta suma suele llamarse “la traza” de la matriz.

La derivadas parciales de segundo orden relevantes son

$$\begin{aligned}\frac{\partial^2 g}{\partial x \partial x} &= 2, \\ \frac{\partial^2 g}{\partial y \partial y} &= 6y, \\ \frac{\partial^2 g}{\partial z \partial z} &= 2xy + 12z^2.\end{aligned}$$

Por lo tanto, el Laplaciano de g es simplemente

$$\nabla \cdot \nabla g = \nabla^2 g = 2xy + 12z^2 + 6y + 2,$$

lo que fue fácilmente calculado a mano.

Sin embargo, para funciones más complicadas, tal vez prefiramos usar Sage. Para esta función g , escribiríamos

Código de Sage

```
1 g(x, y, z) = x^2 + y^3 + z^4 + x*y*z^2
2
3 L(x, y, z) = diff(g, x, x) + diff(g, y, y) + diff(g, z, z)
4
5 print('Laplaciano:')
6 print(L(x,y,z))
7
8 print('Hessiano:')
9 print(diff(g, 2))
```

Obtendríamos entonces la siguiente salida:

```
Laplaciano:
2*x*y + 12*z^2 + 6*y + 2
Hessiano:
[(x, y, z) |--> 2      (x, y, z) |--> z^2      (x, y, z) |--> 2*y*z      ]
[(x, y, z) |--> z^2    (x, y, z) |--> 6*y      (x, y, z) |--> 2*x*z      ]
[(x, y, z) |--> 2*y*z  (x, y, z) |--> 2*x*z    (x, y, z) |--> 2*x*y + 12*z^2]
```

Podemos ver aquí que, en efecto, el laplaciano es la suma de entradas de la diagonal principal (la traza) del Hessiano.

4.24.4 La matriz Jacobiana

Ahora consideremos una función $\vec{f} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$. Una forma de entender una función tal con valores vectoriales, digamos por ejemplo,

$$\vec{f}(x, y, z) = y\vec{i} + x^2\vec{j} + 3\vec{k} = \langle y, x^2, 3 \rangle,$$

es concebirla como si en realidad fuera tres funciones independientes, cada una de la forma $f_i : \mathbb{R}^3 \rightarrow \mathbb{R}$. En este caso particular, estas serían

$$\begin{aligned}f_1(x, y, z) &= y, \\ f_2(x, y, z) &= x^2, \\ f_3(x, y, z) &= 3, \\ \vec{f}(x, y, z) &= \langle f_1(x, y, z), f_2(x, y, z), f_3(x, y, z) \rangle.\end{aligned}$$

Como podemos ver, el papel de f_1 , f_2 y f_3 es indicarnos una coordenada particular de \vec{f} .

La matriz Jacobiana¹⁷ está definida como sigue:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial z} \end{bmatrix}.$$

En nuestro caso particular, tenemos

$$J = \begin{bmatrix} 0 & 1 & 0 \\ 2x & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

En ocasiones, cuando alguien se usa al término “Jacobiano”, en realidad se está refiriendo al determinante de la Matriz Jacobiana. En este caso, el resultado sería 0, pues J tiene una fila de ceros y por lo tanto es evidentemente singular.

Para calcular todo esto en Sage, usamos el siguiente código:

Código de Sage

```
1 f(x,y,z) = [y, x^2, 3]
2
3 print('Función:')
4 print(f)
5 print('Jacobiano:')
6 print(diff(f))
7 print('Determinante Jacobiano:')
8 print(det(f.diff()))
```

Podemos ver aquí que Sage considera el Jacobiano como el significado natural de la palabra “derivada” para una función con argumento e imagen vectoriales. Pensándolo con detenimiento, esto tiene sentido. Específicamente, si la imagen de una función es unidimensional, pero su argumento es multidimensional, entonces el Jacobiano sería igual/idéntico al gradiente como un vector fila. Y Sage considera a su vez el gradiente como el significado natural de la palabra “derivada” para este último tipo de función. También podemos ver aquí que si se desea el determinante de la Matriz Jacobiana, debemos usar explícitamente el comando `det`, como con cualquier otro determinante.

4.24.5 La divergencia

La divergencia es un operador importante, pero no está predefinido en Sage. Vimos que el operador Laplaciano tampoco lo está, y pronto veremos que la rotacional tampoco. Sin embargo, la divergencia es fácil de calcular usando comandos existentes.

Para cualquier función $\vec{f} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$, la divergencia se define por

$$\nabla \cdot \vec{f} = \text{div} \vec{f} = \frac{\partial f_1}{\partial x} + \frac{\partial f_2}{\partial y} + \frac{\partial f_3}{\partial z},$$

que no es nada más que la suma de las entradas de la diagonal principal de la Matriz¹⁸ Jacobiana. En ese sentido, es muy similar al Laplaciano, que es la suma de las entradas de la diagonal principal de la Matriz Hessiana. Obsérvese que para cada punto en el espacio tridimensional ordinario, la divergencia nos proveerá de un número, no un vector. Esto significa que la divergencia de \vec{f} es una función $\mathbb{R}^3 \rightarrow \mathbb{R}$.

Ahora realizamos un ejemplo. Consideremos esta función algo vistosa:

$$\vec{f}(x, y, z) = xy^2\vec{i} + yz^2\vec{j} + zx^2\vec{k},$$

para la cual calcularemos tanto la divergencia como la Matriz Jacobiana. El código se encuentra abajo. Dado que es un ejemplo sencillo, el lector tal vez quiera calcularlo a mano primero.

¹⁷Nombrada en honor a Carl Gustav Jacob Jacobi (1804–1851).

¹⁸Esta suma suele llamarse “la traza” de la matriz.

Código de Sage

```

1  f1(x,y,z) = x*y^2
2  f2(x,y,z) = y*z^2
3  f3(x,y,z) = z*x^2
4
5  f(x, y, z) = (f1(x,y,z), f2(x,y,z), f3(x,y,z))
6
7  div(x, y, z) = diff(f1, x) + diff(f2, y) + diff(f3, z)
8
9  print('Divergencia:')
10 print(div(x,y,z))
11
12 print('Jacobiano:')
13 print(derivative(f))

```

Este código produce la siguiente salida:

```

Divergencia:
x^2 + y^2 + z^2
Jacobiano:
[(x, y, z) |--> y^2      (x, y, z) |--> 0      (x, y, z) |--> 2*x*z]
[(x, y, z) |--> 2*x*y    (x, y, z) |--> z^2     (x, y, z) |--> 0      ]
[(x, y, z) |--> 0        (x, y, z) |--> 2*y*z    (x, y, z) |--> x^2   ]

```

Podemos ver aquí que, en efecto, la divergencia es igual a la suma de las entradas de la diagonal principal (la traza) de la Matriz Jacobiana.

4.24.6 Verificando una vieja identidad

Probablemente el lector estudió el teorema que afirma que la divergencia del gradiente es igual al Laplaciano. En la página 196, calculamos el Laplaciano de

$$g(x, y, z) = x^2 + y^3 + z^4 + xyz^2.$$

Por lo tanto, calculemos la divergencia de su gradiente, y veamos si obtenemos la misma respuesta que antes.

Para calcular el gradiente, escribimos

Código de Sage

```

1  g(x, y, z) = x^2 + y^3 + z^4 + x*y*z^2
2
3  print('Original:')
4  print(g(x,y,z))
5
6  gradiente = derivative(g)
7
8  print('Gradiente:')
9  print(gradiente)

```

que resulta en la salida

```

Original:
x*y*z^2 + z^4 + y^3 + x^2
Gradiente:
(x, y, z) |--> (y*z^2 + 2*x, x*z^2 + 3*y^2, 2*x*y*z + 4*z^3)

```


Ahora, con esto en mente, podemos interpretar las tres componentes de este último resultado como f_1 , f_2 y f_3 . Programaremos Sage para tomar las derivadas parciales y calcular su suma. Usamos el siguiente código:

Código de Sage

```
1 f1(x,y,z) = y*z^2 + 2*x
2 f2(x,y,z) = x*z^2 + 3*y^2
3 f3(x,y,z) = 2*x*y*z + 4*z^3
4
5 divgrad(x,y,z) = diff(f1, x) + diff(f2, y) + diff(f3, z)
6 print('Divergencia del gradiente:')
7 print(divgrad(x, y, z))
```

Así, finalmente tenemos la salida

```
'Divergencia del gradiente:'
2*x*y + 12*z^2 + 6*y + 2
```

Como podemos ver, esta es una coincidencia exacta con el Laplaciano que habíamos calculado en la página 196. Entonces, hemos verificado que la divergencia del gradiente (de esta función particular) es igual al Laplaciano. Esto está muy lejos de probar el teorema en general, pero es bueno saber que al menos podemos verificar instancias individuales.

4.24.7 La rotacional de una función con valores vectoriales

La rotacional de una función $\vec{f} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$, es decir, de la forma

$$\vec{f}(x, y, z) = \langle f_1(x, y, z), f_2(x, y, z), f_3(x, y, z) \rangle,$$

es denotada $\nabla \times \vec{f} = \text{rot } \vec{f}$, y se define por

$$\nabla \times \vec{f} = \text{rot } \vec{f} = \left(\frac{\partial}{\partial y} f_3 - \frac{\partial}{\partial z} f_2 \right) \vec{i} + \left(\frac{\partial}{\partial x} f_1 - \frac{\partial}{\partial z} f_3 \right) \vec{j} + \left(\frac{\partial}{\partial x} f_2 - \frac{\partial}{\partial y} f_1 \right) \vec{k},$$

la cual de seguro no es una fórmula muy fácil de memorizar. En consecuencia, existe un truco para recordarla.

Un truco para recordar la fórmula de la rotacional Personalmente, el autor siempre se ha sentido frustrado con el hecho que la rotacional es tan difícil de calcular, particularmente porque es el operador más importante después del gradiente en cualquier aplicación de ingeniería o física. De hecho, existe una regla nemotécnica semicoherente para recordar cómo calcular la rotacional a mano. Consideremos

$$\begin{aligned} \det \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ f_1 & f_2 & f_3 \end{bmatrix} \\ = \det \begin{bmatrix} \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \end{bmatrix} \vec{i} - \det \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial z} \end{bmatrix} \vec{j} + \det \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix} \vec{k} \\ = \left(\frac{\partial}{\partial y} f_3 - \frac{\partial}{\partial z} f_2 \right) \vec{i} - \left(\frac{\partial}{\partial x} f_3 - \frac{\partial}{\partial z} f_1 \right) \vec{j} + \left(\frac{\partial}{\partial x} f_2 - \frac{\partial}{\partial y} f_1 \right) \vec{k} \\ = \text{rot } \vec{f} \end{aligned}$$

Esencialmente, si uno puede memorizar esta matriz de 3×3 , y calcula su determinante correctamente, entonces puede calcular la rotacional de cualquier función $\mathbb{R}^3 \rightarrow \mathbb{R}^3$.

El lector puede sentirse extrañado por la palabra “semicoherente” que usa el autor para describir esta fórmula. La razón es que $\partial/\partial x$ es un operador, y calcular

$$\frac{\partial}{\partial x} f_2$$

no es una multiplicación de ninguna manera, forma o clase; pero, el proceso de calcular esos determinantes de 2×2 consiste exacta y precisamente de dos multiplicaciones separadas por una sustracción. Sin embargo, si uno sigue ciegamente este proceso, los símbolos sí resultan estar en las posiciones correctas, visualmente hablando, y por lo tanto este truco nemotécnico es útil —aun cuando no tenga sentido desde el punto de vista matemático—. Este truco es el equivalente de nivel universitario de la broma que dice que $(3)(3)$ debería ser igual a 33.

Un ejemplo sencillo sobre la rotacional En ocasiones, un ejemplo sencillo es lo mejor. El siguiente caso fue encontrado en el artículo de Wikipedia “Curl: (mathematics)” el 29 de junio de 2014. Aunque el autor normalmente no citaría Wikipedia, la exposición es notablemente lúcida en este caso. Sería una lástima sustituirlo con un ejemplo más complicado simplemente para evitar el estigma de Wikipedia. Se ha tenido un cuidado extra en asegurarse que no existan errores en el procedimiento, pues Wikipedia es famosamente susceptible a errores.

Consideremos la siguiente función:

$$\vec{c} = y^2 \vec{i} = \langle y^2, 0, 0 \rangle.$$

Dado que f_2 y f_3 son la función cero en este caso, entonces nos enfocamos en f_1 . Podemos ver que

$$\frac{\partial}{\partial x} f_1 = 0 \quad \frac{\partial}{\partial y} f_1 = 2y \quad \frac{\partial}{\partial z} f_1 = 0$$

y así, la fórmula normal de la rotacional rápidamente se simplifica a

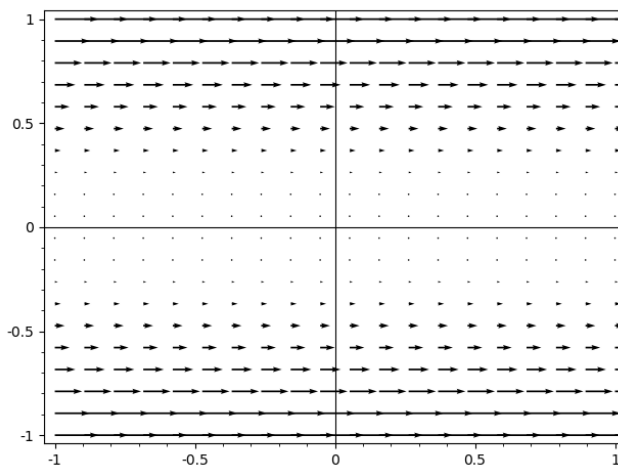
$$\begin{aligned} \nabla \times \vec{f} = \text{rot } \vec{f} &= \left(\frac{\partial}{\partial y} f_3 - \frac{\partial}{\partial z} f_2 \right) \vec{i} + \left(\frac{\partial}{\partial x} f_1 - \frac{\partial}{\partial z} f_3 \right) \vec{j} + \left(\frac{\partial}{\partial x} f_2 - \frac{\partial}{\partial y} f_1 \right) \vec{k} \\ &= (0 - 0) \vec{i} + (0 - 0) \vec{j} + (0 - 2y) \vec{k} \\ &= -2y \vec{k} \\ &= \langle 0, 0, -2y \rangle \end{aligned}$$

Veamos la gráfica del campo vectorial original \vec{c} . Para poder graficarlo en una página bidimensional, ignoraremos la coordenada z completamente. Esto no es problema en este caso, pues nuestro ejemplo no usa la variable z . Podemos escribir

Código de Sage

```
1 var('y')
2 plot_vector_field((y^2,0), (x,-1,1), (y,-1,1))
```

Obtenemos la siguiente imagen:



Como podemos ver, todas las flechas apuntan a la derecha. La coordenada x no es relevante; las flechas cerca del eje x (aquellas con una coordenada y muy pequeña en valor absoluto) tienen una magnitud muy pequeña, mientras que las que están lejos del eje x (aquellas con una coordenada y muy grande en valor absoluto) tienen una magnitud muy grande. Ahora preguntémosnos: si esta gráfica representara el flujo de un fluido, ¿cómo esperaríamos que una hélice se comportara si fuera insertada:

- por encima del eje x ?
- sobre el eje x ?
- por debajo del eje x ?

Tomémonos un momento para pensar realmente acerca de esto. (Se recomienda que el lector no continúe más adelante hasta que haya intentado responder a este experimento mental.)

En cualquier caso, si la hélice estuviera sobre el eje x , no habría rotación. Sin embargo, por encima del eje x , esperaríamos una rotación en el sentido de las manecillas del reloj, y por debajo del eje x , esperaríamos una rotación en sentido contrario a las manecillas del reloj. Esto se debe a que un lado de la hélice está siendo golpeado en mucha mayor medida que el otro. Recordando que una rotación en el sentido de las manecillas del reloj en el plano indica un vector hacia plano, y la rotación contra las manecillas del reloj indica un vector hacia afuera del plano, vemos que nuestra respuesta de $\langle 0, 0, -2y \rangle$ tiene sentido:

- Cuando y es cero, todas las coordenadas de la rotacional son cero, así que la rotacional es nula. (Esto implica que la hélice no rota.)
- Cuando y es positivo, la tercera coordenada de la rotacional es negativa, así que la rotacional apunta hacia el papel. (Esto implica que la hélice rota con las manecillas del reloj.)
- Cuando y es negativo, la tercera coordenada de la rotacional es positiva, así que la rotacional apunta hacia afuera del plano. (Esto implica que la hélice rota contra las manecillas del reloj.)

Ahora debemos calcular esto en Sage. El cálculo manual fue fácil en este ejemplo sencillo, pero en otros casos puede tornarse extremadamente desagradable. El siguiente pedazo de código es muy útil:

Código de Sage

```

1  var('y z')
2
3  f1 = y^2
4  f2 = 0
5  f3 = 0
6
7  rotacional = (diff(f3, y) - diff(f2, z), diff(f1, z) - diff(f3, x), diff(f2, x) -
    ↪ diff(f1, y))
8  print(rotacional)

```

Naturalmente, obtenemos la salida esperada $\langle 0, 0, -2*y \rangle$ de este código.

Un ejemplo más difícil sobre la rotacional Ahora consideremos un ejemplo más complicado, para el cual no querríamos calcular la rotacional a mano. Una función ligeramente más complicada es

$$\vec{f}(x, y, z) = \langle x^2y, x + y + \sin(x), 0 \rangle,$$

para la cual podemos usar el siguiente código:

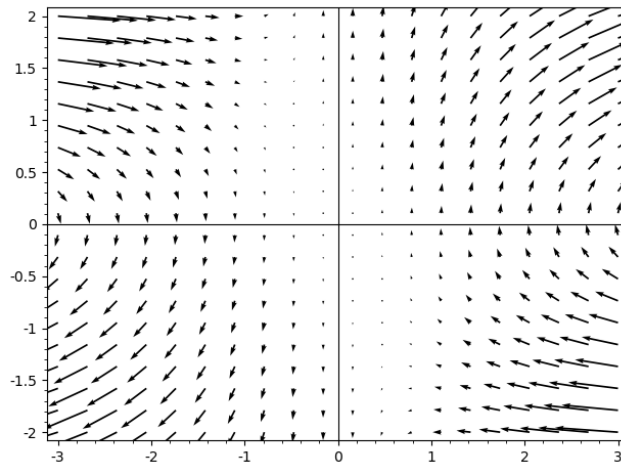
Código de Sage

```

1  var('y z')
2
3  f1 = x^2*y
4  f2 = x + y + sin(x)
5  f3 = 0
6
7  rotacional = (diff(f3, y) - diff(f2, z), diff(f1, z) - diff(f3, x), diff(f2, x) -
    ↪ diff(f1, y))
8  print(rotacional)
9
10 fvec = (f1, f2)
11 plot_vector_field(fvec, (x, -3, 3), (y, -2, 2))

```

Como podemos ver, hemos añadido dos líneas al final para generar una gráfica vectorial, con lo que obtenemos la siguiente imagen:



4.24.8 Desafío: Verificar algunas identidades sobre la rotacional

Aquí tenemos un desafío para el lector. Abra un buen texto de *Cálculo vectorial*.

- Encuentre varias funciones de la forma $g : \mathbb{R}^3 \rightarrow \mathbb{R}$ y verifique que la rotacional del gradiente es el vector cero, $\vec{0} = \langle 0, 0, 0 \rangle$.
- Encuentre varias funciones de la forma $\vec{f} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ y verifique que la divergencia de la rotacional es de hecho 0.

Nota: Este último punto es un poco más difícil dado que ninguno de los operadores mencionados ahí están predefinidos en Sage. Sin embargo, el trozo de código que usamos antes para calcular los dos ejemplos de rotacional deberían ser de mucha ayuda.

4.24.9 Integrales múltiples

El proceso de integración en Sage es el mismo para integrales múltiples que para integrales individuales. Sin embargo, existe una notación alternativa que hace las cosas más fáciles de entender. Por ejemplo,

$$\int_0^1 x e^{-x^2} dx$$

puede ser escrito en Sage como

Código de Sage

```
1 integral(x * exp(-x^2), x, 0, 1)
```

tal como vimos en la sección 1.12 empezando en la página 48. Pero por otro lado, la siguiente sintaxis también es aceptable:

Código de Sage

```
1 integral(x * exp(-x^2), (x, 0, 1))
```

Esta segunda notación parece indeseable pues contiene más caracteres y tiene paréntesis anidados. Sin embargo, ahora veremos que esto tiene mucho más sentido cuando trabajamos con múltiples integrales. Consideremos el siguiente ejemplo:

$$\int_0^1 \int_0^y e^{y^2} dx dy.$$

Con la antigua notación, esto se escribiría

Código de Sage

```
1 var('y')
2 integral(integral(exp(y^2), x, 0, y), y, 0, 1)
```

Usando la nueva notación, escribimos

Código de Sage

```
1 var('y')
2 integral(integral(exp(y^2), (x, 0, y)), (y, 0, 1))
```

Al parecer del autor, así es más fácil de entender que la integral “interna” tiene dominio $0 < x < y$ y que la “externa” tiene dominio $0 < y < 1$. Esta nueva notación es similar a la forma de expresar intervalos que vimos primero en la página 101, bajo el encabezado “Compatibilidad retrógrada”, cuando aprendimos técnicas avanzadas de graficación.

4.25 Usando Sage y L^AT_EX, parte dos

Esta sección está escrita para el lector que ya sabe cómo componer documentos usando el sistema tipográfico L^AT_EX. Si el lector nunca antes usó L^AT_EX, es altamente recomendable para su vida profesional que lo aprenda. Mientras tanto, esta sección puede saltarse sin ninguna pérdida de continuidad.

Aquellos que se dedican a la ciencia y la investigación, inevitablemente deben publicar sus resultados. El estándar *de facto* para escribir texto científico, como el de un artículo, una tesis o un libro, es usar el sistema tipográfico L^AT_EX. Vimos en sección 4.15 en la página 159 que Sage puede traducir sus resultados a L^AT_EX, lo que es una gran ventaja en el proceso de escritura. En efecto, si necesitamos incluir un resultado en algún documento (un artículo, por ejemplo), podemos pedir a Sage que haga los cálculos por nosotros, entonces usamos el comando `latex` y finalmente podemos simplemente copiar y pegar la salida en nuestro código de L^AT_EX.

Este proceso tiene muchas ventajas. Por ejemplo, no requerimos hacer los cálculos manualmente, pues Sage mismo se encarga de ello; no necesitamos traducir manualmente nuestro resultado a L^AT_EX, pues el comando `latex` ya hace eso. También, podemos estar seguros que nuestro documento está libre de errores humanos.

Sin embargo, también existen desventajas en esta forma de trabajar. Es un proceso tedioso moverse constantemente de L^AT_EX a Sage y viceversa. También, los datos con los que estábamos trabajando podrían cambiar mientras estamos escribiendo, en cuyo caso tendríamos que repetir el proceso de cálculo y transcripción. Otra desventaja —aunque menos probable— es que podríamos cometer algún error al copiar de Sage a L^AT_EX. Finalmente, es posible que en futuro queramos actualizar o modernizar nuestro documento (especialmente si es un libro y estamos planeando una nueva edición). En ese caso, tendríamos que dar mantenimiento a por lo menos dos archivos diferentes, uno conteniendo el texto en L^AT_EX y el otro conteniendo nuestro código de Sage. Naturalmente, entre más archivos tengamos que mantener y actualizar, tanto más difícil el trabajo y más susceptibles somos de cometer error.

4.25.1 El paquete SageT_EX

Sage viene con un paquete de L^AT_EX, llamado SageT_EX, que está específicamente diseñado para lidiar con estas desventajas, sin sacrificar ninguna de las ventajas. Este paquete nos permite incluir código de Sage en nuestros documentos de L^AT_EX. El código es automáticamente extraído y ejecutado, y los resultados correspondientes son incluidos en nuestro documento, sin la necesidad de intervención humana. En particular, esto resuelve el problema de tener que mantener y actualizar nuestro trabajo, pues todo el código para producir el documento está contenido en un solo archivo.

No explicaremos el uso de SageT_EX a completitud. Sin embargo, después de completar esta sección, el lector será capaz de incluir código de Sage en sus documentos de L^AT_EX.

4.25.2 Usando el paquete Sage \TeX

Para poder usar el paquete, este debe ser visible para nuestra instalación de \LaTeX . La forma más fácil de hacer esto,¹⁹ es copiar el archivo “sagetex.sty” a la misma carpeta que contiene nuestro documento. Es muy importante usar el mismo sagetex.sty que viene con nuestra instalación de Sage; otras versiones podrían no ser compatibles. El archivo puede encontrarse en la carpeta

```
<SAGE>/local/share/texmf/tex/latex/sagetex,
```

donde <SAGE> indica el directorio donde instalamos Sage.

Una vez hecho esto, podemos *importar* el paquete al escribir a línea

```
\usepackage{sagetex}
```

en el preámbulo de nuestro documento de \LaTeX . Entonces, podemos compilarlo usando la siguiente secuencia de comandos en una terminal:

```
latex <documento>.tex
sage <documento>.sagetex.sage
latex <documento>.tex
```

Aquí, <documento> es el nombre de nuestro documento. La primera llamada a latex²⁰ extrae cada pedazo de código de Sage y lo pone en el archivo “<documento>.sagetex.sage”, el que entonces compilamos llamando a sage, y los resultados son incluidos con la segunda llamada a latex.

El comando \backslash sage El comando \backslash sage extrae su argumento, lo ejecuta, usa la función latex de Sage e incluye el resultado en nuestro documento, todo hecho de manera automática. Por ejemplo, si estamos escribiendo acerca de inversiones y préstamos con interés compuesto, podríamos escribir algo como

```
Si invertimos \$900 al 6\% anual, en 90 días tendremos
$\$ \backslash
```

Esto mostrará lo siguiente en nuestro documento:

```
Si invertimos $900 al 6 % anual, en 90 días tendremos $913,32. Por otro lado, el valor actual
de un préstamo al 5 % compuesto anualmente por 30 años, con un pago anual de $500, es
$7686,23.
```

La ventaja aquí es que no necesitamos el cálculo manualmente, ni requerimos transcribir el resultado desde Sage.

Por supuesto, el comando \backslash sage puede hacer mucho más por nosotros que solo cálculos tediosos: en general, puede ejecutar casi cualquier conjunto de instrucciones de Sage. Como un ejemplo algo más complejo, podemos escribir

```
La factorización del polinomio $p(x)=3x^2+14x+8$ es simplemente
$p(x)=\backslash
```

¹⁹Existen otras formas mucho más avanzadas, pero no las discutiremos aquí, por estar fuera de nuestros propósitos.

²⁰Nótese que este no es el comando de Sage, sino el compilador de \LaTeX .

Obtenemos la siguiente salida:

La factorización del polinomio $p(x) = 3x^2 + 14x + 8$ es simplemente $p(x) = (3x + 2)(x + 4)$. También podemos ver que hay 68 números primos entre 12 y 367, inclusive. Por otro lado, es evidente que la matriz

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

es invertible.

El comando `\sagestr` El comando `\sagestr` trabaja de manera similar a `\sage`, pero no usa la función latex de Sage. Es útil para ejecutar funciones o subrutinas cuyas salidas ya se encuentran en formato de \LaTeX .

Supongamos que existe un comando de Sage llamado “`prod_sum()`”, el cual toma dos enteros positivos, calcula el cociente y el residuo de su división, y usa esta información para reescribir el primer número como un producto y una suma. por ejemplo,

- `prod_sum(15, 2) = 7 × 2 + 1`,
- `prod_sum(17, 5) = 3 × 5 + 2`,
- `prod_sum(1234, 34) = 36 × 34 + 10`.

La salida de este comando ya está en formato de \LaTeX , así que escribimos lo siguiente:

Notemos que el número 20 puede descomponerse de muchas formas como un producto y una suma. Por ejemplo, tenemos que $20 = \text{\sagestr{prod_sum}(20, 3)}$, pero $20 = \text{\sagestr{prod_sum}(20, 5)}$ y también $20 = \text{\sagestr{prod_sum}(20, 7)}$.

El resultado que veríamos en nuestro documento es:

Notemos que el número 20 puede descomponerse de muchas formas como un producto y una suma. Por ejemplo, tenemos que $20 = 6 \times 3 + 2$, pero $20 = 4 \times 5 + 0$ y también $20 = 2 \times 7 + 6$.

Este comando ficticio, `prod_sum`, de hecho puede ser programado por nosotros como una subrutina, lo que aprenderemos en el siguiente capítulo. Sin embargo, si el lector siente curiosidad en este momento, aquí lo tiene:

Código de Sage

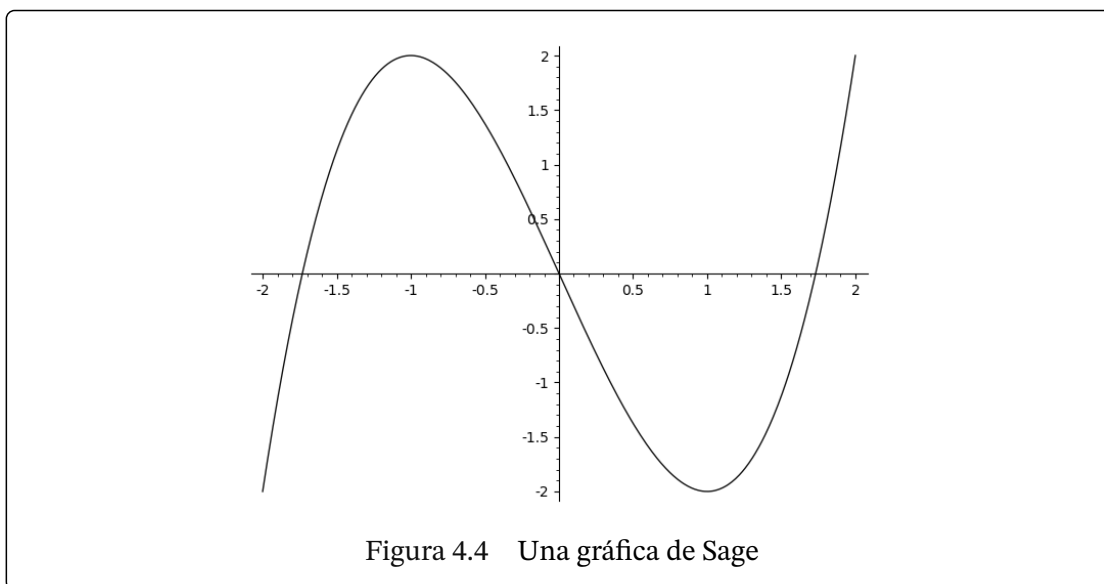
```
1 def prod_sum(m, n):
2     return str(m // n) + '\\times' + str(n) + '+' + str(m % n)
```

Una vez más insistimos que si no entiende cómo funciona este código, no debería preocuparse; podrá entenderlos cuando termine el siguiente capítulo.

El comando `\sageplot` Nos permite incluir gráficos de Sage directamente en nuestro documento. Por ejemplo, si escribimos

```
\begin{figure}[!ht]
  \centering
  \sageplot[scale=0.5]{plot(x^3-3*x, (x,-2,2))}
  \caption{Una gráfica de Sage}
\end{figure}
```

obtenemos la siguiente imagen:



Como podemos apreciar, `\sageplot` acepta los mismos argumentos opcionales que el comando `\includegraphics` de \LaTeX . También acepta otros argumentos opcionales, pero explicarlos está fuera de nuestros propósitos.

El ambiente `sagesilent` Este ambiente extrae su contenido y lo ejecuta, pero no imprime nada en nuestro documento. Es útil para definir variables, constantes, funciones, subrutinas, etc., que pueden ser accedidas más adelante con los comandos `\sage` y `\sagestr`. Por ejemplo,

```
\begin{sagesilent}
  var('x')
  p(x) = -6*x^2 + 11*x + 10
  q(x) = 2*x^2 + 3*x - 20
  ip_2 = integral(p(x), x, 0, 2)
\end{sagesilent}
Los polinomios  $p(x)=\sage{p(x)}$  y  $q(x)=\sage{q(x)}$  tienen el máximo
común divisor  $\sage{gcd(p(x), q(x))}$ . También, es interesante notar que
\begin{equation*}
  \int_0^2 p(x) dx = \sage{ip_2}.
\end{equation*}
```

Veremos lo siguiente en nuestro documento de \LaTeX :

Los polinomios $p(x) = -6x^2 + 11x + 10$ y $q(x) = 2x^2 + 3x - 20$ tienen el máximo común divisor $2x - 5$. También, es interesante notar que

$$\int_0^2 p(x) dx = 26.$$

Podemos ver que, en efecto, los contenidos de `sagesilent` son invisibles, pero Sage los ejecuta de todas maneras.

En el ejemplo que vimos del comando `sagestr`, el ambiente `sagesilent` sería excelente para definir nuestra subrutina `prod_sum`:

```
\begin{sagesilent}
def prod_sum(m, n):
    return str(m // n) + '\\times' + str(n) + '+' + str(m % n)
\end{sagesilent}
```

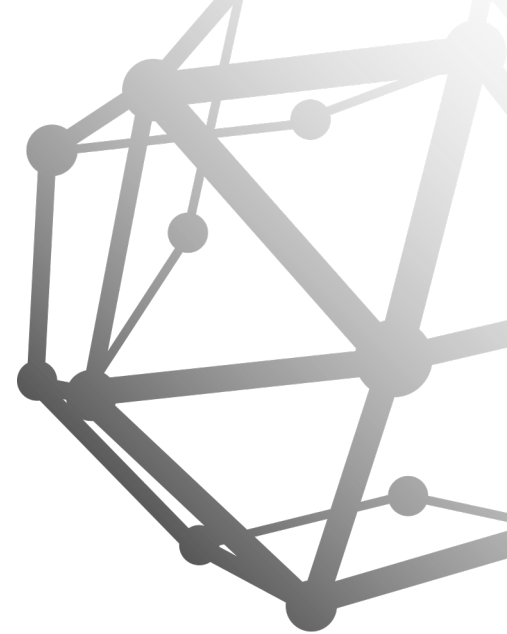
Notemos que el número 20 puede descomponerse de muchas formas como un producto y una suma. Por ejemplo, tenemos que $20 = \text{sagestr}\{\text{prod_sum}(20, 3)\}$, pero $20 = \text{sagestr}\{\text{prod_sum}(20, 5)\}$ y también $20 = \text{sagestr}\{\text{prod_sum}(20, 7)\}$.

4.25.3 La documentación de SageTeX

Evidentemente, SageTeX viene con muchas más características y opciones que las que hemos discutido aquí. Si el lector quiere saber más, la documentación del paquete (en inglés) se puede encontrar en la carpeta

`<SAGE>/local/share/texmf/tex/latex/sagetex/`,

donde `<SAGE>` representa el directorio en el que instalamos Sage.



5

Programando en Sage y Python

Como vimos en los capítulos anteriores, muchas tareas matemáticas importantes pueden realizarse en Sage sin demasiadas líneas de código. Frecuentemente solo un pequeño número de comandos se encargarán de lo que sea que necesitemos. Por otro lado, hay ocasiones en las que buscamos hacer algo más profundo y más complejo. Tareas elaboradas frecuentemente requieren más código, abarcando desde, más o menos, una docena de líneas, hasta enormes programas de 10 000 líneas o más, escritas por varios desarrolladores dispersos alrededor del mundo, a lo largo de un periodo de muchos años.

En este capítulo aprenderemos cómo escribir programas de tamaño intermedio en Sage para realizar algunas tareas matemáticas interesantes. Como Sage está construido sobre Python, aprenderemos un porcentaje respetable de este último en el camino. Por supuesto, Python es un lenguaje de programación mayor y tiene muchas características, y no podemos esperar cubrirlas todas aquí. Sin embargo, veremos cada uno de los comandos más comunes de Python usados en un contexto matemáticamente relevante.

Vale la pena notar que este capítulo está escrito asumiendo que el lector conoce el contenido del capítulo 1, y cómo usar CoCalc, pero no necesariamente el contenido del resto de este libro. El ritmo sugerido para avanzar a través del material aquí presentado es estudiar las secciones 5.1 a 5.7, cada una durante un día. Probablemente los estudiantes menos pacientes y con más práctica puedan hacer dos secciones a la vez, pero esto no es recomendable. Este material es algo pesado y es mejor digerirlo en pedazos pequeños.

Conforme los ejemplos en Sage se hacen más largos, puede resultar tedioso transcribirlos letra por letra en la computadora. Pero no es necesario hacerlo. Simplemente resaltamos el código de la versión PDF (gratis en línea) de este libro, usamos el comando “copiar” en el visor de PDF, vamos a la ventana de Sage y usamos el comando “pegar”. Sin embargo, nótese que es posible que el lector tenga que ajustar el sangrado de las líneas después de pegarlas en Sage.

Similarmente, es muy importante que el lector use CoCalc (<https://cocalc.com/>) para este capítulo, y no el servidor Sage Cell. El autor usa casi exclusivamente este último, y aunque los bloques de instrucciones de 10 a 20 líneas que escribiremos aquí pueden correr bien (y de hecho corren bien) ahí, el lector querrá mantener un registro permanente del código por varias razones. Primero, si hay una pequeña falla en la red, sería un lástima perder todo un programa en el que se ha estado trabajando por una hora o más. Segundo, frecuentemente cuando los expertos programan, suelen recurrir a algún viejo código que han escrito, y lo copian y pegan en uno nuevo. Tercero, si el lector olvida cómo hacer algo, es muy útil poder volver a algún código escrito previamente y ver cómo fue hecho. Como el autor es terco y usa excesivamente el servidor Sage Cell (casi sin usar CoCalc), frecuentemente pierde su trabajo y tiene que empezar desde el principio.

Finalmente, presentaremos algunos desafíos a lo largo de este capítulo. De vez en cuando, después que dos o tres conceptos han sido presentados, se plantea un problema para que el lector lo resuelva por su cuenta. Asegúrese de intentar cada uno de ellos, pues resaltarán enormemente su habilidad de aprender este importante tópico. *La única manera de aprender a programar es programando*. De seguro que el lector encontrará estos ejemplos relativamente sencillos, pero útiles y no triviales. Por otro lado, si no está dispuesto a intentar estos desafíos por su cuenta, entonces no sacará ningún beneficio de leer este capítulo —¡y da lo mismo que ya no proceda más adelante!—.

5.1 Repetición sin fastidio: El bucle `for`

A muchos de nosotros se nos ha enseñado que las computadoras son mejores en la automatización de tareas repetitivas idénticas (o muy similares). Estas son grandes noticias, pues pueden salvar a un ser humano de realizar una actividad tediosa y aburrida, que puede tomar eras en completarse. El mejor ejemplo de esto —y el más comúnmente usado— es llamado el “bucle `for`”. Aprenderemos sobre este ahora.

5.1.1 Usando Sage para generar tablas

Digamos que queremos generar una lista (regular, no de Python) de los 100 primeros cuadrados perfectos. El código para ello sería

Código de Sage

```
1 for i in range(0, 100):    i^2
```

Esto significa que i tomará los valores 0, 1, 2, 3, ..., 99, pero no 100. De esta manera, `range(0, x)` siempre tiene x ítemes en él.

Eso fue fácil, pero no increíblemente útil. Supongamos que estamos graficando a mano y queremos conocer algunos valores de una función en el intervalo $-10 \leq x \leq 10$. Digamos que la función es $f(x) = x^3 - x$. Entonces escribimos

Código de Sage

```
1 for x in range(-10, 11):
2     x^3 - x
```

El sangrado de las líneas es importante, pues indica qué comandos deben ser repetidos por el bucle `for`. Usamos la tecla TAB para ello.

Eso fue interesante, pero no crucial, pues ya hemos aprendido cómo hacer que Sage grafique funciones directamente, en la página 9. Sin embargo, esta es una forma en que las computadoras grafican funciones: evalúan una función en —digamos— 1000 puntos adecuados que están muy juntos entre sí, y grafican esos 1000 puntos, uno en cada ubicación. Para el ojo humano, parece ser una curva muy suave y clara, pero en realidad, es solo una colección de puntos.

A propósito, ¿notó el lector cómo el primer bucle, listando los cuadrados perfectos, estaba todo en una sola línea? Mientras tanto, el bucle con la función $x^3 - x$ está en dos líneas. Aunque es legal dejar todo en una sola línea, se considera buen estilo de programación usar varias. Conforme aprendamos a construir bucles más y más complejos, veremos por qué. Es importante que cada pedazo de código en un programa de computadora sea legible; de otra manera, será difícil repararlo o mejorarlo. No es suficiente que un software computacional “simplemente funcione”.

Ahora veamos un ejemplo de finanzas. Digamos que queremos generar todos los montos finales posibles para un solo depósito de \$ 15 000 de inversión, con una tasa de interés desconocida compuesta anualmente, pero siendo la duración de 5 años. Supongamos que deseamos considerar todas las tasas de interés de la forma 1 %, 2 %, 3 %, 4 %, ..., hasta 29 %. La fórmula para interés compuesto es $A = P(1 + i)^n$, y por lo tanto, escribimos en Sage

Código de Sage

```
1 for i in range(1, 30):
2     N(15_000 * (1 + (i / 100))^5, digits=7)
```

Notemos que si dejamos fuera el `N()`, entonces obtenemos fracciones exactas (números racionales) que lucen realmente extrañas. Intente el lector remover el `N()` y ver qué ocurre —pero asegúrese que cada “(” tiene su “)” correspondiente después de la modificación—. También obsérvese que hemos usado la opción `digits=7` para restringir el redondeo de los resultados hasta el centavo más próximo: cinco dígitos para los dólares y dos dígitos para los centavos. Esto tiene mucho sentido en este caso, ya que nadie pagaría la millonésima parte de un centavo. La opción `digits` la estudiamos a detalle en la página 4

5.1.2 Formateando cuidadosamente la salida

En cualquier caso, este último código produce una salida algo difícil de leer. Alternativamente, podemos escribir

Código de Sage

```
1 for i in range(1, 30):
2     print(i, end='')
3     print(' implica ', end='')
4     print(N(15_000*(1+(i/100))^5, digits=7))
```

Un poco de información adicional puede ayudar aquí. El sangrado de las líneas sirve para indicar los comandos subordinados al bucle `for`, es decir, los que serán repetidos. La función `print` mostrará en pantalla lo que sea que se coloque dentro de sus paréntesis, pero la opción `end=''` indica no empezar una nueva línea. En general, esta opción sirve para especificar con qué texto terminar lo que la función `print` está imprimiendo. Para comprender mejor esta última afirmación, recomendamos que el lector intente las siguientes cuatro variaciones del código anterior por separado:

- añada `end=''` a la última función `print`;
- remueva todos los `end=''`;
- borre el `end=''` después de `' implica '`;
- reemplace `end=''` por `end=' [Hola, mundo] '` en la primera función `print`.

Hagamos nuestro código más poderoso ahora. Si queremos hacer el mismo cálculo, pero cada 1/4 de uno por ciento, podríamos escribir

Código de Sage

```
1 for i in range(1, 120):
2     print(N(i/400), end='')
3     print(' implica ', end='')
4     print(N(15_000*(1+(i/400))^5, digits=7))
```

lo que acarrea tres cambios. Primero, el $(i/100)$ en la última línea se convirtió en $(i/400)$, con el objetivo de hacer los pasos 1/400 (o un cuarto de uno por ciento) en lugar de 1/100 (o uno por ciento). Segundo, el bucle va hasta 119 en lugar de 29, así que nos detendremos en 29,75 %. Finalmente, hemos hecho que el primer comando `print` muestre la proporción real de porcentaje, en lugar del valor de i , así que cambiamos `print(i, end='')` por `print(N(i/400), end='')`. Dejar fuera el `N()` en esta línea produce un resultado muy poco elegante, pues Sage trata de reducir las fracciones que comprenden la razón del interés. El lector puede intentarlo para verlo por sí mismo.

Podemos hacer que nuestro ejemplo con la función $x^3 - x$ también produzca una salida de aspecto más profesional. Podemos escribir

Código de Sage

```
1 for x in range(-10, 11):
2     print('x=', end='')
3     print(x, end='')
4     print(' significa y=', end='')
5     print(x^3 - x)
```

En este punto, nuestro código empieza a hacerse un poco “alto”, en el sentido que estamos usando un gran número de líneas, pero sin lograr mucho con ello. Resulta que podemos ahorrar un poco de espacio vertical al combinar las funciones `print`. Esto está permitido, siempre y cuando separemos los diferentes pedazos de texto usando comas. Por ejemplo, probemos con el siguiente código:

Código de Sage

```
1 for x in range(-10, 11):
2     print('x =', x, 'significa y =', x^3-x)
```

Bajo esta forma abreviada, la función `print` convertirá los argumentos entre sus paréntesis en cadenas de caracteres y las concatenará, añadiendo un espacio entre ellas como separación. Ya habíamos visto brevemente este comportamiento en la página 139.

5.1.3 Usando listas arbitrarias

En ocasiones, la lista de valores que necesitamos es algo arbitraria. Por ejemplo, si queremos evaluar $x^3 - x$ para los valores $-2, 5, 7/4, 3/10$ y $-28/11$ de x , entonces escribiríamos

Código de Sage

```
1 for x in [-2, 5, 7/4, 3/10, -28/11]:
2     print('x =', x, 'significa y =', x^3 - x)
```

Como podemos ver, simplemente hemos reemplazado `range(-10, 11)` con `[-2, 5, 7/4, 3/10, -28/11]`. Este es un ejemplo de cómo Sage usa la estructura de lista de Python. También hemos visto eso en numerosos lugares a lo largo de este libro.

5.1.4 Bucles con acumuladores

También podemos dotar a un bucle de “memoria”, almacenando información en una variable. El uso más común de esta técnica es encontrar la suma de las entradas en una lista, pero también se puede usar para encontrar un valor mínimo, o uno máximo, o el producto de las entradas. Un término realmente antiguo para una variable tal en un bucle `for`, desde los primeros días de la computación, es “acumulador”. El autor encuentra el término útil, pero para un científico computacional, sonaría anticuado.

Por ejemplo, consideremos el siguiente pedazo de código, que calculará la suma de los enteros de 0 a 1000, inclusive. El acumulador es llamado `total`.

Código de Sage

```
1 total = 0
2 for i in range(0, 1_001):
3     total = total + i
4     print(total)
```

Bucles derrochadores Sería un desperdicio, y muy poco inteligente, imprimir cada número del 0 al 1000 mientras los sumamos. El código para hacer esto es

Código de Sage

```
1 total = 0
2 for i in range(0, 1_001):
3     print(i)
4     total = total + i
5     print(total)
```

Nótese el “`print(i)`” entre las sentencias “`for`” y “`total = total + i`”. En este caso, la salida es demasiado larga para mostrarla aquí, con una lista de números enorme, desenvolviéndose hacia abajo en la pantalla por un largo tiempo. Es importante evitar tales prácticas (especialmente si se usan millones, o cantidades mayores, de números), pues importantes recursos computacionales, tales como el ancho de banda y el tiempo de cálculo, serían entonces malgastados. El derroche es particularmente exagerado en un sistema como Sage, pues estamos usando la internet para transferir todos estos valores intermedios desde el servidor de Sage a nuestros navegadores web. Cuando los estudiantes del autor se encuentran frustrados porque sus códigos corren extremadamente lentos, frecuentemente la causa es que tienen sentencias `print` superfluas que desperdician tiempo.

¿Notó el lector cómo el 0 fue incluido, pero el 1001 fue excluido en el anterior cálculo? Esto es similar a los ejemplos en la subsección 5.1.1 en la página 210, donde evaluamos $f(x) = x^3 - x$ sobre el dominio `range(-10, 11)`, siendo el -10 incluido, pero no así el 11. La razón por la cual Python trabaja de esta manera es para que `range(0, 10)` se ejecute exactamente 10 veces, `range(0, 100)` se ejecute exactamente 100 veces y `range(0, 1_000)` se ejecute 1000 veces. A propósito de esto, cada “pasada” o “ejecución” de un bucle `for` es llamada “una iteración”.

Calculando la suma anterior de forma directa Deberíamos indicar un último detalle desde el punto de vista de las matemáticas puras. Existe una fórmula para sumar todos los números en una progresión aritmética. Una progresión aritmética es una lista de números donde la diferencia entre dos números consecutivos, dondequiera en la lista, es siempre una constante fija. Esta es llamada “la diferencia común”. Por ejemplo, consideremos

$$3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, \dots,$$

o alternativamente,

$$72, 69, 66, 63, 60, 57, 54, 51, 48, 45, \dots,$$

así como

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, \dots$$

Como podemos ver, estas tienen una diferencia común de +2, -3 y +1, respectivamente.

La suma de una progresión aritmética está dada por

$$S = \frac{(a + z)(n)}{2},$$

donde a es el primer miembro en la progresión, z es el último, n es el número de miembros y S es la suma final. Podemos calcular nuestro objetivo, más directamente que con el bucle anterior, con

$$S = \frac{(1 + 1000)(1000)}{2} = (1001)(500) = 500\,500,$$

obteniendo la misma respuesta, pero con mucho menos esfuerzo computacional.

5.1.5 Usando Sage para encontrar un límite numéricamente

Digamos que queremos investigar lo que hace la función

$$f(x) = e^{1/(8-x)}$$

alrededor del punto $x = 8$. Entonces podríamos escribir el siguiente código:

Código de Sage

```
1 for i in range(0, 9):
2     xtemp = 8 - 10^(-i)
3     print(N(e^(1 / (8 - xtemp))))
```

Este examina los valores de $f(x)$ en los puntos

$$7; 7.9; 7.99; 7.999; 7.9999; 7.99999; 7.999999; 7.9999999; 7.99999999.$$

Estos números también tienen los nombres

$$8 - 10^0, 8 - 10^{-1}, 8 - 10^{-2}, 8 - 10^{-3}, 8 - 10^{-4}, 8 - 10^{-5}, 8 - 10^{-6}, 8 - 10^{-7}, 8 - 10^{-8}.$$

Evaluando $f(x)$ en estos puntos, obtenemos la salida

```
2.71828182845905
22026.4657948067
2.68811714181614e43
1.97007111401705e434
8.80681822566292e4342
2.80666336042612e43429
3.03321539680209e434294
6.59223253461844e4342944
1.54997674664843e43429448
```

lo que claramente va hacia el infinito. (¡Si 1.549 976 746 648 43e43 429 448 no es lo “suficientemente cerca” a infinito, el autor no imagina qué puede serlo!)

Entonces, cambiando $8 - 10^{-i}$ por $8 + 10^{-i}$ tenemos el código

Código de Sage

```
1 for i in range(0, 9):
2     xtemp = 8 + 10^(-i)
3     print(N(e^(1 / (8 - xtemp))))
```

Este examina los valores de $f(x)$ en los puntos

9; 8,1; 8,01; 8,001; 8,0001; 8,00001; 8,000001; 8,0000001; 8,00000001.

Estos números también tienen los nombres

$8 + 10^0$, $8 + 10^{-1}$, $8 + 10^{-2}$, $8 + 10^{-3}$, $8 + 10^{-4}$, $8 + 10^{-5}$, $8 + 10^{-6}$, $8 + 10^{-7}$, $8 + 10^{-8}$

Evaluando $f(x)$ en estos puntos obtenemos la siguiente salida

```
0.367879441171442
0.0000453999297624849
3.72007597602084e-44
5.07595889754946e-435
1.13548386531474e-4343
3.56294956530937e-43430
3.29683147808856e-434295
1.51693678089873e-4342945
6.45170969282177e-43429449
```

lo que claramente va hacia cero. (¡Si $6.451\,709\,692\,821\,77e-43\,429\,449$ no es lo “suficientemente cerca” a cero, el autor no imagina qué puede serlo!)

Por lo tanto, podemos concluir finalmente que

$$\lim_{x \rightarrow 8^-} f(x) = \infty \quad \text{y también que} \quad \lim_{x \rightarrow 8^+} f(x) = 0,$$

las cuales son dos respuestas interesantes por derecho propio. La conclusión final, sin embargo, debe ser que

$$\lim_{x \rightarrow 8} f(x) \text{ no existe,}$$

pues sabemos que un límite (en general) no existe, si el límite por izquierda y el límite por derecha no son iguales entre sí.

Advertencia La exploración de ese límite requirió un poco de trabajo, pero no mucho, y respondió una pregunta bastante difícil. Así, podemos ver que la técnica de *evaluación numérica de límites* tiene un lugar en las matemáticas. Sin embargo, debemos ser cuidadosos —¡los límites numéricos son peligrosamente poco confiables!—.

Por ejemplo, la suma de los recíprocos de los primeros n números primos tiende a infinito conforme n tiende a infinito. Para ser precisos, lo que esto significa es que la suma

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \frac{1}{13} + \frac{1}{17} + \frac{1}{19} + \frac{1}{23} + \frac{1}{29} + \cdots$$

tiende a infinito. (Este teorema tiene una elegante pero difícil demostración, y no sería agradable pasar por ella en este momento.) Sin embargo, nunca podríamos detectar la divergencia de esta serie en una computadora. En efecto, conforme n crece, esa suma crece como $\log(\log(n))$. (Estamos simplificando esta discusión ligeramente al remover unos cuantos detalles.) En particular, si nuestro estándar de “suficientemente cerca” a infinito fuera solo 1000, tendríamos que esperar hasta que e^{1000} cálculos fueran totalizados. El concepto de e^{1000} es tan grande que el autor ni siquiera sabe cómo empezar a describirlo. Incluso si nuestro estándar de “suficientemente cerca” a infinito fuera nada más que 10, tendríamos que esperar hasta que e^{10} cálculos fuesen realizados, ¡y nótese que e^{10} es un número con 9566 dígitos!

No es necesario decir que si fuéramos a tratar de calcular esto con un bucle `for`, mucho antes que el cálculo terminase, el sol ya se habría convertido en una gigante roja y todos habríamos ardido en el fuego nuclear de nuestra estrella.

5.1.6 Bucles for y Polinomios de Taylor

Puede que el lector sepa que a veces, en cálculo y en cursos de nivel superior, debemos tomar muchas derivadas de una función —especialmente cuando calculamos un Polinomio de Taylor—. Algunos profesores de cálculo enseñan los Polinomios de Taylor temprano en el curso, y otros lo hacen muy tarde; algunos incluso saltan el tema. Si el lector no ha sido expuesto a los Polinomios de Taylor antes, a veces llamados “Series de Taylor”, entonces puede sentirse libre de saltar a la siguiente sección en la página 216. La presente discusión es una excursión no relacionada al resto del capítulo.

Consideremos el siguiente trozo de código:

Código de Sage

```
1 for i in range(0, 10):
2     print(i, '-ésima derivada:', diff(x^3-x, x, i))
```

Como podemos ver, esto calcula las primeras 9 derivadas de $x^3 - x$, la mayoría de las cuales son nulas. También observemos que la derivada cero es la función misma. Debe notarse que Sage puede calcular la Serie de Taylor por nosotros directamente —sin necesidad que determinemos todas esas derivadas— con una sola línea de código. Esto fue explicado en la sección 4.17 en la página 166. Este ejemplo es solo por el propósito de la discusión.

Otra función interesante es la Transformada de Lorentz de la Relatividad Especial:

$$L(v) = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}},$$

donde v es la velocidad de algún objeto y c es la velocidad de la luz. Esta fórmula describe cómo los objetos que se mueven muy rápido se contraen en la dirección del movimiento, se hacen más masivos y el tiempo se ralentiza (o dilata) para ellos.

Para aplicar la herramienta previa a esta fórmula, procedemos como sigue:

Código de Sage

```
1 f(v,c) = 1 / sqrt(1 - (v^2 / c^2))
2 for i in range(0, 10):
3     print(i, '-ésima derivada:', diff(f, v, i))
```

lo que en particular nos dice que la novena derivada es

$$(v, c) \mapsto 893025*v/(c^{10}*(-v^2/c^2 + 1)^{(11/2)}) + 13097700*v^3/(c^{12}*(-v^2/c^2 + 1)^{(13/2)}) + 51081030*v^5/(c^{14}*(-v^2/c^2 + 1)^{(15/2)}) + 72972900*v^7/(c^{16}*(-v^2/c^2 + 1)^{(17/2)}) + 34459425*v^9/(c^{18}*(-v^2/c^2 + 1)^{(19/2)})$$

un cálculo que el autor no querría hacer a mano.

Eliminando espacios indeseados de las funciones print Consideremos la salida del pedazo de código anterior que calcula las derivadas de $x^3 - x$. Por ejemplo, las tres primeras líneas mostradas en pantalla son:

```
0 -ésima derivada: x^3 - x
1 -ésima derivada: 3*x^2 - 1
2 -ésima derivada: 6*x
```

Aunque matemáticamente correcta, esta salida luce algo extraña debido que tenemos un espacio extra entre el valor de i y el sufijo -ésima. Es una molestia menor en el peor de los casos, pero no es algo con lo que tengamos que vivir.

Recordemos que la función `print` concatena los argumentos entre sus paréntesis, usando espacios como separadores entre ellos. Este comportamiento puede ser cambiado con la opción `sep`, que permite especificar un separador diferente para la concatenación. Por ejemplo, podemos reescribir el código anterior como

Código de Sage

```
1 for i in range(0,10):
2     print(i, '-ésima derivada: ', diff(x^3-x, x, i), sep='')
```


donde hemos escrito `sep=' '` para que `print` no añada nada entre las diferentes partes del texto. Entonces no tendremos el espacio entre el valor de `i` y el sufijo “-ésima” —tendremos una salida como “3-ésima” en lugar de “3 -ésima”—. Sin embargo, esto también evitará que aparezca el espacio después de los dos puntos. Dado que en efecto queremos ese espacio, ahora debemos añadirlo manualmente al final del segundo pedazo de texto.

Para comprender esto mejor la opción `sep`, recomendamos que el lector intente las siguientes tres variaciones del código anterior por separado:

- remueva la opción `sep=' '`;
- especifique un espacio como separador reemplazando `sep=' '` con `sep=' '`;
- use la opción `sep='-->'`.

5.1.7 Si el lector ya sabe programar...

Si el lector ya sabe programar, entonces reconocerá en este punto que Sage opera como un lenguaje de programación. De hecho, Sage se ejecuta sobre el lenguaje de programación Python.

- Si sabe programar Python, entonces puede usar cualquiera de los comandos de ese lenguaje directamente en Sage, sin ningún problema.
- Si sabe programar, pero no en Python, entonces puede estar seguro que Python es un lenguaje extremadamente fácil de aprender. Cualquiera con conocimientos de C, C++, Pascal, Java o C# puede aprender Python con apenas escaso esfuerzo.¹
 - Si el lector es un programador experimentado, entonces puede aprender mucho sobre Python en el libro “Inmersión en Python 3”, disponible para descarga gratuita en <https://github.com/jmgaguilera/inmersionenpython3/releases>.
 - Si el lector es un programador intermedio, entonces puede simplemente leer este capítulo. Probablemente lo leerá mucho más rápido que un verdadero principiante. En el curso de esa lectura, aprenderá mucho de Python.

5.1.8 Si el lector aún no sabe programar...

Si el lector no sabe programar en ningún lenguaje, no debe preocuparse de eso en este momento. Definitivamente no es necesario tener esa habilidad para usar Sage. Casi ninguna de las tareas usuales requiere programación, y las demás requieren muy poco —solamente los rudimentos que aprenderemos en las siguientes pocas páginas—. Este capítulo ha sido escrito pensando en el principiante, y será bastante funcional como programador para el final de este (suponiendo que haga todos los ejercicios).

5.2 Escribiendo subrutinas

Los programas de computadora frecuentemente están divididos en pequeños pedazos que llevan a cabo tareas separadas. Para frustración de los estudiantes, el vocabulario difiere de lenguaje a lenguaje. A veces estos pedazos de código son llamados “métodos” o “procedimientos”, pero usualmente se los llama “funciones”. Algunos lenguajes de programación muy antiguos usan el término “subprogramas”. En Python, definitivamente preferimos la palabra “función”.

Esto nos presenta un obstáculo verbal importante en computación matemática. En matemáticas, la palabra “función” significa algo como $f(x) = x^2 + 7x - 12$. ¿Cómo podemos entonces usar este mismo término para indicar simultáneamente $f(x)$ y un pequeño pedazo de código? En consecuencia, a lo largo de este capítulo usaremos el anticuado término “subrutina” para indicar un fragmento de código que realiza alguna tarea, mientras que usaremos la palabra “función” para indicar algo como $f(x) = x^2 + 7x - 12$. Este uso del término “subrutina” es algo no estándar y la mayoría de los expertos en Python estarán sorprendidos al ver que lo usamos aquí. Sin embargo, el autor considera que este capítulo sería ilegible si se usara la palabra “función” para describir pedazos de código y funciones matemáticas a la vez.

Ya sea que los llamemos métodos, procedimientos, subprogramas, funciones o subrutinas, la programación en general (y la programación matemática en particular) consiste en diseñar estos bloques elementales y entonces combinarlos en algún sistema de software más grande.

¹Martin Albrecht, un amigo del autor y un desarrollador mayor de Sage, una vez se refirió a Python como “pseudocódigo ejecutable”.

5.2.1 Un ejemplo: trabajando con monedas

Vamos a empezar con una subrutina muy sencilla, la cual sería parte del programa de una máquina expendedora o una caja registradora. Esta nos responderá con el valor en dólares para cierto número de monedas de cuartos, décimos, vigésimos y céntimos de dólar.² Como clarificación, estas son monedas que tienen los valores \$ 0,25, \$ 0,10, \$ 0,05 y \$ 0,01. Esta es una tarea simple, así que podemos enfocarnos en Python y no en la matemática subyacente. Este es el código que escribiríamos para esta tarea:

Código de Sage

```

1  def contMonedas(cuartos, decimos, vigesimos, centimos):
2      """Esta subrutina recibe datos acerca de una pila de cambio en
3      centavos, con los cuatro parámetros siendo el número de monedas de
4      25, 10, 5 y 1 centavos. Entonces el valor total en dólares de estas
5      es reportado. Se asume que las cantidades de entrada son números
6      naturales. Por favor, no use números complejos como entradas, pues
7      espantará al usuario enterarse que su dinero es imaginario."""
8
9      total = 0.25*cuartos + 0.10*decimos + 0.05*vigesimos + 0.01*centimos
10
11     print('Esto hace un total de $us.', total)

```

Estudiemos la primera línea de este código. El comando `def` indica a Python y Sage que estamos definiendo una nueva subrutina. Seguimos inmediatamente con el nombre de esta (“contMonedas”) y la lista de las variables que requiere como entradas. En este caso particular, estas son cuartos, decimos, vigesimos y centimos. Mientras que en álgebra las variables tienden a ser una sola letra, en programación usamos palabras para no tener que recordar lo que cada variable representa. Los dos puntos al final de la línea son importantes: estos indican que los comandos siguientes están subordinados a la sentencia `def`; el sangrado de las líneas muestra específicamente cuáles están subordinados.

Después de esto, empezando en la segunda línea, colocamos un pequeño texto de documentación, donde describimos lo mejor posible nuestra subrutina.³ Este texto no será visible al momento de ejecución; su propósito es servir de guía para cualquiera que desee reusar o modificar nuestra subrutina. Esto nos incluye a nosotros mismos. Supongamos que mucho tiempo después de programar esto se nos pide hacer lo mismo, pero esta vez con una moneda diferente al dólar y otros tipos de cambio. Fácilmente podemos reusar este código, haciendo las modificaciones necesarias, pues el texto de documentación describe qué significa cada argumento y cómo se calcula la salida.

Como habíamos indicado en la página 15, en Python —y por tanto, en Sage—, no existe ninguna regla que exija el uso de comillas dobles o simples para delimitar cadenas de caracteres. En particular, es permitido por el lenguaje el uso de triples comillas simples (‘ ’ ‘) para delimitar el texto de documentación. Sin embargo, el estándar de buena programación recomienda el uso de comillas triples (""") como arriba. En este libro, seguiremos esta recomendación.

El usuario final de nuestra subrutina cuenta con varios métodos para leer la documentación. Uno de ellos ya lo estudiamos en la sección 1.10 en la página 44: podría escribir `contMonedas?` o `contMonedas??`. Un solo signo de interrogación mostrará solamente el texto de documentación, mientras que el doble signo incluirá adicionalmente el código completo de nuestra subrutina, tal como se ve arriba. Existen otros mecanismos definidos por Python para acceder al texto de documentación, pero no los cubriremos en este libro.

Aunque documentar una subrutina de esta manera no es algo obligatorio, es altamente recomendable y se considera una buena práctica de programación. *El código no documentado es prácticamente inútil.*

Continuando con nuestra subrutina, en la línea número nueve tenemos una fórmula que calcula el valor total de la pila de monedas. Cada moneda es multiplicada por su valor en dólares y el total es puesto en una variable que, como era de esperarse, hemos llamado “total”. La última línea imprime el monto resultante en una forma elegantemente formateada y muy legible, como aprendimos a hacer en la página 211.

Podemos probar nuestra subrutina con 5 cuartos, 4 décimos, 3 vigésimos y 2 céntimos. Escribimos

²**Nota del traductor:** En inglés, estas monedas reciben los nombres *quarters*, *dimes*, *nickels* y *pennies*. Dado que necesitamos palabras en español que describan estos valores para los argumentos de la subrutina de esta subsección, he decidido usar el nombre de sus valores con respecto a la unidad de un dólar.

³En Python (y por tanto en Sage), las triples comillas delimitan texto que se extiende por varias líneas, como en este caso. Tienen muchos usos, pero el más importante es la documentación.

Código de Sage

```
1 contMonedas(5, 4, 3, 2)
```

y recibimos la respuesta correcta:

```
Esto hace un total de $us. 1.820000000000000
```

Ahora, es importante recordar que el orden de los argumentos importa. Considérense las siguientes tres llamadas a esta subrutina:

Código de Sage

```
1 contMonedas(1, 3, 2, 1)
2 contMonedas(1, 1, 2, 3)
3 contMonedas(1, 2, 3, 1)
```

Como podemos ver, las tres llamadas preguntan sobre diferentes pilas de cambio, y estas tendrán diferentes valores. Uno imaginaria que alguien usando esta subrutina en tres días diferentes, que ha olvidado el orden de los argumentos, podría intentar adivinar e introducir esas tres líneas de arriba. La salida producida, que se muestra abajo, ilustra el punto: obtenemos una respuesta incorrecta si ponemos los parámetros en el orden incorrecto.

```
Esto hace un total de $us. 0.6600000000000000
Esto hace un total de $us. 0.4800000000000000
Esto hace un total de $us. 0.6100000000000000
```

A diferencia de la mayoría de los lenguajes de programación, Python tiene una solución para aquellos que no pueden recordar bien el orden de los argumentos. Usando la siguiente notación, etiquetamos cada valor con la variable a la que deseamos que sea asignada. Obtendremos la misma respuesta en cada ocasión.

Código de Sage

```
1 contMonedas(decimos=3, vigesimos=2, centimos=1, cuartos=1)
2 contMonedas(cuartos=1, centimos=1, vigesimos=2, decimos=3)
3 contMonedas(centimos=1, vigesimos=2, decimos=3, cuartos=1)
```

Sage y Python responderán a las líneas anteriores de la siguiente manera, indicando que entendieron lo que esperábamos expresar en cada ocasión.

```
Esto hace un total de $us. 0.6600000000000000
Esto hace un total de $us. 0.6600000000000000
Esto hace un total de $us. 0.6600000000000000
```

Este es un muy buen servicio para aquellos cerebros que tienen problemas con los órdenes, ya sea que tengan dislexia o solo estén temporalmente distraídos. Sin embargo, esto introduce otro problema: debemos recordar cómo escribir el nombre de cada variable. Como puede apreciarse abajo, hemos quitado la “r” de “cuartos”, escribiendo en su lugar “cuatos”:

Código de Sage

```
1 contMonedas(decimos=3, vigesimos=2, centimos=1, cuatos=1)
```

Dado que la computadora no ha visto esta palabra antes, no sabe cómo interpretarla y por lo tanto provee el siguiente mensaje de error:

```
TypeError: contMonedas() got an unexpected keyword argument 'cuatos'
```

Una última cosa antes de continuar. En CoCalc, si escribimos `cont` en una línea propia, por debajo del código que define `contMonedas`, y presionamos TAB sin presionar ENTER, ¡podremos ver que una de las opciones presentadas ante nosotros es nuestra subrutina recientemente creada! CoCalc la considera una parte legítima del mundo en el que estamos operando en ese momento.

5.2.2 Desafío: Una caja registradora

Una tarea interesante sería simular una caja registradora. Supongamos que un pequeño puesto de comidas en una estación del subterráneo vende sodas (a \$ 1,50 cada una), bananas (a \$ 0,75 cada una), sandwiches de varios tipos (cualquiera a \$ 3,75) y botellas de agua (a \$ 2 cada una).

El lector debe escribir una subrutina que tome cinco entradas: las cantidades de sodas, bananas, sandwiches y botellas de agua, así como el monto de dinero recibido (por ejemplo, \$ 10 o \$ 20). El programa debe decir al usuario cuánto cambio, en dólares y centavos, se debe al cliente.

5.2.3 Un ejemplo: Diseñando acuarios

Ahora escribiremos una subrutina que es adecuada para una compañía que construye acuarios transparentes a pedido, ya sea para uso en oficinas de dentistas o cualquier otra oficina de lujo. Los clientes traen órdenes y el precio debe ser correctamente calculado, basado en los tamaños de los seis paneles que componen el acuario rectangular. Resulta que los materiales particulares que esta compañía está usando tienen los siguientes costos:

- el fondo del acuario es de metal, que cuesta 1,3 centavos por pulgada cuadrada;
- la tapa del acuario es de plástico, que cuesta medio centavo por pulgada cuadrada;
- los lados del acuario son de vidrio, que cuesta 5 centavos por pulgada cuadrada.

Dada la preferencia de un cliente para el volumen del acuario, existen muchas opciones para las dimensiones y cada una de ellas tendrá un costo sustancialmente diferente. Consideremos un cliente que quiere un acuario de 12 000 pulgadas cúbicas. (Nótese que este volumen es cerca de 52 galones o 197 litros, así que no es un acuario muy grande.) Podemos calcular los siguientes precios:

- Una elección de $20 \times 30 \times 20$ tiene un costo de \$ 110,80.
- Una elección de $40 \times 30 \times 10$ tiene un costo de \$ 91,60.
- Una elección de $60 \times 20 \times 10$ tiene un costo de \$ 101,60.
- Una elección de $80 \times 15 \times 10$ tiene un costo de \$ 116,60.

Estos fueron calculados usando la subrutina `analizarDisenoAcuario`, que se puede ver en la figura 5.1.

Código de Sage

```

1  COSTO_FONDO = 1.3
2  COSTO_TAPA = 0.5
3  COSTO_LADO = 5
4
5  def analizarDisenoAcuario(largo, ancho, altura):
6      """Aquí, las dimensiones de un acuario son las entradas. Las áreas
7      de los seis paneles que forman el acuario rectangular serán
8      mostradas en pantalla, junto con sus costos. Estos últimos son
9      calculados en términos de las áreas de los paneles y las variables
10     globales COSTO_TAPA, COSTO_FONDO y COSTO_LADO. Finalmente, el
11     volumen y el costo total son mostrados."""
12
13     area_tapa_fondo = largo * ancho
14     area_adelante_atras = largo * altura
15     area_izquierda_derecha = ancho * altura
16
17     print('Los paneles izquierdo/derecho tienen un área de',
18           ↵ area_izquierda_derecha, end=' ')
19     print('con un costo de $', COSTO_LADO*area_izquierda_derecha*0.01)
20     print()
21     print('Los paneles delantero/trasero tienen un área de', area_adelante_atras,
22           ↵ end=' ')
23     print('con un costo de $', COSTO_LADO*area_adelante_atras*0.01)

```

```

22     print()
23     print('El panel superior tiene un área de', area_tapa_fondo, end=' ')
24     print('con un costo de $', COSTO_TAPA*area_tapa_fondo*0.01)
25     print()
26     print('El panel del fondo tiene un área de', area_tapa_fondo, end=' ')
27     print('con un costo de $', COSTO_FONDO*area_tapa_fondo*0.01)
28     print()
29
30     print('El volumen total es', N(largo * ancho * altura))
31
32     costo = COSTO_TAPA * area_tapa_fondo + COSTO_FONDO * area_tapa_fondo + 2 *
33     ↪ COSTO_LADO * area_adelante_atras + 2 * COSTO_LADO * area_izquierda_derecha
34
35     print('El costo total es de $', 0.01 * costo)

```

Figura 5.1 Código para calcular el costo de los materiales para un acuario

Los siguientes comandos fueron usados, uno a la vez, para determinar los precios anteriores:

Código de Sage

```

1  analizarDisenoAcuario(20, 30, 20)
2  analizarDisenoAcuario(40, 30, 10)
3  analizarDisenoAcuario(60, 20, 10)
4  analizarDisenoAcuario(80, 15, 10)

```

Como podemos apreciar, este código tiene algunas características nuevas que probablemente el lector no haya visto antes. Las variables `COSTO_FONDO`, `COSTO_TAPA` y `COSTO_LADO` son usadas para almacenar los costos por pulgada cuadrada de cada parte del acuario. Si en vez de ello hubiésemos usado el número 0,5 en cada lugar donde la variable `COSTO_TAPA` aparece, entonces el código se habría vuelto menos legible. También, el precio podría cambiar. Si este se volviera 0,6 en lugar de 0,5, entonces podría ocurrir que hagamos el cambio en solo unos cuantos lugares, fallando en cambiarlos todos. Entonces el cálculo quedaría arruinado. Sin embargo, al usar una variable para almacenar el valor al principio del programa, solo necesitamos cambiarlo en un lugar. Hacer que el código sea fácil de actualizar y mantener es también extremadamente importante en los mundos de la industria y la academia.

Vimos el uso de variables como estas antes (véase la sección 1.6 en la página 31); sin embargo, ahora introducimos algo de terminología adicional. Cuando una variable es creada y usada exclusivamente dentro de una subrutina, entonces es llamada una *variable local*. Por otro lado, esas tres variables que mencionamos fueron creadas fuera de una subrutina, y por lo tanto son visibles a todas las subrutinas; estas son llamadas *variables globales*. Aunque, en muchos tipos de programación, se recomienda evitar el uso de variables globales, en computación científica y matemática, parecen ser más comunes. Es considerado buen estilo de programación colocar todas las variables globales al principio del programa, de manera que puedan ser fácilmente localizadas —aunque esto no es estrictamente necesario—. Python y Sage no objetarán si se rompe esta regla de etiqueta de programación.

Por otro lado, notemos que los valores de las variables `COSTO_FONDO`, `COSTO_TAPA` y `COSTO_LADO` no cambian durante la ejecución de nuestra subrutina, ni tampoco entre una ejecución y otra. Este tipo de variables recibe el nombre (aparentemente autocontradictorio) de *variables constantes*. Entre los programadores de Python, se considera una buena práctica que a estas se les asocie un nombre completamente en mayúsculas, separando las palabras que lo componen con guiones bajos, como hemos hecho en este caso —aunque esto tampoco es estrictamente necesario—.

Cualquier otra línea del código anterior es una función `print` o una ecuación matemática. Frecuentemente, en computación científica, tareas complejas pueden ser llevadas a cabo con estas herramientas simples.

A propósito, esta subrutina, `analizarDisenoAcuario`, será la base para una página web interactiva usando Sage. Construir estas páginas web puede ser muy divertido y es el tema del capítulo 6. Exploraremos el uso de esta subrutina particular en la sección 6.5 en la página 270.

5.2.4 Desafío: Un silo cilíndrico

Este ejercicio es una pequeña modificación de un problema clásico de cálculo. Una empresa de almacenamiento de suministros de granja está fabricando silos baratos. Cualquier silo particular es un cilindro (digamos, de radio r y altura h) y es tapado con una semiesfera (con el mismo radio r). El problema clásico de cálculo tiene como objetivo producir un silo con cierto volumen específico, pero usando la cantidad (área) mínima de metal. Sin embargo, vamos a añadir algo de realismo y diremos que el metal del cilindro cuesta 25 centavos por pie cuadrado, dado que es fácil de fabricar, mientras que el metal semiesférico cuesta \$ 1,50 por pie cuadrado, pues es más difícil de fabricar. Esperamos minimizar costos, lo que es similar (pero no idéntico) a minimizar el área de la superficie.

El desafío consiste en escribir una subrutina que toma r y h como entradas. Debería calcular e imprimir

- el área de la superficie cilíndrica del silo;
- el área de la superficie semiesférica del silo;
- el costo de cada uno de estos y el costo total, y
- el volumen del silo.

Naturalmente, el lector querrá consultar la internet para conocer las fórmulas apropiadas para el volumen y el área del cilindro y la semiesfera. Frecuentemente, cuando el autor se encuentra ocupado en algún cálculo científico, tiene necesidad de consultar algunas fórmulas menores. Gracias a la internet, encontrar material hoy en día es más fácil de lo que era cuando el autor aprendió a programar en 1988. Por supuesto, podríamos poner las fórmulas aquí. Sin embargo, es un ejercicio más realista si el lector debe buscarlas en línea, por su cuenta. Es una mejor simulación de trabajo en la industria, donde recordaremos los conceptos y procedimientos matemáticos, pero habremos olvidado hace mucho los detalles menores.

5.2.5 Combinando bucles y subrutinas

En ocasiones, particularmente cuando aprendemos por primera vez sobre funciones, es interesante evaluar una función en los números naturales 0, 1, 2, 3, 4, 5, 6, ..., hasta algún número final. La siguiente subrutina nos permitirá hacer justamente eso. Podemos elegir cualquier función que queramos y también podemos elegir cuántos términos deseamos en la sucesión formada.

Código de Sage

```
1 def funcion_a_sucesion(f, cuantos):
2     """Esta subrutina toma una función como primera entrada y la evalúa
3     en los primeros "cuantos" números naturales. Por ejemplo, si cuantos
4     es 4, entonces f es evaluada en 0, 1, 2, 3."""
5     for k in range(0, cuantos):
6         print(f(k))
```

Definamos las siguientes funciones para testar nuestro código:

Código de Sage

```
7 a(x) = 3*x + 5
8 b(x) = 2 - x
9 c(x) = 7 + 10*x
10 d(x) = 8 * exp(-0.1*x)
11 f(x) = 5 * 3^x
12 g(x) = (1/8) * 2^x
13 h(x) = (0.125) * 2^x
```

Ahora podemos verificar la subrutina con la instrucción `funcion_a_sucesion(a, 20)` y ver qué ocurre. Similarmente, podemos usar `b(x)` o `c(x)`, reemplazando `a` con `b` o `c`, y así sucesivamente. El lector notará algunas diferencias interesantes de comportamiento entre `g(x)` y `h(x)`, las cuales son diferentes para Sage a pesar del hecho que para un matemático son la misma función. Inténtelo usted mismo —no arruinaremos la sorpresa—.

Un aspecto realmente elegante de Sage es que podemos escribir código como

Código de Sage

```
1 funcion_a_sucesion(3*x, 20)
```

y Sage deducirá lo que queremos decir. No requerimos definir una función separada como $j(x) = 3x$ para trabajar con $3x$.

5.2.6 Otro desafío: Sumando una sucesión

Volvamos por un momento a la subrutina `funcion_a_sucesion` anterior. El lector debe modificarla para que también pueda tabular la suma total de los números impresos en pantalla. Debe mostrar el total en una línea separada, después de imprimir la sucesión misma.

Luego, pruebe la subrutina usando la función $r(x) = 1/(x + 1,0)^4$ y vea qué ocurre. Un matemático diría que se está calculando la suma de los recíprocos de las cuartas potencias de los enteros positivos. ¿Cuál es el total que se obtiene? Para valores grandes de cuantos, ¿cómo se compara esto a la aproximación decimal del número $\pi^4/90$? Si el lector conoce acerca de la Función Zeta de Riemann⁴, denotada $\zeta(x)$, entonces reconocerá que simplemente hemos calculado $\zeta(4)$. La definición de esta función es

$$\zeta(x) = \sum_{j=1}^{\infty} \frac{1}{j^x},$$

donde j recorre por los enteros positivos y x puede ser un número real mayor que 1 o un número complejo cuya parte real sea mayor que 1.

5.3 Bucles y el Método de Newton

Esta es una sección larga, pero importante. Vamos a desarrollar un pedazo de código para un objetivo matemáticamente útil: encontrar las raíces de una función. Mientras avancemos, vamos a explorar varias formas de abordar esta tarea. Esencialmente, empezaremos con un trozo de código razonablemente bueno e iremos añadiéndole varias características, una a la vez, hasta que tengamos una excelente subrutina. Durante este proceso veremos muchas características de los lenguajes de programación, incluyendo la *construcción if-then-else*, los *bucles while*, *parámetros opcionales* y muchos otros.

5.3.1 ¿Qué es el Método de Newton?

Las siguientes secciones explicarán algunas técnicas de programación útiles y extremadamente comunes. Para proveer de una aplicación matemática genuina e interesante, examinaremos el Método de Newton⁵, un famoso algoritmo para encontrar raíces de funciones o, equivalentemente, soluciones numéricas a ecuaciones.

Esta introducción está escrita asumiendo que el lector no conoce aún dicho método. Si ya lo ha visto antes, como en la clase de cálculo, entonces tal vez quiera solo ojear esta subsección rápidamente o saltarla por completo y proceder a la subsección 5.3.2 en la página 224.

El concepto general aquí es que nos gustaría generar una subrutina que opere de manera muy parecida al comando `find_root` —y de paso, aprenderemos mucho de matemáticas así como de lo que `find_root` realmente hace tras bambalinas—.

Algunas ecuaciones pueden ser resueltas con técnicas algebraicas; otras pueden ser resueltas con trucos creativos —tal como con identidades trigonométricas—. Sin embargo, muchas ecuaciones simplemente no pueden resolverse de manera simbólica. Por ejemplo, supongamos que queremos encontrar una solución a

$$\sin(x) = 1 - x.$$

No existe una manera de resolver esta ecuación usando técnicas ordinarias del álgebra. Sin embargo, usando este procedimiento llamado Método de Newton, podemos obtener una serie de aproximaciones sucesivas. Estas son fáciles de calcular y tienden a volverse muy precisas en unos cuantos pasos.

⁴Nombrada en honor a Georg Friedrich Bernhard Riemann (1826–1866). Es el mismo Riemann de la Geometría Riemanniana (la curvatura del espacio), la Hipótesis de Riemann, y también hizo varias contribuciones al estudio de los números primos.

⁵Como podemos imaginar, este es nombrado en honor a Sir Isaac Newton (1642–1727).

Primero, observemos que una solución a la ecuación anterior sería también una raíz de la siguiente función:

$$f(x) = 1 - x - \sin(x)$$

Empezamos entonces con la siguiente fórmula famosa:

$$x_{sig} = x_{ant} - \frac{f(x_{ant})}{f'(x_{ant})},$$

que es el corazón del Método de Newton. Usaremos $x = 0$ como nuestra conjetura inicial para la raíz. Además, tenemos

$$f(x) = 1 - x - \sin(x) \quad \text{y por lo tanto} \quad f'(x) = -1 - \cos(x),$$

que podemos usar como sigue:

- Con $x = 0$,
 - tenemos $f(0) = 1$ y $f'(0) = -2$.
 - Esto significa que $f/f' = -1/2$.
 - Así, el siguiente valor de x es $0 - (-1/2) = 1/2$.
- Con $x = 1/2$,
 - tenemos $f(1/2) = 0,0205744 \dots$ y $f'(1/2) = -1,87758 \dots$,
 - así que $f/f' = -0,0109579 \dots$
 - El siguiente valor de x es $0,5 - (-0,0109579 \dots) = 0,510957 \dots$
- Con $x = 0,510957$,
 - $f(0,510957) = 0,00000307602 \dots$,
 - mientras que $f'(0,510957) = -1,87227 \dots$.
 - En consecuencia, $f/f' = -0,0000164293 \dots$,
 - lo que significa que el siguiente valor de x será

$$0,510957 - (-0,0000164293) = 0,510973 \dots$$

Nota: Si realizamos este procedimiento con una calculadora de mano, debemos asegurarnos que esté configurada para usar radianes.

Si fuéramos a discutir estos resultados en un correo electrónico o en una publicación matemática, diríamos que $x = 0$ es la primera aproximación, $x = 1/2$ es la segunda, $x = 0,510957$ es la tercera y $x = 0,510973$ es la cuarta.

Todo esto no requirió mucho esfuerzo, pero tampoco fue un cálculo extremadamente fácil, así que podríamos tener curiosidad de qué tan bien lo hicimos. Reemplacemos nuestra cuarta aproximación en ambos lados de la ecuación original y veamos qué tan cerca pudimos llegar a la solución:

- El valor de $\sin(0,510973) = 0,489\,026\,196 \dots$.
- El valor de $1 - 0,510973 = 0,489\,027 \dots$.

Logramos este fenomenal grado de precisión con solo tres usos de la fórmula de Newton. Cuando tengamos una computadora para realizar los cálculos, veremos que podemos obtener una precisión hasta la billonésima parte relativamente rápido.

En realidad, en el procedimiento anterior redondeamos a la sexta cifra significativa en varias etapas, así que probablemente la cuarta aproximación pudo haber sido incluso mejor, hecha en computadora. Usando el código que desarrollaremos en las siguientes páginas, hemos obtenido la quinta aproximación a la solución, que resulta ser

$$x = 0,510\,973\,429\,388\,569 \dots,$$

que logra igualar los lados izquierdo y derecho de la ecuación con 15 decimales —precisión de hasta la mil billonésima parte—. Usando ese código podremos resolver este problema o cualquier otro de una categoría muy amplia de problemas.

Resumen De este ejercicio podemos ver que el Método de Newton tiene dos propiedades muy importantes. Primero, una vez que $f(x)$, $f'(x)$ y la suposición inicial están dadas, entonces el método es muy simple de realizar. Segundo, el procedimiento no es demasiado excitante como para que un ser humano lo use, pues es repetitivo —después de todo, básicamente se reduce usar una fórmula varias veces—. Estas dos características indican que el Método de Newton es ideal para una computadora, y es más claro que este es el caso cuando pensamos que la computadora trabaja con más dígitos de precisión —más de los que estaríamos dispuestos a usar con una calculadora de mano—.

Método de Newton en forma tabular Una nota menor sobre esto es que, si uno está ejecutando este método con una calculadora de mano en lugar de una computadora, entonces puede encontrarlo mucho más fácil si construye una pequeña tabla para organizar los pasos intermedios. Para el ejemplo de arriba tendríamos:

x	$f(x)$	$f'(x)$	razón f/f'
0	1	-2	-1/2
1/2	0,0205744	-1,87758	-0,0109579
0,510957	0,0000307602	-1,87227	-0,0000164293
0,510973			

La práctica hace al maestro Ahora, si el lector desea estar seguro que puede manejar esta técnica computacional a la perfección, tal vez quiera intentar un ejemplo en papel y lápiz, ayudado por una calculadora de mano. Trate de resolver en este caso

$$e^x = 10x,$$

siendo la suposición inicial $x = 4$. Después de hacer esto, probablemente se sentirá ansioso de delegar esta tarea a una computadora. La cuarta aproximación, después de tres usos de la fórmula de Newton, debería ser alrededor de $3,57715206 \dots$, con alguna variación, dependiendo de la marca de la calculadora que se use y cómo esta maneja los redondeos internamente. En efecto, podemos verificar que

$$e^{3,57715206395730} = 35,771\,520\,639\,573\,1 \dots$$

con cualquier calculadora de mano.

5.3.2 El Método de Newton con un bucle for

En la figura 5.2 tenemos un buen ejemplo de un primer intento de implementación del Método de Newton en Sage. Ahora analizaremos este código línea por línea.

Código de Sage

```

1 def metodo_newton(f, x_ant):
2     """Esta subrutina usa el Método de Newton para encontrar una raíz de
3     f, usando una conjetura inicial x_ant. Se ejecutarán 10 iteraciones."""
4
5     f_prima(x) = diff(f, x)
6
7     for j in range(0, 10):
8         print('Iteración =', j)
9         print('x_ant =', x_ant)
10        print('f(x) =', f(x_ant))
11        print("f'(x) =", f_prima(x_ant))
12
13        razon = f(x_ant) / f_prima(x_ant)
14
15        print('razón =', razon)
16
17        x_sig = x_ant - razon
18
19        print('x_sig =', x_sig)
20        print()
21
22        x_ant = N(x_sig)

```

Figura 5.2 El Método de Newton en Sage, versión 1

La primera línea notifica a Sage que estamos a punto de definir una subrutina. Su nombre será “metodo_newton” y tendrá dos parámetros, llamados f y x_{ant} . Como vimos, es extremadamente importante que las siguientes líneas estén sangradas. El grado de sangrado muestra cuáles comandos están subordinados a cuáles otros.

Dado que el Método de Newton requiere conocer la derivada de la función cuya raíz estamos tratando de hallar, no nos sorprende ver la línea número cinco definiendo $f_{\text{prima}}(x)$. Nótese que no podemos usar un apóstrofo para representar “prima”, como en $f'(x)$, pues el apóstrofo ya tiene un significado en Python. Este punto menor fue primero mencionado en la página 47.

Ahora llegamos a un bucle que se ejecutará 10 veces, como está indicado por el comando `for`. Podemos apreciar que el bucle está más que todo formado por funciones `print`, con el objetivo de hacer saber al usuario lo que está ocurriendo. En este punto, la sintaxis del comando `print` es seguramente familiar para el lector, pero, si lo desea, puede consultar la subsección 5.1.2 en la página 211.

Las líneas 13, 17 y 22, que están dentro del bucle, son asignaciones. La primera define la “razón”, como vimos en la subsección previa. La segunda asignación define x_{sig} a partir de x_{ant} de acuerdo a las reglas del Método de Newton. Finalmente, tenemos la línea $x_{\text{ant}} = N(x_{\text{sig}})$, la cual debe resultar algo confusa. Básicamente, durante la ejecución del bucle `for` identificado por —digamos— $j = 3$, el valor de x_{ant} será el valor que x_{sig} tenía durante la ejecución del bucle `for` identificado por $j = 2$. Este comando provee el enlace entre una ejecución del bucle y la siguiente.

La única sorpresa aquí es el uso del comando `N()`, que previamente hemos usado para convertir una forma exacta (algebraica) de algún número —tal vez muy difícil de entender—, a una forma de punto flotante o decimal —que es más fácil de entender, pero en general es necesariamente solo una mera aproximación—. El uso de `N()` aquí es para forzar que Sage calcule aproximadamente y no de forma exacta. Esto puede resultar una sorpresa, pues una de las fortalezas de Sage es su habilidad de calcular exactamente. Sin embargo, veremos un ejemplo en la siguiente subsección que nos revelará el porqué de esta elección.

5.3.3 Probando el código

Los ejemplos que usaremos para probar nuestro código serán las siguientes tres funciones:

$$\begin{aligned}a(x) &= x^5 + x + 1, \\b(x) &= x^x - 5, \\c(x) &= x^3 - 2x + 1.\end{aligned}$$

Tomémonos un momento para asegurarnos que las hemos escrito correctamente en Sage.

Para $a(x)$ podríamos escoger una condición inicial de 0,5, y así escribimos

Código de Sage

```
1 metodo_newton(a, 0.5)
```

obteniendo eventualmente la respuesta de $x = -0,754\,87\dots$, que parece razonable. (Este es el valor final de x_{ant} .) Debemos notar que $f(x_{\text{ant}})$ tiene un valor final de $f(x_{\text{ant}}) = 8,326\,67\dots \times 10^{-17}$, lo que es realmente cerca de cero. Por lo tanto, podemos concluir que esta ejecución de nuestra subrutina es un éxito. Hemos calculado la raíz de un polinomio de grado cinco.

Sin embargo, aún hay espacio para mejoras, como puede observarse retrocediendo a los resultados de la iteración 5. Vemos que $f(x_{\text{ant}}) = -7,621\,68\dots \times 10^{-14}$ en ese punto, lo cual es “suficientemente cerca de cero”. Por lo tanto, hemos gastado tiempo computacional. Eso no es importante cuando estamos analizando las raíces de un solo polinomio, ya que puede haber una diferencia de un cuarto de segundo versus medio segundo. Sin embargo, si nuestro proyecto se convierte en una parte importante de un programa más grande (tal vez siendo ejecutado un millón de veces), entonces desperdiciar la mitad del tiempo computacional sería excepcionalmente insensato. Remediamos este desperdicio unas cuantas páginas más adelante.

Ahora podemos tratar con $b(x)$. Dado que $2^2 = 4$, $3^3 = 27$ y $4 < 5 < 27$, podemos anticipar que la solución a $x^x = 5$ debería encontrarse entre 2 y 3. Usando la condición inicial de 2 terminamos con $x_{\text{ant}} = 2,129\,37\dots$, que parece razonable. Podemos verificar la respuesta dada al pedir el valor de

Código de Sage

```
1 2.129_372_482_760_16 ^ 2.129_372_482_760_16
```

copiando y pegando (en lugar de transcribir). La respuesta de Sage es

5,000000000000003

que está extraordinariamente cerca de 5. Nuevamente vemos que nuestro código inicial es derrochador, pues obtenemos una excelente respuesta tan pronto como en la iteración 4.

Si hubiésemos usado $x = 3$ como la condición inicial, obtenemos el mismo valor final de x_{ant} . Coincide con nuestra anterior respuesta para los 15 decimales mostrados. Vale la pena detenerse un momento para ponderar la precisión implicada por un acuerdo de 15 cifras significativas. Pensemos que fuésemos a calcular el peso de un automóvil de aproximadamente una tonelada métrica. Quince cifras significativas significaría ser precisos al nanogramo más próximo —o una billonésima parte del peso de un sujetapapeles (un gramo)—. También vale la pena mencionar que no acertamos exactamente a 5. Métodos tales como el de Newton producen aproximaciones —frecuentemente excelentes aproximaciones—, pero no respuestas exactas.

Esta comparación entre el comportamiento de $b(x)$ con $x = 2$ y $x = 3$ tiene intención de mostrar que las suposiciones iniciales simplemente necesitan ser *razonables*, y no necesariamente muy buenas. Sin embargo, una suposición inicial *ilógica*, tal como $x = 10$, produce el fracaso. En particular, vemos que en este caso el valor final de $f(x_{ant}) = 4,824\,81 \dots \times 10^5$ no está para nada cerca de cero. Sin embargo, esta falla no debe resultar una sorpresa, pues estamos tratando de hallar valores de x tales que $x^x = 5$. De seguro que todos estaremos de acuerdo que 10^{10} no es aproximadamente 5.

Ahora que tenemos la suficiente confianza de que las cosas funcionan bien, podemos completar nuestra revisión con la función $c(x)$. Con una condición inicial de $x = -1$ vemos que $x_{ant} = -1,618\,03 \dots$ es la respuesta final provista, y $f(x_{ant})$ es mostrado como si fuese exactamente cero. Esto no es precisamente cierto, sino que la larga cadena de ceros indica que $f(x_{ant})$ está tan cerca de cero que la mejor aproximación de su verdadero valor, que puede ser representada (en forma de punto flotante) por Sage, es cero mismo. Si hubiésemos querido ser estrictos, podríamos copiar y pegar lo siguiente en Sage:

Código de Sage

```
1 c(-1.618_033_988_749_89)
```

que evalúa a $2,842\,17 \dots \times 10^{-14}$.

Con una condición inicial de $x = 0$, vemos que el valor final es $x_{ant} = 0,618\,03 \dots$. Esta vez, tenemos que $f(x_{ant}) = -5,551\,11 \dots \times 10^{-17}$. Esto tiene una excelente precisión y podemos estar contentos de que nuestro código funciona para estos ejemplos. Es interesante notar que, en este caso, las condiciones iniciales de $x = -1$ y $x = 0$ ambas convergen a buenas respuestas, pero que estas son diferentes. Esto ocurre cuando f tiene múltiples raíces. Dadas ciertas condiciones, el Método de Newton convergerá a una sola raíz —frecuentemente, pero no siempre, a la más próxima—.

Solo por diversión, podemos reconsiderar nuestro problema de apertura, de la subsección 5.3.1 en la página 222. Deseábamos resolver $\sin(x) = 1 - x$ con respecto a x .

Código de Sage

```
1 u(x) = 1 - x - sin(x)
2 metodo_newton(u, 0)
```

Podemos ver que en la quinta iteración tenemos una respuesta precisa a un factor 10^{-16} y, además, esta coincide con la solución que encontramos a mano. Similarmente,

Código de Sage

```
1 v(x) = e^x - 10*x
2 metodo_newton(v, 4)
```

es otro ejemplo que vimos previamente.

Una observación final es que con estas tres últimas funciones, nuevamente nuestros cálculos fueron derrochadores:

- Para $c(x)$, en la iteración 5 tenemos que f toma un valor de alrededor de una billonésima.
- Para $u(x)$, en la iteración 5 tenemos que f toma un valor de alrededor de $1/9$ de una mil billonésima.
- Para $v(x)$, en la iteración 5 tenemos que f toma el valor de cero.

Todas estas de seguro tienen una “precisión más que suficiente” para cualquier aplicación del mundo real, así que deberíamos buscar una forma de “detenernos antes”.

5.3.4 Representación numérica vs. exacta

Esta subsección explora nuestra anterior subrutina con un poco más de detalle para explicar por qué usamos la función $N()$ en esta nueva y extraña forma. Nos concentraremos en $a(x)$.

Intentemos cambiar $x_ant = N(x_sig)$ por $x_ant = x_sig$ y veamos qué ocurre. Si usamos una condición inicial de $x = 0,5$, no vemos ninguna diferencia; sin embargo, si cambiamos a $x = 1$, definitivamente vemos algo distinto. Vale la pena que el lector lo intente ahora —el resultado es sorprendente—.

Incluso en la iteración 5 vemos la respuesta

$$x_{ant} = \frac{-12\,188\,385\,459\,018\,332\,151\,653\,005\,986\,109\,291\,290\,903\,586}{16\,146\,170\,242\,253\,095\,711\,911\,994\,084\,139\,951\,355\,091\,803},$$

lo que es absurdo para nuestros propósitos —aunque correcto—. Las subsecuentes iteraciones muestran incluso más dígitos. Por supuesto, esto no es lo que teníamos en mente, por lo que deberíamos forzar que Sage haga los cálculos numéricamente. La forma de hacer esto es usar $N()$ en $x_old = N(x_new)$. Esto implica que, al empezar la segunda iteración, hemos garantizado que estamos trabajando numéricamente.

¿Por qué ocurrió este fenómeno de ultralta exactitud con la condición inicial $x = 1$, pero no con $x = 0,5$? La razón tiene que ver con cómo Sage interpreta los números. El número $0,5$ ya es un decimal, así que Sage supone que es una mera aproximación numérica. Sin embargo, el número $1/2$, a diferencia de $0,5$, es asumido exacto. Si usamos $1/2$ en lugar de $0,5$, con $x_old = x_new$ en lugar de $x_old = N(x_new)$, veremos una cantidad ridícula de dígitos para x_{ant} , incluso tan temprano como en la iteración 4.

Es divertido verificar que usando $b(x) = x^x - 5$ con cálculos exactos, se producen resultados incluso más catastróficos.

5.3.5 Trabajando con parámetros opcionales y mandatorios

Tal vez la primera cosa que deberíamos corregir de nuestra subrutina son los cálculos derrochadores. Nos referimos específicamente a las situaciones en las que hemos obtenido $f(x_{ant})$ con una precisión de una mil millonésima parte, pero el algoritmo continua porque especificamos 10 iteraciones. Además, habrá un beneficio secundario a este cambio que será revelado en unos cuantos párrafos.

La forma inteligente de proceder es tratar de tener algún tipo de condición matemática, mediante la cual la subrutina pueda determinar, por sí sola, que es momento de detenerse. Sin embargo, primero vamos a explorar una forma menos sofisticada de manejar esto, pues es mucho más fácil de explicar e implementar: vamos a dejar que el usuario de nuestro código especifique cuántas iteraciones deben ser realizadas. Por supuesto, eso significa que necesitamos un nuevo parámetro para la subrutina, que indicará cuántas iteraciones realizar.

Para lograr esto, realizaremos dos pequeños cambios:

- “`def metodo_newton(f, x_ant):`” se convierte en “`def metodo_newton(f, x_ant, max_iter):`” y
- “`for j in range(0, 10):`” se convierte en “`for j in range(0, max_iter):`”.

El primer cambio establece la tercera entrada o parámetro, que hemos llamado `max_iter`, pues es el máximo número de veces que el bucle `for` debería iterarse. El segundo cambio hace que el valor de `max_iter` tome el lugar del 10 en el ciclo `for` —en otras palabras, el bucle iterará un número `max_iter` de veces en lugar de 10 veces siempre—.

Sin embargo, deberíamos poder hacerlo incluso mejor. Analicemos por qué. Los cambios que hemos hecho arriba alteran nuestros casos de estudio:

- “`metodo_newton(b, 2)`” se debe escribir como “`metodo_newton(b, 2, 10)`” y
- “`metodo_newton(b, 10)`” se debe escribir como “`metodo_newton(b, 10, 10)`”.

Estos cambios no son difíciles y simplemente proveen un valor numérico para el parámetro `max_iter`. Sin embargo, esto no es óptimo por varias razones. Primero, un programador que ha estado usando nuestra antigua subrutina dentro de sus propios programas por un largo tiempo, ahora descubrirá que su código es incapaz de trabajar. Eso es porque las llamadas a nuestra subrutina en sus programas tendrían solamente dos parámetros, siendo que ahora se requieren tres. Como resultado, el código de ese programador no operará apropiadamente y esto provocaría su ira⁶ contra nosotros. Segundo, un nuevo programador, poco experimentado, no debería ser agobiado con tener que entender demasiado acerca del Método de Newton para poder usar nuestra subrutina. Queremos que una audiencia tan grande como sea posible sea capaz de

⁶Este problema, llamado “compatibilidad retrógrada”, es extremadamente serio en cualquier proyecto de software que involucre a más de un programador. Un estudiante que desee tener cualquier tipo de trabajo debe tomarse un momento para darse cuenta que no será el único o la única programando una computadora en su nueva compañía. Por lo tanto, deberá tener mucho cuidado de pensar sobre este asunto en profundidad. “Romper” el código de otras personas al modificar algo sin permiso es una forma extraordinariamente efectiva de ser despedido.

usar nuestro código. Sería mejor si fijamos un valor “estándar” para `max_iter` y entonces establecemos la capacidad de que otros programadores “sobrepongan” su propia elección a la nuestra. El mecanismo para lograr esto en Python es un *parámetro opcional*.

Para implementar un parámetro opcional, debemos cambiar el viejo código “`def metodo_newton(f, x_ant):`”, o el más nuevo “`def metodo_newton(f, x_ant, max_iter):`”, por

Código de Sage

```
1 def metodo_newton(f, x_ant, max_iter=10):
```

El efecto de este cambio es que si un valor de `max_iter` es dado, entonces ese valor será usado; sin embargo, si no se da ninguno, entonces el valor por defecto de 10 será usado en su lugar. Haga el lector estos cambios y entonces tómese un momento para probar cada una de las siguientes llamadas a nuestra subrutina, una a la vez:

Código de Sage

```
1 metodo_newton(b, 2)
2 metodo_newton(b, 2, max_iter=5)
3 metodo_newton(b, 2, 5)
4 metodo_newton(b, 10)
5 metodo_newton(b, 10, max_iter=5)
6 metodo_newton(b, 10, 5)
7 metodo_newton(b, 10, max_iter=100)
8 metodo_newton(b, 10, 100)
```

Vale la pena aclarar por qué algunas de las llamadas anteriores usan el nombre del argumento `max_iter`, mientras que otras no. Resulta que cuando no se usa el nombre de un parámetro en una llamada de una subrutina, Python —y por lo tanto, Sage— asume que ese valor corresponde al parámetro que ocupa la misma posición en la definición de la subrutina. Por ejemplo, en la línea 1 anterior, no tuvimos la necesidad de especificar que `b` y `2` corresponden con los argumentos `f` y `x_ant`, respectivamente; Sage ya lo sabía por la posición que ocupan. Esto también es válido para los argumentos opcionales. Por ejemplo, la segunda y tercera líneas anteriores son equivalentes, pues Sage asume que el número 5 corresponde con el parámetro `max_iter`, debido a que ambos ocupan la tercera posición en la definición de `metodo_newton`. De la misma manera, la cuarta y la quinta llamadas anteriores son equivalentes entre sí, así como la séptima y la octava lo son entre sí. En general, es obligatorio usar el nombre de un parámetro solamente cuando se lo usa en una posición diferente a la que corresponde en la definición de la subrutina. Por ejemplo, la siguiente instrucción es equivalente a la segunda llamada anterior:

Código de Sage

```
9 metodo_newton(max_iter=5, x_ant=2, f=b)
```

Como podemos ver en las líneas 7 y 8 anteriores, dada la terrible suposición inicial de $x = 10$ para la función $b(x) = x^x - 5$, el método de Newton sí convergió eventualmente. Aunque tenemos la extensa salida de 100 iteraciones en la pantalla, es interesante ver dónde el desempeño cambia de “malo” a “bueno”. En particular, vemos que la iteración 23 tiene $f(x_{ant}) = 0,31804\dots$, que realmente no es una buena aproximación a cero, mientras que la iteración 26 hace que $f(x_{ant})$ esté aproximadamente a 19 billonésimas de cero, y la iteración 27 tiene $f(x_{ant})$ a menos de una mil trillonésima de cero.

Cuando el autor aprendió a programar por primera vez a finales de los 1980s y principios de los 1990s, era común en una implementación de un algoritmo iterativo, como el Método de Newton, requerir que el usuario presione la barra espaciadora entre iteraciones, a manera de dar una probada de cómo evoluciona el algoritmo. Esa no es la forma en que el software es escrito en la era actual, pero a pesar de todo, nos presenta un buen experimento mental. Imaginémoslo teniendo que presionar la barra espaciadora para indicar “siguiente iteración” 23 veces, esperando resolver el problema $x^x = 5$ y obteniendo un valor de $f(x_{ant})$ alrededor de 0,318042. ¿Qué tan frustrados estaríamos? Probablemente mucho. La mayoría de nosotros se habría rendido mucho antes de 23 iteraciones, y algunos de nosotros estaríamos tentados de renunciar incluso en la quinta iteración. Sin embargo, una persona muy paciente presionaría el botón 3 o 4 veces más y obtendría una excelente respuesta en la 26^{ta} o 27^{ma} iteración. Podemos tener ahora la certeza que “cuándo detenerse” no es algo obvio y que algún grado de paciencia es requerido.

A propósito, vale la pena notar que los parámetros opcionales no son comunes entre los más importantes lenguajes de programación. Es una característica que Python pone a nuestra disposición, pero de la que muchos lenguajes carecen.

5.3.6 Devolviendo un valor y anidando subrutinas

La experiencia general de trabajar en computación científica o programación numérica consiste en escribir una serie de subrutinas, tal que cada una realiza una tarea importante, pero que en su mayoría logran su cometido llamando a otras subrutinas —a veces aquellas escritas por nosotros mismos, pero, más frecuentemente, subrutinas escritas por otros—. En esta subsección experimentaremos esta situación usando nuestro código anterior para calcular algo que realmente no podría calcularse sin algo semejante al Método de Newton o alguno de los métodos rivales.

Solo como ejemplo, echemos un ojo a las siguientes instrucciones:

Código de Sage

```
1 13 + sin(3*pi + 1/5)
2 log(169) / 2
3 16 - sqrt(9 - x^2)
```

Cada una de estas tiene sentido y es utilizable debido a que una subrutina, como `sin`, `log` o `sqrt`, devuelve un número que puede ser usado con otros números de acuerdo a las reglas de la aritmética.

Similarmente, nuestro `metodo_newton` también debe devolver un número. Solo tiene sentido que esta salida deba ser nuestra mejor aproximación, que es el valor de x_{ant} y x_{sig} al final de la ejecución de la subrutina. Nótese que al finalizar la ejecución, siempre tendremos que $x_{ant} = x_{sig}$, por lo que no es necesario agonizar con la decisión de cuál de ellos elegir. Para lograr la salida de un número, añadimos la línea

Código de Sage

```
26 return x_ant
```

al final de nuestra subrutina. Este comando de Python indica a Sage que use el valor final de x_{ant} como la salida de nuestra subrutina.

Este cambio, junto con los que hicimos en la subsección previa, resultan en el código de la figura 5.3. Aquellas líneas que han cambiado desde la versión 1 tienen la anotación `# cambió` o `# nueva` junto a ellas. El símbolo `#` es muy importante. Cualquier cosa que lo siga en la misma línea de código de Python es completamente ignorada. Esto significa que lo podemos usar para pequeñas anotaciones como `"# cambió"` o `"# nueva"`. También podemos usar `#` para comentarios más sustanciosos, tales como un resumen de una sola línea, indicando al lector de nuestro programa qué es lo que las siguientes pocas líneas de código pretenden hacer. Frecuentemente, si una fórmula proviene de algún artículo o capítulo de un texto, es una buena idea anotar ese hecho en esta categoría de comentario “rápido”.

Código de Sage

```
1 def metodo_newton(f, x_ant, max_iter=10):      # cambió
2     """Una implementación del Método de Newton para encontrar una raíz
3     de f(x), dada una suposición inicial para x. Por defecto, 10
4     iteraciones serán realizadas, a menos que el parámetro opcional
5     max_iter sea ajustado a otro valor."""    # cambió
6
7     f_prima(x) = diff(f, x)
8
9     for j in range(0, max_iter):              # cambió
10         print('Iteración =', j)
11         print('x_ant =', x_ant)
12         print('f(x) =', f(x_ant))
13         print("f'(x)=", f_prima(x_ant))
14
15         razon = f(x_ant) / f_prima(x_ant)
16
17         print('razón =', razon)
18
19         x_sig = x_ant - razon
20
```

```

21     print('x_sig =', x_sig)
22     print()
23
24     x_ant = N(x_sig)
25
26     return x_ant      # nueva

```

Figura 5.3 El Método de Newton en Sage, versión 2

Al principio de esta sección prometimos que usaríamos `metodo_newton` para completar una tarea computacional legítima. La que hemos escogido es calcular el autologaritmo. Dado algún número real positivo y , queremos hallar x tal que $x^x = y$. Nuestro ejemplo previo de $b(x) = x^x - 5$ llevaría a cabo esta tarea para el caso específico de $y = 5$, pero ahora escribiremos código que debería trabajar para cualquier real positivo y .

Probablemente el lector adivina que, en lugar de usar $b(x) = x^x - 5$, usaremos $b(x) = x^x - y$. Sin embargo, este no es el único cambio, pues necesitamos hacer una suposición inicial. Es fácil para un ser humano usar el razonamiento para esta tarea, pero una computadora carece del poder de la razón. Necesitamos una regla fácil e ingeniosa para producir una condición inicial rápidamente —incluso si esta no es muy buena—. Requeriremos una que es bastante pequeña incluso para valores grandes de y . Por ejemplo, consideremos $6^6 = 46\,656$ y notemos que, incluso con un número de 5 dígitos, basta el pequeño valor de $x = 6$. Este podría ser buena elección, pero existen otras opciones.

Al final, elegimos hacer la condición inicial igual a $1 + \log_{10}(y)$, donde \log_{10} representa el logaritmo común. Esto no es tan terrible, como ilustran los siguientes ejemplos:

$$\begin{array}{rclcl}
 1 + \log_{10}(46656) & = & 4,6689 \dots & \text{en lugar de} & 6 \\
 1 + \log_{10}(27) & = & 1,431\,36 \dots & \text{en lugar de} & 3 \\
 1 + \log_{10}(4) & = & 0,602\,05 \dots & \text{en lugar de} & 2
 \end{array}$$

El autor debe ser honesto aquí y confesar que tuvo que experimentar unas cuantas veces hasta que encontró una función de selección de valor inicial que consideró adecuada.

Finalmente, pero aún importante, debemos decidir cómo usar los parámetros opcionales. En este caso, decidimos fijar `max_iter=100`, de manera que podamos estar seguros que tenemos una buena respuesta. Hay una desventaja al usar 100 iteraciones, sin embargo —la salida textual será extremadamente larga y el usuario tendrá que desplazarse en la pantalla para leerla—. Aprenderemos cómo mitigar este problema usando control de verbosidad en la página 232.

El trozo final de código resultante puede encontrarse en la figura 5.4. Nótese que este código absolutamente requiere de la versión 2 o superior de nuestra subrutina `metodo_newton`, pues la versión 1 no tiene una sentencia `return`.

Código de Sage

```

1  def autologaritmo(y):
2      """Esta subrutina devuelve x tal que x^x = y, donde y es un número
3      real positivo."""
4
5      g(x) = x^x - y
6      aprox = 1 + log(y, 10)
7
8      respuesta = metodo_newton(g, aprox, max_iter=100)
9
10     # Lo que sigue es meramente código de test
11     test = respuesta ^ respuesta
12     print(respuesta, '^', respuesta, '=', test)
13     print('Lo deseado:', y)
14
15     return respuesta

```

Figura 5.4 Código para calcular el autologaritmo, usando `metodo_newton` (versión 2 o superior).

Ahora podemos probar nuestra subrutina escribiendo

Código de Sage

```
1 autologaritmo(17)
```

5.3.7 Desafío: Encontrar rectas tangentes paralelas

El objetivo de este desafío es, dadas dos funciones $a(x)$ y $b(x)$, encontrar un valor de x , denotado por x_p , tal que las rectas tangentes a ambas funciones en $x = x_p$ sean paralelas. Esto debe hacerse llamando a nuestra implementación del Método de Newton. Los parámetros son $a(x)$ y $b(x)$, y la salida debe ser el valor de x_p . Para un desafío aun más difícil, la subrutina debería devolver ambas rectas tangentes en la forma $y = mx + b$. (Advertencia: ¡esta es una tarea muy difícil!)

Aquí tenemos algunos ejemplos:

- Para $a(x) = x^2 + 2$ y $b(x) = x^3 - x$, una respuesta es $x_p = 1$.
 - Obsérvese que $a'(1) = 2$ y $b'(1) = 2$.
 - Obtenemos una recta tangente para $a(x)$ definida por $y = 2x + 1$.
 - Obtenemos una recta tangente para $b(x)$ definida por $y = 2x - 2$.
 - Para una representación de esta situación, véase la gráfica izquierda en la figura 5.5.
- Para $a(x) = x^2 + 2$ y $b(x) = x^3 - x$, la otra solución es $x_p = -1/3$.
 - Obsérvese que $a'(-1/3) = -2/3$ y $b'(-1/3) = -2/3$.
 - Obtenemos una recta tangente para $a(x)$ definida por $y = (-2/3)x + 17/9$.
 - Obtenemos una recta tangente para $b(x)$ definida por $y = (-2/3)x + 74/999$.
 - Para una representación de esta situación, véase la gráfica derecha en la figura 5.5.
- La subrutina identificaría solo una de estas dos soluciones, a menos que el lector hiciese algo muy sofisticado.

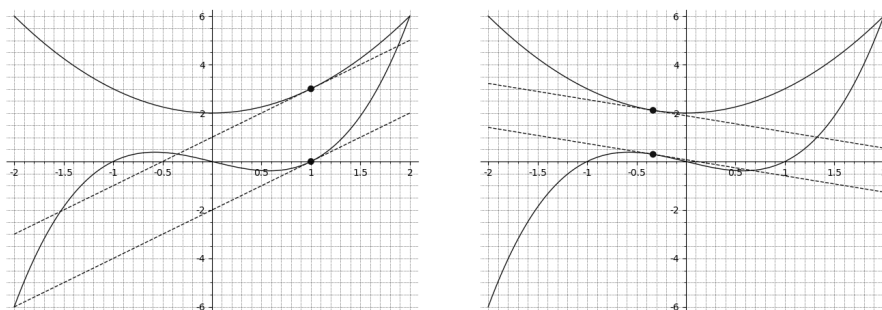


Figura 5.5 Los dos casos de rectas tangentes paralelas.

Para un ejemplo diferente, consideremos $a(x) = \sqrt{x}$ y $b(x) = x^2$. La única solución es $x_p = \sqrt[3]{1/16}$. Obsérvese que

$$a'(x) = b'(x) = \sqrt[3]{\frac{1}{2}} \approx 0,793\,700\,525 \dots,$$

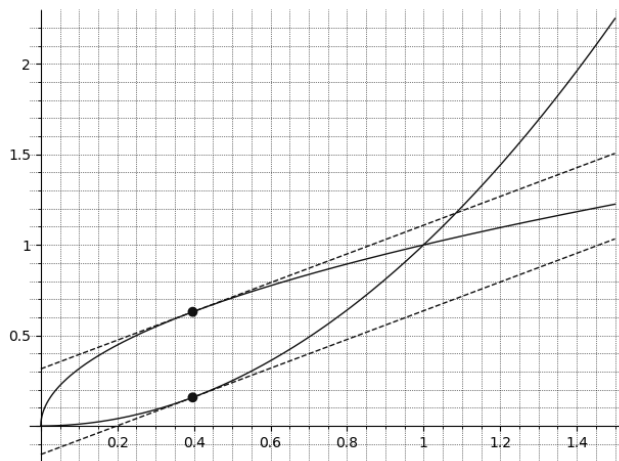
implicando que las rectas tangentes son

$$y = (0,793\,700\,525 \dots)x + 0,314\,980\,262 \dots$$

para $a(x)$ y

$$y = (0,793\,700\,525 \dots)x - 0,157\,490\,131 \dots$$

para $b(x)$. Esta situación se resume en el siguiente gráfico:



5.3.8 Desafío: El Método de Halley

Edmond Halley (1656–1742) fue un astrónomo y físico por quien el cometa Halley es nombrado. Sin embargo, también fue amigo de Sir Isaac Newton y contribuyó a ampliar el campo de estudio que eventualmente se conoció como cálculo.

Un descubrimiento menor de Halley, que ha sido casi totalmente olvidado, es un método para encontrar raíces, que es muy similar al trabajo de Newton con Joseph Raphson. Por ejemplo, hasta 2007, no existió un artículo sobre este método en Wikipedia, donde de hecho, muchas otras fórmulas oscuras pueden encontrarse. En cualquier caso, aquí tenemos la fórmula subyacente:

$$x_{sig} = x_{ant} - \frac{f(x_{ant}) f'(x_{ant})}{[f'(x_{ant})]^2 - \frac{1}{2} f(x_{ant}) f''(x_{ant})}$$

Como podemos apreciar rápidamente, es más complicada y requiere además el conocimiento de la segunda derivada. Sin embargo, tiene una excelente convergencia aproximando una raíz r de $f(x)$, suponiendo que $f'(r) \neq 0$ y $f''(r) \neq 0$; también, $f'''(x)$ debe existir, por lo que ciertas funciones están excluidas. En situaciones donde $f'(x)$ y $f''(x)$ pueden calcularse fácilmente, el Método de Halley puede ser un algoritmo extremadamente eficiente para buscar raíces.

Este desafío consiste en modificar nuestro código de manera que use el Método de Halley en lugar del Método de Newton. Sin embargo, tenemos una sugerencia: antes que cualquier cosa, vale la pena almacenar los valores de $f(x)$, $f'(x)$ y $f''(x)$ en “variables temporales”. Por ejemplo, $a = f(x_{ant})$, y similarmente para b y c conteniendo la primera y segunda derivadas, respectivamente. Entonces la larga fórmula de Halley de arriba será mucho más fácil de ingresar en la computadora.

5.4 Una introducción al control de flujo

Vamos a expandir nuestras capacidades en esta sección, mientras programamos en Python y Sage, para incluir estructuras que controlan el flujo de un programa. La más común de estas, por mucho, es la construcción “if-then” (“si-entonces”). Veremos varios ejemplos de esta, además de cómo detener una subrutina temprano en la ejecución (por ejemplo, si ha terminado sus cálculos antes de lo esperado) y cómo anunciar un mensaje de error personalizado cuando algo sale mal.

5.4.1 Control de verbosidad

El control de verbosidad nos permite limitar cuánto de la salida en pantalla de una subrutina obtenemos. En este momento, estamos desarrollando la subrutina `metodo_newton`, así que en verdad deseamos la salida completa —para permitirnos estudiar lo que realmente está ocurriendo—. Sin embargo, nuestro código podría llegar a ser parte de un programa más largo algún día. Todos esos detalles impresos en pantalla destruirían el rendimiento del algoritmo, pues cualquier tarea relacionada con I/O (Input/Output o Entrada/Salida) es siempre mucho más lenta que un cálculo.

El plan, por lo tanto, típicamente involucra reunir todas las funciones `print` en un mismo punto y “protegerlas” con una sentencia `if`. Este es nuestro primer encuentro con la construcción `if`, y es una forma sencilla de entender el uso de este comando. Combinaremos esto con un nuevo parámetro, llamado “`verboso`”. Como hicimos con `max_iter`, este será opcional. Si el parámetro `verboso` es fijado a `True`, imprimiremos; caso contrario, no lo haremos.

El código mismo está dado en la figura 5.6. Aquellas líneas que han cambiado (desde la versión 2 dada en la figura 5.3) tienen la anotación “# cambió” a su lado; aquellas que han sido movidas tienen la anotación “# movida”; las líneas nuevas están indicadas con “# nueva”.

Código de Sage

```

1  def metodo_newton(f, x_ant, max_iter=10, verboso=False):    # cambió
2      """Una implementación del Método de Newton para encontrar una raíz
3      de f(x), dada una suposición inicial para x. Por defecto, 10
4      iteraciones serán realizadas, a menos que el parámetro opcional
5      max_iter sea ajustado a otro valor. Use verboso=True para ver todos
6      los pasos intermedios."""    # cambió
7
8      f_prima(x) = diff(f, x)
9
10     for j in range(0, max_iter):
11         razon = f(x_ant) / f_prima(x_ant)    # movida
12         x_sig = x_ant - razon    # movida
13
14         if verboso==True:    # nueva
15             print('Iteración =', j)
16             print('x_ant =', x_ant)
17             print('f(x) =', f(x_ant))
18             print("f'(x)=", f_prima(x_ant))
19             print('razón =', razon)
20             print('x_sig =', x_sig)
21             print()
22
23         x_ant = N(x_sig)
24
25     return x_ant

```

Figura 5.6 El Método de Newton en Sage, versión 3

Aunque parece haber una gran cantidad de cambios, en realidad no hay mucha diferencia:

- Hemos añadido el parámetro opcional `verboso`.
- Hemos movido todas las funciones `print` a un mismo lugar. Requiere un poco de cuidado decidir dónde ponerlas. Tienen que estar lo suficientemente tarde en el código para asegurar que `razon` y `x_sig` hayan sido calculados. Sin embargo, si las colocamos después de `x_ant = N(x_sig)`, entonces la distinción entre `x_ant` y `x_sig` desaparecería.
- Todas las líneas con funciones `print` se han sangrado un nivel adicional.
- Añadimos la instrucción `if verboso==True:`.

El sangrado del texto es muy importante. Por ejemplo:

- Las funciones `print` se han sangrado muy a la derecha, lo que muestra que están subordinadas a las sentencias `if` y `for`. Esto significa que la sentencia `if` será ejecutada por cada iteración del bucle `for` y las funciones `print` serán ejecutadas solo si el parámetro `verboso` es `True`.
- La sentencia `x_ant = N(x_sig)` está menos sangrada, lo que muestra que está subordinada a la sentencia `for`, pero no a la estructura `if` —en otras palabras, el hecho que el parámetro `verboso` sea `True` o `False` es irrelevante para esta línea, que será ejecutada en cualquier caso—. Por otro lado, como esta instrucción está subordinada al ciclo `for`, esta será ejecutada en cada iteración.
- La sentencia `return x_ant` está muy poco sangrada. Esto significa que no está subordinada al ciclo `for` ni a la sentencia `if`. Sin embargo, sí está subordinada a `def`, lo que implica que de hecho es parte de la subrutina. Solo será ejecutada una vez, no en cada iteración.

Podemos probar este código con cualquiera de los siguientes comandos:

Código de Sage

```
1 metodo_newton(b, 2)
2 metodo_newton(b, 2, verboso=True)
3 metodo_newton(b, 2, max_iter=5)
4 metodo_newton(b, 2, max_iter=5, verboso=True)
5 metodo_newton(b, 2, 5, True)
6 metodo_newton(verboso=True, x_ant=2, max_iter=5, f=b)
```

Como podemos apreciar, estas tres últimas líneas son equivalentes. Recordemos que podemos escribir los parámetros de una subrutina en cualquier orden, siempre que indiquemos qué parámetro corresponde con qué valor. Si no lo hacemos, Sage asignará los valores a los parámetros en el mismo orden que en la línea `def` que define la subrutina. Esta es una regla que Sage heredó de Python.

Tenemos un tecnicismo menor que debemos discutir ahora. En la sentencia `if`, vemos dos signos igual, “==”. Hasta este punto, hemos usado uno solo. La clave aquí es que si estamos *comprobando* igualdad, entonces usamos dos signos igual, pero si estamos *causando* igualdad, entonces usamos uno solo. Por ejemplo, si quisiéramos forzar la verbosidad en nuestra subrutina, escribiríamos `verboso=True` en algún punto. Sin embargo, en la sentencia `if`, necesitamos verificar la igualdad, no forzarla, así que usamos el doble signo de igualdad en `if verbose==True:` para hacerlo.

A propósito de esto, si el lector sabe programación, muy probablemente notó que podríamos haber escrito simplemente “`if verboso:`” en lugar de “`if verboso == True:`”. En efecto, si suponemos que `verboso=True`, resulta que la comprobación `verboso == True` es verdadera; sin embargo, si `verboso=False`, resulta que `verboso == True` es falsa. Naturalmente, esto es válido para cualquier variable de tipo booleana⁷. Entonces, ¿cuál es la forma preferida de escribir la sentencia `if`? Se considera un buen estilo de programación escribir “`if verboso:`”. Sin embargo, hemos usado —y seguiremos usando— la otra notación por ser más clara para el principiante. El lector es libre de usar cualquiera.

La presencia de un parámetro opcional para la verbosidad es una característica del estilo de programación del autor. Por supuesto, la mayoría de los programadores usan esta técnica en algún punto u otro, pero casi cualquier subrutina que escribe el autor tiene esta opción. No solo hace mucho más fácil depurar el código, sino que se considera un buen hábito.

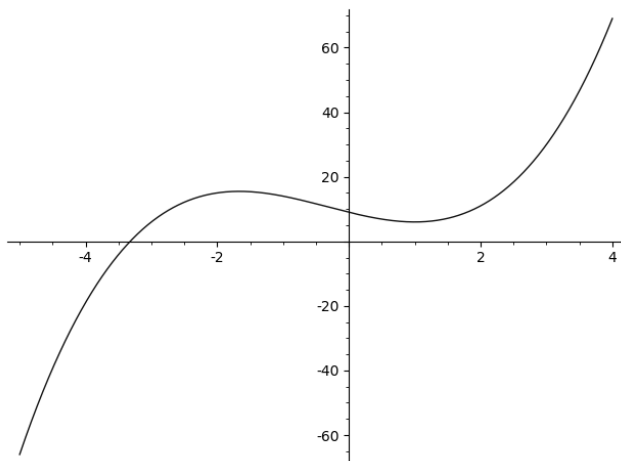
La noción de control de verbosidad es común en computación numérica. Algunas veces, esta incluso aparece en el humor de los analistas numéricos. En cierta ocasión el autor estaba divagando sobre un tema relativamente sin importancia, cuando un colega se sintió frustrado y bromeó diciendo “¡Greg, suficiente! ¡Verboso es igual a falso!”.

5.4.2 Interludio teórico: Cuando el Método de Newton enloquece

Consideremos la siguiente función, que después de todo es solo un polinomio cúbico, y por lo tanto no es particularmente intimidante.

$$p(x) = x^3 + x^2 - 5x + 9.$$

Incluso su gráfica no es particularmente inusual, como podemos ver abajo (mostrando el dominio $-5 < x < 4$).



⁷Que solo puede tomar uno de dos valores: `True` o `False`.

Sin embargo, veamos qué ocurre cuando intentamos con la condición $x = 2$ en el Método de Newton para $p(x)$. La respuesta que obtenemos es un mensaje de error, que se reproduce en la figura 5.7. Como podemos ver, el problema es que durante la iteración 2, tenemos que $f'(x) = 0$.

Podríamos pensar que este es un caso aislado, pero estaríamos equivocados. Aquí tenemos tres ejemplos más:

- $q(x) = x^3 + x^2 - 56x + 155$ con condición inicial $x = 3$,
- $r(x) = x^3 + x^2 - 85x + 323$ con condición inicial $x = 3$ y
- $s(x) = x^3 - x^2 - 8x + 15$ con condición inicial $x = 1$.

La cuestión clave aquí es que el Método de Newton requiere que dividamos por $f'(x)$, y en estos ejemplos vemos que $f'(x)$ se hace cero. Dado que no es permitido dividir por cero, la subrutina falla. (La jerga para “dividir por cero” es “explotar”). Los siguientes ejemplos son polinomios que explotan en la primera iteración y no sobreviven incluso para la segunda.

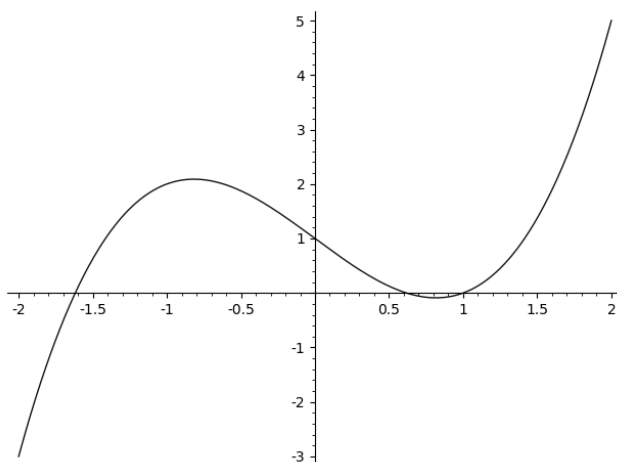
Código de Sage

```
1 p(x) = x^2 + 8*x - 9
2 newton_method(p, -4)
3
4 q(x) = x^4 - 2*x^2
5 newton_method(q, -1)
6 newton_method(q, 0)
7 newton_method(q, 1)
```

De hecho, es relativamente raro caer *exactamente* sobre cero. Pero otras dificultades pueden ocurrir cuando $f'(x)$ es aproximadamente cero. Consideremos nuestro ejemplo previo $c(x)$, que nuevamente es un polinomio cúbico muy poco intimidante:

$$c(x) = x^3 - 2x + 1$$

Incluso su gráfica no es particularmente inusual, como puede apreciarse abajo (mostrando $-2 < x < 2$).



Por lo tanto, podemos imaginar que tenemos una función de buen comportamiento...hasta que escribimos lo siguiente:

Código de Sage

```
1 metodo_newton(c, -0.436_485, max_iter)
```

o incluso peor:

Código de Sage

```
1 metodo_newton(c, -0.436_485_285_968_722)
```

En la computadora del autor, el segundo ejemplo no hace ningún progreso hasta la iteración 79. Eso se debe a que $f'(x)$ pasa extremadamente cerca de cero.

```

Iteración = 0
x_ant = 2
f(x) = 11
f'(x) = 11
razón = 1
x_sig = 1

Iteración = 1
x_ant = 1.000 000 000 000 00
f(x) = 6.000 000 000 000 00
f'(x) = 0.000 000 000 000 000
-----
ValueError                                Traceback (most recent call last)
/Scientific/SageMath/local/lib/python3.7/site-packages/sage/symbolic/expression.pyx in
sage.symbolic.expression.Expression._div_
(build/cythonized/sage/symbolic/expression.cpp:24515)()
 3608             else:
-> 3609                 x = left._gobj / _right._gobj
 3610                 return new_Expression_from_GEx(left._parent, x)

ValueError: power::eval(): division by zero

During handling of the above exception, another exception occurred:

ZeroDivisionError                        Traceback (most recent call last)
/home/diego/Joint Work with Diego Sejas/bard.sagetex.sage.py in metodo_newton(f, x_ant,
max_iter)
 7283             print("f'(x)=", f_prima(x_ant))
 7284
-> 7285             razon = f(x_ant) / f_prima(x_ant)
 7286
 7287             print('razón =', razon)

/Scientific/SageMath/local/lib/python3.7/site-packages/sage/structure/element.pyx in
sage.structure.element.Element.__truediv__
(build/cythonized/sage/structure/element.c:12796)()
 1714             cdef int c1 = classify_elements(left, right)
 1715             if HAVE_SAME_PARENT(c1):
-> 1716                 return (<Element>left)._div_(right)
 1717             if BOTH_ARE_ELEMENT(c1):
 1718                 return coercion_model.bin_op(left, right, truediv)

/Scientific/SageMath/local/lib/python3.7/site-packages/sage/symbolic/expression.pyx in
sage.symbolic.expression.Expression._div_
(build/cythonized/sage/symbolic/expression.cpp:24595)()
 3613             # See http://docs.cython.org/docs/wrapping_CPlusPlus.html
 3614             if 'division by zero' in str(msg):
-> 3615                 raise ZeroDivisionError("symbolic division by zero")
 3616             else:
 3617                 raise

ZeroDivisionError: symbolic division by zero

```

Figura 5.7 La salida de la subrutina `metodo_newton` (versión 2) cuando falla porque trata de dividir por cero.

La pregunta es ahora: ¿qué vamos a hacer acerca de todo esto? Pues bien, resulta que estos ejemplos representan elecciones fenomenalmente desafortunadas de una condición inicial. Si en su lugar hiciésemos una elección ligeramente mejor (tal vez primero graficando la función), veríamos que cada ejemplo de esta subsección converge rápida y tranquilamente.

De acuerdo a esto, lo que las buenas implementaciones hacen es que, si $f'(x)$ llega a ser cero o muy próximo a cero, avisan sobre un error de condición para que el usuario pueda volver a intentar con una nueva y diferente elección para la condición inicial. Nosotros haremos esto muy pronto, en la subsección 5.5.1 en la página 239.

5.4.3 Deteniendo el Método de Newton antes

En varias ocasiones hemos notado que los cálculos continúan ejecutándose mucho después que una respuesta bastante exacta ha sido calculada. Esto es un desperdicio, particularmente si nuestra subrutina fuera llamada millones de veces dentro de un programa más grande. Por ejemplo, procedimientos de minimización multivariada frecuentemente involucran una ejecución entera del Método de Newton para cada una de sus iteraciones, pero un millón de iteraciones del solucionador multivariable no sería algo muy inusual.

En consecuencia, deberíamos crear algún tipo de interrupción. Si $f(x)$ es más pequeño que algún umbral particular, tal vez una mil millonésima, entonces deberíamos detener el cálculo y reportar la respuesta final. Deberíamos tomarnos un momento para decidir si este umbral será predeterminado o especificado por el usuario. Un valor predeterminado es mejor para usuarios que no estén muy familiarizados con el Método de Newton, mientras que permitir que sea manualmente especificado sería mejor para los expertos. Lo mejor de ambos mundos puede lograrse usando un parámetro opcional. El usuario principiante no necesita preocuparse (o tan siquiera saber) del parámetro opcional, pero el experto puede especificar un valor cuando lo requiera. También, dado que es poco probable que este umbral tenga que ser cambiado muchas veces, usar un parámetro opcional nos evita tener demasiadas cosas entre los paréntesis cada vez que la subrutina es llamada. El nombre tradicional para un umbral de este tipo es “épsilon”. La letra griega estilizada ε es frecuentemente usada para representar un valor de “suficientemente cerca” en matemática computacional.

En la figura 5.8 podemos encontrar el nuevo código, o versión 4, del Método de Newton. Las nuevas líneas clave son

Código de Sage

```
12 if abs(f(x_ant)) < epsilon:
13     return x_ant
```

La primera dice (en lenguaje de Python) lo que un matemático escribiría como “si $|f(x)| < \varepsilon$, entonces”; la segunda línea, es decir la sentencia `return`, indica a la subrutina retornar o devolver la respuesta final. Está implícito en esa sentencia `return` que la subrutina ha completado sus cálculos y que nada más del código restante será ejecutado hasta la siguiente llamada. Así, es muy común que una subrutina tenga muchas sentencias `return`, pero que, cuando cualquiera de ellas es alcanzada, la ejecución termine.

Código de Sage

```
1 def metodo_newton(f, x_ant, max_iter=10, verboso=False, epsilon=10-(9)): # cambi
2     """Una implementación del Método de Newton para encontrar una raíz
3     de f(x), dada una suposición inicial para x. Por defecto, 10
4     iteraciones serán realizadas, a menos que el parámetro opcional
5     max_iter sea ajustado a otro valor. Use verboso=True para ver
6     todos los pasos intermedios. El algoritmo terminará antes si
7     |f(x)| < epsilon.""" # cambi
8
9     f_prima(x) = diff(f, x)
10
11     for j in range(0, max_iter):
12         if abs(f(x_ant)) < epsilon: # nueva
13             return x_ant # nueva
14
15         razon = f(x_ant) / f_prima(x_ant)
16         x_sig = x_ant - razon
```

```

17
18         if verboso==True:
19             print('Iteración =', j)
20             print('x_ant =', x_ant)
21             print('f(x) =', f(x_ant))
22             print("f'(x) =", f_prima(x_ant))
23             print('razón =', razon)
24             print('x_sig =', x_sig)
25             print()
26
27         x_ant = N(x_sig)
28
29     return x_ant

```

Figura 5.8 El Método de Newton en Sage, versión 4

Algunos pensamientos acerca del valor absoluto La forma de escribir el valor absoluto en Sage es usando el comando `abs`, como vimos primero en la página 10.

Es interesante pensar por qué, desde el punto de vista matemático, se usa el valor absoluto. Imaginemos que hubiésemos establecido que la condición de terminación de nuestra subrutina fuese $f(x) < \epsilon$ en lugar de $|f(x)| < \epsilon$. Consideremos $f(x) = 9 - x^2$ y supongamos que un usuario confundido usa la condición inicial $x = 100$. Entonces tendríamos $f(100) = -91$. ¿Es un buen punto para detenerse? ¿Es $x = 100$ una aproximación a una raíz de $f(x) = 9 - x^2$? ¡Por supuesto que no! Sin embargo, $-91 < 10^{-9}$ es cierto. Para evitar esto, necesitamos los símbolos de valor absoluto, para poder verificar si $|-91| < 10^{-9}$, que en efecto es falso, como deseamos.

Si el lector toma un curso de *Análisis Real*, entonces verá otros muchos ejemplos donde “ $|f(x)| < k$ ” es usado como abreviación de $-k < f(x) < k$. Este es un truco muy común en Análisis Matemático.

Probando la terminación temprana Podemos verificar que este nuevo código de hecho funciona, escribiendo

Código de Sage

```

1  b(x) = x^x - 5
2  metodo_newton(b, 2, verboso=True)

```

que termina su ejecución muy temprano. Después de la iteración 3, tenemos x_{sig} mostrado como la respuesta final, pues se ha determinado que $f(x_{sig})$ es menor que `epsilon`, fijado en su valor original de una mil millonésima.

Por supuesto, algunos usuarios podrían requerir mayor precisión. En ese caso, el código

Código de Sage

```

1  b(x) = x^x - 5
2  newton_method(b, 2, epsilon=10^(-15), verboso=True)

```

nos muestra cómo podemos lograr una precisión de mil billonésimas. Es interesante que solo una iteración adicional es requerida para ello.

En general, es bueno permitir que el usuario controle detalles específicos, como el estándar de “suficientemente bueno”. Por ejemplo, alguien estimando el déficit presupuestario de un país probablemente estaría feliz con una precisión al dólar o centavo más cercano, y no querría desperdiciar cálculos para obtener una precisión de una mil millonésima de dólar.

5.4.4 La lista de comparaciones

Cuando escribimos una sentencia `if`, existen seis construcciones comunes. Estas están listadas abajo en su forma matemática normal, así como el código correcto en Sage o Python para cada una.

Matemática	Sage o Python
si $k < 6$, entonces	<code>if k < 6:</code>
si $k > 6$, entonces	<code>if k > 6:</code>
si $k = 6$, entonces	<code>if k == 6:</code>
si $k \leq 6$, entonces	<code>if k <= 6:</code>
si $k \geq 6$, entonces	<code>if k >= 6:</code>
si $k \neq 6$, entonces	<code>if k != 6:</code>

Nótese que aunque muchas personas encuentran que escribir `if k == 6` más atractivo que `if k >= 6`, Python no aceptará esa notación; la sintaxis correcta es la segunda. La forma de no olvidar esto es recordar que es común decir “es mayor o igual que”, pero es muy poco común decir “es igual a o mayor que”. También, debemos tener cuidado de usar `==` (dos signos de igualdad) en lugar de `=` (un solo signo de igualdad) en cualquier sentencia `if`, como aprendimos en la subsección 5.4.1 en la página 234.

5.4.5 Desafío: ¿Cuántos primos menores que o iguales a un x dado hay?

Hemos estado trabajando con el Método de Newton por un buen tiempo, así que será agradable considerar un desafío que no tiene nada que ver con este. Consideremos la siguiente pregunta:

Para cualquier entero $x > 1$, ¿cuántos números primos pueden encontrarse entre 1 y x , inclusive?
 Más formalmente, ¿cuántos primos p hay tales que $1 \leq p \leq x$?

Aquí damos una pista: el lector puede usar un bucle `for` que cuente 1, 2, 3, ..., $x - 1$, x . Entonces, para cada uno de esos enteros z , puede preguntar a Sage si es un primo con `is_prime(z)`, que devolverá ya sea `True` o `False`, de manera correspondiente. Usando una sentencia `if`, puede incrementar o no un contador, dependiendo del resultado obtenido con esta función.

En caso que el lector desee comprobar su solución, la función `prime_pi` de Sage realiza exactamente esta tarea. Esta suele denotarse por $\pi(x)$ (no debe confundirse con el número π) y se denomina *función contador de números primos*.

Ahora, para un desafío extra (una vez que se haya completado el anterior), se debe aumentar un parámetro opcional extra, `listar`, que por defecto es `False`. Sin embargo, si el usuario lo cambia a `True`, entonces la subrutina deberá imprimir la lista de todos los primos que descubra mientras cuenta de 1 a x .

5.5 Más conceptos sobre control de flujo

En esta sección exploraremos el uso del comando `raise` para situaciones inesperadas, además de estudiar un control más matizado basado en un concepto llamado la construcción “`if-then-else`”.

5.5.1 Levantando una “excepción”

En todo tipo de programas, uno tiene que manejar situaciones en las que algo salió mal. Estas pueden ser una división por cero (como discutimos en la subsección 5.4.2 en la página 234), un número negativo donde no se esperaba uno, o cualquiera de una multitud de otras situaciones. El término formal para este tipo de situaciones es *excepciones*. En el caso del Método de Newton, no queremos que $f'(x)$ sea cero o se aproxime mucho a cero.

El comando de Python `raise` (levantar) es usado para anunciar que una excepción debe ser generada porque alguna condición inusual ha sido detectada. Sin embargo, también da la oportunidad de componer un mensaje de error legible para un humano, que se espera pueda ayudar al usuario a deducir qué salió mal. A propósito de esto, “levantar una excepción” (“`raise an exception`”) es a veces referido como “arrojar un excepción” (“`throw an exception`”) en otros lenguajes de programación como Java. El código de la figura 5.9 levantará una excepción si $f'(x) = 0$ mientras se ejecuta el Método de Newton.

Código de Sage

```

1 def metodo_newton(f, x_ant, max_iter=10, verboso=False, epsilon=10^(-9),
2   ↪ peligro=10^(-4)): # cambio
3     """Una implementación del Método Newton para encontrar una raíz de
4     f(x), dada una suposición inicial para x. Por defecto, 10
5     iteraciones serán realizadas, a menos que el parámetro opcional
6     max_iter sea ajustado a otro valor. Use verboso=True para ver todos
7     los pasos intermedios. El algoritmo terminará antes si
8     |f(x)| < epsilon. El algoritmo levantará una excepción si
9     |f'(x)| < peligro.""" # cambio
10
11     f_prima(x) = diff(f, x)
12
13     for j in range(0, max_iter):
14         if abs(f(x_ant)) < epsilon:
15             return x_ant
16
17         if abs(f_prima(x_ant)) < peligro: # nueva
18             raise RuntimeError('f_prima(x) se hizo muy pequeña! '
19                               + 'Por favor, pruebe otra condición inicial.') # nueva
20
21         razon = f(x_ant) / f_prima(x_ant)
22         x_sig = x_ant - razon
23
24         if verboso==True:
25             print('Iteración =', j)
26             print('x_ant =', x_ant)
27             print('f(x) =', f(x_ant))
28             print("f'(x) =", f_prima(x_ant))
29             print('razón =', razon)
30             print('x_sig =', x_sig)
31             print()
32
33         x_ant = N(x_sig)
34
35     return x_ant

```

Figura 5.9 El Método de newton en Sage, versión 5

Por ejemplo, el código

Código de Sage

```

1 c(x) = x3 - 2*x + 1
2 metodo_newton(c, -0.436_485_285_968_722, verboso=True, max_iter=20)

```

resultará en la salida mostrada en la figura 5.10. La indicación “Por favor, pruebe otra condición inicial”, en la última línea de la salida, causará que el usuario obediente escriba algo similar a

Código de Sage

```

1 c(x) = x3 - 2*x + 1
2 metodo_newton(c, -0.4, verboso=True, max_iter=20)

```

y entonces Sage tranquilamente devolverá la solución correcta, es decir 0,618 03 ... (Muchos otros números podrían haber sido usados en lugar de -0,4 como condición inicial.)

```

-----
RuntimeError                                Traceback (most recent call last)
/home/diego/Joint Work with Diego Sejas/bard.sagetex.sage.py in metodo_newton(f,
x_ant, max_iter, verbose, epsilon, peligro)
7334
7335         if abs(f_prima(x_ant)) < peligro:         # nueva
-> 7336             raise RuntimeError('¡f_prima(x) se hizo muy pequeña! '
7337                 + 'Por favor, pruebe otra condición inicial.')    #
nueva
7338

RuntimeError: ¡f_prima(x) se hizo muy pequeña! Por favor, pruebe otra condición
inicial.

```

Figura 5.10 La salida generada por el comando `raise` en `metodo_newton`.

El cambio primario en el código que permite esta característica es el par de líneas

```

Código de Sage
1  raise RuntimeError('¡f_prima(x) se hizo muy pequeña! '
2      + 'Por favor, pruebe otra condición inicial.')
```

que notifica a Python y Sage que una excepción ha ocurrido y que todo cálculo debería detenerse. También informa que la excepción es de la categoría `RuntimeError` (error en tiempo de ejecución), además de proveer un mensaje de error fácilmente legible. No distinguiremos en este libro entre los varios tipos de excepciones, pero libros más avanzados sobre programación en Python puede que lo hagan.

Notemos también cómo hemos usado el símbolo `+` para hacer que un mensaje de error muy largo quepa en dos líneas. No está permitido introducir un salto de línea dentro de un texto entre comillas simples o dobles. Por lo tanto, hemos escrito nuestro mensaje de error en dos pedazos —cada uno encajando en una sola línea— y entonces hemos usado el símbolo de suma para indicar a Sage que estos constituyen en realidad un solo texto largo. Podemos ver la salida generada en la figura 5.10. Como siempre, la última línea de un mensaje de error en Sage es el más importante.

Existen muchas ventajas al usar el comando `raise` para notificar al usuario de un error mayor, en lugar de solo detener la ejecución. Primero, podríamos imaginar que en un programa grande, algunas secuencias de subrutinas podrían llamar un solucionador multivariado sofisticado, que a su vez llamaría a nuestro `metodo_newton`. Si solo devolvemos un valor (tal como 0) e imprimimos un mensaje de error, no estamos “notificando” a las capas superiores que algo ha salido mal y el programa entero podría hacer cosas muy extrañas como consecuencia. Sin embargo, el comando `raise` detendría la ejecución y prevendría que el programa más grande haga algo muy inesperado. Segundo, existe una construcción `try-except` en python, que no podremos explorar aquí. Por medio de esta, el usuario puede responder y solucionar el error, y tal vez intentar ejecutar la subrutina otra vez. Tercero, el comando `raise` nos permite ofrecer un buen y útil mensaje de error al usuario.

Como sin duda observó el lector, muchos mensajes de error en Sage son extremadamente difíciles de comprender. Esto de seguro le ha causado algo de frustración en varias ocasiones. Esa es la razón por la que debemos intentar evitar causarle el mismo sufrimiento a nuestros usuarios, y debemos crear mensajes de error informativos —así como esperar que futuras generaciones de desarrolladores de Sage hagan algo similar—.

5.5.2 La construcción `if-then-else`

En la subsección 5.2.5 en la página 221 escribimos una subrutina, llamada `funcion_a_sucesion`, que evalúa una función en los primeros números naturales, 0, 1, 2, 3, ..., hasta un superior requerido. Sin embargo, ese código fallará si en algún momento dividimos por cero. Podemos demostrarlo con esta función:

$$f(x) = \frac{x^2 - 4x + 3}{x^2 - 5x + 6}.$$

Ahora utilizaremos un concepto muy poderoso en programación computacional, llamado la construcción “if-then-else” (“si-entonces-sino”). Esta es similar a la sentencia if (si) que usamos antes, pero ahora estamos añadiendo la cláusula else (sino). La idea es que primero verificaremos (con un if) si el denominador es cero o no. Si es cero, entonces imprimiremos “n.e.”, por “no existe”; si no es cero, entonces simplemente imprimiremos el valor de la función entera en ese número natural —y aquí entra el “else”—.

Consideremos el código siguiente:

Código de Sage

```

1 def funcionRacionalASucesion(num, denom, cuantos):
2     """Esta subrutina toma el numerador y denominador de una función
3     racional como primera y segunda entradas, respectivamente, y evalúa
4     la función en los primeros 'cuantos' números naturales. Por ejemplo,
5     si cuantos es 4, entonces f es evaluada en 0, 1, 2, 3. Cualquier polo
6     encontrado es reportado como 'n.e.' (no existe)."""
7     for j in range(0, cuantos):
8         if (denom(x=j) == 0):
9             print('n.e.', end=' ')
10        else:
11            value = num(x=j) / denom(x=j)
12            print(value, end=' ')
13    print()

```

Ahora tenemos dos parámetros para comunicar la función a la subrutina, pues necesitamos el numerador y el denominador por separado. El tercer parámetro nos indica cuántas evaluaciones de la función debemos hacer. El ciclo for solamente nos lleva a través de los primeros cuantos números naturales, es decir 0, 1, 2, ..., cuantos – 2, cuantos – 1. A continuación de esto, en la octava línea, evaluamos el denominador en el punto $x = j$ y verificamos si es cero. En la novena línea, solamente ejecutada si el denominador es cero, imprimimos “n.e.” (como abreviación de “no existe”). La línea número diez es el comando else, que significa que las líneas once y doce —que están sangradas bajo el else y por lo tanto, subordinadas a este— serán ejecutadas solamente si el denominador no es cero. Calculamos el valor de $f(x)$ en $x = j$, que no es nada más que el numerador dividido por el denominador (línea once), y lo imprimimos (en la línea doce).

Seguramente el lector notó que hemos usado la opción `end=' '` en las dos primeras funciones `print`. Esto lo hicimos con el objetivo de forzar a que estas impriman un espacio en lugar del usual salto de línea. De esta manera, todos los resultados se mostrarán en una misma línea de texto. La última función `print` (línea trece) añade un salto de línea final para terminar la impresión de los resultados. Así, si llamamos múltiples veces a la misma subrutina, cada ejecución imprimirá sus resultados por separado. Ya habíamos discutido a detalle sobre la opción `end` en la subsección 5.1.2 en la página 211.

Probemos nuestra subrutina con las siguientes instrucciones:

Código de Sage

```

1 f(x) = x^2 - 4*x + 3
2 g(x) = x^2 - 5*x + 6
3 funcionRacionalASucesion(f, g, 10)

```

Obtenemos la siguiente salida, y podemos estar satisfechos que la subrutina trabaja correctamente.

1/2 0 n.e. n.e. 3/2 4/3 5/4 6/5 7/6 8/7

La cláusula `else` nos ahorra tiempo, pero también hace que el código sea más legible. En efecto, el siguiente código es equivalente al anterior, pero más tedioso de escribir y leer.

Código de Sage

```

1  def funcionRacionalASucesion(num, denom, cuantos):
2      """Esta subrutina toma el numerador y denominador de una función
3      racional f como primera y segunda entradas, respectivamente, y
4      evalúa la función en los primeros 'cuantos' números naturales. Por
5      ejemplo, si cuantos es 4, entonces f es evaluada en 0, 1, 2, 3.
6      Cualquier polo encontrado es reportado como 'n.e.' (no existe)."""
7      for j in range(0, cuantos):
8          if denom(x=j) == 0:
9              print('n.e.', end=' ')
10
11         if denom(x=j) != 0:
12             valor = num(x=j) / denom(x=j)
13             print(valor, end=' ')
14     print()

```

Por lo tanto, el comando `else` no es estrictamente necesario, pero sí que hace las cosas más sencillas para el programador. Las ventajas de tener `else` disponible son más fáciles de apreciar cuando manejamos sentencias `if-then-else` complejas dentro de otras. No llegaremos a eso en este capítulo, pero otros libros de Python puede que lo hagan.

Una nota histórica es que, aunque no existe un comando “then” (“entonces”) en Python, ni en C/C++, ni en Java, el concepto es aún llamado “if-then-else”. Eso se debe a que algunos lenguajes de programación muy antiguos, como BASIC y Pascal, usaban esa palabra como comando. La nomenclatura refleja la memoria histórica.

5.5.3 Un desafío sencillo: El procesamiento del fin del semestre, parte 1

Para este desafío, vamos a escribir una pequeña subrutina que recibirá la nota de un estudiante en una escala de 0 a 100 y decidirá su estado en la universidad. El nombre de la subrutina debe ser `decidirEstado` y debe tener dos parámetros: primero, el nombre del estudiante y segundo, su nota.

Antes que el procesamiento de los datos comience, la subrutina debe verificar si la nota es menor que cero o mayor que cien, en cuyo caso levantará una excepción. Después de eso, si la nota es menor a 50, entonces debe declararse al estudiante en periodo de prueba académica. En una sola línea debe imprimirse su nombre seguido por las palabras “está bajo prueba académica”. Por otro lado, si el promedio del estudiante es 50 o mayor, entonces, en una sola línea, debe imprimirse su nombre y “está aprobado”.

Si el lector se encuentra en una región con una escala diferente de aprobación/reprobación a la propuesta aquí, puede adaptar este desafío a las reglas locales.

Es importante que se imprima cada resultado en una sola línea, como se ha indicado, porque más adelante (en la subsección 5.7.9) crearemos otra subrutina que llama a esta, pero para varios estudiantes.

5.5.4 Un desafío más difícil: El procesamiento del fin del semestre, parte 2

Esta es una modificación del ejercicio anterior. Recordemos que un promedio p es válido si y solo si $0 \leq p \leq 100$. Para cualquier promedio p en ese rango, el estado del estudiante debería darse de acuerdo a la siguiente regla:

100	≥	p	≥	75	:	Destacado
75	>	p	≥	50	:	Normal
50	>	p	≥	25	:	Periodo de prueba académica
25	>	p	≥	0	:	Expulsión académica

Nuevamente, si el lector se encuentra en una región con una escala diferente a la propuesta aquí, puede adaptar este desafío a las reglas locales.

Aquí tenemos una ayuda: el lector tal vez quiera intentar con tres construcciones `if-then-else`, es decir, tener un `if-then-else` gigantesco externo, para colocar otro `if-then-else` dentro de la “región del `then`”, y otro dentro de la “región del `else`”. Esto requiere de mucho cuidado con el sangrado de las líneas.

5.6 Bucles while versus bucles for

Como resulta ser, el bucle `for` es el más común en programación matemática, pero tiene una competencia cercana. El bucle `while` (mientras) es también muy importante y ocurre frecuentemente. Un ciclo `while` continuará siempre y cuando una prueba lógica retorne `True`; si en algún momento retorna `False`, el ciclo se detendrá.

No hay nada más en esta historia, ahora que hemos descrito la única diferencia entre los bucles `for` versus los bucles `while`. Por lo tanto, veamos un ejemplo.

5.6.1 Una cuestión sobre factoriales

Frecuentemente, cuando se enseña o aprende sobre factoriales, permutaciones, combinaciones y otros similares, es útil tratar de elegir números de un tamaño particular. Por simplicidad, restrinjámonos a los factoriales. Imaginemos que queremos encontrar los valores de n alrededor de los cuales $n!$ es aproximadamente mil millones, un billón o tal vez solo mil. Podríamos hacer esto por prueba y error, pero no es una actividad muy agradable. Sería más conveniente tener una subrutina que directamente nos provea una respuesta. Específicamente, dado algún número y , deseamos encontrar el valor más pequeño de x tal que $x! \geq y$.

Dado que $0! = 1$, podemos empezar con $x = 0$, y deberíamos contar hacia arriba hasta que alcancemos el primer valor de x tal que $x! \geq y$. Probemos con el siguiente código:

Código de Sage

```
1 def primer_factorial_mayor_que(y):
2     """Esta subrutina devuelve el entero más pequeño x tal que x! >= y."""
3     supuesto = 0
4     while factorial(supuesto) < y:
5         supuesto = supuesto + 1
6
7     return supuesto
```

Esta subrutina empezará con un valor para `supuesto` de 0. Seguirá ejecutando el comando `supuesto = supuesto + 1` mientras siga cumpliéndose que `factorial(supuesto) < y` evalúa a `True`. Sin embargo, el instante mismo en que evalúe a `False`, detendrá el bucle y ya no continuará con el incremento. En palabras sencillas, el momento mismo en que `factorial(supuesto) >= y`, el bucle se detiene y el flujo del programa continuará con la siguiente línea, que en este caso es `return supuesto`.

Podemos testar esto con unos cuantos números y ver si funciona. Por ejemplo,

Código de Sage

```
1 primer_factorial_mayor_que(130)
```

devuelve 6, pues $5! = 120$ y $6! = 720$. También podemos tratar de verificar con lo siguiente:

Código de Sage

```
1 print(primer_factorial_mayor_que(10^6))
2 print(factorial(9))
3 print(factorial(10))
```

5.6.2 Desafío: Encontrar el número primo siguiente

Para este desafío, vamos a reproducir uno de los comandos predefinidos en Sage. En la subsección 5.4.5 en la página 239, tuvimos el reto de contar el número de primos p tales que $1 \leq p \leq x$, presumiblemente con el uso de un bucle `for` y de la función `is_prime`. Ahora, nos gustaría encontrar el primo más pequeño que sea mayor que x . En español simple, nos gustaría encontrar el siguiente primo después de x , aunque no exista razón para asumir que x mismo es primo. Por ejemplo, el primo más pequeño mayor que 13 es 17 y el primo más pequeño mayor que 15 es también 17.

Lo que es interesante acerca de este desafío es que claramente no podemos usar un ciclo `for`, pues no tenemos idea de cuántos enteros tendremos que verificar. Por ejemplo, el siguiente primo después de 113 es 127, un salto de 14 enteros; el siguiente primo después de 107 es 109, un salto de 2 enteros; el siguiente primo después de 360 653 es 360 749, un salto de 96 enteros. Para números más grandes, la situación es incluso más curiosa. El siguiente primo después de 1 693 182 318 746 371 es 1 693 182 318 747 503, un salto de 1132 enteros. En contraste, el siguiente primo después de este último es 1 693 182 318 747 523, un salto de solo 20 enteros.

Por lo tanto, se debe crear una pequeña subrutina que cuente hacia arriba desde un número (la entrada) hasta que alcance un primo. Este, por supuesto, será mayor que la entrada (porque contamos hacia arriba, no hacia abajo) y debe ser retornado por la subrutina.

Se puede comprobar fácilmente nuestro trabajo en este desafío, pues existe la función `next_prime`, predefinida en Sage. Podemos apreciar cuán lejos han llegado nuestras habilidades, pues ya estamos programando subrutinas equivalentes a comandos de Sage.

Para un desafío extra, se puede tratar de escribir el código de manera que nunca se moleste en verificar un número par mayor que 2, pues estos obviamente no pueden ser primos. Si la entrada de la subrutina es impar, esto es fácil de lograr —el lector debe, por lo tanto, detectar y responder a entradas pares adecuadamente para cumplir este desafío extra—.

5.6.3 El Método de Newton con un bucle `while`

Ahora que entendemos los ciclos `while`, el autor debe confesar que se ha sentido incómodo usando un ciclo `for` en el Método de Newton. La aplicación realmente pide a gritos un bucle `while`. El plan general es repetir el proceso iterativo hasta que la aproximación cumpla el estándar requerido de ser menor que `epsilon` (en valor absoluto). Sin embargo, podemos tener una “ruta de escape” si un máximo permitido de iteraciones es excedido.

El código está dado en la figura 5.11. Hemos marcado las líneas nuevas con “# nueva” y las que han resultado alteradas con “# cambió”, como hicimos antes. Nótese que hemos cambiado el valor por defecto de `max_iter` a 1000, para reflejar el hecho de que esta raramente será la forma en que la subrutina termine; normalmente, terminará al obtener una aproximación menor que `epsilon` (en valor absoluto).

Código de Sage

```

1  def metodo_newton(f, x_ant, max_iter=1_000, verboso=False, epsilon=10^(-9),
↪  peligro=10^(-4)):    # cambió
2      """Una implementación del Método de Newton para encontrar una raíz
3      de f(x), dada una suposición inicial para x. Por defecto, un máximo
4      de 1000 iteraciones serán realizadas, a menos que el parámetro
5      max_iter sea ajustado a otro valor. Use verboso=True para ver todos
6      los pasos intermedios. El algoritmo terminará antes si |f(x)|<epsilon.
7      El algoritmo levantará una excepción si |f'(x)|<peligro."""
8
9      f_prima(x) = diff(f, x)
10     iteraciones = 0
11
12     while abs(f(x_ant)) >= epsilon:    # nueva
13         iteraciones = iteraciones + 1    # nueva
14
15         if iteraciones >= max_iter:    # nueva
16             print('Advertencia: Límite de iteraciones alcanzado.')    # nueva
17             break    # nueva
18
19         if abs(f_prima(x_ant)) < peligro:
20             raise RuntimeError('f_prima(x) se hizo muy pequeña! ' +
21                 'Por favor, intente otra condición inicial.')
22
23         razon = f(x_ant) / f_prima(x_ant)
24         x_sig = x_ant - razon

```

```

25
26     if verboso==True:
27         print('Iteración =', iteraciones)      # cambió
28         print('x_ant =', x_ant)
29         print('f(x) =', f(x_ant))
30         print("f'(x) =", f_prima(x_ant))
31         print('razón =', razon)
32         print('x_sig =', x_sig)
33         print()
34     x_ant = N(x_sig)
35
36     # Ahora el bucle ha terminado.      (# nueva)
37     if verboso==True:      # nueva
38         print(iteraciones, 'iteraciones fueron requeridas.')      # nueva
39         print('f(respuesta) =', f(x_ant))      # nueva
40
41     return x_ant

```

Figura 5.11 El Método de Newton en Sage, versión 6

Vemos en este código también la cuestión de cómo terminar el ciclo `while` después de `max_iter` iteraciones. Como podemos apreciar en las líneas 15 y 16, usamos una sentencia `if` para detectar la condición y luego imprimir una advertencia para el usuario. A continuación (línea 17), usamos el comando `break`, que aún no vimos. El comando `break` abandonará inmediatamente del bucle actual, pero continuará con el flujo de programa después del fin del bucle en el código fuente. (En otras palabras, continúa el programa en el punto específico donde este iría una vez que el ciclo terminara normalmente.) Esto es diferente al comando `return`, que termina tanto el ciclo como la subrutina.

Un tecnicismo que vale la pena mencionar es que, si tenemos varios bucles anidados, el comando `break` aplica solamente al más interno.

En nuestro código, una vez que nos topamos con el comando `break` o cuando el bucle `while` termina naturalmente, ejecutamos algunas sentencias `print` si el parámetro opcional `verboso` es `True` y entonces retornamos la respuesta final (líneas 37–41).

5.6.4 El impacto de las raíces repetidas

Finalmente, pero no por ello menos importante, nos gustaría mostrar al lector un fenómeno interesante en el Método de Newton. Con la versión 6 de nuestra subrutina escrita, definamos los siguientes dos polinomios:

Código de Sage

```

1  f(x) = (x-1)^5
2  g(x) = (x-1) * (x-10) * (x-20) * (x+10) * (x+20)

```

A continuación, intentemos

Código de Sage

```

1  metodo_newton(f, 3, verboso=True)

```

Notaremos que el valor de $f'(x)$ se hace demasiado pequeño y nuestra subrutina aborta la ejecución. Podemos evitar esta interrupción al fijar el parámetro opcional `peligro` en 10^{-9} en lugar de 10^{-4} , que es el valor por defecto. Entonces, escribimos

Código de Sage

```

1  metodo_newton(f, 3, verboso=True, peligro=10^-9)

```

¡y vemos que 22 iteraciones fueron requeridas! En contraste,

Código de Sage

```
1 metodo_newton(g, 3, verboso=True, peligro=10^-9)
```

nos provee una respuesta en tan solo 4 iteraciones. Es una aproximación de mayor calidad, con $g(x_{ant})$ igual a $-1,75 \cdots \times 10^{-11}$. Por el contrario, $f(x_{ant})$ fue igual a $6,99 \cdots \times 10^{-10}$, considerablemente peor.

Como podemos ver, $f(x)$ tiene una raíz con multiplicidad cinco; mientras tanto, $g(x)$ tiene cinco raíces con multiplicidad uno. Podemos escoger polinomios diferentes, pero una raíz repetida muchas veces tendrá, en efecto, una convergencia más lenta que en un polinomio del mismo grado sin raíces repetidas. Este es un teorema que frecuentemente es demostrado en una clase de *Análisis Numérico*, aunque no siempre en el primer semestre de esa materia. La prueba es muy difícil para incluirla aquí, pero, si el lector experimenta con polinomios con raíces repetidas, verá este fenómeno en acción.

5.6.5 Factorización mediante división al tanteo

Ahora continuamos con nuestro tema de tratar de programar pequeños pedazos de código que realizan tareas similares a las capacidades de Sage. Aquí vamos a intentar la factorización, como el comando `factor` de Sage, pero con un enfoque extremadamente directo: simplemente dividiremos por números primos hasta que hayamos factorizado nuestro objetivo.

El número objetivo es inicialmente el que queremos factorizar. Vamos a dividir el “número objetivo actual” (que se actualiza en cada iteración y por lo tanto puede cambiar) por primos sucesivos, empezando con 2. Si un número primo particular lo divide, lo imprimiremos como parte de la factorización y reemplazaremos el objetivo con el cociente —y *aún no* pasaremos al siguiente primo—. Si un número primo no divide el objetivo, no lo imprimiremos como parte de la factorización y pasaremos al siguiente primo, identificado por el comando `next_prime` de Sage. Nos detendremos cuando el número objetivo sea 1.

Una cuestión que debemos solucionar es cómo decidir si un número es divisible por otro. Los matemáticos suelen usar un truco llamado *aritmética modular*, pero no es conveniente que expliquemos aquí cómo usarla en Sage, pues no todos los lectores conocerán esa área de estudio. Si resulta que el lector en efecto la conoce, entonces sabe que x es divisible por y si y solo si $x \bmod y$ es igual a cero. Si resulta que el lector no conoce la aritmética modular, entonces simplemente confíe en el autor cuando dice que

Código de Sage

```
1 if (objetivo % primo_actual) == 0:
```

es equivalente a preguntar si `objetivo` es divisible por `primo_actual`. Alternativamente, se puede pensar que el operador `%` anterior calcula el residuo de la división entera entre dos números. Por ejemplo, $19 \% 5$ es igual a 4, pues $19 = 5 \times 3 + 4$, mientras que $15 \% 5$ es 0, pues 15 es divisible por 5 sin residuo.

A propósito de esto, la operación “hermana” del residuo entero es la división entera, representada en Sage por el operador `//`, que dará el resultado de una división, pero redondeando hacia abajo al entero más próximo. En otras palabras, $14 // 7$, $15 // 7$ y $20 // 7$ son todos iguales a 2, pero $13 // 7$ es 1 y $21 // 7$ es 3. En cambio, el operador `/` representa la división ordinaria. Si ambas operaciones concuerdan (dan el mismo resultado), entonces eso significa que el denominador divide exactamente el numerador. Por lo tanto, podemos tratar

Código de Sage

```
1 if (a // b) == (a / b):
```

como una forma alternativa a la instrucción no existente, aunque muy deseada, `if ("a es divisible por b"):`.

Una propiedad interesante que relaciona los operadores de división entera `//` y residuo `%` es que $a = b(a//b) + a \% b$, para cualesquiera números enteros a y b .

El código de la subrutina `factorDivisionTanteo` siguiente puede resultar un poco confuso al principio; es una subrutina matemáticamente no trivial y, por lo tanto, vale la pena luchar un poco con ella.

Código de Sage

```

1 def factorDivisionTanteo(x):
2     """Usando división al tanteo, esta subrutina imprimirá la
3     factorización prima de la entrada x. Se asume que x es un
4     entero mayor que 1."""
5
6     print('Calculando la factorización de', x, ': ', end=' ')
7     objetivo = x
8     primo_actual = 2
9
10    while objetivo > 1:
11        if (objetivo % primo_actual) == 0:
12            # objetivo es divisible por primo_actual.
13            print(primo_actual, end=' ')
14            objetivo = objetivo / primo_actual
15        else:
16            # objetivo no es divisible por primo_actual.
17            primo_actual = next_prime(primo_actual)
18
19    print()

```

El lector puede notar que hemos usado la opción `end=' '` en las dos primeras funciones `print` con el objetivo de forzarlas a imprimir un espacio, en lugar del salto de línea usual al final del texto. De esta manera, todos los factores de la entrada se muestran en una misma línea de texto, separados por un espacio cada uno. Por otro lado, la función `print` al final de la subrutina añade un salto de línea final, lo que nos permitirá realizar múltiples experimentos a la vez, con los resultados de cada uno mostrados por separado. Ya habíamos discutido a detalle la opción `end` en la subsección 5.1.2 en la página 211.

Probemos nuestro código con las siguientes instrucciones:

Código de Sage

```

1 factorDivisionTanteo(25)
2 factorDivisionTanteo(26)
3 factorDivisionTanteo(27)

```

Obtenemos las salidas de alta calidad siguientes:

```

Calculando la factorización de 25 : 5 5
Calculando la factorización de 26 : 2 13
Calculando la factorización de 27 : 3 3 3

```

El lector puede intentar remover la función `print` solitaria al final de la subrutina y probar nuevamente con las tres líneas de código anterior, solo para ver qué ocurre. La salida de menor calidad demostrará por qué se requiere ese `print()`.

Dado que la subrutina representa un procedimiento matemáticamente no trivial, para poder comprender mejor el código anterior, usemos una técnica llamada *prueba de escritorio*. Básicamente, esta consiste en tomar un ejemplo particular y ejecutar el código manualmente, instrucción por instrucción, llevando nota del resultado de cada paso. Esto debería darnos una mayor comprensión del funcionamiento de nuestra subrutina. Hagamos la prueba de escritorio para `factorDivisionTanteo(20)` como sigue:

Calculando la factorización de 20 :

objetivo = x	objetivo = 20
primo_actual = 2	primo_actual = 2
objetivo > 1	True
objetivo% primo_actual	20% 2 = 0
(objetivo% primo_actual) == 0	True

Calculando la factorización de 20 : 2

objetivo = objetivo / primo_actual	objetivo = 20 / 2 = 10
objetivo > 1	True
objetivo% primo_actual	10% 2 = 0
(objetivo% primo_actual) == 0	True

Calculando la factorización de 20 : 2 2

objetivo = objetivo / primo_actual	objetivo = 10 / 2 = 5
objetivo > 1	True
objetivo% primo_actual	5% 2 = 1
(objetivo% primo_actual) == 0	False
primo_actual = next_prime(primo_actual)	primo_actual = next_prime(2) = 3
objetivo > 1	True
objetivo% primo_actual	5% 3 = 2
(objetivo% primo_actual) == 0	False
primo_actual = next_prime(primo_actual)	primo_actual = next_prime(3) = 5
objetivo > 1	True
objetivo% primo_actual	5% 5 = 0
(objetivo% primo_actual) == 0	True

Calculando la factorización de 20 : 2 2 5

objetivo = objetivo / primo_actual	objetivo = 5 / 5 = 1
objetivo > 1	False

Una prueba aun mejor de nuestra subrutina se haría usando un puñado de enteros consecutivos, como abajo:

Código de Sage

```
1 for y in range(19, 50):
2     factorDivisionTanteo(y)
```

Si el lector intenta esta prueba, verá que nuestro código funciona bien.

Sin embargo, cabe notar que la subrutina no funciona correctamente con una entrada no positiva. Tendremos que corregir eso. Por ejemplo, solamente se imprime “Calculando la factorización de -10 :” si intentamos

Código de Sage

```
1 factorDivisionTanteo(-10)
```

Algo similar ocurre con cero.

5.6.6 Minidesafío: División al tanteo, deteniendo antes

Nuestra subrutina es muy ineficiente en el sentido que, si le preguntamos acerca de 1 000 003 (que resulta ser primo), esta tiene que verificar todos los primos, 2, 3, 5, 7, ..., 999 983, 1 000 003, solo terminando cuando alcanza este último. El siguiente lema nos ayudará con esto:

Si un entero positivo n no es divisible por ninguno de los números primos menores o iguales que \sqrt{n} , entonces n debe ser primo.

Demostración. Supongamos que n no es primo. Entonces consideremos la factorización prima de n , es decir

$$n = (p_1)(p_2)(p_3) \cdots (p_\ell),$$

donde $\ell \geq 2$, pues n no es primo. Sabemos que no existe un primo que divida n , que sea menor o igual que \sqrt{n} , así que tenemos que

$$\begin{aligned} p_1 &> \sqrt{n}, \\ p_2 &> \sqrt{n}, \\ p_3 &> \sqrt{n}, \\ &\vdots \\ p_\ell &> \sqrt{n}. \end{aligned}$$

Combinando todo esto (que solo es permitido porque todos los números involucrados son positivos) nos lleva a:

$$\begin{aligned} (p_1)(p_2)(p_3) \cdots (p_\ell) &> (\sqrt{n})^\ell \\ (p_1)(p_2)(p_3) \cdots (p_\ell) &> n^{\ell/2} \\ n &> n^{\ell/2} \end{aligned}$$

Dado que $\ell \geq 2$, entonces $\ell/2 \geq 1$ y $n^{\ell/2} > n^1$. Ahora tenemos

$$n > n^{\ell/2} \quad \text{y} \quad n^{\ell/2} > n \quad \text{implicando} \quad n > n,$$

que es claramente una contradicción. Por lo tanto, nuestra suposición es falsa, y n es un número primo. \square

En conclusión, debemos verificar si `primo_actual` es mayor que la raíz cuadrada de `objetivo`. Si es así, entonces podemos detener libremente la ejecución del bucle `while` pues `objetivo` es primo. Este desafío consiste en modificar el código existente de arriba para hacer aprovechar este atajo. Un buen caso para experimentar podría ser

$$x = 1\,000\,730\,021.$$

Aunque estamos haciendo una mejora de nuestra subrutina, aún debemos arreglar la inhabilidad de factorizar números no positivos. Como sabemos que podemos factorizar enteros positivos, una solución sencilla se presenta por sí misma. Dado un entero negativo x , deberíamos factorizar $-x$ en su lugar, que claramente será positivo. Deberemos poner “-1” en frente de la factorización para reflejar el hecho que el número original es negativo. Finalmente, si algún usuario pregunta por la factorización de 0, entonces debemos levantar una excepción. Por ejemplo, si usamos el comando predefinido de Sage, y preguntamos `factor(0)`, entonces obtendremos un mensaje de error, terminando con

```
ArithmeticError: factorization of 0 is not defined
```

5.6.7 Desafío: Una caja registradora mejorada

Ahora vamos a mejorar el código que escribimos en el desafío de la subsección 5.2.2, que simula una caja registradora. En lugar de decirle al usuario que su cambio es \$4,75, se debe indicar que el cambio es cuatro billetes de un dólar y tres monedas de un cuarto. Software de este tipo es requerido en cajas registradoras en las que las monedas saltan de un lado de la máquina a una bandeja de donde el comprador las recoge. Estas registradoras, aunque costosas, se han vuelto muy populares en las tiendas durante las últimas décadas, pues mucha gente que, en otro caso estaría trabajando en las cajas registradoras, son incapaces (!) de determinar los cambios correctamente. Es un poco confuso para algunos calcular cómo hacer esto con objeto de entregar la mínima cantidad posible de monedas. Ese es un objetivo que vale la pena realizar, de manera que la máquina no debe ser vuelta a llenar con monedas muy frecuentemente. También, el comprador estaría molesto si fuese a recibir 475 centavos de cambio.

Aquí tenemos un caso abreviado: Si sabemos que debemos devolver \$3,90 de cambio, solo como ejemplo, deberíamos primero dar tantos billetes de dólar como podamos (en este caso, 3 billetes, pues \$4 es demasiado). Ahora debemos dar 90 centavos, y deberíamos entregar tantos cuartos como podamos (en este caso, 3 cuartos, pues equivalen a 75 centavos que está bien, pero \$1,00 es demasiado). Pues bien, ahora quedan 15 centavos, de manera que deberíamos entregar tantos décimos como podamos (en este caso, solo 1, pues \$0,20 es demasiado). Finalmente, quedan 5 centavos, y eso es claramente un vigésimo que entregar. Una forma fácil de programar esto es con cinco ciclos `while` (uno para billetes de dólar, uno para cuartos, uno para décimos, uno para vigésimos y uno para céntimos) o mediante el uso astuto del comando `floor` de Sage.

5.7 Cómo funcionan los arreglos y las listas

Hemos tenido muchos encuentros con el concepto de listas a lo largo de este libro. La estructura de lista en Python reemplaza conceptos antiguos como arreglos y listas enlazadas en otros lenguajes de programación —pero si el lector nunca ha programado antes, no tiene que preocuparse por el significado de todo esto por ahora—. Aquí aprenderemos algunos trucos acerca de cómo operar con listas.

Coincidentemente, también tendremos alguna exposición a la forma en que el comando `plot` de hecho genera gráficas. El autor desea expresar agradecimientos especiales a Brian Knaeble por incentivarlo a incluir este subtópico, pues estudiantes de niveles iniciales frecuentemente sienten curiosidad acerca de cómo Sage y las calculadoras grafican funciones.

5.7.1 Listas de puntos y graficación

El pequeño pedazo de código a continuación nos permitirá explorar cómo las computadoras lidian con la graficación. Vamos a trabajar con una función sencilla, $f(x) = x^3 - x$, y escribiremos una subrutina que la dibujará en el intervalo $-3/2 < x < 3/2$. Aquí tenemos el código que analizaremos:

Código de Sage

```

1  f(x) = x^3 - x
2  lista_de_puntos = []
3  verboso = False
4
5  for k in range(-150, 151):
6      x = 0.01 * k
7      y = f(x)
8      nuevo_punto = (x, y)
9      if verboso == True:
10         print(nuevo_punto)
11
12     lista_de_puntos = lista_de_puntos + [nuevo_punto]
13
14  scatter_plot(lista_de_puntos)
```

Primero, definimos nuestra función y luego empezamos con una lista vacía de puntos (esas son nuestras dos primeras líneas). La lista vacía es un par de corchetes sin nada más entre ellos. Entonces, rápidamente definimos una variable `verboso` (en la línea 3), que será conveniente mas adelante. A continuación de esto (empezando en la línea 5), sigue el bucle `for` que hace la mayor parte del trabajo.

Por cada iteración a través del bucle `for`, calculamos una coordenada x . Aunque k tomará los valores enteros de -150 hasta 150 , cada coordenada x estará muy cerca de la siguiente. En efecto, multiplicamos k por $0,01$ (línea 6), forzando que x se mueva desde $-1,5$ hasta $1,5$ en incrementos de $0,01$. La razón por la que $k = 151$ y $x = 1,51$ no son incluidos tiene que ver con cómo funciona el comando `range`, sobre lo que ya discutimos en la subsección 5.1.1 en la página 210. La coordenada y no es nada más que $f(x)$ (línea 7). Entonces (línea 8), construimos un punto, que es un par ordenado, es decir, “ x coma y ”. Puede resultar interesante para el lector cambiar `verboso` a `True` para poder ver la lista. A continuación (línea 12), actualizamos la lista de puntos sumando la antigua lista, con una que consiste del nuevo punto solamente. Finalmente, cuando todos los valores de k han sido procesados y el ciclo `for` ha sido completado, creamos un gráfico de dispersión con el comando `scatter_plot` (línea 14).

La instrucción

Código de Sage

```

12  lista_de_puntos = lista_de_puntos + [nuevo_punto]
```

puede requerir un poco más de explicación. Esta indica a Sage que la nueva lista de puntos debería ser igual a la antigua, más el nuevo punto. El signo de suma que se ha usado ahí resulta en concatenación (o fusión) de las dos listas. Sin embargo, a diferencia de la Teoría de Conjuntos, el orden importa y las entradas duplicadas pueden ocurrir. Exploraremos las diferencias entre conjuntos y listas de Python en poco.

El hecho de que tuvimos que escribir el nombre `lista_de_puntos` dos veces es ligeramente inconveniente, pues este es un poco largo. Para aliviar este trabajo, Python de hecho tiene un atajo:

```
Código de Sage
12 lista_de_puntos.append(nuevo_punto)
```

que añadirá un solo elemento al final de la lista dada.

Ahora la pregunta natural sería: “¿es así como Sage lo hace?”. Lo que acabamos de ver es el algoritmo básico que casi siempre es usado en herramientas como calculadoras graficadoras o viejos programas matemáticos. Sage de hecho toma algunos puntos en pequeños intervalos igualmente espaciados lo largo del eje x , en lugar de solo un punto por intervalo. También, esos puntos representativos dentro del intervalo no están igualmente espaciados, sino que son escogidos aleatoriamente. Ya explicamos el porqué en la sección 3.2. Por otro lado, Sage ocasionalmente detectará que se requieren más valores en ciertos subintervalos, porque la función “enloquece” ahí dentro, y evaluará puntos extra en esos subintervalos.

5.7.2 Desafío: Creando el comando `plot`

Modifiquemos el código que acabamos de escribir para hacerlo más general. Primero, deberíamos convertirlo en una subrutina propiamente dicha usando `def` y un nombre apropiado. Segundo, deberíamos tomar algunas entradas: la función a ser graficada, el punto de inicio y el punto final del intervalo. Tercero, deberíamos permitir un parámetro opcional que indique cuántos puntos desea el usuario sobre el intervalo de graficación. Probablemente 100 sea un buen valor por defecto para ese parámetro.

La parte difícil es construir una fórmula tal que, dados (1) el inicio del intervalo, (2) el final del intervalo, (3) el número de puntos deseado y (4) la variable de iteración del bucle, provea la coordenada x . Esa es la principal dificultad de este desafío.

5.7.3 Operaciones con listas

Como vimos antes, sumar dos listas las concatena:

```
Código de Sage
1 a = [2, 3, 4]
2 b = [1, 3]
3
4 print(a + b)
5 print(b + a)
```

En este caso, obtenemos la siguiente salida:

```
[2, 3, 4, 1, 3]
[1, 3, 2, 3, 4]
```

Aquí podemos ver realmente la diferencia entre las listas de Python y los conjuntos. Primero que nada, en un conjunto matemático, ningún elemento aparece más de una vez. Sin embargo, aquí vemos que las entradas pueden aparecer varias veces. También, en conjuntos matemáticos, el orden no es relevante; en listas de Python sí lo es.

En ocasiones es útil saber la longitud de una lista. Para ello usamos el comando `len`. De hecho, vimos brevemente esto en la página 144, cuando discutimos acerca de determinar cuántos divisores podría tener un número.

```
Código de Sage
1 print(len(a))
2 print(len(b))
```

El operador `in` es generalmente usado con listas, junto con la sentencia `if` para determinar acciones a realizar, dependiendo de la pertenencia de un valor a una lista. Como podemos ver, 2 es parte de nuestra lista `a`, pero no de `b`. De acuerdo a esto, las siguiente líneas devolverán `True` y `False`, respectivamente, como uno esperaría.

Código de Sage

```
1 print(2 in a)
2 print(2 in b)
```

Para acceder a miembros específicos de una lista, usamos corchetes. Los científicos computacionales cuentan desde 0 en lugar de 1, debido a buenas razones, aunque difíciles de explicar, que tienen que ver con ubicaciones en la memoria. Por lo tanto, la primera entrada es la entrada 0, la segunda es la 1, la tercera es la 2, y así sucesivamente. De manera interesante, hay ocasiones en las que requerimos la última entrada o la penúltima o la antepenúltima. Aunque el autor desconoce de otros lenguajes de programación que puedan hacer esto (puede haber algunos), Python nos permite llegar directamente a estas, pidiendo la entrada `-1` en la lista, que es la última, o la `-2`, que es la penúltima, etc. El lector puede usar el código de abajo para probar esta característica:

Código de Sage

```
1 C = [1, 2, 3, 4, 5, 6]
2 print(C[0])
3 print(C[1])
4 print(C[2])
5 print()
6 print(C[-1])
7 print(C[-2])
8 print(C[-3])
```

También podemos remover elementos de una lista. El método `remove` se encarga de eso.

Código de Sage

```
9 C.remove(5)
10 print(C)
```

En ocasiones, en programación matemática, podríamos querer sumar los elementos de una lista. La función `sum` hace esto por nosotros. Mucho menos comúnmente usada, pero en ocasiones útil, es la función `prod`, que multiplicará todos los miembros de una lista. El código de abajo correctamente calcula una suma de 28 y un producto de 5040 para los primeros siete enteros positivos.

Código de Sage

```
1 E = [1, 2, 3, 4, 5, 6, 7]
2 print('Suma:', sum(E))
3 print('Producto:', prod(E))
```

También existe una función `shuffle` en Python, que es muy útil para aleatorizar el orden de una lista. Puede ser útil para crear juegos o en experimentos científicos donde la aleatoriedad es importante por razones estadísticas. El autor la usa para hacer listas tales que puede llamar a sus estudiantes en orden aleatorio y estar seguro que no llamará a alguien dos veces.

Código de Sage

```

1 lista = ['Alice', 'Bob', 'Charlie', 'Diane', 'Edward', 'Francis', 'Greg',
  ↪ 'Harriet']
2 print(lista)
3 shuffle(lista)
4 print(lista)
5 shuffle(lista)
6 print(lista)
7 shuffle(lista)
8 print(lista)
9 shuffle(lista)

```

Este código produjo la siguiente salida en la computadora donde se escribió este libro, aunque el lector podría obtener algo diferente. (Se supone que todo esto es pseudoaleatorio, después de todo.)

```

['Alice', 'Bob', 'Charlie', 'Diane', 'Edward', 'Francis', 'Greg', 'Harriet']
['Francis', 'Diane', 'Harriet', 'Edward', 'Greg', 'Bob', 'Charlie', 'Alice']
['Diane', 'Greg', 'Edward', 'Charlie', 'Alice', 'Bob', 'Francis', 'Harriet']
['Diane', 'Charlie', 'Greg', 'Harriet', 'Alice', 'Francis', 'Edward', 'Bob']

```

Existen muchos comandos para ayudarnos a trabajar con listas en Python. Una excelente referencia puede encontrarse en la dirección URL abajo. Es un libro titulado “Inmersión en Python 3”, dedicado a la enseñanza de Python, que está dirigida a programadores experimentados. Sin embargo, no hará ningún daño que el lector eche un ojo a la sección 2.4 para apreciar cuántos comandos relacionados con listas hay en Python.

<https://github.com/jmgaguilera/inmersionenpython3/releases>

Recomendamos algunos tutoriales más sencillos en “Otras lecturas” en la página 260, que son más adecuados para principiantes.

5.7.4 Bucles a través de un arreglo

En esta subsección exploraremos cómo podemos generar reportes a partir de arreglos. Frecuentemente, uno tiene una parte de un programa encargada de calcular algo, otra parte generando estadísticas de esos datos y, finalmente, una tercera parte para mostrar los resultados en un formato útil. Aquí simularemos una pequeña pieza de un software de administración. Escribiremos una subrutina que genera un reporte de ingresos basado en una lista de nombres y un arreglo de ingresos.

Evidentemente, las listas de nombres e ingresos serán nuestras entradas. Primero, debemos verificar que sean de la misma longitud —y si no, levantaremos una excepción con el comando `raise`—. Entonces recorreremos todos los nombres con un ciclo `for`. Imprimiremos una línea dando el nombre y el ingreso correspondiente. Aquí tenemos el código:

Código de Sage

```

1 def generarReporteIngresos(nombres, ingresos):
2     """Esta subrutina toma dos listas, una de nombres y otra de
3     ingresos. Si estas tiene longitudes distintas, una excepción es
4     levantada. Entonces, un reporte es generado, con cada nombre y el
5     ingreso entregado mostrados en una línea por administrador."""
6     if len(nombres) != len(ingresos):
7         raise RuntimeError('¡La lista de nombres y la lista'
8         + ' de ingresos no tienen la misma longitud! Por'
9         + ' favor, intente nuevamente.')
10
11     for i in range(0, len(nombres)):
12         print(nombres[i], 'entregó', ingresos[i])

```

Abajo tenemos una forma de testar nuestro código:

Código de Sage

```
1 nombres = ['Ned', 'Ed', 'Ted', 'Jed', 'Fred']
2 ingresos = [200*1_000, 300*1_000, 400*1_000, 100*1_000, 600*1_000]
3
4 generarReporteIngresos(nombres, ingresos)
```

Deberíamos recibir la siguiente salida:

```
Ned entregó 200000
Ed entregó 300000
Ted entregó 400000
Jed entregó 100000
Fred entregó 600000
```

Puede existir la duda de por qué verificamos las longitudes de las listas y por qué levantamos una excepción si son distintas. Si `nombres` es más corta que `ingresos`, entonces habrán entradas en esta última que nunca serán accesadas —los ingresos sin nombres asociados—. Si `nombres` es más larga que `ingresos`, entonces nuestro código solicitará una entrada de esta última que no existe —el ingreso de la primera persona sin ingreso asociado— y una excepción será levantada automáticamente por Python (aunque no usemos el comando `raise`).

Minidesafío: Calcular el ingreso total Al autor le gustaría presentar un minidesafío ahora: ¿Puede el lector cambiar la subrutina que acabamos de escribir de manera que también imprima el ingreso total (al final del reporte de ingresos), al terminar de ejecutarse? Se puede elegir hacer esto con un acumulador (véase la subsección 5.1.4 en la página 212) o se puede hacer también con la función `sum` para listas (véase la subsección 5.7.3 en la página 253). ¡La elección es a gusto!

5.7.5 Desafío: Reporte de ganancia de la compañía

Para este desafío, el lector tendrá que escribir una subrutina que acepta tres listas como entradas. La primera es una lista de nombres, segunda es una de ingresos y la tercera es de costos. Estas representan a los administradores de cada división de una corporación, así como los ingresos y egresos de sus divisiones respectivas. La subrutina generará reportes de ganancia trimestrales, que consisten en una lista de nombres seguidos de las ganancias, en una línea propia para cada división. Sin embargo, si las listas ingresadas no tienen longitudes iguales, se debe levantar una excepción. Finalmente, no se debe olvidar la línea del fondo del reporte: la ganancia total.

5.7.6 Promediando un puñado de números

En esta subsección escribiremos una subrutina que promedie las calificaciones de los estudiantes, pero queremos descartar la nota más baja. Como esta es una práctica poco común fuera de USA, tomaremos un momento para explicar el concepto.

Para nuestros lectores fuera de USA, frecuentemente los estudiantes universitarios en ese país son evaluados con un examen escrito corto cada semana o cada dos semanas, llamado simplemente *prueba*, en lugar de tener solo un examen al final del semestre (o del año) como es estándar —por ejemplo— en UK. De hecho, cuando el autor da clases, pueden haber de 12 a 14 de estas pruebas, un examen final largo y un examen semilargo a medio semestre, llamado *examen de medio término*. Si uno lo piensa por un momento, esto implica una gran diferencia. Si un estudiante toma *Cálculo II* y *Cálculo III* con el autor, este terminará pasando por cerca de 32 eventos tipo examen distintos durante el año académico. En cambio, en el modelo Británico clásico, los estudiantes solo rendirían dos exámenes, o tal vez solo uno.⁸

En cualquier caso, teniendo 13 disponibles, algunos docentes *descartarán* la nota más baja. Esta es una forma de adaptarse al hecho que algún estudiante puede tener un mal día, no durmió bien, o tal vez tiene resaca, el día de una prueba. La nota más baja es excluida de su registro y las restantes 12 notas son promediadas. Para un ejemplo concreto,

⁸Debe notarse que algunas universidades del Reino Unido han adoptado el modelo americano. Más aún, el autor está al tanto de que ciertas universidades en Alemania frecuentemente tienen sesiones de problemas escritos semanalmente en sus clases de matemáticas, que es básicamente el mismo concepto que en el modelo americano.

imaginemos una clase de seis pruebas, donde los puntajes de una persona fueron 100, 80, 90, 70, 80 y 90. En ese caso, el 70 quedará descartado y el promedio será

$$\frac{100 + 80 + 90 + 80 + 90}{5} = \frac{440}{5} = 88,$$

que es equivalente a B+ en USA. Esto es significativamente mejor que 85, el promedio sin descartar notas, que es equivalente a B.

Tenemos el código para esta aplicación a continuación:

Código de Sage

```

1  def promediarNotas(notas):
2      """Esta subrutina toma como entrada una lista de notas entre 0 y
3      100, inclusive. Entonces calculará el promedio de estas, después de
4      descartar el puntaje más bajo."""
5      total = 0
6      mas_baja = 101
7
8      for n in notas:
9          if (n < 0) or (n > 100):
10             raise RuntimeError('¡Lista no válida de notas!')
11
12             total = total + n
13
14             if n < mas_baja:
15                 mas_baja = n
16
17     numerador = total - mas_baja
18     denominador = len(notas) - 1
19
20     return N(numerador / denominador, digits=3)
```

Al principio, uno puede estar sorprendido por la línea

Código de Sage

```

8  for n in notas:
```

que es un tipo de sintaxis del bucle `for` que, aunque no hemos manejado mucho en este libro, sí lo habíamos explicado en la subsección 5.1.3 en la página 212. Aquí, estamos diciendo que `n` debe tomar el valor de cada uno de los miembros de la lista `notas` exactamente una vez.

Como podemos ver, nuestro código levanta una excepción si una nota es menor que cero o mayor que cien. Hemos usado la palabra clave `or` (“o”), que significa que la sentencia `if` se ejecutará si *cualquiera* de las condiciones $n < 0$ o $n > 100$ se cumple. En otras situaciones podemos usar la palabra clave `and` (“y”) similarmente, para verificar si *ambas* condiciones se cumplen.

También tenemos un acumulador, llamado `total`. La variable `mas_baja` requiere algo de explicación. Si estamos en medio del bucle de nuestro código y vemos una nota `n` más pequeña que `mas_baja`, entonces necesitamos actualizar esta última con este valor *más bajo* de `n`. Cuando el bucle termina, de seguro que `mas_baja` es el valor más pequeño de `n` con el que nos hemos topado.

Lo único que queda es escoger un valor inicial para `mas_baja` al empezar la subrutina. Escogimos 101 porque ninguna nota puede ser mayor que 100. En otras palabras, si una nota mayor a 100 es encontrada, la subrutina levantará una excepción. Siempre tenemos garantizado encontrar un puntaje en la lista de entrada que es menor que 101. De esta manera, `mas_baja` siempre será un puntaje de la lista —y, de hecho, será el más bajo—.

Podemos testar nuestra subrutina con la siguiente instrucción:

Código de Sage

```

1  promediarNotas([100, 80, 90, 70, 80, 90])
```

Obtenemos el promedio de 88, como se esperaba.

5.7.7 Minidesafío: Promediando con y sin descarte de notas

Una modificación divertida de la subrutina anterior será un buen desafío para el lector. Se debe añadir un parámetro adicional llamado *misericordia*. La nota más baja debe ser descartada si *misericordia* es *True*, y debe ser incluida si es *False*. El valor por defecto debe ser este último.

5.7.8 Minidesafío: Duplicando la nota más alta

Otra modificación al minidesafío anterior consiste en hacer que los profesores misericordiosos tomen en cuenta la nota más alta como doble, en lugar de descartar la más baja. El lector puede intentar eso.

5.7.9 Desafío: El procesamiento del fin de semestre, parte 3

Aquí tenemos un excelente ejemplo de cómo pequeñas subrutinas pasan a formar parte de subrutinas más grandes y, eventualmente, de programas. Consideremos los códigos que escribimos en las subsecciones 5.5.3 y 5.5.4. Ahora se nos pide escribir una subrutina que tomará una lista con los nombres de los estudiantes y una lista con sus notas (en una escala de 0 a 100, como antes). Si estas tienen longitudes distintas, se debe levantar una excepción; si tienen el mismo tamaño, llamamos a nuestras subrutinas previamente programadas, en las subsecciones mencionadas, para cada estudiante en la lista. Un reporte debe ser generado, mostrando el nombre del estudiante, su promedio y, finalmente, su destino. Es probable que el lector deba modificar ligeramente los viejos códigos para lograr esto.

Si el lector se encuentra en una región con una escala de notas diferente a la propuesta aquí, puede adaptar este desafío a las reglas locales.

5.7.10 Algo útil: Devolviendo solamente raíces reales, racionales o enteras

Previamente, en la sección 1.8 en la página 35, aprendimos cómo usar el comando `solve` para hallar las raíces de un polinomio. Naturalmente, este incluye raíces que son números complejos. Sin embargo, en ocasiones solo deseamos las que son reales, o racionales, o enteras.

El lector probablemente ha visto una notación que luce como \mathbb{C} , \mathbb{R} , \mathbb{Q} y \mathbb{Z} , para representar los conjuntos de números complejos, reales, racionales y enteros, respectivamente. Para representarlos, Sage tiene la notación `CC`, `RR`, `QQ` y `ZZ`. Podemos usar el comando `in` como si estos fuesen listas de Python, aun cuando en realidad son conjuntos infinitos.⁹

Por ejemplo, si estuviésemos interesados en

$$f(x) = x^9 - 10x^8 + 40x^7 - 100x^6 + 203x^5 - 290x^4 + 260x^3 - 200x^2 + 96x,$$

podríamos escribir

Código de Sage

```
1 f(x) = x^9-10*x^8+40*x^7-100*x^6+203*x^5-290*x^4+260*x^3-200*x^2+96*x
2 respuestas = solve(f(x)==0, x)
3
4 for x in respuestas:
5     print(x)
```

Esto imprimiría todas las nueve raíces, es decir

$$\{0, 1, 2, 3, 4, i, 2i, -2i, -i\}.$$

⁹De hecho, si el lector conoce acerca de los diferentes tamaños de infinito, \mathbb{R} y \mathbb{C} tienen el tamaño más grande entre estos conjuntos, el “infinito no contable”, a veces llamado \aleph_1 . Por otro lado, \mathbb{Q} y \mathbb{Z} tienen el tamaño más pequeño de entre todos los infinitos, el “infinito contable”, a veces llamado \aleph_0 . El símbolo \aleph es la letra Hebrea “aleph”.

Pero tal vez no queremos ver las raíces imaginarias. Podemos intentar lo siguiente en su lugar

Código de Sage

```
1 f(x) = x^9-10*x^8+40*x^7-100*x^6+203*x^5-290*x^4+260*x^3-200*x^2+96*x
2 respuestas = solve(f(x)==0, x)
3
4 for x in respuestas:
5     if x.rhs() in RR:
6         print(x)
```

lo que nos restringirá a la raíces reales solamente, excluyendo las cuatro puramente imaginarias.

A propósito de este código, `rhs()` significa “right hand side” (“lado derecho”). El lado izquierdo puede accederse con `lhs()`, que es abreviación de “left hand side”, pero esto sería inútil en este caso, pues el lado izquierdo es simplemente x .

Ahora cambiemos $f(x)$ por $g(x)$ en el código anterior, donde

$$g(x) = 3x^7 - 2x^6 - 15x^5 + 10x^4 + 6x^3 - 4x^2 + 24x - 16,$$

teniendo cuidado de hacer los cambios dos veces —una vez en la definición del polinomio y una vez en el comando `solve`—. Si mantenemos el `RR`, veremos solamente cinco raíces. Sin embargo, si cambiamos `RR` por `CC`, entonces tendremos las siete raíces, es decir

$$\left\{2, i, \sqrt{2}, \frac{2}{3}, -i, -\sqrt{2}, -2\right\}.$$

Por otro lado, podríamos usar `QQ`, con lo que veríamos las tres raíces racionales, es decir $\{2, 2/3, -2\}$. Finalmente, si usamos `ZZ`, entonces obtenemos las dos raíces enteras, es decir $\{2, -2\}$.

Podríamos reemplazar $g(x)$ con

$$h(x) = x^4 - x^3 - x^2 - x - 1$$

e intentar el código anterior usando `CC`. Obtenemos cuatro raíces algebraicas explícitas (con un formato perturbadoramente complicado). Una de esas raíces está dada en la figura 5.12, y las otras tienen estructuras similares. Si cambiamos `CC` por `RR`, entonces descubriremos que dos de esas raíces son reales, lo que significa que las dos restantes son complejas. Finalmente, si cambiamos `RR` por `QQ`, observaremos que ninguna es racional.

$$\left(\frac{-1}{12}\right)\sqrt[6]{\frac{\left(\frac{1}{2}\right)^{\frac{2}{3}}\left(4\left(\frac{1}{2}\right)^{\frac{2}{3}}\left(3\sqrt[3]{563\sqrt{3}-65}\right)^{\frac{2}{3}}+11\left(\frac{1}{2}\right)^{\frac{2}{3}}\left(3\sqrt[3]{563\sqrt{3}-65}\right)^{\frac{1}{3}}-56\right)}{\left(3\sqrt[3]{563\sqrt{3}-65}\right)^{\frac{2}{3}}}}+\left(\frac{-1}{2}\right)\sqrt[6]{\frac{\left(\frac{-1}{3}\right)\left(\frac{1}{2}\right)^{\frac{2}{3}}\left(3\sqrt[3]{563\sqrt{3}-65}\right)^{\frac{2}{3}}-\frac{13\sqrt[6]{6}\left(\frac{1}{2}\right)^{\frac{2}{3}}}{2\sqrt[3]{\frac{4\left(\frac{1}{2}\right)^{\frac{2}{3}}\left(3\sqrt[3]{563\sqrt{3}-65}\right)^{\frac{2}{3}}+11\left(\frac{1}{2}\right)^{\frac{2}{3}}\left(3\sqrt[3]{563\sqrt{3}-65}\right)^{\frac{1}{3}}-56}}+\frac{28\left(\frac{1}{2}\right)^{\frac{2}{3}}}{3\left(3\sqrt[3]{563\sqrt{3}-65}\right)^{\frac{2}{3}}}}+\frac{11}{6}+\frac{1}{4}}{\left(3\sqrt[3]{563\sqrt{3}-65}\right)^{\frac{2}{3}}}}$$

Figura 5.12 Una de las raíces de $h(x)$ en la subsección 5.7.10

5.7.11 Notación alternativa para listas

Sage cuenta con un método alternativo para crear listas, que puede resultar muy útil. Supongamos que deseamos crear una lista de los primeros 15 números enteros positivos. Podemos escribir

Código de Sage

```
1 [1, ..., 15]
```

lo que nos devolverá

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Pero Sage es aún más flexible. Por ejemplo, si deseamos ser más explícitos, como alternativa al comando anterior, podemos escribir

Código de Sage

```
1 [1, 2, 3, ..., 15]
```

y Sage entenderá lo que queremos decir.

Por supuesto, esta notación no sería muy útil si solo pudiéramos crear listas de elementos que están separados por una unidad. En efecto, es posible especificar un “paso” para incrementar los elementos. Consideremos el siguiente comando:

Código de Sage

```
1 [1, ..., 15, step=3]
```

Este nos mostrará

```
[1, 4, 7, 10, 13]
```

Y naturalmente, podemos especificar un paso negativo si queremos una lista decreciente:

Código de Sage

```
1 [15, ..., 1, step=-3]
```

devolverá la lista

```
[15, 12, 9, 6, 3]
```

Por otro lado, usando este mismo método, ¿es posible definir listas de números con un paso que no es necesariamente un entero! Esta es una ventaja de esta notación con respecto al comando `range`, que solo nos permite usar pasos enteros. Por ejemplo,

Código de Sage

```
1 [0, ..., 1, step=0.1]
```

nos devolverá los números entre cero y uno, inclusive, separados por una distancia de 0,1:

```
[0.0000000000000000, 0.1000000000000000, 0.2000000000000000, 0.3000000000000000,
0.4000000000000000, 0.5000000000000000, 0.6000000000000000, 0.7000000000000000,
0.8000000000000000, 0.9000000000000000, 1.0000000000000000]
```

Finalmente, podemos empezar una lista con números arbitrarios:

Código de Sage

```
1 [2/7, -2/3, 3/4, -1, 1, ..., 15, step=3]
```

Esto nos dará como resultado:

```
[2/7, -2/3, 3/4, -1, 1, 4, 7, 10, 13]
```

Estas listas, como cualquier otra, pueden ser usadas en bucles como

Código de Sage

```
1 for n in [0, ..., 1, step=0.1]:
2     print(n)
```

Minidesafío: Creando una tabla de costo por mil En la subsección 4.2.1 definimos la función `costoPorMil1`. Usando el conocimiento adquirido en este capítulo y el nuevo método de construcción de listas que hemos aprendido, el lector puede escribir una subrutina “`tablaCostoPorMil`”, que acepte como datos el número máximo de años T , la tasa de interés anual mínima $rmin$ y la tasa de interés anual máxima $rmax$, e imprima una tabla de costo por mil para préstamos de un año, dos años, tres años, hasta T años, con tasas de interés entre $rmin$ y $rmax$ con separación de 0,05 (es decir, un 5%).

A continuación, mostramos la tabla de costo por mil para $T = 3$ años, con $r_{min} = 0,3$ y $r_{max} = 0,55$:

Año: t =1	interés: r = 0.3000000000000000	CPT: 97.4871269883382
Año: t =1	interés: r = 0.3500000000000000	CPT: 99.9629873361132
Año: t =1	interés: r = 0.4000000000000000	CPT: 102.471482861710
Año: t =1	interés: r = 0.4500000000000000	CPT: 105.012300979526
Año: t =1	interés: r = 0.5000000000000000	CPT: 107.585118538978
Año: t =1	interés: r = 0.5500000000000000	CPT: 110.189602401911
Año: t =2	interés: r = 0.3000000000000000	CPT: 55.9128203601027
Año: t =2	interés: r = 0.3500000000000000	CPT: 58.5186187146798
Año: t =2	interés: r = 0.4000000000000000	CPT: 61.1877355229640
Año: t =2	interés: r = 0.4500000000000000	CPT: 63.9189033436050
Año: t =2	interés: r = 0.5000000000000000	CPT: 66.7107754820753
Año: t =2	interés: r = 0.5500000000000000	CPT: 69.5619344943605
Año: t =3	interés: r = 0.3000000000000000	CPT: 42.4515767434827
Año: t =3	interés: r = 0.3500000000000000	CPT: 45.2360413217689
Año: t =3	interés: r = 0.4000000000000000	CPT: 48.1101508692478
Año: t =3	interés: r = 0.4500000000000000	CPT: 51.0706049660580
Año: t =3	interés: r = 0.5000000000000000	CPT: 54.1139146418230
Año: t =3	interés: r = 0.5500000000000000	CPT: 57.2364450859662

Como se puede observar, los resultados tienen una precisión demasiado alta. ¿Puede el lector reducir la precisión para tener resultados más legibles?

5.8 ¿A dónde ir a partir de aquí?

Es la esperanza del autor que este capítulo haya sido útil y tal vez incluso divertido para el lector. Aprender a programar puede abrir nuevos mundos llenos de oportunidades en investigación, academia o industria. También puede incrementar dramáticamente el salario de uno.

5.8.1 Otros recursos sobre programación en Python

Aquí tenemos algunos recursos que permitirán al lector incrementar sus habilidades de programación.

- El libro “Inmersión en Python 3”, de Mark Pilgrim, traducido al español por José Miguel González Aguilera, es un excelente recurso, disponible en formatos electrónico y físico, e incluso cuenta con su propia página web (en inglés). Sin embargo, está dirigido a programadores experimentados, que conocen otros lenguajes de programación, pero no Python. Está disponible para descarga gratuita en la URL

<https://github.com/jmgaguilera/inmersionenpython3/releases>.

- Otro libro muy recomendado es *Aprenda a pensar como un programador con Python*, por Allen Downey, Jeffrey Elkner y Chris Meyers, y traducido al español por varios. Está disponible para descarga gratuita en la URL

<https://argentinaenpython.com/quiero-aprender-python/>,

- Un gran libro electrónico para aprender Python es *El tutorial de Python*, por Guido van Rossum, el genio creador de ese lenguaje de programación. Está disponible en español gracias a la comunidad Python Argentina en la URL

<https://argentinaenpython.com/quiero-aprender-python/>,

junto con otros muchos recursos útiles.

Nota: Ni el lector ni el traductor están en posición de juzgar entre estos tres recursos.

- Un sitio web extremadamente útil para el principiante en programación es

<http://www.pythontutor.com/>

Está disponible para aprender ya sea en Python, Java, C, C++ y otros. En particular, para trabajar con Python, podemos ingresar directamente a

<http://www.pythontutor.com/visualize.html#mode=edit>

Ahí veremos una celda muy similar a la del servidor Sage Cell, donde podemos escribir o copiar un programa. Entonces, si presionamos el botón “Visualize Execution” (“Visualizar Ejecución”), se iniciará la ejecución en modo interactivo. A la izquierda veremos nuestro código con una flecha verde, que nos muestra la instrucción que se acaba de ejecutar, y una flecha roja, indicándonos qué instrucción se ejecutará a continuación. Justo por debajo tenemos los botones “Forward” (“Adelante”) y “Back” (“Atrás”), que nos permiten avanzar o retroceder, respectivamente, un paso en la ejecución. A la derecha tendremos disponible el resultado de llevar a cabo la instrucción recién ejecutada.

Esto es lo que conocimos como una *prueba de escritorio* en la subsección 5.6.5.¹⁰ Esto es útil para el lector curioso que desea saber qué hace la computadora cuando se le pide correr un programa, cómo es el flujo del programa, cómo realiza las operaciones que pedimos.

- Finalmente, pero no por eso menos importante, la página web oficial de Sage cuenta con recursos en español para aprender Sage. Pueden encontrarse en

<http://www.sagemath.org/es/>

5.8.2 ¿Qué hemos dejado fuera de este libro?

Dependiendo de la universidad, hemos cubierto aproximadamente la mitad de lo que normalmente se enseña en una clase de *Ciencia Computacional I* de un semestre. Lo que sigue es un resumen de lo que normalmente se enseñaría durante la segunda mitad, para aquellos que sientan curiosidad.

Tipos de datos: Los datos en programación computacional pueden venir en varias formas. Enteros, números de punto flotante (que son aproximaciones de los números reales) y cadenas (texto entre comillas triples, dobles o simples) son los más comunes. En la mayoría de los lenguajes de programación, uno debe declarar sus variables y especificar sus tipos de datos. Las declaraciones de variables no son difíciles o confusas, pero son importantes porque nuestros programas no se ejecutarán sin ellas en dichos lenguajes. Pueden existir cuestiones sutiles acerca de la elección del tipo de dato correcto.

Operadores lógicos: Vimos el uso de `or` en la subsección 5.7.6, pero un curso típico ofrecería algunas situaciones más desafiantes donde el uso ingenioso de `or` y `and` es importante.

Capturando excepciones que son levantadas: Normalmente, cuando un comando `raise` es encontrado, significa que algún tipo de error catastrófico ha ocurrido y que el programa debe terminar con un mensaje de error, informando al usuario acerca de qué salió mal. Sin embargo, existe otra forma de lidiar con el comando `raise`. Usando la construcción `try-except`, es posible que el código que llama a la subrutina que produce la ofensa pueda remediar la situación. El programa no es detenido, sino que en cambio, un código especial para lidiar con el problema es ejecutado. Desafortunadamente, no podemos cubrir esos detalles aquí.

Sentencias `if-then-else` anidadas: La construcción `if-then-else` es una excelente forma de lidiar con dos posibilidades. Para tres, cuatro o más, uno puede hacer una cadena de estas con una cláusula “else if”. En Python, el comando `elif` es una abreviación para esta.

Manejando sentencias `case/switch`: Para situaciones donde puede haber una docena o más de posibilidades, una larga cadena de comandos `if-then-elif-elif-elif-else` sería muy confusa de leer y depurar. La mayoría de los lenguajes de programación tienen una forma de separar “casos”. Conocer esto puede ser muy útil.

Alcance: La breve discusión en la página 220 sobre “variables locales” versus “variables globales” es de hecho solo una pequeña probada de un tema moderadamente largo, llamado “alcance”. Entender cómo funciona el alcance de las variables puede ser importante al depurar programas de computadora muy complejos.

¹⁰La prueba de escritorio es un método para entender y verificar nuestro código, ejecutando cada instrucción y anotando el resultado correspondiente.

Recursividad: Tanto en matemáticas como en ciencia computacional, la recursividad se refiere a cuando una función o una subrutina se llama a sí misma. Por ejemplo, $n! = (n)(n-1)!$ o $x^n = (x)(x^{n-1})$. Esto no es muy distinto a las pruebas por inducción. Los ejemplos que tenemos aquí son muy sencillos, pero existen otros más sofisticados y elegantes, que pueden ser matemática y computacionalmente poderosos.

Trabajando con cadenas: Hemos trabajado bastante con datos numéricos a lo largo de este capítulo. Similarmente, uno puede trabajar con cadenas (texto). Se puede buscar cadenas, hacer sustituciones, darles un formato elegante y todo tipo de tareas interesantes. Esta es la forma en que los procesadores de texto funcionan.

Aserciones: Estas constituyen una excelente forma de depurar algún pedazo de código que no está trabajando de la forma en que se suponía. Una aserción es una sentencia lógica, algo similar a lo que daríamos a una sentencia `if`, pero es algo de lo que uno está relativamente seguro que debe ser verdadero, en el momento en que un comando `assert` es ejecutado. Por ejemplo, `assert x>3`. Si resulta que esta afirmación es falsa, el programa inmediatamente se detiene por completo y somos notificados que una aserción ha fallado. Este es un método muy eficaz para encontrar situaciones donde hemos asumido que algo debería ser cierto, pero en la práctica no siempre lo es.

Otras técnicas de depuración: Aprender a depurar programas es la tarea más vital de todas para un programador. Esto consume la vasta mayoría del tiempo de desarrollo en la práctica. La mayoría de los programadores desarrollan esta habilidad lentamente a lo largo del tiempo, pero puede ser un reto divertido. De hecho, algunas preguntas interesantes para un examen pueden presentar a un estudiante una tarea y un pedazo de código que *casi* la cumple, pero no en realidad. El estudiante debe identificar el error y postular cómo modificar el código para que opere correctamente.

Actuar como intérprete/compilador: Otro excelente ejercicio es dar a un estudiante un pedazo de código y preguntarle “¿qué hace esto?”. El estudiante debe pretender ser una computadora y recorrer el código siguiendo el flujo de control. Esto es lo que conocimos como *prueba de escritorio* en la subsección 5.6.5, aunque solo presentamos una introducción muy elemental. Esta es también una excelente forma de aprender y comprender nuevos y complicados algoritmos. También es un método honrado por el tiempo para depurar un programa: uno puede verificar su código ejecutando manualmente cada instrucción paso a paso con papel y lápiz, y detectar así si existen o no errores.

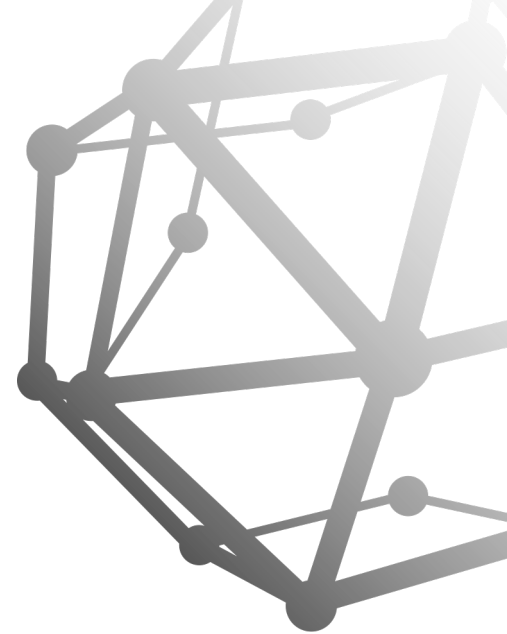
Simulaciones: Generalmente, una simulación modela algún fenómeno complicado en ingeniería, física, química, biología o, incluso, sociología. ¿Cómo puede una entidad tan compleja ser modelada en una computadora? La idea es “rebanar” la circunstancia estudiada en muchos (tal vez millones) de pequeños pedazos de tiempo. Cada rebanada puede ser modelada con un modelo que no necesita ser impresionantemente preciso, tal como una colección de funciones lineales. Como cada corte es pequeño, y existen muchos de ellos, la simulación aún puede producir una salida de alta fidelidad. Este es básicamente el concepto de integral aplicado a la ciencia computacional. Muchas situaciones imposibles de tratar analíticamente pueden ser enfocadas y estudiadas de esta manera. El arte de las simulaciones es la más sencilla y la más útil de las ideas de esta lista.

Nota: Si un sistema es dividido en pequeñas regiones del espacio en lugar del tiempo, esto se llama el “Método de Elementos Finitos”, y es la más poderosa técnica en uso hoy en día para situaciones complejas de ingeniería. Muchos logros tecnológicos de las últimas tres décadas solo han sido posibles debido al Método de Elementos Finitos. Lastimosamente, este es casi siempre enseñado solamente en posgrado y no a estudiantes de pregrado, y el autor no comprende por qué.

Programación orientada a objetos: Es difícil describir la programación orientada a objetos en pocas palabras. Sin embargo, trataremos de darle un pellizco al tema. En este punto, el lector sabe qué son datos y lo que una subrutina hace. Una colección de subrutinas (llamadas *métodos*) y variables que contienen datos (llamadas *campos*) son puestas juntas para formar una *clase*. Por ejemplo, en el sistema de procesamiento de fin de semestre podría haber una clase llamada “Estudiante”, con variables para el nombre, el promedio y un número único de identificación. Entonces, existirían muchos objetos del tipo Estudiante, uno para cada alumno en la universidad. Dado que los datos y el código están íntimamente ligados, uno puede realizar eficientemente muchas tareas que de otra manera serían difíciles y tediosas.

Naturalmente, para universidades que utilizan trimestres o cuatrimestres, la división de temas sería algo diferente a la que se presenta aquí.

Más importante que cualquiera de estos puntos, en una clase real de Ciencia Computacional, se nos daría una batería de ejemplos para estudiar y ejercicios para programar. Estos son fenomenalmente valiosos y son una parte importante del desarrollo de un programador.



6

Construyendo páginas web interactivas con Sage

En este capítulo vamos a aprender cómo construir una página web interactiva, en ocasiones llamada simplemente “interactivo”, “app” o “applet”, que puede ser publicada en la web. A diferencia de otros usos de Sage, esta característica es particularmente poderosa por una simple razón: cualquier persona con acceso a internet puede encontrar nuestras páginas interactivas, y sin ningún conocimiento de Sage, o incluso sin mucho conocimiento de matemáticas, las personas pueden mover algunos deslizadores y ver cómo cambia una gráfica o cómo algún concepto matemático se desenvuelve ante sus ojos. Dado que muchos estudiantes aprenden visualmente, este concepto puede resultar fenomenalmente útil para aquellos que, de otra manera, estarían luchando para aprender algún tópico de matemáticas.

Por ejemplo, los estudiantes en clases de nivel elemental (en la universidad o incluso en secundaria), pueden recibir una URL y empezar a usar las applets con muy poca intervención del profesor. No se requiere experiencia con Sage o cualquier otro software de álgebra computacional. De hecho, la mayoría de los estudiantes ni siquiera notarán que Sage estuvo involucrado en el proceso.

Personalmente, el autor estuvo involucrado con Sage por 5 años y 10 meses antes de aprender cómo hacer interactivos, durante la vacación de invierno entre 2012–2013. Cuando finalmente lo hizo, estuvo fascinado por lo fácil que es. De haberse dado cuenta de lo sencillo que es construir páginas web interactivas, ciertamente habría empezado muchos años antes.

6.1 Nuestros ejemplos

Este capítulo hará referencia a cinco ejemplos diversos, pero matemáticamente elementales:

Recta tangente: Este será nuestro ejemplo principal. Construiremos este interactivo lentamente, desde cero, conforme vamos recorriendo el cuerpo de este capítulo. El objetivo es graficar una recta móvil, tangente a un simple polinomio de tercer grado. Destacaremos cómo el autor construyó esta applet al usar un proceso de seis etapas.

Onda sinusoidal: Este ejemplo será el desafío primario para el lector. Después que terminemos de explorar el interactivo de la recta tangente, se le pedirá construir esta applet por sí mismo.

Acuario óptimo: En la subsección 5.2.3 en la página 219, desarrollamos una subrutina que calcula el costo de un acuario con base en sus dimensiones. Sería estupendo desarrollar un código que evolucione esta subrutina en una página web interactiva. Resulta que este será un proceso relativamente simple.

Graficadora polinomial: Esta es una applet muy rápida de armar, destinada a mostrar cómo construir menús desplegables y casillas de verificación para nuestros interactivos. Es realmente simple y directo. Tendremos un menú desplegable con una selección de tres polinomios, y uno puede indicar si le gustaría tener un cuadriculado en la gráfica o no, usando una casilla de verificación.

Integral definida: Previamente, el autor pensó que este sería un buen desafío para el lector —el rol ahora tomado por el interactivo “Onda sinusoidal”—. Resulta que existen algunos tecnicismos aquí que deben ser tomados en cuenta, por lo que esta tarea es un poco más difícil de lo que primero se supuso. Esto quedará como un proyecto para el lector particularmente motivado.

Dónde puede encontrarse el código En caso de quedar atascado en el proceso de construcción de alguno de estos proyectos, se puede consultar el código del autor —disponible para cada etapa— en la página

<http://www.sage-para-estudiantes.com/interactivos>.

Hay un archivo zip ahí y un directorio para cada uno de estos cinco proyectos.

6.2 El proceso de seis etapas para construir interactivos

El siguiente proceso multietapa es cómo el autor procede para construir sus interactivos, y esta es la forma en que nuestras explicaciones estarán organizadas. En particular, veremos cómo la applet de la “Recta tangente” fue desarrollada a través de estas etapas en sucesión.

Etapla 0: Desarrollamos el concepto —nos tomamos un tiempo para concretar precisamente qué es lo que queremos que nuestro interactivo haga—. Debemos pensar esto no solo como un diseño, sino como un modelo de comportamiento: si se hace clic aquí, esto ocurre; si se hace clic allá, esta otra cosa pasa, *et cetera*.

Etapla 1: Diseñamos una subrutina en Sage (usando `def`) que represente una ronda, un paso, ciclo o fase de nuestra applet.

Etapla 2: Pulimos esta subrutina hasta que quede “perfecta”, con todas las entradas y salidas como queremos que sean, incluyendo colores y otros detalles.

Etapla 3: Convertimos nuestro código en una subrutina interactiva en CoCalc, para lo cual contamos con los comandos “@interact” y “slider”. Describiremos este proceso en detalle. Es realmente más fácil de lo que suena; el comando `slider` hace todo el trabajo.

Etapla 4: Insertamos nuestra subrutina interactiva en una plantilla de página web. El autor tiene una disponible¹ que ha estado usando por un largo tiempo, en la cual podemos copiar y pegar nuestro código.

Etapla 5: Damos mayor detalle a la página web como haríamos con cualquier otra, añadiendo algunos párrafos acerca de las matemáticas involucradas, unas cuantas instrucciones y tal vez la motivación. Finalmente, subimos esta nueva página a nuestro sitio web de la misma manera que haríamos con cualquier otra página.

Los prerrequisitos para construir interactivos Al escribir este capítulo, el autor ha procurado apuntar a la audiencia más amplia posible. Básicamente, hay cuatro ingredientes necesarios en los conocimientos del lector para que sea capaz de comprender a detalle este material:

- (1) Se asume un conocimiento muy básico de HTML. Solamente se requiere una familiaridad muy mínima con ese lenguaje, suficiente como para que el lector pueda hacer sus propias páginas web (ordinarias, no interactivas).
- (2) Es necesario tener algo de conocimiento general de Sage. Por ejemplo, familiaridad con los contenidos de la mayoría del capítulo 1 de este libro.
- (3) Por otro lado, programaremos en Sage vía Python, por lo que también sería útil leer el capítulo 5. Sin embargo, solo usaremos la instrucción `def` para crear algunas subrutinas, además de usar algo de control de flujo, así que un entendimiento superficial de ese capítulo será suficiente (digamos, las subsecciones 5.1 y 5.2). Más aun, debe notarse que si el lector tiene experiencia con cualquier lenguaje de programación (por ejemplo, C, C++, Java, Perl, Fortran, ...), entonces podrá entender fácilmente el código y deducir lo que ocurre sin ningún conocimiento previo de Python o del capítulo 5.
- (4) Finalmente, el lector debe estar por lo menos un poco familiarizado con la interfaz de CoCalc.

¹Facilitada por el Prof. Jason Grout.

6.3 El interactivo de la recta tangente

Ahora seguiremos la construcción del interactivo “Recta tangente” tal como el autor lo hizo con su método de seis etapas.

Etapla 0: Desarrollo del concepto Antes de realmente programar cualquier cosa, debemos tratar de definir precisamente lo que deseamos que haga nuestro interactivo. Generalmente, el autor tiene tres cosas en mente, que analizaremos a continuación.

Primero, queremos encontrar *el mejor ejemplo posible* para comunicar la idea matemática —no debe ser demasiado complejo ni debe ser demasiado simple—. En este caso, queremos graficar una recta tangente y mostrar a los estudiantes tempranos de cálculo (tal vez con menos de tres semanas en el curso) cómo luce una de tales rectas. Por lo tanto, es pedagógicamente útil escoger que la función a graficar sea extremadamente simple —como un polinomio—. Escogimos la función $f(x) = x^3 - x$ en este caso.

Segundo, tratemos de enfocarnos en qué es lo que el usuario puede cambiar y qué no tiene permitido cambiar. Los aspectos que puede modificar se convertirán en los deslizadores del interactivo. En este caso, el usuario debería ser capaz de especificar la coordenada x del punto donde la recta tangente se dibuja, pero no la coordenada y , que debe ser calculada por nuestro código. De esta forma, el usuario no podrá escoger un punto que no esté sobre la curva $y = x^3 - x$.

Tercero, podemos hacer un bosquejo de cómo lucirá la salida. Debemos preguntarnos cómo podemos exponer óptimamente las circunstancias involucradas para mostrar suficiente detalle —pero no demasiado como para confundir al usuario—.

Ahora que el concepto está finalizado, podemos ingresar a nuestra cuenta de CoCalc y empezar a trabajar.

Etapla 1: Diseñamos una subrutina de Sage El código para la etapa 1 del interactivo de la recta tangente puede encontrarse en la figura 6.1.

Código de Sage

```

1  f(x) = x^3 - x
2  f_prima(x) = diff(f(x), x)
3
4  def tangente_en_punto(x_0):
5      y_0 = f(x_0)
6      m = f_prima(x_0)
7
8      # Dado que  $y - y_0 = m(x - x_0)$ , sabemos que
9      #  $y - y_0 = m*x - m*x_0$ ,
10     #  $y = m*x + y_0 - m*x_0$ .
11     # Por lo tanto,  $b = y_0 - m*x_0$ .
12
13     b = y_0 - m * x_0
14
15     P1 = plot(f(x), -2, 2, color='blue')
16     P2 = plot(m*x + b, -2, 2, color='tan')
17     P3 = point((x_0, y_0), color='red', size=100)
18
19     P = P1 + P2 + P3
20     P.show()
21
22     print('x_0 =', x_0)
23     print('y_0 = f(x_0) =', y_0)
24     print("m = f'(x_0) =", m)
25     print('Recta tangente: y = (', m, ') * x + (', b, ')')
26     print()

```

Figura 6.1 El código para la etapa 1 del interactivo de la “Recta tangente”

En las primeras dos líneas de nuestro programa, primero declaramos la función $f(x) = x^3 - x$ y entonces calculamos su derivada. Hacemos que Sage calcule la derivada por nosotros, aunque de seguro pudimos haber definido $f'(x) = 3x^2 - 1$ directamente. La razón de esto es que posiblemente, más adelante, podríamos querer usar otras funciones, como $f(x) = \sin(x)$, por ejemplo. De la forma en que hemos procedido, excluimos la posibilidad de que podamos olvidar cambiar la derivada de $f'(x) = 3x^2 - 1$ a $f'(x) = \cos(x)$ en ese caso.

Después de eso (línea 4), usamos el comando `def` para definir nuestra subrutina, que muy acertadamente hemos llamado `tangente_en_punto`. Esta es el corazón de la etapa 1 de este proceso para construir una applet; esta representará el núcleo del interactivo entero. Nuestra subrutina evolucionará lentamente hasta convertirse en una página web interactiva funcional.

Naturalmente, una recta tangente puede ser producida cuando se tienen tres ingredientes: la pendiente, la coordenada x y la coordenada y . La coordenada x vendrá de un deslizador cuando el interactivo esté completo. Pero por ahora es un número real que recibimos como entrada (la variable x_0). La mayoría de los interactivos tienen varios deslizadores cuando necesitan más parámetros.

Tenemos que calcular la coordenada y y la pendiente nosotros mismos en nuestra subrutina. Esos números resultan de evaluar $f(x)$ y $f'(x)$ en x_0 , respectivamente (líneas 5 y 6). Una vez equipados con la pendiente de la recta tangente, necesitamos calcular su ordenada en el origen. La forma en la que el autor enseña esto es hacer que sus estudiantes tomen la fórmula punto-pendiente

$$y - y_0 = m(x - x_0)$$

y entonces usen álgebra para convertir esta a la forma $y = mx + b$. Sin embargo, cuando se escribe un programa de computadora, uno debe tener una fórmula explícita para la ordenada en el origen, aquí denotada por b .

Las líneas 8 a 11 del código son *comentarios* y no tienen ningún efecto para Sage o Python. Se usan para indicar al usuario cuál *debería* ser el efecto del pedazo de código siguiente, según piensa el programador. Tengamos siempre en cuenta que no son las palabras infalibles de un ser omnisciente, sino una declaración de las intenciones del programador. Cualesquiera líneas que empiecen con `#` son comentarios y serán ignorados por la computadora. Aquí, hemos usado comentarios en esas líneas para obtener la fórmula $b = y_0 - mx_0$.

Dos comandos `plot` y un comando `point` vienen a continuación en el código (líneas 15–17). Primero, queremos graficar la función $f(x)$ en el intervalo $-2 \leq x \leq 2$, y hemos escogido el color azul para ello. Segundo, queremos graficar la recta tangente $y = mx + b$ en el mismo intervalo, y hemos escogido el color canela (“tan”, en inglés), solo porque la palabra “tangente” empieza con esas letras. Adicionalmente, hemos añadido un gran punto rojo en la intersección, para ayudar al usuario a dirigir su mirada. Hemos elegido el color rojo porque usualmente indica algo importante. Recordemos que la superposición de gráficas en Sage se logra mediante una “suma”. Por lo tanto, a continuación (en la línea 19), hemos sumado las tres gráficas y luego las hemos mostrado (en la línea 20).

No es mala idea además dar algunos detalles sobre el resultado al usuario, especialmente si esta applet es para propósitos educativos. Con eso en mente, en las líneas 22–25, mostramos los valores de x_0 , $f(x_0)$, la pendiente y la ecuación de la recta tangente. Estos detalles no están de más; después de todo, no se pueden leer directamente de la gráfica, pero sí la complementan perfectamente. Si la sintaxis exacta para la forma en que las comas y comillas interactúan resulta confusa para el lector, no permita que esto constituya una barrera por ahora —no es importante para nuestro progreso—.

Si el lector requiere un recordatorio acerca de la suma de gráficas para hacer superposición, puede consultar la página 18. Para un recordatorio del comando `point`, vea la página 16.

Etapla 2: Pulimos la subrutina Una vez que tenemos una subrutina funcional, es una buena idea jugar un poco con ella y afinarla exactamente como la queremos. Esto no es porque sea imposible hacer cambios después. Al contrario, el autor ha hecho modificaciones incluso en la etapa 5. Sin embargo, resulta cada vez más irritante para el programador hacer cambios en etapas posteriores. Por lo tanto, nos tomamos un tiempo para revisar los colores, las fronteras del gráfico o cualquier otra cosa. En este caso particular, testamos nuestra subrutina con las instrucciones siguientes:

Código de Sage

```
1  tangente_en_punto(0.5)
2  tangente_en_punto(1.5)
3  tangente_en_punto(-1.75)
```

Aunque el resultado está muy lejos de ser desagradable, hay aún espacio para mejoras. Por ejemplo, en el último de los casos anteriores, la recta es relativamente empinada (con pendiente de 8,1875), por lo que la línea de hecho sube hasta algo más que $y = 27$ antes de salir del área de graficación. Esto hace que la curva cúbica luzca diminuta, pues solo llega

hasta $y = 6$. Por lo tanto, parece que debemos acotar las coordenadas y de las gráficas. Hacemos esto para ambas, $f(x)$ y la recta tangente, restringiéndolas a $-6 \leq y \leq 6$, pues ese es el rango de $f(x) = x^3 - x$ sobre el dominio $-2 \leq x \leq 2$. Si observamos las líneas 15 y 16 en la figura 6.2, veremos que hemos añadido las opciones `ymin` y `ymax` del comando `plot`.

Código de Sage

```

1  f(x) = x^3 - x
2  f_prima(x) = diff(f(x), x)
3
4  def tangente_en_punto(x_0):
5      y_0 = f(x_0)
6      m = f_prima(x_0)
7
8      # Dado que  $y - y_0 = m(x - x_0)$ , sabemos que
9      #  $y - y_0 = mx - m*x_0$ ,
10     #  $y = mx + y_0 - m*x_0$ .
11     # Por lo tanto,  $b = y_0 - m*x_0$ .
12
13     b = y_0 - m * x_0
14
15     P1 = plot(f(x), -2, 2, color='blue', ymin=-6, ymax=6, gridlines='minor')
16     P2 = plot(m*x + b, -2, 2, color='tan', ymin=-6, ymax=6)
17     P3 = point((x_0, y_0), color='red', size=50)
18
19     P = P1 + P2 + P3
20     P.show()
21
22     print('x_0 =', N(x_0,digits=4))
23     print('y_0 = f(x_0) =', N(y_0,digits=4))
24     print("m = f'(x_0) =", N(m,digits=4))
25     print('Recta tangente: y = (', N(m,digits=4), ') * x + (', N(b,digits=4), ')')
26     print()

```

Figura 6.2 El código para la etapa 2 del interactivo “Recta tangente”

Por otro lado, podemos ver que el punto rojo que hemos añadido es un poco grande, así que reducimos su tamaño de 100 a 50 (línea 17). Nótese que este valor es el área del punto, no su radio, por lo que esta acción no corta el diámetro a la mitad. En realidad, como el área es la que se corta a la mitad, el diámetro es reducido por $1/\sqrt{2} \approx 0,707106 \dots$.

También, puede que un cuadrículado de fondo sea útil en el gráfico. Si observamos el comando para `P1` (línea 15), notaremos que hemos añadido la opción `gridlines='minor'`. Dado que `P1` es la gráfica “del fondo” de las tres que hemos superpuesto, este cuadrículado se mostrará por debajo de las otras.

Finalmente, tal vez sería buena idea restringir un poco la precisión numérica de x_0 , y_0 , m y en la ecuación de la recta tangente que imprimimos para el usuario. Probablemente, cuatro decimales sean más que suficientes. Podemos ver en las funciones `print` (líneas 22–25) que hemos usado la función `N()` de Sage, con el parámetro `digits` fijado en 4. Esto lo aprendimos en la página 4.

Después de todas estas modificaciones, intentamos los casos de prueba anteriores, con $x = 0,5$, $x = 1,5$ y $x = -1,75$. Los resultados lucen mucho mejor ahora, así que es tiempo de la etapa 3.

Etapla 3: Hacemos que la subrutina sea interactiva Ahora aprenderemos acerca de la instrucción `@interact` y cómo construir deslizadores para los parámetros de nuestra subrutina. Haremos tres cambios a nuestro código ahora, dos de los cuales son menores; pero con el tercero requeriremos algo de explicación.

En particular, si nos concentramos en las siguientes líneas de la figura 6.2:

Código de Sage

```
1 f(x) = x^3 - x
2 f_prima(x) = diff(f(x), x)
3
4 def tangente_en_punto(x_0):
5     y_0 = f(x_0)
```

notaremos algunos cambios a continuación:

Código de Sage

```
1 f(x) = x^3 - x
2 f_prima(x) = diff(f(x), x)
3
4 @interact
5 def tangente_en_punto(x_0=slider(-2,2,0.1,-1.5,label='Coordenada x')):
6     y_0 = f(x_0)
```

Primero, vemos que hemos añadido el comando `@interact` (“interactivo”, en español), en una línea propia, por encima del comando `def` que inicia nuestra subrutina. Esto indica a Sage que estamos creando un interactivo.

Segundo —y esto requiere algo de explicación—, hemos usado el comando `slider` (“deslizador”, en español) para cambiar x_0 de una entrada numérica a un deslizador que el usuario puede manipular. Los dos primeros parámetros le dicen a Sage que la variable x_0 estará restringida al intervalo $-2 \leq x \leq 2$. El tercer parámetro indica la *granularidad*. En este caso, queremos que cada pequeño movimiento represente $1/10$. Así que, si el deslizador está en $x_0 = 0,5$, las dos posiciones siguientes serían $0,4$ a la izquierda y $0,6$ a la derecha. El cuarto parámetro representa el valor inicial. En este caso, el deslizador se posicionará inicialmente en $x_0 = -1,5$, aunque el usuario puede moverlo luego a otros valores. Finalmente, el quinto parámetro (que es opcional) mostrará una etiqueta a modo de descripción.

El tercer cambio consiste en que, dentro de CoCalc, no llamaremos más a nuestra subrutina con comandos como `tangente_en_punto(-1.75)`. Esto es porque ya no estamos pasando x_0 como un número, sino que estamos usando el deslizador para cambiar ese valor y llamar a la subrutina automáticamente. Se puede pensar que el deslizador ahora es la entrada de la subrutina, de manera que, cuando lo movemos y lo soltamos en la posición $0,5$ —por ejemplo—, estamos implícitamente haciendo la llamada `tangente_en_punto(0.5)`.

En este punto, hemos construido una subrutina interactiva dentro de CoCalc. Deberíamos tomarnos un momento para experimentar con ella y ver cómo funciona. Nótese que cuando hacemos clic y arrastramos el deslizador, debemos dejar de presionar el botón del ratón antes que la applet reaccione.

Para que el lector pueda ver cómo lucirá el interactivo, hemos puesto una captura de pantalla en la figura 6.3.

Etapla 4: Insertamos en una plantilla web En esta etapa, copiamos y pegamos nuestra subrutina interactiva de CoCalc a un archivo HTML, construyendo de esa manera una página web interactiva rudimentaria. Podemos usar cualquier plantilla HTML que deseemos; sin embargo, el autor también pone una a disposición del lector. Esta puede ser encontrada como “plantillagenerica.html” en la página web

<https://www.sage-para-estudiantes.com/interactivos>.

También se muestra en la figura 6.5.

De seguro que el lector sabe cómo copiar y pegar, pero, en ocasiones, cosas muy pequeñas pueden ir mal en este proceso. Por lo tanto, a riesgo de parecer infantiles, detallaremos los pasos ahora muy cuidadosamente, como parte del proceso de preparación de la plantilla. Se le ruega al lector que no se sienta insultado por el nivel de detalle.

Los pasos exactos para preparar la plantilla HTML:

- (1) Resaltamos el código de la etapa 3 en CoCalc.
- (2) Damos la instrucción de “copiar” a nuestro navegador. Podemos usar el botón derecho del mouse y seleccionar la instrucción en el menú contextual, o podemos usar el teclado².

²En muchos sistemas operativos, tales como UNIX, LINUX o Windows, la combinación de teclas para esta acción es “Ctrl+C”; en sistemas Apple es “command-C”. Esto puede variar, dependiendo de la configuración de cada computadora.

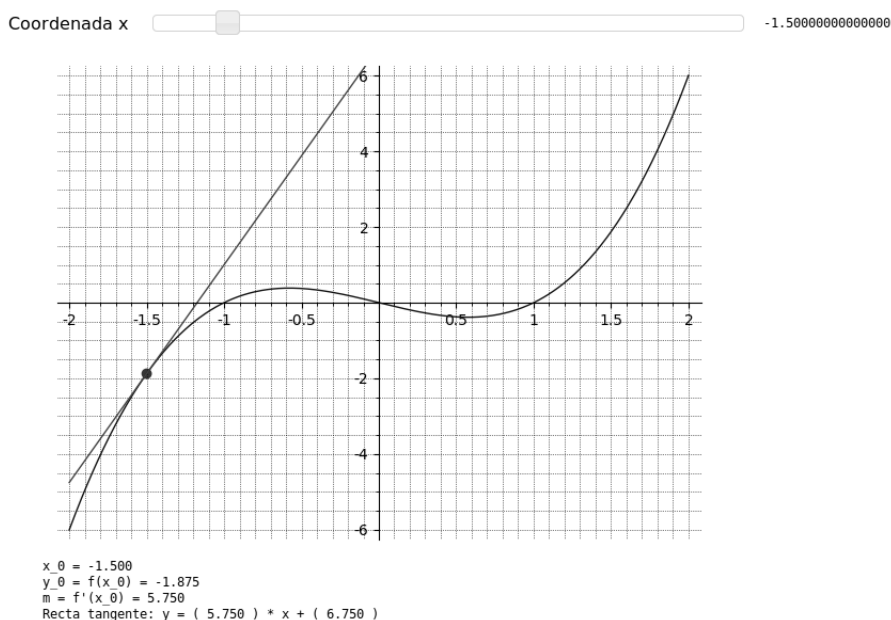


Figura 6.3 Una captura de pantalla del interactivo “Recta tangente”

- (3) Localizamos la región marcada “Debe copiar y pegar su función interactiva de Sage aquí” en el archivo de la plantilla HTML y la resaltamos, incluyendo cualquier línea con el símbolo #. Para editar archivos HTML, podemos usar cualquier editor o ambiente de desarrollo —incluso CoCalc mismo—.
- (4) Damos la instrucción de “pegar” a nuestro editor HTML. Podemos usar el menú contextual con el clic derecho del mouse, o podemos usar el teclado.³
- (5) Reemplazamos “ElTítuloAquí por primera vez” por un título adecuado, como “Explorando la recta tangente”.
- (6) Reemplazamos “ElTítuloAquí por segunda vez”, repitiendo el título anterior.
- (7) Ubicamos y reemplazamos “SuNombreAquí”.
- (8) Ubicamos y reemplazamos “LaFechaAquí”.
- (9) Guardamos el archivo usando la instrucción “Guardar como” e introduciendo un nombre que consideremos adecuado —por ejemplo, “recta_tangente.html”—.

Ahora podemos ver el archivo HTML en nuestro navegador de internet. Como podemos apreciar, la página es muy básica —apenas un esqueleto—. Si presionamos el gran botón que dice “Lanzar applet interactiva ahora”, entonces nuestra applet aparece. Tomémonos un momento para disfrutar nuestro trabajo. ¡Hemos convertido algo que era solamente un concepto en una verdadera página web interactiva!

Etapas 5: Damos detalle a la página web En este punto podemos editar nuestra página web como haríamos con cualquier otra. En la plantilla tenemos regiones para “Generalidades”, “Instrucciones” y “Discusión”. Por supuesto, podemos tener tantos encabezados de este tipo como queramos. Similarmente, podemos colocar tantos párrafos como deseemos.

La página web interactiva final de la recta tangente —el resultado de esta etapa— puede ser hallada en el sitio web www.sage-para-estudiantes.com/interactivos. El lector la encontrará bajo el título “Explorando la recta tangente”.

³En muchos sistemas operativos, tales como UNIX, LINUX o Windows, la combinación de teclas para esta acción es “Ctrl+V”; en sistemas Apple es “command-V”. Esto puede variar, dependiendo de la configuración de cada computadora.

6.4 Un desafío para el lector

Ahora ya conocemos lo que tiene que ocurrir cuando estamos creando una página web interactiva en Sage. Alentamos—incluso rogamos— al lector tomarse un momento ahora para intentarlo por sí mismo. No existe mejor forma de verificar si el entendimiento propio es total o de identificar vacíos.

Se puede elegir cualquiera de muchas tareas, pero recomendamos la siguiente: Imaginemos que vamos a dar una clase acerca de ondas sinusoidales, como las que ocurren en la luz, electricidad, sonido y otros similares. Por ejemplo, tal vez queremos explicar al estudiante el rol de la amplitud (en voltios) y la velocidad angular (en radianes por segundo). En resumen, imaginemos que queremos construir un interactivo que graficará

$$f(t) = A \sin(\omega t),$$

donde A y ω están controlados por deslizadores.

Para la gráfica, recomendamos restringirse a los intervalos $-10 \leq x \leq 10$ y $-5 \leq y \leq 5$. Por otro lado, aunque uno podría ver ventajas en forzar que la amplitud A sea positiva, puede resultar interesante para los estudiantes aprender el efecto de valores negativos, así que recomendamos $-5 \leq A \leq 5$ voltios. En el caso de ω , puede resultar confuso explicar lo que $\omega = 0$ representa, así que tal vez $0,5 \leq \omega \leq 5$ es una buena elección. Adicionalmente, recomendamos una granularidad de $1/4$ de voltio para A y de $1/2$ radianes por segundo para ω .

El lector debe realizar esta tarea, pero, si desea, puede comparar su trabajo con los archivos del autor. Es importante que primero intente completar las cinco etapas antes de hacer la comparación. Sin embargo, si se estanca, siéntase libre de echar una mirada antes en pleno proceso.

6.5 El interactivo del acuario óptimo

El objetivo general aquí es presentarle a un estudiante una pregunta al inicio de un curso (o una unidad de algún curso), de manera que

- (1) sea capaz de entenderla en ese momento,
- (2) aún no la pueda resolver,
- (3) desee resolverla y
- (4) será capaz de resolver después de completar el curso (o esa unidad).

El curso en este caso es *Cálculo I* y la unidad es *optimización*. Aquí, presentamos un problema donde las dimensiones para el diseño de un acuario son requeridas. El objetivo es construir un acuario de 64 000 pulgadas cúbicas, con una razón largo/ancho de 2,0, pero con costos mínimos. El costo de la tapa es de 0,5 centavos por pulgada cuadrada, los lados cuestan 5 centavos por pulgada cuadrada y el fondo cuesta 1,3 centavos por pulgada cuadrada. El diseño del acuario⁴ debe cumplir los dos requisitos de volumen y razón largo/ancho (al 1 % más próximo), pero tratando de alcanzar un costo mínimo.

Por supuesto, este problema es fácil de resolver con técnicas de cálculo, pero el objetivo es mostrar a los estudiantes que es difícil de resolver sin esa herramienta (solo con la intuición). Uno imagina una clase llena de estudiantes, la mayoría de los cuales no podrá considerar simultáneamente las condiciones del volumen y la razón largo/ancho. De entre aquellos que sí puedan hacerlo, la mayoría se detendrá en ese punto, ignorando la minimización del costo.

Entonces, uno puede preguntar a los estudiantes qué costo lograron obtener, y quedará claro que no todos tienen el mismo resultado. A continuación, el profesor puede pronunciar la solución óptima, de manera que los estudiantes puedan ver que en realidad es más barato que lo que ellos obtuvieron, aunque no sea por mucho. Eventualmente, el profesor debería resolver el problema con álgebra y cálculo explícitos, muchas lecciones después, cuando la unidad sobre optimización haya sido completada.

El lector debe crear un interactivo para esta clase, de tal forma que las dimensiones del acuario (largo, ancho y alto) puedan ser especificadas por los alumnos mediante deslizadores. La applet deberá imprimir las áreas de los paneles que componen el acuario, la razón largo/ancho, el volumen y el costo total. Además, se deberá indicar al usuario si las condiciones de volumen y razón largo/ancho se satisfacen o no mediante un mensaje.

⁴Para una clase de estudiantes de mediana edad que están “regresando a la escuela”, el mismo problema puede ser planteado en el contexto de diseñar un almacén con bajo costo de construcción, teniendo costos específicos para el piso, el techo y las paredes.

6.6 Selectores y casillas de verificación

Abajo tenemos una applet muy sencilla para graficar algunos polinomios. El autor la escribió solo para poder mostrar cómo se pueden hacer menús desplegables o casillas de verificación para los interactivos en Sage. Aquí tenemos el código:

Código de Sage

```

1 @interact
2 def graficadora_polynomial(mipoli=selector([x^2-4,x^3-x,x^2+1],label='Polinomio:'),
3   ↪ ),
4   ↪ cuadriculado=False):
5     if cuadriculado == True:
6         P = plot(mipoli, -3, 3, ymin=-5, ymax=5, gridlines='minor')
7     else:
8         P = plot(mipoli, -3, 3, ymin=-5, ymax=5)
9
10    show(P)

```

Vemos un nuevo comando, `selector`, que recibe una lista de Python como argumento. Este construirá un menú desplegable cuyas opciones son los elementos de la lista —en este caso, los tres polinomios mostrados en el código anterior—. El parámetro `label` añadirá una etiqueta descriptiva —como “Polinomio:”— para informar al usuario el propósito del menú. (Aunque en este caso es obvio qué hace el menú, en otras situaciones puede no serlo.)

Por otro lado, tenemos el parámetro opcional `cuadriculado`, que es `False` por defecto, de la misma manera como teníamos `verboso=False` en la página 232. Cuandoquiera que incluyamos un parámetro opcional en un interactivo, el cual puede tomar los valores `True` o `False`, Sage sabrá que deseamos tener una casilla de verificación. En este caso, el gráfico del polinomio seleccionado se mostrará con o sin el cuadriculado de fondo, dependiendo de si `cuadriculado` es `True` o `False`, respectivamente.

El resto del código no es nada nuevo para nosotros. La figura 6.4 muestra una captura de pantalla de esta applet; pero nótese que el menú no está desplegado.

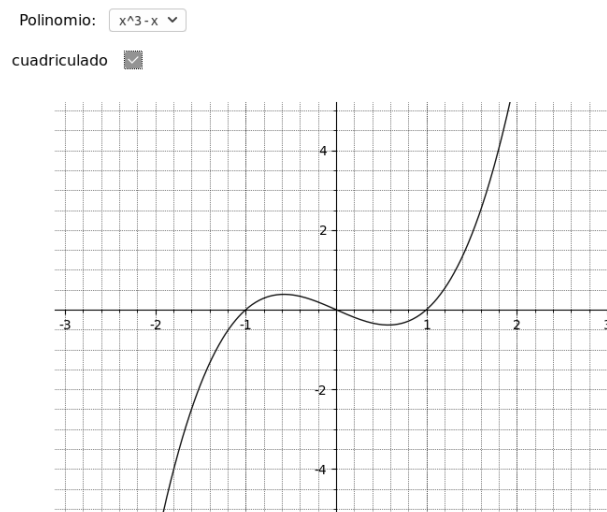


Figura 6.4 Una captura de pantalla de la applet graficadora de polinomios

6.7 El interactivo de la integral definida

En principio, este interactivo parecía simple. Deseamos explicar el significado de la integral definida y hemos escogido una función sencilla para ello, $y = x^2 - 1$. Los límites de integración son inicialmente $-1,5 < x < 1$, pero el usuario puede cambiar cualquiera de ellos (o ambos). El área incluida por la integral definida es pintada de color gris para mostrar al estudiante cómo se suele sombrear entre la curva y el eje x . Usando las técnicas de “Graficando una integral” de la subsección 3.1.4 en la página 93, podemos construir tal gráfica en Sage. Por lo tanto, a primera vista, esto parecería ser un interactivo fácil de construir.

Resulta que esto no es totalmente cierto. Primero, debemos responder al caso en que el estudiante intercambie los límites inferior y superior. Esto, en particular, causará que la integral que calcule nuestra applet tenga el signo opuesto, pues

$$\int_a^b f(x) dx = - \int_b^a f(x) dx.$$

Eso podemos dejarlo en manos de la computadora. Sin embargo, debemos tener cuidado de cómo graficar el área considerada en ese caso.

Segundo, Sage reacciona mal si la región a ser sombreada tiene área cero. En otras palabras, si $a = b$, el código de Sage que se encarga del sombreado no funciona correctamente, lo que genera un mensaje de error. Por lo tanto, una cosa que hizo el autor en su código para este interactivo es modificar artificialmente el ancho de la región para que tenga 10^{-12} unidades más de lo que debería. Así, el área nunca es cero y la queja es suprimida.

Finalmente, el autor decidió añadir una gran flecha. Si los límites de integración están en el orden normal, de izquierda a derecha, se grafica una flecha verde en esa dirección; si los límites están “invertidos”, de derecha a izquierda, se grafica una flecha roja en esa dirección. Esto ayudará a explicar a los estudiantes lo que ocurre cuando se intercambian los límites de la integral.

Como podemos ver, algunos de estos cambios son un poco inesperados. Sin embargo, el lector tiene el código del autor (para todas las etapas del proceso de construcción) a su disposición, para que pueda ver cómo progresan las cosas. Véase la dirección

www.sage-para-estudiantes.com/interactivos

plantilla.html

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>ElTituloAquí por primera vez</title>
    <script src="https://sagecell.sagemath.org/static/jquery.min.js"></script>
    <script src="https://sagecell.sagemath.org/embedded_sagecell.js"></script>
    <script>
$(function () {
  // Convertimos *todo* div de clase 'compute' en una celda de Sage
  sagecell.makeSagecell({inputLocation: 'div.compute',
                                template:      sagecell.templates.minimal,
                                evalButtonText: 'Lanzar applet interactiva ahora'});
});
    </script>
  </head>
  <body style="width: 1000px;">

    <! Puede empezar a modificar a partir este punto!>
    <! Puede empezar a modificar a partir este punto!>
    <! Puede empezar a modificar a partir este punto!>

    <h1>ElTituloAquí por segunda vez</h1>
```

```
<p>Una applet energizada por SageMath y MathJax.</p>
<p>(Por SuNombreAqui)</p>
```

```
<hr>
```

```
<h2>Generalidades</h2>
```

```
<p>blah blah blah</p>
```

```
<p>yada yada yada</p>
```

```
<p>blah blah blah</p>
```

```
<h2>Instrucciones</h2>
```

```
<p>blah blah blah</p>
```

```
<p>yada yada yada</p>
```

```
<div class="compute">
```

```
<script type="text/x-sage">
```

```
###
```

```
## Debe copiar y pegar su función interactiva aquí.
```

```
###
```

```
</script>
```

```
</div>
```

```
<h2>Discusión</h2>
```

```
<p>blah blah blah</p>
```

```
<p>yada yada yada</p>
```

```
<p>blah blah blah</p>
```

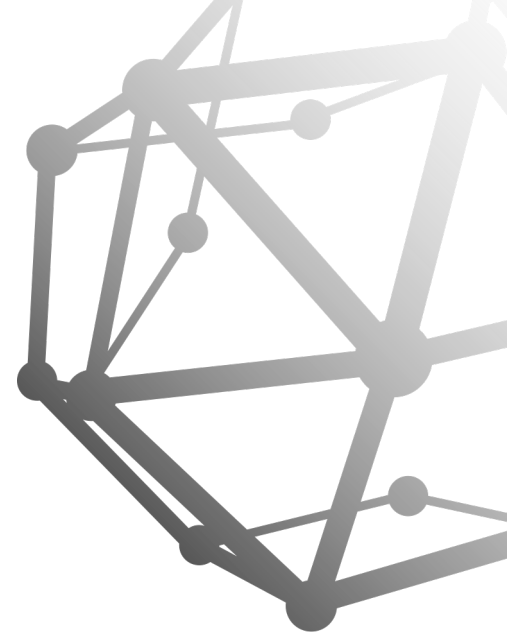
```
<hr>
```

```
Modificado por última vez: LaFechaAquí.
```

```
</body>
```

```
</html>
```

Figura 6.5 La plantilla HTML sugerida para una página web interactiva —basada en el trabajo del Prof. Jason Grout—.



A **¿Qué hacer cuando uno está frustrado?**

Las computadoras no perdonan en lo que se refiere a la sintaxis. Para los usuarios que no están acostumbrados a ese nivel de rigurosidad, esto puede ser muy frustrante. Aquí presentamos una lista de sugerencias que el lector debería usar cuando Sage rechaza su código por razones que no puede determinar. Es la esperanza del autor que estas 14 preguntas cubrirán la mayoría de los casos.

1. ¿Está usando multiplicación implícita? Este caso particular es mejor explicado con un ejemplo:

Código de Sage

```
1 solve(x^3 - 5x^2 + 6x == 0, x)
```

no es correcto; se debe escribir en cambio

Código de Sage

```
1 solve(x^3 - 5*x^2 + 6*x == 0, x)
```

En otras palabras, Sage necesita ese asterisco o símbolo de multiplicación entre los coeficientes 5 y 6, y los términos a los cuales están acompañando, x^2 y x , respectivamente.

Por alguna razón, las funciones trigonométricas realmente causan esta confusión a algunos usuarios. Sage rechazará

Código de Sage

```
1 f(x) = 5cos(6x)
```

pero aceptará

Código de Sage

```
1 f(x) = 5 * cos(6*x)
```

Sin embargo, para aquellos usuarios que encuentren esto *extremadamente frustrante*, existe una forma de evitar este requerimiento. Si colocamos la línea

Código de Sage

```
1 implicit_multiplication(True)
```

al inicio de nuestra sesión de CoCalc, entonces entraremos en *modo de multiplicación implícita*. Por ejemplo, consideremos el siguiente código:

Código de Sage

```
1 implicit_multiplication(True)
2 g(x) = (12 + 3x) / (-12 + 2x)
3 plot(g, 2, 8, ymin=-30, ymax=30)
```

Al momento de escribir estas líneas (3 de junio de 2019), esto funcionará en CoCalc, pero no en el servidor Sage Cell. Nótese que la definición de g arriba no tiene un asterisco entre el 3 y la primera x , ni entre el 2 y la segunda x . Como estamos en modo de multiplicación implícita, esos dos asteriscos omitidos son perdonados.

2. ¿Olvidó declarar una variable? La declaración de variables puede ser algo confuso para aquellos que no han hecho programación en los años previos a aprender Sage. En lenguajes como C, C++ y Java, todas las variables deben ser declaradas —sin excepciones—. En Sage, la variable x está predefinida; todas las demás deben ser declaradas, exceptuando las constantes.

Por ejemplo, si requerimos x , y y z para resolver un problema, y también requerimos tener una constante $g = 9.82$, entonces debemos declarar y y z con `var('y z')`, pero g queda implícitamente definida cuando escribimos `g = 9.82`.

Por razones de estilo (y para eliminar riesgos), algunos usuarios prefieren declarar x de todas maneras. Por lo tanto, ellos escribirían `var('x y z')`. No causa ningún impacto en Sage la declaración redundante de x . Sin embargo, visualmente hay una diferencia —trata a x de una manera más igualitaria con respecto a y y z —. Esto también es considerado una buena práctica de programación, pues puede ocurrir que previamente hayamos sobrescrito la definición de x —por ejemplo, haciendo `x = 2`—, lo que será problemático más adelante si queremos usar x como una variable.

3. ¿Se interrumpió la conexión de internet? Frecuentemente, si Sage deja de responder repentinamente, puede deberse a que el internet se ha desconectado. Por supuesto, esto puede ser causado por un problema con la señal, o simplemente porque el servicio no está disponible por unos momentos.

Puede parecer tonto incluir esto en la lista, pero vale la pena mencionarlo. Dado que Sage Cell y CoCalc operan a través de la internet, si la conexión se cae, entonces habrá que restaurarla para continuar usando Sage.

π . ¿Apareció un enorme mensaje de error? A veces, en Sage, nos aparecerá un enorme mensaje de error, del orden de las 40 líneas de longitud. Esto puede resultar intimidante o descorazonador, incluso para los usuarios experimentados.

La clave para entender estos mensajes es mirar principalmente la última línea o las dos últimas líneas. Eso es todo lo que necesitamos saber con respecto a este tema. Sin embargo, explicaremos *por qué* es este el caso en los siguientes dos párrafos, que solo van dirigidos al lector curioso.

Lo que ocurre cuando ejecutamos un comando es que nuestro código llama a una porción de Sage, que probablemente llama a otra de sus partes, e incluso puede que una tercera llamada sea hecha a una parte diferente. Pero entonces el flujo es derivado a alguna herramienta como Maxima, GAP, R o Singular, que constituye uno de los bloques elementales sobre los que está construido Sage mismo. Esta transferencia de flujo puede involucrar otras llamadas extra a funciones. Las líneas de código que estaban siendo ejecutadas cuando el error ocurrió son impresas, con los niveles más profundos dados primero y los niveles más próximos a la superficie (la interfaz de usuario) dados al final.

Lo más probable es que el error esté en nuestro código (las últimas líneas impresas) o, menos probablemente, en Sage (el siguiente lote de líneas). Los paquetes que forman los bloques elementales de Sage (por ejemplo, Maxima, Singular, GAP y otros) están muy bien testados y depurados —esos están en los dos tercios superiores del enredo de líneas impresas—. Esta es la razón por la que debemos observar las últimas pocas líneas para obtener información que nos ayude a deducir cuál es el error. Frecuentemente la última línea es la más informativa.

4. ¿El servidor Sage Cell no responde? Digamos que estamos trabajando en el servidor Sage Cell y nos detenemos para ir a almorzar. Entonces volvemos, después de tal vez algo más de una hora. Probablemente la *sesión* haya *expirado*. Para continuar trabajando sin perder lo que habíamos escrito en la celda, hacemos lo siguiente:

- resaltamos nuestro código,
- le damos la instrucción “Copiar” al navegador web,
- recargamos la página web del servidor Sage Cell,

- le damos la instrucción “Pegar” al navegador web y
- presionamos “Evaluar”.

De hecho, el autor hace esto tan seguido, que se ha convertido en una rutina. Frecuentemente, tal vez incluso una vez cada veinte minutos, realiza estos cinco pasos como una forma de asegurarse que su sesión se “mantenga viva”.

Otro pensamiento al respecto es que si el código en el que estamos trabajando es más largo que diez líneas, entonces tal vez deberíamos trabajar en CoCalc, de manera que nuestro trabajo sea más permanente y accesible en meses/años futuros cuando lo necesitemos.

5. ¿Está en una cafetería o un hotel? El tipo de internet público en una cafetería o en un hotel puede causar problemas en Sage, pero son fácilmente remediados. Primero que nada, los enrutadores de un hotel frecuentemente prealmacenan las páginas para todos sus usuarios. Por ejemplo, es muy probable que muchos huéspedes verán sitios sobre el pronóstico del clima o noticieros web famosos. ¿Por qué descargar esos sitios para cada usuario una y otra vez, cuando pueden ser descargados una sola vez y almacenados localmente? La forma más fácil de evitar esto es usando el protocolo https en lugar de http. En realidad, en el servidor Sage Cell y CoCalc, este es el comportamiento por defecto, pero nunca está de más verificar al escribir:

```
https://sagecell.sagemath.org/
https://cocalc.com
```

Pero hay otra cuestión completamente diferente. Muchas cafeterías requieren que uno se reautentifique en la red cada 10 o 20 minutos. Esto es con el propósito de hacer que el uso de internet por largos periodos de tiempo sea poco atractivo ahí. De esta manera, los estudiantes que hacen algún trabajo académico no acaparán un asiento por dos horas.

Usualmente, estas reautenticaciones no interactúan bien con Sage, por lo que ni siquiera se muestra una pantalla pidiendo reingresar el código de la red. En ese caso, solo necesitamos abrir una nueva ventana del navegador y escribir la URL de cualquier sitio famoso mayor que no hayamos visitado en esta sesión web particular, y entonces veremos la página de reautenticación. Una vez reingresado el código necesario, podemos continuar usando Sage libremente.

6. ¿Hay paréntesis o corchetes desbalanceados? En Sage, así como en matemáticas, debemos tener un “(” por cada “)” y un “[” por cada “]”. Esto es absolutamente obligatorio.

Aquí hay un viejo truco que el autor aprendió en colegio para trabajar con matemáticas en papel y lápiz. También funciona muy bien en computadoras. Para ver si hemos errado con los paréntesis, leemos una línea particular de izquierda a derecha. Empezando desde cero, cada vez que encontramos un “(”, sumamos uno, y cada vez que nos topamos con un “)”, restamos uno. El número en nuestras cabezas nunca debe ser negativo y debe resultar ser cero cuando la línea termina. Si cualquiera de estas condiciones son violadas, entonces hicimos algo mal.

7. ¿Algún comando está mal escrito? En ocasiones, la sintaxis de un comando no es clara. Por ejemplo, ¿la instrucción correcta es `A.inverse()`, `A.invert()` o `A.find_inverse()`? ¿se escribe `find_root` o `find_roots`? Esas distinciones son muy difíciles de recordar, pero debemos hacerlo de manera correcta.

El remedio para este dilema es escribir parte del comando y entonces presionar la tecla TAB. Si existe un solo comando al que nos podemos estar refiriendo, Sage lo completará por nosotros; si existen varios comandos que encajan bien con lo escrito, veremos una lista desplegable con las opciones disponibles. Tratamos esto por primera vez en la página 5.

8. ¿Están los parámetros de una función en el orden correcto? Otra sorpresa desagradable puede ocurrir cuando los parámetros de una función están en un orden incorrecto. Por ejemplo, el comando `taylor` para determinar Polinomios de Taylor y el comando `slider` para crear applets (páginas web interactivas) tienen cuatro parámetros. Consecuentemente, si tratamos de adivinar el orden de esos parámetros, frecuentemente lo haremos incorrectamente.

La solución es escribir el comando, en una línea propia, y añadir el operador “?” a continuación. Entonces presionamos ENTER. Esto mostrará el texto de documentación (o archivo de ayuda) para ese comando, el cual especificará la sintaxis —incluyendo los parámetros—. Vimos esto por primera vez en la página 44.

9. ¿Los saltos de línea están bien posicionados? En ocasiones, Python y Sage simplemente insisten en que ciertas partes del código no contengan saltos de línea. Por otro lado, el ancho de las páginas de este libro, medido en caracteres, es mucho menor que el de una celda en el servidor Sage Cell o en CoCalc. Por lo tanto, en ocasiones hemos

tenido que añadir saltos de línea en este libro (denotados con un “↵”), de manera que el código encaje en la página, pero puede que no siempre tenga sentido hacerlo de esa forma en la pantalla.

Lo único que el autor puede recomendar es experimentar y ser paciente mientras se prueban y aprenden varias posibilidades.

9³/₄. ¿Hay algún problema con el sangrado? Este es casi el mismo problema que el anterior. En Python, el sangrado del texto es realmente importante. Si se está usando Sage sin construcciones de programación, como bucles `for` y `while`, entonces tal vez no necesitamos preocuparnos por el sangrado. Sin embargo, si estamos programando Python a través de Sage, como aprendimos en el capítulo 5, el sangrado es significativo, ya que muestra cuáles comandos están subordinados a cuáles construcciones de control de flujo. Tal vez leer el capítulo 5 ayude.

Lastimosamente, al copiar y pegar el código de la versión electrónica de este libro al servidor de Sage Cell, el autor frecuentemente ha encontrado que el sangrado es destruido o mutilado. La solución, por supuesto, es borrar los espacios corruptos y volver a sangrar las líneas con la tecla TAB a través de todo el código. Esto tomará menos de un minuto, como uno puede imaginar.

10. ¿Faltan los dos puntos? Continuando con las peculiaridades de Python que frustran a los estudiantes, si estamos programando en ese lenguaje a través de Sage, como aprendimos en el capítulo 5, entonces debemos recordar poner dos puntos después de sentencias que tienen otras instrucciones subordinadas. Esto incluye a los ciclos `for`, los ciclos `while`, las definiciones de subrutinas y la construcción `if-then-else`. Véase el capítulo 5 para ejemplos.

11. ¿Hay espacios en los números grandes? En español, los números grandes tienen espacios como separadores cada tres dígitos.¹ Otros lenguajes definen reglas similares. Por ejemplo, consideremos lo siguiente:

- La notación americana/británica: 1,234,567.89.
- La notación europea continental: 1.234.567,89.
- La notación española: 1 234 567,89.
- La notación en Sage: 1234567.89 o, mejor aun, 1_234_567.89.

Como podemos ver, si estamos acostumbrados a alguna de estas formas de escritura, debemos abandonar la costumbre cuando ingresamos números grandes en Sage. En particular, no podemos usar espacios para separar los miles, millones, etc.; eso está definitivamente prohibido.

13. ¿Múltiples líneas de salida en el servidor Sage Cell? Imaginemos a alguien enseñando Álgebra Universitaria. El profesor escribe

Código de Sage

```
1 factor(x^2 - 5*x + 6)
```

en el servidor Sage Cell, obteniendo la usual respuesta útil. Entonces, para mostrar a los estudiantes un patrón, el instructor escribe las siguientes cuatro líneas en la celda:

Código de Sage

```
1 factor(x^2 - 5*x + 6)
2 factor(x^2 - 6*x + 8)
3 factor(x^2 - 7*x + 10)
4 factor(x^2 - 8*x + 12)
```

¡Solamente la factorización del último polinomio se muestra! ¿Por qué ocurre esto? Porque Sage solo imprime la última línea de los cálculos de una celda.

¹En realidad, la regla es un poco más complicada. Por ejemplo, para un número entero, se usan pequeños espacios cada tres dígitos, a menos que el número solo tenga cuatro dígitos; las partes entera y decimal de un número se tratan por separado, etc. Para mayores detalles, el lector puede consultar el Diccionario Panhispánico de Dudas en la dirección <http://lema.rae.es/dpd/>.

En cambio, este profesor podría escribir

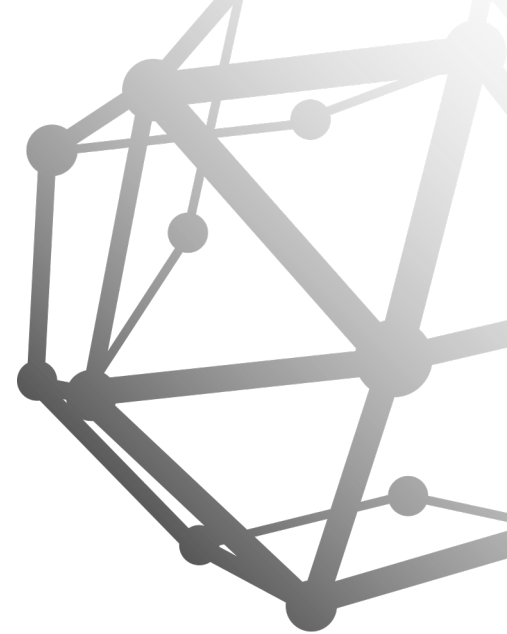
Código de Sage

```
1 print(factor(x^2 - 5*x + 6))
2 print(factor(x^2 - 6*x + 8))
3 print(factor(x^2 - 7*x + 10))
4 print(factor(x^2 - 8*x + 12))
```

y entonces, las cuatro líneas apropiadas de la salida son mostradas.

14. ¿Aún no funciona? Si esta lista de consejos no resolvió el problema, entonces podemos encontrar ayuda en el siguiente foro. Sin embargo, dado que está administrado por expertos en Sage de forma totalmente voluntaria, puede que tengamos que esperar desde unas cuantas horas hasta varios días por una respuesta. Ciertamente es útil (y más rápido) buscar viejas preguntas por una respuesta a nuestra duda, antes de hacer nuevas preguntas propias. En general, es un muy buen servicio.

<https://ask.sagemath.org>



B Haciendo la transición a CoCalc.com

El capítulo 1 de este libro asume que estamos usando el servidor Sage Cell. Eso es excelente para tareas pequeñas o incluso de tamaño medio. Sin embargo, es casi seguro que eventualmente querramos hacer el cambio a CoCalc. En general, si estamos escribiendo código de más de 10–20 líneas, definitivamente deberíamos hacer el cambio de todas maneras. Aunque ambas interfaces tienen ventajas y desventajas, el servidor Sage Cell está pensado solamente para tareas simples o cálculos rápidos.

B.1 ¿Qué es CoCalc?

CoCalc es un espacio de trabajo virtual en línea para cálculos, investigación, colaboración y creación de documentos. Un navegador web es todo lo que necesitamos para escapar del espacio confinado de nuestro escritorio y trasladarnos a la nube.

Proyectos: Los bloques principales de construcción para trabajar en CoCalc son los Proyectos. Creamos uno o más proyectos en nuestra cuenta para poder organizar nuestro trabajo en ambientes separados. Cada proyecto consiste de archivos accesibles solamente por nosotros y otros colaboradores en el mismo proyecto.

Muchos tipos de archivos con los cuales trabajar: Podemos trabajar con una variedad de lenguajes de programación y ambientes de desarrollo en CoCalc. Por ejemplo, los archivos con la extensión `.sagews` son para trabajar con Hojas de Cálculo de Sage, un archivo `.ipynb` inicia las implementaciones de CoCalc del Cuaderno Jupyter y un archivo `.tex` abre un editor para documentos de \LaTeX . También podemos trabajar con Java, Python, R, Julia, C, C++, Haskell, Scala, Fortran, OCaml, GAP, Macaulay2, Maxima, PARI/GAP, Singular, HTML, Markdown, NLTK, Pandas, Scikit-Learn, Statsmodels, TensorFlow y otros.

Colaboración: Nosotros mismos y nuestros colaboradores podemos editar archivos al mismo tiempo. Nuestros cambios son sincronizados entre todos los colaboradores en tiempo real. Podemos comunicarnos con otros por medio de un “chat lateral” para cada archivo, así como en salas de chat separadas. El chat es una herramienta esencial para la enseñanza y la investigación —incluso podemos incluir fórmulas de \LaTeX en nuestros mensajes—.

Simplicidad: Dado que los servidores son mantenidos por el equipo de administradores de sistemas de CoCalc, los profesores y estudiantes están libres de la carga de configurar y dar mantenimiento a los servidores, o mantener el software actualizado.

Ahorro en costos: Existen muchas razones por las que la Computación en la Nube es económica. Normalmente, los recursos computacionales no se usan durante las horas en las que los estudiantes y los matemáticos están dormidos —por ejemplo, desde las 3 AM hasta las 9 AM—. Ahora que las máquinas en la nube están al servicio

del mundo entero todo el tiempo, las zonas horarias ya no importan. El problema con la antigua forma de hacer las cosas —donde las grandes universidades tendrían cada una su propio servidor— es que no se puede comprar 0,1 servidores. El costo más bajo sería el de comprar un servidor, pero tal vez una pequeña universidad de artes liberales solo necesita el equivalente a 0,2 en poder computacional. La “huella de carbono” de todo ese tiempo de servidor desperdiciado también debería tomarse en consideración. El ahorro en costos (y el uso más eficiente de recursos energéticos) alcanzado a través de la agregación de servicios es a veces llamado “desfragmentación de la capacidad”.

Respaldo e historial de edición: CoCalc conserva lo siguiente para nuestros archivos:

- Viaje en el tiempo: un historial de ediciones realizadas usando el editor por defecto de CoCalc, con resolución de dos segundos. Podemos ver cada versión de cada archivo editado de esta forma, mostrando el autor de cada cambio (hasta ese nivel de resolución).
- Instantáneas: una serie de registros de solo lectura del sistema de archivos de nuestro proyecto, que nos permiten recuperar versiones antiguas de archivos que no hayan sido editados con el editor por defecto, sino que fueron alterados por otros medios.
- Respallos fuera del sitio: para restaurar el servicio y todo nuestro código con todos nuestros datos, en caso de un desastre mayor.

Combinadas, estas “medidas de durabilidad” nos dan un nivel de confiabilidad mucho más alto que al almacenar nuestros archivos en nuestra propia computadora.

Las cuentas de CoCalc están disponibles ya sea gratis o pagadas. Los proyectos en las cuentas gratis tienen limitaciones en los recursos computacionales, no pueden originar conexiones a sitios externos y puede parecer que corren más lento. Las suscripciones pagadas ofrecen mejoras tales como acceso a internet y alojamiento en sistemas con más recursos computacionales. Las suscripciones están disponibles para cuentas personales, así como para la administración de cursos con cualquier número de estudiantes.

CoCalc requiere que creamos una cuenta. Hacerlo toma cerca de 25 segundos —solo necesitamos ingresar una dirección de correo electrónico, elegir una contraseña, escribir nuestro nombre y aceptar los términos de uso—. En contraste, no se necesita crear una cuenta para el servidor Sage Cell.

B.2 Hojas de cálculo de Sage en CoCalc

Todo trabajo en CoCalc toma lugar dentro de proyectos. Cada proyecto tiene su propio espacio de archivos de usuario, que puede ser dividido en carpetas, subcarpetas y así sucesivamente, como cualquier sistema de archivos moderno. Para empezar con una hoja de cálculo de Sage, abrimos un archivo con la extensión `.sagews`.

Una hoja de cálculo de Sage es como una sucesión de varias celdas como la del servidor Sage Cell, con la excepción que cada celda está consciente del contenido de las anteriores.¹ Podemos escribir algunas líneas de código y entonces presionar SHIFT-ENTER o hacer click en el botón “Run” (“Correr” o “Ejecutar”), lo que es análogo a presionar “Evaluate” en el servidor Sage Cell. Cuando presionamos SHIFT-ENTER, una línea horizontal es dibujada, que separa la hoja de cálculo en bloques de código. SHIFT-ENTER evalúa el bloque actual y, bajo este, la salida aparece.

Como las hojas de cálculo son archivos, podemos tener muchas de ellas, copiarlas, renombrarlas, moverlas, imprimirlas, guardarlas para siempre o borrarlas. Hay mucho más sobre las hojas de cálculo de Sage en el manual de usuario en línea (en inglés) de CoCalc, en esta dirección:

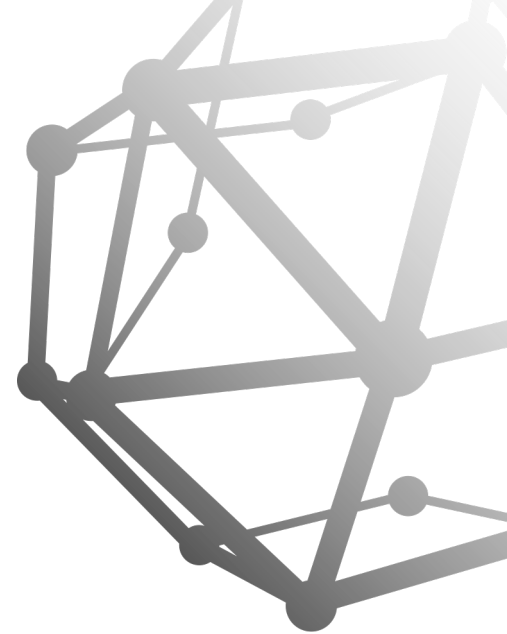
<https://doc.cocalc.com/sagews.html>

B.3 Otras características de CoCalc

Muchas otras características de CoCalc incluyen el uso de \LaTeX , Cuadernos Jupyter, Python, R, administración de cursos y chat en línea. No entraremos en detalles acerca de estas características aquí. El lector puede encontrar más información (en inglés) en esta dirección:

<http://doc.cocalc.com>

¹Esto significa que si —por ejemplo— definimos una variable en una celda, esta estará disponible y podrá ser usada en cualquiera de las celdas siguientes.



C

Otros recursos sobre Sage

Existen muchos recursos acerca de Sage. Por desgracia, para el lector de habla exclusivamente hispana, los mejores y más completos se encuentran escritos en inglés, aunque existe un esfuerzo en marcha para tenerlos disponibles en otros idiomas. A continuación presentamos una lista de recursos en línea que el lector puede consultar dudas y avanzar sus conocimientos. Para aquellos sitios que cuentan con una contraparte en español (al momento de escribir estas líneas), se ha indicado primero la dirección en español y luego la genérica, indicado claramente la que nos lleva a nuestro idioma.

Como resulta ser la naturaleza de la internet, es probable que alguna de estas direcciones ya no esté disponible o haya cambiado al momento en que el lector lea este apéndice. De ser así, el autor y el traductor harán un esfuerzo razonable para tener una lista de direcciones electrónicas actualizadas en la página del libro. De la misma manera, si alguna de estas páginas apareciera en un futuro en español, la dirección se hará disponible en el sitio web del libro.

La lista maestra de recursos de ayuda: En las siguientes URLs se puede encontrar un registro de todos los ítems que podrían ir en esta lista. Esto incluye tutoriales, guías de inicio rápido, materiales de referencias, libros, tours en línea y videos. Hay un pequeño aunque creciente número de recursos en otros idiomas.

<https://www.sagemath.org/help.html>
<https://doc.sagemath.org/>

Tutoriales temáticos: Este sitio web es el hogar de un gran número de tutoriales temáticos de Sage. Estos están orientados a áreas comunes de las matemáticas, como *Cálculo*, *Teoría de Grupos*, *Programación Lineal* o *Combinatoria Algebraica*.

http://doc.sagemath.org/html/en/thematic_tutorials/index.html

Foro de preguntas y respuestas: Este sitio web es un lugar donde cualquiera puede hacer preguntas acerca de Sage, y un experto en el tema responderá con prontitud. Existe una vibrante actividad ahí dado que Sage tiene una gran comunidad.

<https://ask.sagemath.org>

Tour de características: Si deseáramos mostrar a algunos amigos, familiares o no matemáticos de qué se trata Sage, existen dos tours de características de este software/lenguaje. Uno es algo matemáticamente pesado:

<https://www.sagemath.org/tour-quickstart.html>

El otro es acerca de los hermosos gráficos y applets interactivas que puede crear Sage:

<https://www.sagemath.org/tour-graphics.html>

Explorando las matemáticas con Sage: Este sitio web/libro electrónico, de Paul Lutus, hace un excelente trabajo explicando la cultura “open source” (de código abierto) del desarrollo de Sage. También tiene problemas de aplicación muy interesantes. Al visitar esta página, recuérdese que en su parte superior hay un menú desplegable que muestra todas las secciones; de otro modo, no se podrá navegar más allá de la introducción.

<https://arachnoid.com/sage/index.html>

Tour en línea: Dirigido a docentes de matemáticas, el siguiente tour web puede ser muy informativo acerca de algunas de las características avanzadas de Sage, así como también de las más básicas.

https://doc.sagemath.org/html/es/a_tour_of_sage/index.html (en español)

https://doc.sagemath.org/html/en/a_tour_of_sage/index.html (en inglés)

Tarjetas de referencia rápida: Muchos usuarios de Sage han preparado tarjetas de referencia rápida para ayudar al aprendizaje de los comandos y servir de apoyo cuando se usa Sage.

<https://wiki.sagemath.org/quickref>

El manual: El “Sage Reference Manual” (“Manual de referencia de Sage”) puede ser encontrado en

<https://www.sagemath.org/doc/reference/>,

¡pero no es para principiantes! El lector está advertido que constituye un documento de —literalmente— miles de páginas. Es como una enciclopedia —no está pensada para ser leída de tapa a tapa, sino para consultar algo específico que se requiera en el momento—.

Tutorial oficial en línea: Este es el tutorial oficial de Sage, dirigido principalmente a estudiantes graduados, estudiantes de último año de matemáticas y docentes.

<http://doc.sagemath.org/html/es/tutorial/> (en español)

<http://doc.sagemath.org/html/en/tutorial/index.html> (en inglés)

Tutorial en línea: El siguiente tutorial en línea y libro electrónico, de Mike O’Sullivan y David Monarres, contiene algunos temas que no pudimos tratar en este libro. Estos incluyen *Álgebra Abstracta*, *Teoría de Códigos* y la construcción de proyectos grandes de programación en Sage.

<https://mosullivan.sdsu.edu/Teaching/sdsu-sage-tutorial/index.html>
(versión en línea)

<https://mosullivan.sdsu.edu/Teaching/sdsu-sage-tutorial/SDSUSageTutorial.pdf>
(versión PDF)

Tutorial numérico: Si el lector está haciendo un curso o una investigación en Computación Numérica, el siguiente tutorial es ideal:

http://doc.sagemath.org/html/en/thematic_tutorials/numerical_sage/index.html

Tutoriales para docentes: Como el sitio web se describe a sí mismo: “Este es un conjunto de tutoriales desarrollados para los talleres ‘Sage: Using Open-Source Mathematics Software with Undergraduates’ de la Asociación Matemática de América durante los veranos¹ de 2010–2012. [...]La audiencia original de estos tutoriales fueron docentes de matemáticas de instituciones de pregrado con poca o ninguna exposición a la programación o software matemático computacional. Dado que la experiencia con la computadora requerida es mínima, este debería ser útil para cualquiera que venga con poca de tal experiencia en Sage”.

<https://doc.sagemath.org/html/en/prep/index.html>

¹Financiamiento otorgado por la National Science Foundation bajo la concesión DUE 0817071.

Tutoriales de inicio rápido: Estos están orientados hacia cursos específicos de matemática de pregrado, y pretenden ser comprensibles para un estudiante que acaba de completar esos cursos particulares. Más aun, todos son muy cortos; muchos son de una sola página. Todos están localizados en la misma URL que el tutorial anterior, pero están listados al final de la página como apéndices.

<https://doc.sagemath.org/html/en/prep/index.html>

Libros que usan Sage: Al momento de la traducción de este libro al español, existen 41 libros listados en el sitio web de Sage que usan este software para su desarrollo. Por supuesto, algunos de ellos tratan temas exóticos orientados hacia la investigación, dirigidos principalmente a estudiantes de doctorado en matemáticas. Sin embargo, otros muchos son completamente adecuados para estudiantes de pregrado.

<https://www.sagemath.org/library-publications.html#books>

Uno de estos libros para estudiantes de pregrado, que además se encuentra disponible en español, es “Álgebra Abstracta”, del Profesor Thomas W. Judson, con adiciones del Profesor Robert A. Beezer. La traducción fue hecha por Antonio Behn, y está disponible para lectura gratuita en las direcciones

<http://abstract.ups.edu/aata-es/> (en español) <http://abstract.ups.edu/aata/aata.html> (en inglés)

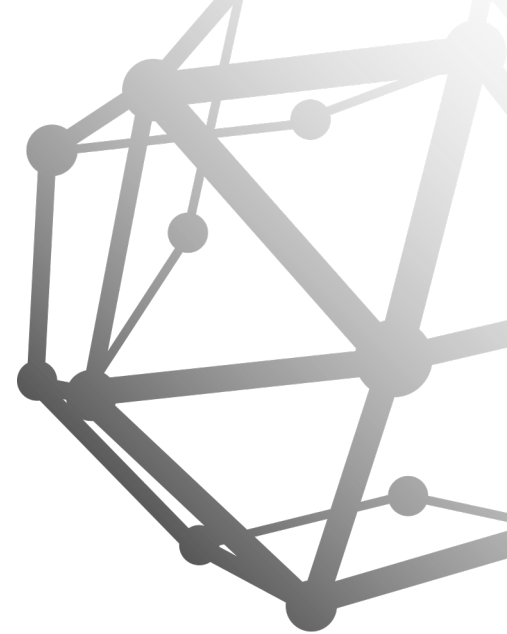
Minicurso en video en línea acerca de Sage: Un amigo del autor, Travis Scrimshaw, ha creado una serie de cuatro videos que cubren una gran parte de Sage. Fueron grabados en julio de 2013, por lo que están ligeramente desactualizados, pero son un excelente recurso de aprendizaje. Estos totalizan 4 horas y 55 minutos de grabación.

Parte I: <https://www.youtube.com/watch?v=FXbHLWtY7s4>

Parte II: <https://www.youtube.com/watch?v=9fPf1TQgF1Y>

Parte III: <https://www.youtube.com/watch?v=cbBkH8Sgryw>

Parte IV: <https://www.youtube.com/watch?v=cyfv9v16s9M>



D Sistemas lineales con infinitas soluciones

Esta sección más propiamente pertenece a un texto de Álgebra Lineal que a una guía de Sage. Sin embargo, el autor encuentra que los estudiantes frecuentemente entienden de manera incorrecta este material —durante y después del curso de Álgebra Lineal, e incluso en cursos de niveles más elementales donde estas situaciones pueden ocurrir—. Por lo tanto, aquí tenemos una discusión completa sobre cómo lidiar con sistemas de ecuaciones lineales con infinitas soluciones. Estos son bastante raros, pero no desconocidos en las aplicaciones. Este tema es sencillo, algo profundo y divertido de explorar.

D.1 El ejemplo de apertura

El sistema de ecuaciones dado por

$$\begin{aligned} 20w + 40x + 2y + 86z &= 154 \\ 4w + 8x + 5y + 31z &= 17 \\ w + 2x + 3y + 13z &= -1 \\ -8w - 16x - 4y - 44z &= -52 \end{aligned}$$

tiene la matriz dada a continuación:

$$A = \left[\begin{array}{cccc|c} 20 & 40 & 2 & 86 & 154 \\ 4 & 8 & 5 & 31 & 17 \\ 1 & 2 & 3 & 13 & -1 \\ -8 & -16 & -4 & -44 & -52 \end{array} \right]$$

Resulta que A tiene la siguiente RREF¹:

$$\text{RREF}(A) = \left[\begin{array}{cccc|c} 1 & 2 & 0 & 4 & 8 \\ 0 & 0 & 1 & 3 & -3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

Vemos aquí que existen dos filas de ceros y no vemos la familiar y bienvenida *diagonal principal* de unos: la matriz identidad de 4×4 no está presente. Por lo tanto, estamos en alerta por la posibilidad de un número infinito de soluciones.

¹Recordemos que RREF es la sigla en inglés para “Reduced Row Echelon Form” o, en español, “Forma Escalonada Reducida por Filas”.

Existe una forma sistemática de lidiar con esta situación. Primero, vamos a identificar entradas especiales de la matriz llamadas “pivotes” (y a veces, “anclas”). Entonces, clasificaremos cada variable como libre o determinada (a veces llamada también “ligada”). Finalmente, reescribiremos el sistema de ecuaciones con todas las variables determinadas a un lado de la igualdad y las variables libres, junto con las constantes, en el lado opuesto. Una vez hecho esto, habremos identificado cada una de las infinitas soluciones del sistemas de ecuaciones.

Llamaremos a la entrada no nula más a la izquierda de cada fila con el nombre de “pivote” o “ancla”. La fila 1 tiene A_{11} como pivote y la fila 2 tiene A_{23} como pivote. Dado que las filas 3 y 4 no tienen entradas no nulas (propiamente, deberíamos decir que son filas nulas), entonces no tienen pivotes. Frecuentemente, cuando el autor trabaja con papel y lápiz, suele encerrar en un círculo o subrayar las entradas de una matriz que son pivotes, como se ve a continuación:

$$\text{RREF}(A) = \left[\begin{array}{cccc|c} \textcircled{1} & 2 & 0 & 4 & 8 \\ 0 & 0 & \textcircled{1} & 3 & -3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

Como ya hemos aprendido, cada columna en la matriz de un sistema representa una variable, excepto por la columna derecha, que representa las constantes. Por supuesto, cada columna o contiene un pivote o no lo contiene. Cualquier columna (excepto la derecha) es llamada *efectiva*, y su variable *determinada o ligada*, si contiene un pivote. Similarmente, cualquier columna (excepto la derecha) es llamada *defectiva*, y su variable *libre*, si no contiene un pivote.

La regla del párrafo anterior puede parecer algo confusa, pero trabajemos con ella solo una vez y veamos lo que hace por nosotros.

- En la columna 1 tenemos un pivote, por lo que esta es efectiva y w es una variable determinada.
- En la columna 2 no tenemos pivote, por lo que esta es defectiva y x es una variable libre.
- En la columna 3 tenemos un pivote, por lo que esta es efectiva y y es una variable determinada.
- En la columna 4 no tenemos pivote, por lo que esta es defectiva y z es una variable libre.

En realidad eso no fue difícil. Ahora tomaremos la RREF, traduciremos las filas (no nulas) al álgebra y entonces moveremos las constantes y las variables libres a la derecha del signo de igualdad, dejando solo las variables determinadas a la izquierda. Primero, la conversión literal o no interpretada de la RREF en álgebra es

$$\begin{aligned} w + 2x + 0y + 4z &= 8 \\ 0w + 0x + y + 3z &= -3 \end{aligned}$$

Ahora, movamos las variables libres (x y z en este caso) al lado derecho de los signos de igualdad, para reunir las con las constantes ahí. Obtenemos

$$\begin{aligned} w &= -2x - 4z + 8 \\ y &= -3z - 3 \end{aligned}$$

Sin embargo, el autor en ocasiones prefiere escribir

$$\begin{aligned} w &= -2x - 4z + 8 \\ x &\text{ es libre} \\ y &= -3z - 3 \\ z &\text{ es libre} \end{aligned}$$

Lo que esto significa es que podemos escoger cualesquiera valores para x y z , pero una vez hecho esto, entonces w y y están determinadas por esas elecciones.

Construyamos siete soluciones específicas, escogiendo siete pares de valores para x y z . Obtenemos, por ejemplo:

- (1) Si $x = 0$ y $z = 0$, entonces $w = 8$ y $y = -3$.
- (2) Si $x = 1$ y $z = 0$, entonces $w = 6$ y $y = -3$.
- (3) Si $x = 0$ y $z = 1$, entonces $w = 4$ y $y = -6$.
- (4) Si $x = 1$ y $z = 1$, entonces $w = 2$ y $y = -6$.
- (5) Si $x = 2$ y $z = 3$, entonces $w = -8$ y $y = -12$.
- (6) Si $x = -4$ y $z = -6$, entonces $w = 40$ y $y = 15$.
- (7) Si $x = \pi$ y $z = \sqrt{2}$, entonces $w = -2\pi - 4\sqrt{2} + 8$ y $y = -3\sqrt{2} - 3$.

Tomémonos un momento para verificar que cada una de estas asignaciones de las variables en verdad satisfacen cada una de las ecuaciones originales. Hagámoslo a mano para las primeras cinco; usemos Sage para las dos últimas. En particular, para la quinta entrada de nuestra lista, escribimos

Código de Sage

```
1 w = 40
2 x = -4
3 y = 15
4 z = -6
5 print(20*w + 40*x + 2*y + 86*z)
6 print(4*w + 8*x + 5*y + 31*z)
7 print(w + 2*x + 3*y + 13*z)
8 print(-8*w - 16*x - 4*y - 44*z)
```

Vemos exactamente lo que deseábamos, es decir, obtenemos los lados derechos de nuestras ecuaciones originales resultan ser 154, 17, -1 y -52 . Similarmente, podemos reemplazar los valores de nuestras cuatro variables por aquellos de cualquiera de nuestras soluciones en la lista anterior. Más precisamente, en el caso de la séptima entrada arriba, escribimos

Código de Sage

```
1 w = 8 - 2*pi - 4*sqrt(2)
2 x = pi
3 y = -3 - 3*sqrt(2)
4 z = sqrt(2)
5 print(20*w + 40*x + 2*y + 86*z)
6 print(4*w + 8*x + 5*y + 31*z)
7 print(w + 2*x + 3*y + 13*z)
8 print(-8*w - 16*x - 4*y - 44*z)
```

Podemos ver que obtendremos la salida que deseamos: 154, 17, -1 y -52 .

D.2 Otro ejemplo

Imaginemos que se nos pide hallar todas las soluciones del sistema de ecuaciones lineales

$$3w - 3y + 6z = 51$$

$$x + 4y + 5z = 19$$

$$-w + x + 5y + 3z = 2$$

$$2w + 5x + 18y + 29z = 129$$

Empezaríamos convirtiendo esto en una matriz:

$$A = \left[\begin{array}{cccc|c} 3 & 0 & -3 & 6 & 51 \\ 0 & 1 & 4 & 5 & 19 \\ -1 & 1 & 5 & 3 & 2 \\ 2 & 5 & 18 & 29 & 129 \end{array} \right]$$

que tiene como su RREF a

$$\text{RREF}(A) = \left[\begin{array}{cccc|c} \textcircled{1} & 0 & -1 & 2 & 17 \\ 0 & \textcircled{1} & 4 & 5 & 19 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

Primero, identificamos los pivotes. Como podemos ver, los hemos marcado encerrándolos en un círculo, como antes. Luego, debemos clasificar las variables como *libres* o *determinadas*. En este caso, dado que w y x tienen pivotes, entonces son variables determinadas. De manera similar, y y z no tienen pivotes, por lo que son variables libres.

Ahora escribimos la traducción literal no interpretada de la matriz RREF(A) al álgebra, obviando las filas nulas:

$$w - y + 2z = 17$$

$$x + 4y + 5z = 19$$

Finalmente, movemos las variables libres (y y z , en este caso) a la derecha del signo de igualdad, para obtener

$$w = y - 2z + 17$$

$$x = -4y - 5z + 19$$

y es libre

z es libre

Tal vez, si se nos pide hallar cuatro soluciones distintas, podíamos escribir:

(1) Si $y = 0$ y $z = 0$, entonces $w = 17$ y $x = 19$.

(2) Si $y = 0$ y $z = 1$, entonces $w = 15$ y $x = 14$.

(3) Si $y = 1$ y $z = 0$, entonces $w = 18$ y $x = 15$.

(4) Si $y = 1$ y $z = 1$, entonces $w = 16$ y $x = 10$.

Algo más de terminología Hemos usado los términos “variable libre” y “variable determinada”, así como “columna defectiva” y “columna defectiva”. El término *sistema degenerado* indica² cualquier sistema de ecuaciones lineales donde la solución no es única —independientemente de si no tiene solución o tiene una infinidad—. Frecuentemente, las columnas efectivas de la RREF de una matriz se suelen llamar también “columnas elementales”. En términos sencillos, son aquellas cuyas entradas son todas nulas, excepto por el pivote, que es igual a 1. De manera correspondiente, las columnas defectivas también se llaman “degeneradas”.

Para distinguir entre sistemas de ecuaciones lineales con infinitas soluciones, usaremos el concepto de *grados de libertad*. El número de grados de libertad es la cantidad de variables libres. Aquí, ambos sistemas que hemos visto hasta este punto tienen dos grados de libertad. En poco, veremos muchos otros con un solo grado de libertad. También existen sistemas con tres o más, pero parecen ser poco comunes en aplicaciones y, por lo tanto, no son enfatizados aquí. También, una solución donde todas las variables son determinadas y no existen variables libres (una solución única), se dice que tiene cero grados de libertad. Normalmente, uno no utiliza este vocabulario en sistemas sin soluciones.

El número de grados de libertad de un sistema de ecuaciones $A\vec{x} = \vec{b}$ es también llamado *la nulidad derecha* de la matriz A . Notemos con cuidado que A es la matriz que describe el sistema de ecuaciones, pero sin la columna de constantes. El número de variables determinadas es llamado *el rango* de la matriz A . Este aparece en muchos lugares a lo largo del estudio del álgebra matricial. El número de grados de libertad es el número de columnas de A menos el rango de A , nuevamente teniendo cuidado de excluir la columna de constantes.

Un desafío para el lector Aquí tenemos cuatro ejemplos sencillos que el lector puede usar para desafiarse a sí mismo, a modo de ver si ha entendido bien todo lo anterior. Las soluciones generales a estos están dadas al final de este apéndice, así como las soluciones particulares que resultan de dar a todas las variables libres el valor 1. El lector también debe generar 3–4 soluciones específicas y ver si satisfacen el sistema o no.

$$M_1 = \left[\begin{array}{cccc|c} 1 & 2 & -1 & 0 & 1 \\ 3 & 6 & -3 & 2 & 7 \\ 2 & 4 & -2 & -1 & 0 \\ 7 & 14 & -7 & 6 & 19 \end{array} \right] \quad M_2 = \left[\begin{array}{cccc|c} 1 & 3 & 0 & 0 & 1 \\ 3 & 9 & 2 & 0 & 7 \\ 2 & 6 & -1 & -4 & -32 \\ 7 & 21 & 6 & 5 & 59 \end{array} \right]$$

$$M_3 = \left[\begin{array}{ccc|c} 4 & 20 & -24 & 16 \\ 7 & 35 & -42 & 28 \\ 1 & 5 & -6 & 4 \end{array} \right] \quad C_2 = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 4 & 5 & 6 & 16 \\ 7 & 8 & 9 & 25 \end{array} \right]$$

²La palabra “degenerado” es, en la era moderna, extremadamente grosera. Sin embargo, en un tiempo debió ser cortés o, al menos, socialmente aceptable. El término fue usado en el tercer cuarto del siglo XX para describir a cualquier persona que era significativamente diferente con respecto a las normas sociales. Aquí, un sistema lineal de ecuación “normal” es uno con solución única. Cualquier desviación de ese comportamiento esperado es etiquetado de “degenerado”.

A propósito, esta última matriz es la misma C_2 que vimos antes en la página 27. En aquel entonces, determinamos que $C_2.\text{rref}()$ es igual a

$$\begin{bmatrix} 1 & 0 & -1 & -1 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

y entonces nos detuvimos indicando que “hay infinitas soluciones”. Sin embargo, ahora podemos producir soluciones con las variables libres y determinadas, como hemos hecho a lo largo de este apéndice.

D.3 Explorando más profundo

En este punto, la mayor parte del tema es familiar para el lector, pero los siguientes detalles menores le darán una mejor percepción.

D.3.1 Una interesante reexaminación de las soluciones únicas

Retornemos brevemente a la matriz B de la subsección 1.5.3, “Matrices y Sage, parte uno”, en la página 23. La matriz original y su RREF son

$$B = \left[\begin{array}{cccc|c} -1 & 2 & 1 & -5 & 6 \\ 0 & 0 & -1 & -1 & -5 \\ 1 & 3 & 3 & -1 & 0 \\ -1 & 5 & -1 & 0 & -1 \end{array} \right] \xrightarrow{\text{se convierte en}} \text{RREF}(B) = \left[\begin{array}{cccc|c} \textcircled{1} & 0 & 0 & 0 & -543/65 \\ 0 & \textcircled{1} & 0 & 0 & -68/65 \\ 0 & 0 & \textcircled{1} & 0 & 268/65 \\ 0 & 0 & 0 & \textcircled{1} & 57/65 \end{array} \right]$$

Nuevamente, hemos marcado los pivotes encerrándolos en círculos. Como podemos ver, todas las columnas son elementales y ninguna es degenerada; todas las variables están determinadas y ninguna es libre. Por esta razón tenemos una y solo una solución:

$$w = -543/65, \quad x = -68/65, \quad y = 268/65, \quad z = 57/65.$$

D.3.2 Dos ecuaciones y cuatro incógnitas

Sería un poco raro, pero alguien podría pedirnos producir soluciones al siguiente sistema de dos ecuaciones:

$$\begin{aligned} 11w + 88x - 11y + 13z &= 47 \\ 17w + 136x - 17y + 19z &= 65 \end{aligned}$$

La razón por la cual esto luce extraño es que, aunque tenemos 4 incógnitas, solo tenemos 2 ecuaciones. Pareciera que contamos con “muy pocas” ecuaciones y que no existe suficiente información dada en el problema para resolverlo. De hecho, esta intuición es correcta. Resulta que existe un teorema que indica que no podemos obtener una solución única: cuando hay menos ecuaciones que incógnitas, entonces el sistema no tiene solución o tiene infinitas soluciones. No probaremos este resultado aquí, pero podemos explorar este ejemplo específico.

Primero, es extremadamente directo convertir este sistema a una matriz, aunque sea una que luce un poco extraña, pues es de orden 2×5 .

$$A = \left[\begin{array}{cccc|c} 11 & 88 & -11 & 13 & 47 \\ 17 & 136 & -17 & 19 & 65 \end{array} \right].$$

Podemos pedir a Sage que encuentre la RREF y entonces identificamos los pivotes. La RREF está dada abajo, con los pivotes encerrados en círculos:

$$\text{RREF}(A) = \left[\begin{array}{cccc|c} \textcircled{1} & 8 & -1 & 0 & -4 \\ 0 & 0 & 0 & \textcircled{1} & 7 \end{array} \right]$$

Ahora, usando nuestros métodos estándares, terminamos con

$$\begin{aligned} w &= -8x + y - 4 \\ x &\text{ es libre} \\ y &\text{ es libre} \\ z &= 7 \end{aligned}$$

¿Qué significa esto? Significa que podemos escoger cualesquiera valores para x y y . Sin embargo, una vez que hemos hecho nuestra elección, w está determinada por la fórmula $w = -4 - 8x + y$. En contraste, la variable z es muy particular y está por siempre fija en $z = 7$. Es interesante notar que tenemos dos grados de libertad, pero simultáneamente, no hay filas nulas en la RREF.

D.3.3 Dos ecuaciones y cuatro incógnitas, pero sin soluciones

Es posible que el lector curioso pueda estar interesado en un sistema con 2 ecuaciones y 4 variables, pero sin soluciones. Aquí tenemos un ejemplo obvio:

$$w + x + y + z = 1$$

$$w + x + y + z = 2$$

Claramente, no hay forma en la que $w + x + y + z$ pueda ser simultáneamente igual a 1 e igual a 2.

D.3.4 Cuatro ecuaciones y tres incógnitas

Consideremos el sistema de 4 ecuaciones y 3 incógnitas dado a continuación:

$$x + 2y + 3z = 31$$

$$4x + 5y + 6z = 73$$

$$7x + 8y - z = 35$$

$$5x + 7y + 9z = 104$$

Este produce la matriz

$$G = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 31 \\ 4 & 5 & 6 & 73 \\ 7 & 8 & -1 & 35 \\ 5 & 7 & 9 & 104 \end{array} \right],$$

cuya RREF es

$$\text{RREF}(G) = \left[\begin{array}{ccc|c} \textcircled{1} & 0 & 0 & 5 \\ 0 & \textcircled{1} & 0 & 1 \\ 0 & 0 & \textcircled{1} & 8 \\ 0 & 0 & 0 & 0 \end{array} \right].$$

Como podemos observar, hay tres pivotes —que hemos encerrado en círculos—. Por lo tanto, cada una de las variables está determinada y ninguna es libre. Finalmente, concluimos que existe una única solución, con $x = 5$, $y = 1$ y $z = 8$. Notemos que la solución es única a pesar de la presencia de una fila nula.

D.4 Ideas erróneas

Ahora trataremos algunas ideas erróneas en las que los estudiantes tienden a caer sobre este tema de infinitas soluciones a un sistema de ecuaciones lineales.

Una idea errónea mayor Frecuentemente, los estudiantes imaginan que si existe una fila de ceros, entonces habrán infinitas soluciones. Vimos en el apéndice D.3.2 un caso en que la matriz tenía una fila de ceros e infinitas soluciones. Sin embargo, en el apéndice D.3.4 vimos otra matriz con una fila de ceros, pero con una única solución. Por lo tanto, ¿qué es lo que dice nos una matriz cuando tiene una fila nula?

Si tenemos el mismo número de variables que ecuaciones, entonces podemos usar la regla que aprendimos en la subsección 1.5.8 en la página 26. De otra manera, una fila de ceros no nos dice absolutamente nada. En ese caso, debemos identificar los pivotes y determinar cuáles variables son libres y cuáles determinadas —como hemos estado haciendo a lo largo de este apéndice—.

Una idea errónea menos común En ocasiones, los estudiantes imaginarán que una columna es efectiva si todas sus entradas son cero, excepto por una que es igual a uno. Aunque esto es casi siempre verdadero, aquí tenemos un contraejemplo que muestra que esta creencia es falsa. Consideremos la matriz

$$Q = \left[\begin{array}{ccc|c} \textcircled{1} & 0 & 1 & 3 \\ 0 & \textcircled{1} & 0 & 2 \\ 0 & 0 & 0 & 0 \end{array} \right].$$

Como podemos ver, la columna de cada variable está llena de ceros, excepto por una entrada que es igual a uno. Un estudiante con esta idea errónea imaginaria que cada variable está determinada y que ninguna es libre. Este estudiante confundido diría que hay una única solución: $x = 3$, $y = 2$ y $z = 0$. Sin embargo, eso es claramente incorrecto.

En particular, dirijamos nuestra atención a la tercera columna, donde hay un único 1 y todas las entradas restantes son nulas. Esa columna parece ser elemental, pero dado que el 1 solitario no es un pivote, no es elemental, sino degenerada. Aquí podemos apreciar el valor genuino de marcar los pivotes con círculos. En este caso, estos son Q_{11} y Q_{22} ; pero Q_{13} no lo es. Como la columna tres no tiene pivote, z es una variable libre. Entonces, debemos escribir

$$\begin{aligned} x &= -z + 3 \\ y &= 2 \\ z &\text{ es libre} \end{aligned}$$

para describir el conjunto infinito de soluciones.

D.5 Definiciones formales de la REF y la RREF

Para el lector curioso, las definiciones formales de pivote, REF³ y RREF están dadas a continuación.

- Una entrada de una matriz es un *pivote* o *ancla* si y solo si es el primer elemento no nulo a la izquierda en su fila. (Por lo tanto, por definición, nunca existen entradas no nulas a la izquierda de un pivote.)
- Se dice que una matriz está en *forma escalonada por filas*, o *REF*, si se cumplen las siguientes dos condiciones:
 - La cantidad de entradas nulas a la izquierda del pivote aumenta de fila en fila. (Es decir, no existen elementos no nulos ni a la izquierda ni por debajo de un pivote.)
 - Por debajo de una fila nula no existe una no nula.
- Se dice que una matriz está en *forma escalonada reducida por filas*, o *RREF*, si se cumplen las siguientes tres condiciones:
 - La matriz está en forma escalonada por filas.
 - Todos los pivotes son iguales a 1.
 - Cada pivote es el único elemento no nulo de su respectiva columna. (Es decir, no existen entradas no nulas ni a la izquierda, ni arriba, ni abajo de un pivote.)

Nota: En ocasiones, una matriz (que no está necesariamente en forma escalonada) es tal que no existen entradas no nulas ni a la izquierda, ni a la derecha, ni arriba, ni abajo de los pivotes, siendo todos estos iguales a 1. Por ejemplo:

- Una matriz que se obtiene al “mezclar” las filas o columnas de la identidad un cierto número de veces (incluso cero) se llama *matriz de permutación*. Cuando resolvemos un sistema de ecuaciones con una matriz de permutación asociada, veremos que los valores de las incógnitas son solamente las constantes, aunque probablemente en otro orden. Estas matrices aparecen en varias ramas del Álgebra Lineal, tanto puras como aplicadas.
- Una matriz rectangular (con más filas que columnas o viceversa), cuyas únicas entradas son los pivotes (iguales a 1) y ceros, se llama *matriz de enroque* debido a algunos problemas interesantes relacionados a la matemáticas y el ajedrez, aunque también aparecen en problemas de combinatoria.

³**Nota del traductor:** Row echelon form o forma escalonada por filas. Aunque Sage no tiene un comando REF, usamos las siglas en inglés, como hemos hecho a lo largo de este libro.

D.6 Notaciones alternativas

Existe una notación más sofisticada para explorar los espacios de infinitas soluciones. Si el lector ya ha estudiado vectores, entonces encontrará esta sección extremadamente interesante; si no está muy cómodo manejando vectores, puede terminar la lectura de este apéndice aquí. Alternativamente, puede saltar al apéndice D.7, “interpretaciones geométricas en tres dimensiones”, presentado a continuación, si lo desea.

Consideremos la solución del primer ejemplo de este apéndice, pero escribiremos $x = x$ en lugar de “ x es libre” y $z = z$ en lugar de “ z es libre”. Obtenemos

$$\left. \begin{array}{l} w = -2x - 4z + 8 \\ x = x \\ y = -3z - 3 \\ z = z \end{array} \right\} \Rightarrow \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ 0 \\ -3 \\ 0 \end{pmatrix} + x \begin{pmatrix} -2 \\ 1 \\ 0 \\ 0 \end{pmatrix} + z \begin{pmatrix} -4 \\ 0 \\ -3 \\ 1 \end{pmatrix}.$$

Ahora consideremos la solución del segundo ejemplo de este apéndice, pero escribiendo $y = y$ en lugar de “ y es libre” y $z = z$ en lugar de “ z es libre”. Obtenemos

$$\left. \begin{array}{l} w = y - 2z + 17 \\ x = -4y - 5z + 19 \\ y = y \\ z = z \end{array} \right\} \Rightarrow \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 17 \\ 19 \\ 0 \\ 0 \end{pmatrix} + y \begin{pmatrix} 1 \\ -4 \\ 1 \\ 0 \end{pmatrix} + z \begin{pmatrix} -2 \\ -5 \\ 0 \\ 1 \end{pmatrix}.$$

Para asegurarse que lo ha entendido bien, el lector puede realizar los siguientes dos desafíos:

- (1) Convertir la solución de M_3 en la página 292 a esta notación vectorial.
- (2) Convertir la solución de C_2 en la página 292 a esta notación vectorial.

D.7 Interpretaciones geométricas en tres dimensiones

Existen formas notablemente útiles de visualizar geoméricamente este tipo de problemas. De la misma manera en que $2x + 3y = 6$ representa una recta en el plano, $x + 2y + 3z = 6$ representa un plano flotando en el espacio. Cuando resolvemos un sistema de 2 ecuaciones lineales con 2 incógnitas, generalmente estamos calculando el punto donde se intersectan las rectas representadas por esas ecuaciones. Sin embargo, existen dos “casos menores”. Uno se da cuando las rectas son paralelas y por lo tanto no se intersectan. El otro es cuando ambas rectas son de hecho la misma y hay un infinito número de soluciones.

Cuando tenemos un sistema de 3 ecuaciones con 3 incógnitas, estamos intersectando tres planos en el espacio. Usualmente, estos se intersectan en un solo punto común; pero, una vez más, existen casos menores, y esta vez son cuatro:

- El primero es que posiblemente los tres planos no tengan un punto en común. Por ejemplo, de manera similar a dos paredes opuestas y el techo de un cuarto rectangular. Dado que las paredes opuestas son paralelas, no tendrán puntos en común.
- El segundo es cuando los tres planos tienen una recta en común. Imaginemos un libro de tapa dura, parcialmente abierto y con páginas muy rígidas. Los tres planos pueden ser cualquiera de estas páginas o las cubiertas. Estas tienen el lomo del libro en común.
- El tercero —y verdaderamente raro— de estos casos es que los tres planos son de hecho el mismo y cualquier punto de este satisface las tres ecuaciones simultáneamente. Sin embargo, nótese que esto no significa que cualquier punto del espacio tridimensional es una solución, sino que un punto debe estar en el plano para ser solución.

- Un cuarto —y con muy poco sentido— caso es el de la siguiente matriz:

$$\left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right],$$

donde cada una de las tres ecuaciones es en efecto la ecuación trivial,

$$0x + 0y + 0z = 0$$

y, por lo tanto, cada punto en el espacio 3-dimensional las satisface. Es decir, la solución es nuestro universo tridimensional entero.

D.7.1 Visualización de los anteriores casos

Usando Sage, cada uno de estos casos pueden mostrarse gráficamente. Sin embargo, siendo tridimensionales, pertenecen a una pantalla a colores, donde el mouse pueda mover las gráficas para verlas de varios ángulos y los planos puedan hacerse transparentes. No podemos incluir imágenes significativas de tal tipo en las páginas impresas de este libro. En cambio, el autor ha diseñado un interactivo para el lector. Esta applet está titulada “Visualizando infinitas soluciones en un sistema lineal de 3 ecuaciones y 3 incógnitas”, y puede ser encontrado en la página

www.sage-para-estudiantes.com/interactivos.

Tres ejemplos son mostrados ahí: para el caso “cero grados de libertad”, se usa la matriz A de la página 19; para el caso “un grado de libertad”, se usa la matriz C_2 que apareció en las páginas 27 y 293; para el caso “dos grados de libertad”, se usa la matriz M_3 de la página 292.

D.7.2 Interpretaciones geométricas en dimensiones superiores

Los problemas con cuatro o más variables requieren las nociones de espacios de dimensión superior e hiperplanos. Por ejemplo, con 2 ecuaciones y 4 incógnitas en la RREF, frecuentemente tenemos que las soluciones resultan ser un plano flotando en el espacio 4-dimensional. Con 3 ecuaciones y 4 incógnitas, la solución es frecuentemente un hiperplano de 3 dimensiones en el mismo espacio 4-dimensional. El hiperplano tiene la misma geometría que el universo en el que vivimos. Todo esto suena muy complicado, pero de hecho puede ponerse aun peor.

Con 5 incógnitas y 2 ecuaciones en la RREF, las soluciones suelen ser un subespacio de 3 dimensiones flotando en un espacio 5-dimensional. (Esos son tres grados de libertad, a propósito.) Una forma de imaginar esto es un universo más grande y más complicado que el nuestro, con un pequeño subespacio ahí dentro siendo nuestro “universo de cada día”. Este último es el conjunto solución.

Después de mucho trabajo, los matemáticos pueden aprender a entender estas ideas formalmente en el sentido del álgebra. Sin embargo, solo unas cuantas personas pueden de hecho *visualizar* estos objetos. La vasta mayoría de nosotros, incluyendo al autor mismo, no podemos hacerlo —para muchos de nosotros, la visualización termina en 3 dimensiones—. Por lo tanto, no exploraremos más allá el punto de vista geométrico en dimensiones superiores.

D.8 Notación paramétrica

Muchos textos insisten que las variables libres sean reescritas como s y t cuando hay dos grados de libertad, y solo t cuando un grado de libertad. Esta es, según opina el autor, una elección muy extraña, pues parece difuminar la línea entre variables libres y determinadas. (Solo por completitud, indicamos que cuando hay tres grados de libertad, las variables libres serían cambiadas a r , s y t .)

Si el lector se siente impulsado a seguir esta notación, entonces tendríamos un muy sencillo (aunque muy innecesario) paso extra cuando procesamos el primer ejemplo de este apéndice. Es decir, tendríamos el siguiente cambio:

$$\left. \begin{array}{l} w = -2x - 4z + 8 \\ x = x \\ y = -3z - 3 \\ z = z \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} w = -2s - 4t + 8 \\ x = s \\ y = -3t - 3 \\ z = t \end{array} \right.$$

y, similarmente, para el segundo ejemplo:

$$\left. \begin{array}{l} w = y - 2z + 17 \\ x = -4y - 5z + 19 \\ y = y \\ z = z \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} w = s - 2t + 17 \\ x = -4s - 5t + 19 \\ y = s \\ z = t \end{array} \right.$$

A primera vista, esta modificación no parece lograr nada. Sin embargo, esta notación no carece totalmente de méritos. Por ejemplo, un sistema con un grado de libertad contiene una t , que es siempre una variable libre. Similarmente, con dos grados de libertad, aparecerán una s y una t , de manera que no habrá dudas de cuáles y cuántas variables son libres. Pero nótese que cosas similares se pueden decir de la notación que hemos usado a lo largo de este apéndice, empleando la palabra “libre”. Más aun, es una pena convertir un problema de 4 variables en uno de 6 —los problemas de 4 variables son lo suficientemente difíciles por sí mismos—. Finalmente, t es frecuentemente confundida con el símbolo $+$ y s con el número 5. Considerando todo esto, el autor piensa que es mejor evitar introducir variables nuevas.

D.9 Soluciones a los desafíos de este apéndice

(1) Para M_1 , la solución general es

$$\begin{array}{l} w = -2x + y + 1 \\ x \text{ es libre} \\ y \text{ es libre} \\ z = 2 \end{array}$$

Una solución particular es $w = 0, x = 1, y = 1, z = 2$.

(2) Para M_2 , la solución general es

$$\begin{array}{l} w = -3x + 1 \\ x \text{ es libre} \\ y = 2 \\ z = 8 \end{array}$$

Una solución particular es $w = -2, x = 1, y = 2, z = 8$.

(3) Para M_3 , la solución general es

$$\begin{array}{l} x = -5y + 6z + 4 \\ y \text{ es libre} \\ z \text{ es libre} \end{array}$$

Una solución particular es $x = 5, y = 1, z = 1$.

(4) Para C_2 , la solución general es

$$\begin{array}{l} x = z - 1 \\ y = -2z + 4 \\ z \text{ es libre} \end{array}$$

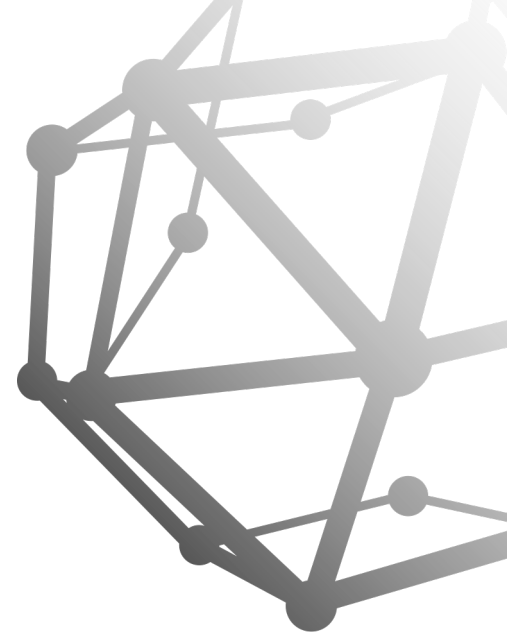
Una solución particular es $x = 1, y = -2, z = 1$.

(5) Aquí transformamos la solución para M_3 a notación vectorial:

$$\left. \begin{array}{l} x = -5y + 6z + 4 \\ y = y \\ z = z \end{array} \right\} \Rightarrow \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \\ 0 \end{pmatrix} + y \begin{pmatrix} -5 \\ 1 \\ 0 \end{pmatrix} + z \begin{pmatrix} 6 \\ 0 \\ 1 \end{pmatrix}.$$

(6) Aquí transformamos la solución para C_2 a notación vectorial:

$$\left. \begin{array}{l} x = z - 1 \\ y = -2z + 4 \\ z = z \end{array} \right\} \Rightarrow \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -1 \\ 4 \\ 0 \end{pmatrix} + z \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}.$$



E Instalando Sage en una computadora personal

La respuesta acostumbrada del autor cuando alguien le pregunta acerca de instalar Sage en su computadora personal es "¿estás seguro que quieres hacer eso?"

Razones en contra de una instalación local Parte de la genialidad de Sage es que provee software matemático innovador a cualquiera que tiene acceso a internet. Desde cualquier navegador web, uno puede usar el servidor Sage Cell para cálculos pequeños y CoCalc para cálculos mayores.

Mantener nuestro trabajo en CoCalc tiene muchas ventajas sobre una instalación local. Por ejemplo, si nuestra laptop se pierde o es robada, no perdemos nuestros archivos. Si requerimos trabajar mientras viajamos, podemos usar cualquier computadora con conexión a internet. Es muy fácil compartir nuestros archivos con otros en CoCalc, simplemente dándoles acceso electrónico. Si nuestro código necesita ejecutarse por unas cuantas horas, no requerimos mantener nuestra computadora "ligada" al proceso cuando lo estamos ejecutando en CoCalc. Además, todos los paquetes de software estarán siempre actualizados, sin ningún esfuerzo extra de nuestra parte. Personalmente, el autor nunca ha tenido una instalación local de Sage.

En el aula, esto es un punto particularmente importante de enfatizar. Si un profesor pide a un grupo de 40 estudiantes que hagan una instalación local, tendrá que lidiar con los diferentes sistemas operativos, las diferentes máquinas y configuraciones, y la inexperiencia de los estudiantes. Muchos de ellos pueden ser computacionalmente letrados, pero probablemente los 2–3 peores alumnos de esos 40 serán significativamente menos instruidos en estos temas. Conforme las computadoras se hacen más y más fáciles de usar, el nivel promedio de habilidades de los usuarios tiende a decrecer. Por ejemplo, el concepto de árbol de directorios y subdirectorios es ahora uno de los cuales los estudiantes encuentran confusos y difíciles de entender a la primera. Los estudiantes son frecuentemente intimidados por y hostiles a la "línea de comandos", o no están familiarizados con esta en el peor de los casos.

Frecuentemente, los estudiantes que tienen algún resentimiento en contra de su profesor (por poner estándares altos o exámenes difíciles), se encuentran imposibilitados de presentar una queja al Jefe de Departamento o al Decano en esos términos, así que pueden recurrir a la dificultad de usar un software de computadora obligatorio como base de su queja. Por ejemplo, "pero, señor Decano XYZ, no lo entiende. ¡Ni siquiera pude instalar el software!"

Razones a favor de una instalación local Pensándolo bien, existen también algunas ventajas de realizar una instalación local. Aunque el acceso al internet de banda ancha es ubicuo en muchos países, este no es tanto el caso de la China rural y el África Subsahariana¹, por nombrar algunos. Incluso este es el caso en algunas regiones de países avanzados, como por ejemplo (según le comentaron al autor), partes de la Wisconsin rural y otras pequeñas regiones

¹Muchos miembros de la comunidad Sage están involucrados con el Instituto Africano para las Ciencias Matemáticas, una comunidad con una atracción hacia Sage por la ausencia de costo.

escasamente pobladas de Estados Unidos. Una enorme desventaja tanto del servidor Sage Cell como de CoCalc es que requieren de una conexión a internet. Más aun, existen padres y madres que se reúsan a permitir que sus hijos tengan acceso a internet por cuestiones religiosas. Este puede ser un factor para profesores de secundaria. Uno de los asistentes de investigación de pregrado del autor creció en uno de tales ambientes domésticos.

Por el lado positivo, aunque instalar Sage en una máquina requiere mucho tiempo y esfuerzo, el resultado será que se ejecuta mucho más rápido. La mayor parte del retraso en el uso de Sage es el tiempo de tránsito entre nuestras computadoras y los servidores. Para aquellos con una conexión lenta de internet, este es un factor importante.

Encontrando los instaladores Los instaladores de Sage están organizados en un árbol de directorios en servidores disponibles alrededor del mundo. Sin importar el sistema operativo o el tipo de computadora que tenga, el lector deberá navegar este árbol de directorios. Para facilitar esta tarea, a continuación presentamos los pasos que se deben seguir.

- (1) En los párrafos siguientes, indicaremos unas URLs de donde se pueden descargar los instaladores. Introducimos la dirección que corresponda con nuestro sistema operativo.
- (2) Veremos una página que nos mostrará una lista de *mirrors* (o sitios web *espejo*²) donde están alojados los instaladores. Seleccionamos el servidor más cercano a nuestra posición actual (o cualquiera de ellos, alternativamente).
- (3) Se nos mostrará entonces una lista de arquitecturas de computadora disponibles (Intel, PowerPC, 32 bits, 64 bits, Itanium, etc.). Seleccionamos la que corresponde con nuestra computadora.
- (4) Finalmente, tenemos una lista de instaladores disponibles. Descargamos el requerimos.

Instrucciones para usuarios de Unix y Linux Se pueden encontrar binarios ejecutables para Unix y Linux en esta URL:

<https://www.sagemath.org/download-linux.html>

Para instalar Sage, se deben ejecutar los siguientes pasos:

- (1) Copiar el instalador en alguna carpeta.
- (2) Descomprimir el archivo. Esto puede lograrse haciendo click derecho y seleccionando la opción “extraer aquí” (o alguna similar) o usando algún software especializado como `file-roller`. También puede lograrse mediante la terminal, usando el comando adecuado para el tipo de archivo:
 - **tar:** `tar xvf <instalador>`.
 - **tar.gz o tgz:** `tar zxvf <nombre-del-instalador>`.
 - **tar.xz:** `tar -xJvf <nombre-del-instalador>`.
 - **tar.bz2:** `tar -xjf <nombre-del-instalador>`.
 - **lzma:** `tar --lzma -xf <nombre-del-instalador>`.
 - **lrz:** `lrzuntar <nombre-del instalador>` (puede que necesite instalar “lrzip”).
- (3) Usando la terminal, moverse a la carpeta de Sage y ejecutar `./sage`.

Nota: Aunque los archivos están principalmente pensados para sistemas tipo Debian/Ubuntu, deberían poder ejecutarse en cualquier otro tipo de sistema Linux.

Instrucciones para usuarios de OSX Si el lector está usando una Mac y tiene OSX, entonces debe recurrir a la siguiente URL:

<https://www.sagemath.org/download-mac.html>

Para instalar Sage se deben ejecutar los siguientes pasos:

- (1) Descargar el dmg en alguna parte y hacer doble click en él.
- (2) Arrastrar la carpeta de sage a algún lugar, por ejemplo, a la carpeta /Aplicaciones.
- (3) Usar el buscador para visitar la carpeta de sage que acaba de copiar y haga doble click en el ícono “sage”.
- (4) Elija la opción de ejecutarlo con “Terminal”:
 - Seleccione “Aplicaciones” y entonces “Todas las aplicaciones” en el menú desplegable “Habilitar:”.
 - Cambie el menú “Aplicaciones” a “Utilidades”. A la izquierda, busque y seleccione “Terminal”. Haga click en “Abrir” y en el siguiente cuadro de diálogo seleccione “Actualizar”.
- (5) Sage debería emerger en una ventana.

²Llamados así por ser todos idénticos entre sí.

- (6) Para el cuaderno gráfico, escriba `notebook()` y siga las instrucciones, las cuales son para abrir Firefox o Safari (a su elección) en la URL `http://localhost:8000`. Si está en una máquina de un solo usuario, usar `inotebook()` es un poco más fácil.

Nota: Escriba a

`https://groups.google.com/group/sage-support`
si tiene cualquier pregunta.

Nota: Si el lector es un gurú del OSX y quiere hacer los pasos 4–6 más agradables, únase a `sage-devel` y hágaselo saber a la comunidad Sage. ¡Su ayuda será enormemente apreciada por muchas personas!

`https://groups.google.com/group/sage-devel`

Instrucciones para compilar Sage en UNIX, Linux y OSX Es posible que el lector también pueda querer compilar Sage completo desde cero, especialmente si los archivos binarios no funcionan o si es un desarrollador de software experimentado. A final de cuentas, esto es parte de la diversión del software open source (de código abierto). Como Sage es un sistema grande y complejo, esto puede tomar más tiempo del que uno anticipe en un principio. Esos usuarios osados y experimentados deben visitar esta URL:

`https://www.sagemath.org/download-source.html`

También, en una terminal, puede usarse el comando

`git clone git://git.sagemath.org/sage.git/`

pero el lector debe saber mayores detalles sobre el software git.

Para compilar Sage se deben ejecutar los siguientes pasos:

- (1) Asegúrese de contar con por lo menos 3 GB de espacio en su disco duro y de tener todas las dependencias instaladas:

- **Para UNIX/Linux:** gcc, make, m4, perl, ranlib y tar. En sistemas Debian/Ubuntu modernos, necesitará instalar `dpkg-dev`.
- **Para OSX:** Xcode. Debe tener la versión más reciente. Para versiones pre-Lion de OSX, puede descargar Xcode de

`https://developer.apple.com/downloads/`.

Para OSX Lion, puede instalarlo usando la App Store. Con Xcode 4.3 o posterior, necesitará instalar las “Herramientas de Línea de Comandos”: en el menú “Archivo”, seleccione “Preferencias”, luego la pestaña “Descargas” y entonces “Instalar”.

- (2) Copiar el instalador a alguna carpeta.
- (3) Descomprimir el archivo. Esto puede lograrse haciendo click derecho y seleccionando la opción “extraer aquí” (o alguna similar) o usando algún software especializado como `file-roller`. También puede lograrse mediante la terminal, usando el comando adecuado para el tipo de archivo:

- **tar:** `tar xvf <instalador>`.
- **tar.gz o tgz:** `tar zxvf <nombre-del-instalador>`.
- **tar.xz:** `tar -xJvf <nombre-del-instalador>`.
- **tar.bz2:** `tar -xjf <nombre-del-instalador>`.
- **lzma:** `tar --lzma -xf <nombre-del-instalador>`.
- **lrz:** `lrzuntar <nombre-del instalador>` (puede que necesite instalar “lrzip”).

- (4) Usando la terminal, ingresar a la carpeta de Sage y ejecutar la siguiente secuencia de comandos:

```
./configure --with-python=3
make
```

Mayores detalles sobre este proceso pueden encontrarse en el archivo `README.txt` disponible en el mismo directorio del que descargó el instalador.

Instrucciones para usuarios de Microsoft Windows Existen dos formas de instalar Sage en Microsoft Windows. A partir de la versión 8.0 de Sage, está disponible un instalador nativo que puede encontrarse en la siguiente URL:

`https://www.sagemath.org/download-windows.html`

También puede encontrarse en la página de GitHub siguiente:

`https://github.com/sagemath/sage-windows/releases`

Para instalar Sage, se deben ejecutar los siguientes pasos:

- (1) Ejecutar el instalador. Por defecto, se realiza una instalación para un solo usuario, lo cual es adecuado para computadoras personales. Pero también se puede hacer una instalación para múltiples usuarios.
 - **Para un solo usuario:** No se requiere ejecutar con permisos de administrador.
 - **Para varios usuarios:** Se requiere ejecutar con permisos de administrador y luego seleccionar la opción “Install for all users” (“Instalar para todos los usuarios”), cuando se presente la oportunidad.
- (2) Presionar “Siguiente”. El asistente de instalación se encargará de todo el proceso, que puede tardar varios minutos.
- (3) Al finalizar la instalación, tendrá la opción de crear íconos en el escritorio, o en el menú de inicio, o ambos. Se recomienda al lector seleccionar la segunda o la tercera opción.

Nota: Se añadirán tres íconos de acceso directo para Sage. El primero, llamado simplemente “SageMath”, lanza la terminal con Sage listo para recibir comandos. El segundo ícono, llamado “SageMath Notebook” ejecuta el cuaderno interactivo de Sage en el navegador web. Finalmente, el tercer ícono, “SageMath Shell”, lanza una terminal que incluye las utilidades más comunes de la línea de comandos de Linux. Esta última opción es recomendada solamente para usuarios avanzados y acostumbrados a sistemas tipo Unix. ¡El lector está advertido!

Mayores detalles sobre Sage en Windows pueden encontrarse en

<https://wiki.sagemath.org/SageWindows>

Finalmente, debemos mencionar que existe una forma más tradicional de instalar Sage en una plataforma Windows. Resulta que el lector deberá instalar VirtualBox, que es una máquina virtual, que encapsulará Sage. Este es un proceso altamente testado y seguro. Se le ha dicho al autor que no existe retraso en el desempeño, comparado con instalaciones en Linux y Mac. Sin embargo, debemos recalcar que, a partir de la versión 8.0 de Sage, se recomienda usar preferentemente los instaladores nativos. Si el lector desea en cambio continuar adelante con este método tradicional, puede seguir los siguientes pasos:

- (1) Ir a la dirección

<http://www.sagemath.org/download.html>

- (2) Seleccionar el servidor más cercano a su ubicación (o cualquier otro).
- (3) Presionar en el link “OVA Image”.
- (4) Descargar el archivo OVA en alguna carpeta.
- (5) Descargar la máquina virtual VirtualBox de la dirección

<https://www.virtualbox.org/>

- (6) Instalar VirtualBox y ejecutarlo.
- (7) En el menú “Archivo”, seleccionar “Importar servicio virtualizado”.
- (8) En la ventana que aparece, seleccionar el archivo OVA de Sage y presionar “Siguiente”.
- (9) Finalmente aparecerá una ventana que muestra las configuraciones con las que se ejecutará la máquina virtual. A menos que sepa lo que está haciendo, es recomendable aceptar estas configuraciones por defecto.

Nota: Cuando se quiera lanzar la máquina virtual, es probable que se deba modificar las configuraciones para seleccionar el tipo y versión del sistema operativo en el que se ejecutará Sage.

Mayores detalles sobre este proceso pueden encontrarse en la URL

<https://wiki.sagemath.org/SageAppliance/Installation>

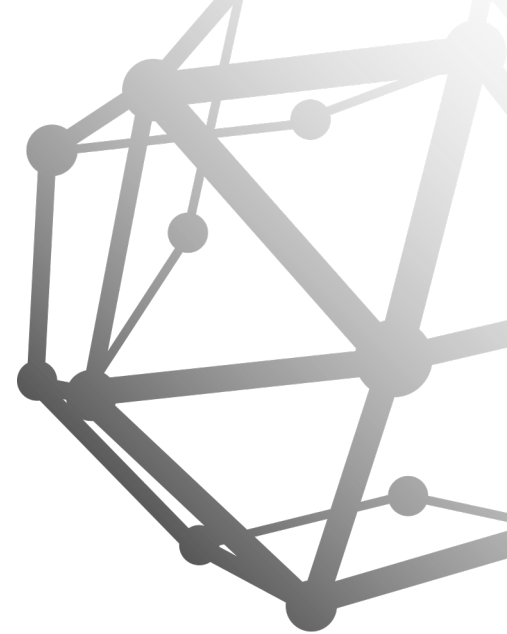
El LiveUSB de Sage Este es un archivo de arranque para USB, a partir del cual se puede ejecutar Sage directamente, sin necesidad de instalarlo. El archivo (también llamado “imagen”) es todo un sistema operativo que puede lanzarse en la computadora sin necesidad de hacer ninguna instalación. Puede encontrarse en esta URL:

<https://www.sagemath.org/download-liveusb.html>

La imagen descargada debe grabarse en un dispositivo USB, usando un software especializado que lo convierta en un *dispositivo ejecutable* (bootable device).

Este método tiene muchas ventajas. Por mencionar algunas:

- (1) El contenido puede copiarse de un dispositivo USB a otro. En esta transferencia, se puede especificar qué archivos deben copiarse y cuáles no. Esto es útil para colaborar entre colegas que requieran una copia de nuestro software y nuestro trabajo. Esto también es útil en un aula en la que los alumnos no tienen acceso a internet.
- (2) La imagen puede intentar repararse sola en caso de daños, casi siempre con éxito.
- (3) La instalación cuenta con mucho más software científico que solo Sage. Por ejemplo, Scilab, R, RStudio, Python, \LaTeX y otros están preinstalados.
- (4) Se puede controlar si los cambios hechos a la imagen (como archivos nuevos, software recién instalado, etc.) permanecen almacenados o son borrados para la siguiente ocasión que se ejecute.
- (5) El dispositivo USB es portátil y se puede llevar a una conferencia o una demostración, sin necesidad de llevar toda una computadora personal.



F **Índice de comandos por nombre y por sección**

Aquí presentamos un índice completo de todos los comandos de Sage en este libro. Primero, los mostramos ordenados por temas (por sección); luego, los mostramos ordenados alfabéticamente (por nombre). Muchas gracias a Thomas Suiter, quien compiló la primera versión de este índice.

Comandos ordenados por sección

Sección	Comando	Descripción
1.2	<code>sqrt</code>	La función raíz cuadrada
1.2	<code>N</code>	Devuelve una aproximación decimal con una cantidad especificada de cifras significativas (15 por defecto)
1.2	<code>n</code>	Sinónimo de “N”
1.2	<code>numerical_approx</code>	Versión larga del comando “N”
1.2	<code>exp</code>	La función exponencial natural, es decir e^x
1.2	<code>log</code>	Calcula el logaritmo en cualquier base (el logaritmo natural por defecto).
1.3	<code>sin</code>	La función trigonométrica “seno.” ^{en} radianes
1.3	<code>cos</code>	La función trigonométrica “coseno.” ^{en} radianes
1.3	<code>tan</code>	La función trigonométrica “tangente.” ^{en} radianes
1.3	<code>csc</code>	La función trigonométrica “cosecante.” ^{en} radianes
1.3	<code>sec</code>	La función trigonométrica “secante.” ^{en} radianes
1.3	<code>cot</code>	La función trigonométrica “cotangente.” ^{en} radianes
1.3	<code>arcsin</code>	La función trigonométrica inversa (del seno) arcoseno en radianes
1.3	<code>arccos</code>	La función trigonométrica inversa (del coseno) arcocoseno en radianes
1.3	<code>arctan</code>	La función trigonométrica inversa (de la tangente) arcotangente en radianes
1.3	<code>arccsc</code>	La función trigonométrica inversa (de la cosecante) arcocosecante en radianes
1.3	<code>arcsec</code>	La función trigonométrica inversa (de la secante) arcocosecante en radianes
1.3	<code>arccot</code>	La función trigonométrica inversa (de la cotangente) arcocotangente en radianes
1.3	<code>asin</code>	Notación abreviada para la función “arcsin”
1.3	<code>acos</code>	Notación abreviada para la función “arccos”
1.3	<code>atan</code>	Notación abreviada para la función “arctan”
1.3	<code>acsc</code>	Notación abreviada para la función “arccsc”
1.3	<code>asec</code>	Notación abreviada para la función “arcsec”
1.3	<code>acot</code>	Notación abreviada para la función “arccot”
1.4	<code>plot</code>	Grafica una función dada de una variable en el plano coordenado
1.4	<code>abs</code>	La función valor absoluto
1.4	<code>point</code>	Usado para identificar/marcar un solo punto en una gráfica
1.4.2	<code>show</code>	Usado para mostrar el gráfico resultante, cuando se superpone un gran número de gráficas una sobre otra
1.5.3	<code>matrix</code>	Usado para definir una matriz, dado el número de filas y columnas, y/o sus entradas
1.5.3	<code>rref</code>	Calcula la <i>forma escalonada reducida por filas</i> de una matriz dada
1.5.4	<code>print</code>	Usado para imprimir información en la pantalla del usuario
1.7	<code>expand</code>	Devuelve la forma expandida de un polinomio (o función racional), sin usar paréntesis
1.7	<code>factor</code>	Devuelve la versión factorizada de un polinomio o un entero
1.7	<code>gcd</code>	Calcula el <i>máximo común divisor</i> de dos polinomios o dos enteros
1.7	<code>divisors</code>	Devuelve una lista de todos los enteros positivos que dividen un entero dado

Sección	Comando	Descripción
1.8.1	<code>solve</code>	Devuelve las soluciones simbólicas de una ecuación o sistema de ecuaciones
1.8.1	<code>var</code>	Declara una nueva variable para representar alguna cantidad desconocida
1.9	<code>find_root</code>	Determina una aproximación numérica de x tal que $f(x) = 0$
1.10	<code>search_doc</code>	Busca en Sage todos los textos de documentación que mencionan los términos especificados
1.11	<code>diff</code>	Determina la derivada de una función especificada en términos de variables predefinidas
1.11	<code>derivative</code>	Sinónimo de “diff”
1.12	<code>integral</code>	Determina la integral de una función especificada en términos de variables predefinidas
1.12	<code>integrate</code>	Sinónimo de “integral”
1.12	<code>numerical_integral</code>	Calcula la integral numéricamente, devolviendo la mejor aproximación primero, seguida de la incertidumbre
1.12	<code>partial_fraction</code>	Calcula las fracciones parciales de una función racional
1.12	<code>erf</code>	La “función de error”, a veces llamada “distribución gaussiana” o “distribución normal”
1.12	<code>assume</code>	Permite al usuario declarar una suposición
2.4.3	<code>lcm</code>	Encuentra el <i>mínimo común múltiplo</i> de dos enteros (véase también 4.6.1)
2.6.1	<code>factorial</code>	Calcula el factorial de un entero no negativo
3.1.1	<code>axes_labels</code>	Etiqueta los ejes en una gráfica
3.1.3	<code>arrow</code>	Dibuja un flecha en una gráfica
3.1.3	<code>text</code>	Añade texto a una gráfica
3.3	<code>polar_plot</code>	Grafica una función en coordenadas polares
3.4	<code>implicit_plot</code>	Toma una función f a dos variables y grafica la curva $f(x, y) = 0$
3.5	<code>contour_plot</code>	Toma una función a dos variables y grafica las curvas de contorno (o conjuntos de nivel) de esa función
3.5.1	<code>sum</code>	Usado para calcular la sumatoria de una expresión (propriadamente mostrado en 4.19) o la suma de las entradas de una lista
3.6	<code>parametric_plot</code>	Toma dos o tres funciones y grafica una curva paramétrica
3.7	<code>plot_vector_field</code>	Toma dos funciones a dos variables y grafica un campo vectorial
3.7.1	<code>plot3d</code>	Crea una gráfica 3D de una función a dos variables
3.7.2	<code>vector</code>	Crea un único vector en \mathbb{R}^n
3.7.2	<code>norm</code>	Calcula la norma (o longitud) de un vector en \mathbb{R}^n
3.8	<code>plot_loglog</code>	Grafica una función de una variable en el plano coordenado con escalas logarítmicas
3.8	<code>scatter_plot</code>	Grafica una serie de puntos de datos predefinidos (véase también la sección 4.9)
3.9.2	<code>nth_root</code>	Un método para encontrar la n -ésima raíz real (véase la sección 3.9.2)
3.9.2	<code>sign</code>	Devuelve ya sea un -1 o un $+1$, dependiendo de si x es negativa o positiva, respectivamente
4.4.2	<code>identity_matrix</code>	Crea la matriz identidad de orden n
4.4.3	<code>solve_right</code>	Resuelve el problema matriz-vector $A\vec{x} = \vec{b}$, donde \vec{x} es desconocido
4.4.3	<code>solve_left</code>	Resuelve el problema vector-matriz $\vec{x}A = \vec{b}$, donde \vec{x} es desconocido

Sección	Comando	Descripción
4.4.3	augment	Aumenta columnas o bloques a la derecha de una matriz predefinida
4.4.3	echelon_form	Determina la forma escalonada de una matriz
4.4.4	inverse	Calcula la inversa de una matriz (si existe)
4.4.5	left_kernel	Calcula el conjunto de vectores \vec{n} tales que $\vec{n}A = \vec{0}$, para una matriz A
4.4.5	right_kernel	Calcula el conjunto de vectores \vec{n} tales que $A\vec{n} = \vec{0}$, para una matriz A
4.4.5	right_nullity	Calcula la dimensión del espacio de vectores \vec{n} tales que $A\vec{n} = \vec{0}$, para una matriz A
4.4.5	left_nullity	Calcula la dimensión del espacio de vectores \vec{n} tales que $\vec{n}A = \vec{0}$, para una matriz A
4.4.6	det	Calcula el determinante de una matriz cuadrada
4.5	dot_product	Calcula el producto punto de dos vectores
4.5	cross_product	Calcula el producto cruz de dos vectores
4.6	next_prime	Encuentra el siguiente número primo mayor que n
4.6	is_prime	Verifica la primalidad de un número
4.6	prime_range	Devuelve todos los números primos en el rango deseado
4.6.1	lcm	Encuentra el mínimo común múltiplo de dos números (Véase también 2.4.3)
4.6.2	nth_prime	Encuentra el n -ésimo número primo
4.6.3	euler_phi	Indica cuantos enteros positivos (desde 1 hasta n) son coprimos de n .
4.6.4	sigma	Devuelve la suma de los enteros positivos que dividen un entero positivo dado
4.6.4	len	Devuelve la longitud de cualquier lista
4.6.6	mod	Calcula la reducción modular de un entero
4.7	min	Devuelve el número más pequeño de una lista
4.7	max	Devuelve el número más grande de una lista
4.7.1	floor	A veces llamada <i>función entero mayor</i> , redondea x hacia abajo
4.7.1	ceil	A veces llamada <i>función entero menor</i> , redondea x hacia arriba
4.7.2	binomial	Calcula el coeficiente binomial o el valor de la función “elección” (para combinaciones)
4.7.2	list	Devuelve una lista de lo que se pide
4.7.2	Permutations	La entrada es una lista de ítems; la salida es una lista de todas las posibles permutaciones de esa lista
4.7.3	sinh	La función seno hiperbólico, en radianes
4.7.4	cosh	La función coseno hiperbólico, en radianes
4.7.5	tanh	La función tangente hiperbólica, en radianes
4.7.5	coth	La función cotangente hiperbólica, en radianes
4.7.5	sech	La función secante hiperbólica, en radianes
4.7.5	csch	La función cosecante hiperbólica, en radianes
4.7.5	arcsinh	La función arcoseno hiperbólico (inversa de sinh), en radianes
4.7.5	arccosh	La función arcocoseno hiperbólico (inversa de cosh), en radianes
4.7.5	arctanh	La función arcotangente hiperbólico (inversa de tanh), en radianes
4.7.5	arccoth	La función arcocotangente hiperbólico (inversa de coth), en radianes
4.7.5	arcsech	La función arcoseno hiperbólico (inversa de sech), en radianes
4.7.5	arccsch	La función arcocosecante hiperbólico (inversa de csch), en radianes

Sección	Comando	Descripción
4.7.5	asinh	Notación abreviada para la función “arcsinh”
4.7.5	acosh	Notación abreviada para la función “arcosh”
4.7.5	atanh	Notación abreviada para la función “artanh”
4.7.5	acoth	Notación abreviada para la función “arcoth”
4.7.5	asech	Notación abreviada para la función “arcsech”
4.7.5	acsch	Notación abreviada para la función “arccsch”
4.8	limit	Usado para calcular el límite (o los límites laterales) de una función
4.9	scatter_plot	Grafica una serie de puntos de datos predefinidos (véase también la sección 3.8)
4.10	find_fit	Encuentra la recta (u otra curva pedida) de mejor ajuste para una lista de puntos de datos
4.11	hex	Convierte un número a base hexadecimal
4.11	bin	Convierte un número a base binaria
4.11	oct	Convierte un número a base octal
4.11	int	Redondea un número al entero más próximo
4.12	sudoku	Encuentra una solución al puzzle del Sudoku
4.13	timeit	Cronometra el tiempo que Sage toma en ejecutar un conjunto de instrucciones
4.15	latex	Devuelve la representación de un objeto, escrita en \LaTeX
4.16.2	eigenvalues	Calcula los autovalores correspondientes a una matriz
4.16.2	eigenvectors_right	Dada una matriz A , devuelve un vector \vec{v} tal que $A\vec{v} = k\vec{v}$ para alguna constante escalar k
4.16.2	eigenvectors_left	Dada una matriz A , devuelve un vector \vec{v} tal que $\vec{v}A = k\vec{v}$ para alguna constante escalar k
4.16.2	charpoly	Calcula el polinomio característico de una matriz
4.16.2	minpoly	Calcula el polinomio mínimo de una matriz
4.16.3	as_sum_of_permutations	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	cholesky	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	frobenius	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	gram_schmidt	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	hessenberg_form	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	hermite_form	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	jordan_form	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	LU	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	QR	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	rational_form	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	smith_form	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	symplectic_form	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	weak_popov_form	Una de las 14 descomposiciones matriciales predefinidas
4.16.3	zigzag_form	Una de las 14 descomposiciones matriciales predefinidas
4.16.4	transpose	Transpone una matriz

Sección	Comando	Descripción
4.17.1	taylor	Calcula el Polinomio de Taylor de una función
4.18.1	minimize	Encuentra un mínimo de una función
4.19	sum	Usado para calcular la sumatoria de una expresión (primero mostrada en 3.5.1) o la suma de las entradas de una lista
4.20	continued_fraction	Calcula la expansión en fracciones continuas de un número real
4.20	convergents	Convierte la expansión en fracciones continuas de un número en una lista perezosa (lazy list), que representa una sucesión convergente al número
4.21.1	MixedIntegerLinearProgram	Usado para inicializar un programa lineal o un programa lineal entero mixto
4.21.1	new_variable	Crea un conjunto infinito de variables para un programa lineal
4.21.1	add_constraint	Añade restricciones a un programa lineal
4.21.1	set_objective	Establece la función objetivo de un programa lineal
4.21.1	get_values	Devuelve los valores de las variables en un programa lineal
4.21.2	remove_constraint	Remueve una restricción de un programa lineal
4.22.1	function	Declara una nueva función desconocida
4.22.1	resolve	Usado para resolver ecuaciones diferencial ordinaria de primer o segundo orden, posiblemente con condiciones iniciales
4.22.3	plot_slope_field	Genera la gráfica de un campo de pendientes de una función dada
4.23.1	laplace	Calcula la Transformada de Laplace
4.23.3	inverse_laplace	Calcula la Transformada Inversa de Laplace
5.1.1	for	Usado para crear un sentencia de un bucle for
5.1.1	in	Usado en bucles for para especificar que una variable pertenece a una lista (ver también 5.7.3)
5.1.1	range	Usando como abreviación para crear una lista de números igualmente espaciados
5.2.1	def	Usado para definir una nueva subrutina
5.3.6	return	Usado par termina una subrutina y devuelve el valor que se le indica
5.4.1	if	Usado para crear sentencias <i>if-then</i>
5.5.1	raise	Usado para levantar una excepción cuando se está lidiando con errores
5.5.1	RuntimeError	Usado para especificar que ha habido un error cuando se ejecutaba una subrutina
5.5.2	else	Usado en conjunción con la sentencia <i>if-then</i> , se encarga del caso en que la condición del “if” es False
5.6.1	while	Usado para crear sentencias de bucle while
5.6.3	break	Termina el bucle actual
5.7.1	append	Añade un nuevo ítem a la cola (el final) de una lista
5.7.3	in	Devuelve True o False, dependiendo de si un elemento está en un conjunto dado o no, respectivamente (ver también 5.1.1)
5.7.3	remove	Remueve un ítem de la cola (el final) de una lista
5.7.3	prod	Multiplica todos los elementos en una lista
5.7.3	shuffle	Reordena aleatoriamente los elementos de una lista
5.7.5	or	Devuelve True si cualquiera de dos sentencias es True, y False en otro caso

Sección	Comando	Descripción
5.7.5	and	Devuelve True si ambas de dos sentencias son True, y False en otro caso
5.7.10	rhs	Devuelve el <i>lado derecho de una ecuación como una expresión</i>
5.7.10	lhs	Devuelve el <i>lado izquierdo</i> de una ecuación como una expresión
6.1	@interact	Indica a Sage que se va a crear un interactivo (página web interactiva)
6.1	slider	Crea un deslizador para alguna entrada de una subrutina en un interactivo
6.5	selector	Crea un menú desplegable para alguna entrada de una subrutina en un interactivo, de donde el usuario puede escoger una opción
C	implicit_multiplication	Permite activar o desactivar el modo de multiplicación implícita de Sage; es decir, permite o desautoriza escribir $5x+6y$ en lugar del usual $5*x + 6*y$

Comandos ordenados alfabéticamente

Comando	Ubicación	Descripción
@interact	6.1	Indica a Sage que se va a crear un interactivo (página web interactiva)
abs	1.4	La función valor absoluto
acos	1.3	Notación abreviada para la función “arccos”
acosh	4.7.5	Notación abreviada para la función “arccosh”
acot	1.3	Notación abreviada para la función “arccot”
acoth	4.7.5	Notación abreviada para la función “arccoth”
acsc	1.3	Notación abreviada para la función “arccsc”
acsch	4.7.5	Notación abreviada para la función “arccsch”
add_constraint	4.21.1	Añade restricciones a un programa lineal
and	5.7.5	Devuelve True si ambas de dos sentencias son True, y False en otro caso
append	5.7.1	Añade un nuevo ítem a la cola (el final) de una lista
arccos	1.3	La función trigonométrica inversa (del coseno) arcocoseno en radianes
arccosh	4.7.5	La función arcocoseno hiperbólico (inversa de cosh), en radianes
arccot	1.3	La función trigonométrica inversa (de la cotangente) arcocotangente en radianes
arccoth	4.7.5	La función arcocotangente hiperbólico (inversa de coth), en radianes
arccsc	1.3	La función trigonométrica inversa (de la cosecante) arcocosecante en radianes
arccsch	4.7.5	La función arcocosecante hiperbólico (inversa de csch), en radianes
arcsec	1.3	La función trigonométrica inversa (de la secante) arcocosecante en radianes
arcsech	4.7.5	La función arcoseno hiperbólico (inversa de sech), en radianes
arcsin	1.3	La función trigonométrica inversa (del seno) arcoseno en radianes
arcsinh	4.7.5	La función arcoseno hiperbólico (inversa de sinh), en radianes
arctan	1.3	La función trigonométrica inversa (de la tangente) arcotangente en radianes
arctanh	4.7.5	La función arcotangente hiperbólico (inversa de tanh), en radianes
arrow	3.1.3	Dibuja un flecha en una gráfica
as_sum_of_permutations	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
asec	1.3	Notación abreviada para la función “arcsec”
asech	4.7.5	Notación abreviada para la función “arcsech”
asin	1.3	Notación abreviada para la función “arcsin”
asinh	4.7.5	Notación abreviada para la función “arcsinh”
assume	1.12	Permite al usuario declarar una suposición
atan	1.3	Notación abreviada para la función “arctan”
atanh	4.7.5	Notación abreviada para la función “arctanh”
augment	4.4.3	Aumenta columnas o bloques a la derecha de una matriz predefinida
axes_labels	3.1.1	Etiqueta los ejes en una gráfica
bin	4.11	Convierte un número a base binaria
binomial	4.7.2	Calcula el coeficiente binomial o el valor de la función “elección” (para combinaciones)
break	5.6.3	Termina el bucle actual
ceil	4.7.1	A veces llamada <i>función entero menor</i> , redondea x hacia arriba

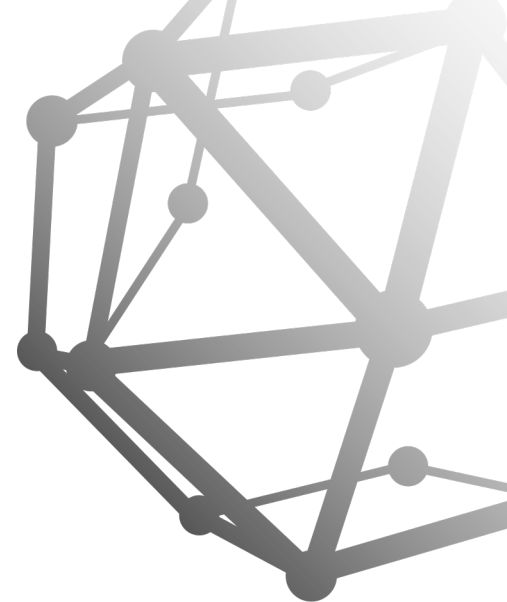
Comando	Ubicación	Descripción
charpoly	4.16.2	Calcula el polinomio característico de una matriz
cholesky	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
continued_fraction	4.20	Calcula la expansión en fracciones continuas de un número real
contour_plot	3.5	Toma una función a dos variables y grafica las curvas de contorno (o conjuntos de nivel) de esa función
convergents	4.20	Convierte la expansión en fracciones continuas de un número en una lista perezosa (lazy list), que representa una sucesión convergente al número
cos	1.3	La función trigonométrica “coseno”, en radianes
cosh	4.7.4	La función coseno hiperbólico, en radianes
cot	1.3	La función trigonométrica “cotangente”, en radianes
coth	4.7.5	La función cotangente hiperbólica, en radianes
cross_product	4.5	Calcula el producto cruz de dos vectores
csc	1.3	La función trigonométrica “cosecante”, en radianes
csch	4.7.5	La función cosecante hiperbólica, en radianes
def	5.2.1	Usado para definir una nueva subrutina
derivative	1.11	Sinónimo de “diff”
desolve	4.22.1	Usado para resolver ecuaciones diferencial ordinaria de primer o segundo orden, posiblemente con condiciones iniciales
det	4.4.6	Calcula el determinante de una matriz cuadrada
diff	1.11	Determina la derivada de una función especificada en términos de variables predefinidas
divisors	1.7	Devuelve una lista de todos los enteros positivos que dividen un entero dado
dot_product	4.5	Calcula el producto punto de dos vectores
echelon_form	4.4.3	Determina la forma escalonada de una matriz
eigenvalues	4.16.2	Calcula los autovalores correspondientes a una matrix
eigenvectors_left	4.16.2	Dada una matriz A , devuelve un vector \vec{v} tal que $\vec{v}A = k\vec{v}$ para alguna constante escalar k
eigenvectors_right	4.16.2	Dada una matriz A , devuelve un vector \vec{v} tal que $A\vec{v} = k\vec{v}$ para alguna constante escalar k
else	5.5.2	Usado en conjunción con la sentencia <i>if-then</i> , se encarga del caso en que la condición del “if” es False
erf	1.12	La “función de error”, a veces llamada “distribución gaussiana” o “distribución normal”
euler_phi	4.6.3	Indica cuantos enteros positivos (desde 1 hasta n) son coprimos de n .
exp	1.2	La función exponencial natural, es decir e^x
expand	1.7	Devuelve la forma expandida de un polinomio (o función racional), sin usar paréntesis
factor	1.7	Devuelve la versión factorizada de un polinomio o un entero
factorial	2.6.1	Calcula el factorial de un entero no negativo
find_fit	4.10	Encuentra la recta (u otra curva pedida) de mejor ajuste para una lista de puntos de datos

Comando	Ubicación	Descripción
find_root	1.9	Determina una aproximación numérica de x tal que $f(x) = 0$
floor	4.7.1	A veces llamada <i>función entera mayor</i> , redondea x hacia abajo
for	5.1.1	Usado para crear un sentencía de un bucle for
frobenius	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
function	4.22.1	Declara una nueva función desconocida
gcd	1.7	Calcula el <i>máximo común divisor</i> de dos polinomios o dos enteros
get_values	4.21.1	Devuelve los valores de las variables en un programa lineal
gram_schmidt	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
hermite_form	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
hessenberg_form	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
hex	4.11	Convierte un número a base hexadecimal
identity_matrix	4.4.2	Crea la matriz identidad de orden n
if	5.4.1	Usado para crear sentencias <i>if-then</i>
implicit_multiplication	C	Permite activar o desactivar el modo de multiplicación implícita de Sage; es decir, permite o desautoriza escribir $5x+6y$ en lugar del usual $5*x + 6*y$
implicit_plot	3.4	Toma una función f a dos variables y grafica la curva $f(x, y) = 0$
in	5.1.1	Usado en bucles for para especificar que una variable pertenece a una lista (ver también 5.7.3)
in	5.7.3	Devuelve True o False, dependiendo de si un elemento está en un conjunto dado o no, respectivamente (ver también 5.1.1)
int	4.11	Redondea un número al entero más próximo
integral	1.12	Determina la integral de una función especificada en términos de variables predefinidas
integrate	1.12	Sinónimo de “integral”
inverse	4.4.4	Calcula la inversa de una matriz (si existe)
inverse_laplace	4.23.3	Calcula la Transformada Inversa de Laplace
is_prime	4.6	Verifica la primalidad de un número
jordan_form	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
laplace	4.23.1	Calcula la Transformada de Laplace
latex	4.15	Devuelve la representación de un objeto, escrita en \LaTeX
lcm	2.4.3	Encuentra el <i>mínimo común múltiplo</i> de dos enteros (véase también 4.6.1)
lcm	4.6.1	Encuentra el mínimo común múltiplo de dos números (Véase también 2.4.3)
left_kernel	4.4.5	Calcula el conjunto de vectores \vec{n} tales que $\vec{n}A = \vec{0}$, para una matriz A
left_nullity	4.4.5	Calcula la dimensión del espacio de vectores \vec{n} tales que $\vec{n}A = \vec{0}$, para una matriz A
len	4.6.4	Devuelve la longitud de cualquier lista
lhs	5.7.10	Devuelve el <i>lado izquierdo</i> de una ecuación como una expresión
limit	4.8	Usado para calcular el límite (o los límites laterales) de una función
list	4.7.2	Devuelve una lista de lo que se pide
log	1.2	Calcula el logaritmo en cualquier base (el logaritmo natural por defecto).
LU	4.16.3	Una de las 14 descomposiciones matriciales predefinidas

Comando	Ubicación	Descripción
<code>matrix</code>	1.5.3	Usado para definir una matriz, dado el número de filas y columnas, y/o sus entradas
<code>max</code>	4.7	Devuelve el número más grande de una lista
<code>min</code>	4.7	Devuelve el número más pequeño de una lista
<code>minimize</code>	4.18.1	Encuentra un mínimo de una función
<code>minpoly</code>	4.16.2	Calcula el polinomio mínimo de una matriz
<code>MixedIntegerLinearProgram</code>	4.21.1	Usado para inicializar un programa lineal o un programa lineal entero mixto
<code>mod</code>	4.6.6	Calcula la reducción modular de un entero
<code>N</code>	1.2	Devuelve una aproximación decimal con una cantidad especificada de cifras significativas (15 por defecto)
<code>n</code>	1.2	Sinónimo de “N”
<code>new_variable</code>	4.21.1	Crea un conjunto infinito de variables para un programa lineal
<code>next_prime</code>	4.6	Encuentra el siguiente número primo mayor que n
<code>norm</code>	3.7.2	Calcula la norma (o longitud) de un vector en \mathbb{R}^n
<code>nth_prime</code>	4.6.2	Encuentra el n -ésimo número primo
<code>nth_root</code>	3.9.2	Un método para encontrar la n -ésima raíz real (véase la sección 3.9.2)
<code>numerical_approx</code>	1.2	Versión larga del comando “N”
<code>numerical_integral</code>	1.12	Calcula la integral numéricamente, devolviendo la mejor aproximación primero, seguida de la incertidumbre
<code>oct</code>	4.11	Convierte un número a base octal
<code>or</code>	5.7.5	Devuelve True si cualquiera de dos sentencias es True, y False en otro caso
<code>parametric_plot</code>	3.6	Toma dos o tres funciones y grafica una curva paramétrica
<code>partial_fraction</code>	1.12	Calcula las fracciones parciales de una función racional
<code>Permutations</code>	4.7.2	La entrada es una lista de ítems; la salida es una lista de todas las posibles permutaciones de esa lista
<code>plot</code>	1.4	Grafica una función dada de una variable en el plano coordenado
<code>plot_loglog</code>	3.8	Grafica una función de una variable en el plano coordenado con escalas logarítmicas
<code>plot_slope_field</code>	4.22.3	Genera la gráfica de un campo de pendientes de una función dada
<code>plot_vector_field</code>	3.7	Toma dos funciones a dos variables y grafica un campo vectorial
<code>plot3d</code>	3.7.1	Crea una gráfica 3D de una función a dos variables
<code>point</code>	1.4	Usado para identificar/marcar un solo punto en una gráfica
<code>polar_plot</code>	3.3	Grafica una función en coordenadas polares
<code>prime_range</code>	4.6	Devuelve todos los números primos en el rango deseado
<code>print</code>	1.5.4	Usado para imprimir información en la pantalla del usuario
<code>prod</code>	5.7.3	Multiplica todos los elementos en una lista
<code>QR</code>	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
<code>raise</code>	5.5.1	Usado para levantar una excepción cuando se está lidiando con errores
<code>range</code>	5.1.1	Usando como abreviación para crear una lista de números igualmente espaciados
<code>rational_form</code>	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
<code>remove</code>	5.7.3	Remueve un ítem de la cola (el final) de una lista

Comando	Ubicación	Descripción
remove_constraint	4.21.2	Remueve una restricción de un programa lineal
return	5.3.6	Usado par termina una subrutina y devuelve el valor que se le indica
rhs	5.7.10	Devuelve el <i>lado derecho de una ecuación como una expresión</i>
right_kernel	4.4.5	Calcula el conjunto de vectores \vec{n} tales que $A\vec{n} = \vec{0}$, para una matriz A
right_nullity	4.4.5	Calcula la dimensión del espacio de vectores \vec{n} tales que $A\vec{n} = \vec{0}$, para una matriz A
rref	1.5.3	Calcula la <i>forma escalonada reducida por filas</i> de una matriz dada
RuntimeError	5.5.1	Usado para especificar que ha habido un error cuando se ejecutaba una subrutina
scatter_plot	3.8	Grafica una serie de puntos de datos predefinidos (véase también la sección 4.9)
scatter_plot	4.9	Grafica una serie de puntos de datos predefinidos (véase también la sección 3.8)
search_doc	1.10	Busca en Sage todos los textos de documentación que mencionan los términos especificados
sec	1.3	La función trigonométrica “secante.” ^{en} radianes
sech	4.7.5	La función secante hiperbólica, en radianes
selector	6.5	Crea un menú desplegable para alguna entrada de una subrutina en un interactivo, de donde el usuario puede escoger una opción
set_objective	4.21.1	Establece la función objetivo de un programa lineal
show	1.4.2	Usado para mostrar el gráfico resultante, cuando se superpone un gran número de gráficas una sobre otra
shuffle	5.7.3	Reordena aleatoriamente los elementos de una lista
sigma	4.6.4	Devuelve la suma de los enteros positivos que dividen un entero positivo dado
sign	3.9.2	Devuelve ya sea un -1 o un $+1$, dependiendo de si x es negativa o positiva, respectivamente
sin	1.3	La función trigonométrica “seno.” ^{en} radianes
sinh	4.7.3	La función seno hiperbólico, en radianes
slider	6.1	Crea un deslizador para alguna entrada de una subrutina en un interactivo
smith_form	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
solve	1.8.1	Devuelve las soluciones simbólicas de una ecuación o sistema de ecuaciones
solve_left	4.4.3	Resuelve el problema vector-matriz $\vec{x}A = \vec{b}$, donde \vec{x} es desconocido
solve_right	4.4.3	Resuelve el problema matriz-vector $A\vec{x} = \vec{b}$, donde \vec{x} es desconocido
sqrt	1.2	La función raíz cuadrada
sudoku	4.12	Encuentra una solución al puzzle del Sudoku
sum	4.19	Usado para calcular la sumatoria de una expresión (primero mostrada en 3.5.1) o la suma de las entradas de una lista
sum	3.5.1	Usado para calcular la sumatoria de una expresión (propriadamente mostrado en 4.19) o la suma de las entradas de una lista
symplectic_form	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
tan	1.3	La función trigonométrica “tangente.” ^{en} radianes
tanh	4.7.5	La función tangente hiperbólica, en radianes

Comando	Ubicación	Descripción
<code>taylor</code>	4.17.1	Calcula el Polinomio de Taylor de una función
<code>text</code>	3.1.3	Añade texto a una gráfica
<code>timeit</code>	4.13	Cronometra el tiempo que Sage toma en ejecutar un conjunto de instrucciones
<code>transpose</code>	4.16.4	Transpone una matriz
<code>var</code>	1.8.1	Declara una nueva variable para representar alguna cantidad desconocida
<code>vector</code>	3.7.2	Crea un único vector en \mathbb{R}^n
<code>weak_popov_form</code>	4.16.3	Una de las 14 descomposiciones matriciales predefinidas
<code>while</code>	5.6.1	Usado para crear sentencias de bucle <code>while</code>
<code>zigzag_form</code>	4.16.3	Una de las 14 descomposiciones matriciales predefinidas



Índice alfabético

$2^{B!}$ mód N (cálculo eficiente), 85
 $2^{10\,000!}$ mód N (cálculo eficiente), 85
 $<$, 239
 $=$, 239
 $>$, 239
 $B!$, 83
 erf , 55
 η , 71
 \gcd , 33, 140
 \geq , 239
 ∞ , 150
 lcm , 76, 141
 \leq , 239
 \log , 6
 \log_{10} , 230
 μ , 71
 ∇ , 111, 130
 \neq , 239
 mód, 247
 ∂ , 129
 π , 4, 31, 178
 π (función), 239
 sinc , 96
 e , 4
 g (constante, aceleración gravitacional), 168
 i , 5
 n -ésimo primo, 142
 tamaño, 142
 \LaTeX , 93, 159, 193, 204
El Ensayador, 184
 Sage
 \TeX , 204
 documentación, 208
 uso, 205
 $\operatorname{SageTeX}$, 60
 $\operatorname{RR.pi}()$, 105
 $!=$, 239

\wedge , 1, 132
 $**$, 2
 $*$, 1, 32, 131, 139
 $+\operatorname{Infinity}$ (constante de Sage), 150
 $+$, 1, 16, 32, 62, 139, 241, 251
 $-\operatorname{Infinity}$ (constante de Sage), 150
 $-\infty$ (constante de Sage), 55
 $-$, 1, 32, 139
 $//$, 147, 247
 $/$, 1, 247
 $0b$, 156
 $0o$, 156
 $0x$, 156
 \leq , 180, 181, 239
 $<$, 239
 $==$, 181, 234, 239
 \geq , 180, 181, 239
 $>$, 239
 $??$, 45, 217
 $?$, 44, 163, 217
 $\operatorname{@interact}$, 268, 271
 $\operatorname{ArithmeticError}$ (excepción), 250
 CC , 257
 $\operatorname{Infinity}$ (constante de Sage), 8, 150
 I (constante de Sage), 5, 7
 $\operatorname{LU}()$, 164
 $\operatorname{MixedIntegerLinealProgram}()$, 180
 $\operatorname{N}()$, 4, 72, 105, 210, 227
 $\operatorname{NameError}$ (excepción), 12
 None (Constante de Python), 181
 N , 267
 $\operatorname{Permutations}()$, 148
 QQ , 257
 $\operatorname{QR}()$, 164
 $\operatorname{RR}()$, 119
 RR , 257
 $\operatorname{RuntimeError}()$, 241

- RuntimeError (excepción), 241
- TypeError, 218
- ValueError (excepción), 54, 56, 192
- ZZ, 257
- ZeroDivisionError (excepción), 235
- #, 229
- \sageplot (comando de \LaTeX), 206
- \sagestr (comando de \LaTeX), 206, 207
- \sage (comando de \LaTeX), 205, 207
- \, 165
- %, 146, 247
- \, 132, 133
- abs(), 120, 237
- abs, 10
- acosh(), 149
- acos, 9
- acoth(), 149
- acot, 9
- acsch(), 149
- acsc, 9
- add_constraint(), 180, 181
- all (opción de comando), 7
- and, 256
- append(), 252
- arccosh(), 149
- arccos, 9
- arccoth(), 149
- arccot, 9
- arccsch(), 149
- arccsc, 9
- arcsech(), 149
- arcsec, 9
- arcsinh(), 149
- arcsin, 9
- artanh(), 149
- arctan, 9
- as_sum_of_permutations(), 164
- asech(), 149
- asec, 9
- asinh(), 149
- asin, 9
- aspect_ratio (opción de comando), 106
- assume(), 57, 176, 192
- atanh(), 149
- atan, 9
- augment(), 132, 135
- axes_labels(), 89
- axes (opciones de comando), 101
- axes (opción de comando), 13, 103
- binary(), 157
- binary (opción de comando), 183
- binomial(), 147, 148
- break, 245
- can't convert complex to float, 118
- ceil(), 147
- charpoly(), 163
- cholesky(), 164
- colorbar (opción de comando), 106
- color (opción de comando), 47
- continued_fraction(), 178
- contour_plot(), 103
- contours (opción de comando), 108
- convergents(), 178
- cosh(), 149
- cos, 8
- coth(), 149
- cot, 8
- cross_product(), 139
- csch(), 149
- csc, 8
- def, 217, 225
- derivative(), 46, 115, 129, 130, 199
- desolve(), 184, 187
- det(), 137
- detect_poles (opción de comando), 14
- diff(), 46, 48, 129, 130, 184, 185, 195, 197, 215
- digits (opción de comando), 4, 72, 210, 267
- divisors(), 34, 144
- dot_product(), 139
- echelon_form(), 134
- eigenvalues(), 163
- eigenvectors_left(), 163
- eigenvectors_right(), 163
- else, 242, 243
- end (opcion de comando), 248
- end (opción de comando), 211, 242
- euler_phi(), 143
- expand(), 33, 124, 175
- exp, 6
- e (constante de Sage), 4
- facecolor (opción de comando), 116
- factor(), 33, 34, 140, 250
- factorial(), 147
- factorial, 84
- factor, 62
- fill (opción de comando), 93, 103
- find_fit(), 154
- find_root(), 42
- floor(), 147
- fontsize (opción de comando), 92
- for, 210, 212, 249
- frame(opción de comando), 91
- frobenius(), 164
- full_simplify(), 56, 175
- function(), 184
- gcd(), 33, 140, 141
- gcd, 62
- get_values(), 180, 182
- gram_schmidt(), 164
- gridlines (opción de comando), 16, 91, 101, 116, 267
- headaxislength (opción de comando), 188
- headlength (opción de comando), 188
- hermite_form(), 164
- hessengerg_form(), 164
- hex(), 156
- ics (opción de comando), 187
- if-then-else, 242
- if, 232, 234, 239, 242, 253, 256

`implicit_plot()`, 100
`identity_matrix()`, 131
`integer` (opción de comando), 183
`integral()`, 48, 49, 51, 203
`integrate()`, 56
`inverse()`, 134
`inverse_laplace()`, 193
`in`, 210, 212, 253, 257
`is_prime()`, 140
`i` (constante de Sage), 5, 7
`jordan_form()`, 164
`kernel()`, 137
`labels` (opción de comando), 103
`label` (opción de comando), 268, 271
`lambda`, 119
`laplace()`, 191
`latex()`, 160, 193
`latex`, 60
`lcm()`, 141
`left_kernel()`, 135
`left_nullity()`, 136
`legend_label` (opción de comando), 89
`len()`, 144, 148, 252
`lhs()`, 258
`limit()`, 149
`linestyle`, 69
`linestyle` (opción de comando), 94
`list()`, 148
`math domain error`, 99
`matrix`, 21, 23
`maximization` (opción de comando), 180
`max`, 147
`min()`, 147
`minimiza()`, 170
`minimize()`, 172
`minpoly()`, 164
`mod()`, 146
`n()`, 4
`new_variable()`, 180–183
`next_prime`, 140, 245, 247
`nonnegative` (opción de comando), 181
`norm()`, 113, 139
`nth_prime()`, 142
`nth_root()`, 119
`numerical_approx`, 4
`numerical_integral()`, 52
`oct()`, 157
`oo` (constante de Sage), 55
`oo` (constante de Sage), 174
`or`, 256
`parametric_plot()`, 108
`partial_fraction()`, 53
`pi` (constante de Sage), 4, 105
`plot()`, 9, 29, 89, 91, 101, 267
`plot3d()`, 111
`plot_loglog()`, 116
`plot_points` (opción de comando), 98, 103, 110
`plot_slope_field()`, 188
`plot_vector_field()`, 110, 201
`point()`, 16, 92, 265
`polar_plot()`, 96
`prime_pi()`, 239
`prime_range()`, 140
`print()`, 29, 139, 211, 215, 242
`print`, 22, 59
`prod()`, 253
`raise`, 239, 241
`range()`, 210, 212, 249
`rank()`, 136
`rational_form`, 164
`remove()`, 253
`remove_constraint()`, 182
`return`, 229, 237
`rhs()`, 258
`right_kernel()`, 136
`right_nullity()`, 136
`rref()`, 135
`rref`, 22
`sagesilent` (ambiente de \LaTeX), 207
`sagetex.sty` (archivo), 205
`save()`, 58
`scatter_plot()`, 116, 151, 251
`search_doc()`, 45
`sech()`, 149
`sec`, 8
`selector()`, 271
`sep` (opción de comando), 215
`set_objective()`, 180, 181
`show()`, 18, 59, 92, 120, 181
`shuffle()`, 253
`sigma()`, 144
`sign()`, 120
`simplify_full()`, 56
`sinh()`, 149
`sin`, 8
`size` (opción de comando), 16, 92, 265
`slider()`, 268
`smith_form()`, 164
`solution_dict` (opción de comando), 35
`solve()`, 35, 37, 180, 257
`solve_left()`, 132
`solve_right()`, 132
`sqrt`, 3
`subdivide` (opción de comando), 132, 135
`sudoku()`, 157
`sum()`, 105, 174, 253, 255
`symplectic_form()`, 164
`tanh()`, 149
`tan`, 8
`taylor()`, 167
`text()`, 92
`then`, 243
`timeit()`, 158
`transpose()`, 165
`var()`, 36
`var`, 123
`vector()`, 113, 131
`verbose` (opción de comando), 170, 172

`weak_popov_form()`, 164
`while`, 244
`xmax` (opción de comando), 120
`xmin` (opción de comando), 120
`x` (variable predefinida de Sage), 36
`ymax` (opción de comando), 14, 90, 267
`ymin` (opción de comando), 14, 267
`zigzag_form()`, 164
`\sage`, 60
`dir` (opción de comando), 150

 aceleración, 82, 189
 aceleración gravitacional, 78, 168
 aceleración gravitacional, 102
 acetileno, 74
 acuario, 219, 270
 acumulador, 212, 255, 256
 adición, 1
 adición de polinomios, 32
 agua, 74, 75
 Al-Sabi Thabit ibn Qurra al Harrani, 145
 Alexandre-Théophile Vandermonde, 26
 algoritmo de graficación, 95
 altitud, 82
 altitud inicial, 78, 79
 altura, 82
 antiderivada, 48
 anualidad creciente, 125
 anualidad decreciente, 125
 Análisis de Entrada-Salida de Leontief, 65
 app, véase interactivo 263
 applet, véase interactivo 263
 aproximación de Stirling, 85
 aproximación decimal, 4, 31
 aproximación numérica, 227
 aproximación polinomial de funciones, 166
 aproximación racional, 178
 apóstrofo, 47
 arcocosecante, 9
 arcocosecante hiperbólica, 149
 arcocoseno, 9
 arcocoseno hiperbólico, 149
 arcocotangente, 9
 arcocotangente hiperbólica, 149
 arcosecante, 9
 arcosecante hiperbólica, 149
 arcoseno, 9
 arcoseno hiperbólico, 149
 arcotangente, 9
 arcotangente hiperbólica, 149
 aritmética modular, 83, 146
 arrastre hidrodinámico, 189
 arreglos, véase listas 251
 aritmética modular, 247
 ataque factorial de Pollard, 83
 protección contra el, 86
 atmósfera (unidad de presión), 70
 aumentar y hallar la RREF (estrategia), 133
 autocompletado de comandos, 44

autologaritmo, 230
 autovalor, 160, 163
 multiplicidad, 163
 autovector, 160, 163
 autovector derecho, 160
 ayuda, 44
 asíntota vertical, 14

 backslash (símbolo), 132, 133
 balance químico, 74
 balística, 78
 base b , 145
 binomial, 147
 biología, 70
 bucle, 210, 212, 221, 244
 iteración, 212
 bucle `for`, 210, 212, 256
 bucle `while`, 244
 bucles `for` vs. bucles `while`, 245
 bugs, 45

 caja registradora, 219, 250
 calidad del gráfico, 98, 109, 114
 calificaciones, 243, 255, 257
 cambio de variable, 124, 127, 128, 222
 cambio de variables, 190
 campo eléctrico, 87, 112, 114
 campo vectorial, 110, 201
 cantidad de divisores de un entero, 144
 cantidad de dígitos de un entero, 158
 capacidad de embarque, 73
 capacidad de producción, 73
 capital, 125
 carbono, 75
 cardioide, 98
 cargas eléctricas, 87, 112
 casilla de verificación, 271
 centiPoise (unidad de viscosidad), 71
 cero (de una función), 41
 cianuro de sodio, 75
 ciclo, 244
 circunferencia, 108
 Cobb-Douglas, 107, 117
 cociente entero, 147
 coeficiente de arrastre, 190
 coeficiente binomial, 147
 coeficientes binomiales, 175
 coerción, 105
 colaboración, 57
 columnas, 21
 combinación, 147, 148
 combustión, 74, 75, 77
 comentario (código), 229, 266
 comparación, 239
 compartir, 57
 Compartir (botón), 57
 compatibilidad retrógrada, 227
 completado de comandos, 5
 componente horizontal de la velocidad, 80

- componente vertical de la velocidad de disparo, 79
- composición, 31, 124, 127, 128
- compra de Alaska a Rusia por Estados Unidos, 159
- condiciones iniciales, 187
- conducción del calor (en una barra), 104
- conjuntos de nivel, 102
- constante de gravitación universal, 128
- constante de integración, 49, 78, 79
- convergencia
 - integral de Laplace, 192
- convergentes, 178
- coordenadas polares, 96
 - θ , 96
 - r , 96
- coprimos, 141, 142
- corchetes, 141
- cortes de Gomory, 181, 183
- cosecante, 8
- cosecante hiperbólica, 149
- cosecante hiperbólica inversa, 149
- coseno, 8
- coseno hiperbólico, 149
- coseno hiperbólico inverso, 149
- costo mínimo, 73
- costo por mil, tabla de, 126
- costos, 66, 67
- costos, función de, 67, 69
- cotangente, 8
- cotangente hiperbólica, 149
- cotangente hiperbólica inversa, 149
- Coulomb (unidad de carga eléctrica), 112
- CPT (costo por mil), 126
- criptografía, 82, 137
- Criptosistema RSA, 82
- Criterio de Sylvester, 112
- cuencas de atracción, 170
- curva de demanda, 90
- curva exponencial, 153, 155
- curva precio-demanda, 25
- Curvas de Lissajous, 108
- cálculo vectorial, 194
- cáncer, muerte por, 70
- código abierto, 45
- código de barras 2D, 57
- Código de Barras Matriciales, 57
- Código de Respuesta Rápida, 57
- código fuente, 45
- código QR, 57
- Código Quick Response, 57
- códigoQR, 57
- decaimiento radiactivo, 153, 155
- decimales exactos, 145
 - denominadores finitos
 - base 10, 145
 - base 2, 145
 - base b , 145
 - denominadores permisibles
 - base 10, 145
 - base 2, 145
 - base b , 145
- decimales finitos, 145
- declaración de variables, 36
- degree, 136
- demanda, 66
- demanda lineal con respecto al precio, 66
- demostración asistida por computadora, 177
- densidad de puntos, 99
- derivación, 46
- derivada, 46, 111, 129, 215, 223
- derivada de orden superior, 48, 130
- derivadas parciales, 111, 129, 172, 195
- desbalance químico, 74
- descenso por gradiente, 170
- desigualdades, 179
- deslizador, 268
- determinante, 137
- diferenciación, 46
- dirección de más rápido ascenso, 115
- dirección de más rápido descenso, 115
- distancia de vuelo, 78, 80
- distribución de los números primos, 142
- Dive into Python 3, 254, 260
- divergencia, 198
- divergencia de la rotacional, 203
- divergencia del gradiente, 199
- divergente, integral, 53
- divisibilidad, 247, 249
- división, 1
- división al tanteo, 247
- división entera, 247
 - residuo, 247
- división por cero, 235, 241
- divisores de un entero, 34, 144
- divisores propios, 145
- dióxido de carbono, 74
- dominio de una función, 120
- docstring, 44
- documentación, 44, 260
- documentación oficial de Sage, 45
- ecuaciones a una variable, 35
- ecuaciones cuadráticas, 36
- ecuaciones cúbicas, 41
- ecuaciones diferenciales, 184, 189
- ecuaciones diferenciales con valor inicial, 187
- ecuaciones lineales, 37
- ecuaciones no lineales, 37
- ecuaciones normales, 166
- ecuaciones paramétricas, 108
- ecuación cuadrática, 80, 81
- ecuación matricial por derecha, 132
- ecuación matricial por izquierda, 132
- ecuación química, 74
- Edmond Halley, 232
- eficiencia, 212, 225, 227, 232, 237, 249
- elipse, 108
- embarque, 72

- empuje, 82, 189
- Encapsulated Postscript, 58
- enfermedad cardiovascular, muerte por, 70
- enlace web permanente, 57
- enlace web temporal, 57
- EPS, 58
- error (cuadrático), 153
- error en tiempo de ejecución, 241
- error relativo, 31
- errores, 45
- errores comunes, 3
- escala log-log, 116
- escala log-log-log, 117
- escala logarítmica, 116
- espacio log-log, 116
- espacio log-log-log, 117
- espiral exponencial, 98
- etiuetado de los ejes, 89
- evaluación, 123, 124, 222
- evaluación a medias, 105, 124
- evaluación numérica, 105, 213, 214
- evaluación numérica vs exacta, 105
- evaluación numérica vs. exacta, 227
- Evaluar (botón), 1
- Evaluate (botón), 1
- excepción, 239
 - arrojar una, 239
 - levantar una, 239
- explotar, *véase* división por cero 235
- exponenciación, 1, 2, 118
- exponenciación matricial, 132
- exponenciales, 5
- factores primos, 83
- factorial, 147, 244
- factorización, 82, 247
 - mediante división por tanteo, 247
- factorización de enteros, 34, 140
- factorización de polinomios, 33
- factorización LU, 164
- factorización matricial, 164
- factorización PLU, 164
- falla de motor, 82
- filas, 21
- finanzas, 210
- flujo de un fluido, 201
- for, 256
- forma canónica, 164
- forma escalonada de una matriz, 134
- forma escalonada por filas, 21
- forma escalonada reducida por filas, 19, 132
- forma expandida de un polinomio, 33
- forma factorizada de un polinomio, 33
- formato de la salida, 211, 215, 242, 248
- fosfato tricálcico, 75
- fracciones continuas, 178
- fracciones parciales, 52
- fractorización LUP, 164
- fuego contra-batería, 81
- fuerza eléctrica, 112
- fuidos, 70
- funcines lementales
 - raíz cuadrada, 3
- funcines trigonométricas
 - coseno, 8
 - seno, 8
 - tangente, 8
- funciones a dos variables, 171
- funciones elementales, 3
 - logaritmo, 6
 - base 42, 6
 - binario, 6
 - común, 6
 - natural, 6
 - raíces de orden superior, 4
 - raíz cuadrada, 7
- funciones hiperactivas, 95
- funciones matemáticas, 28
- funciones multivariadas, 129
- funciones trigonométricas, 8
- funciones trigonométricas hiperbólicas, 149
- funciones trigonométricas hiperbólicas inversas, 149
- funciones trigonométricas inversas, 8
 - arcocosecante, 9
 - arcocoseno, 9
 - arcocotangente, 9
 - arcosecante, 9
 - arcoseno, 9
 - arcotangente, 9
- funciones trigonométricas recíprocas, 8
 - cosecante, 8
 - cotangente, 8
 - secante, 8
- función ϕ , 143
- función σ , 144
- función τ , 144
- función (matemática), 216
- función (programación), *véase* subrutina 216
- función a dos variables, 105
- función a una variable, 130
- función contador de números primos, 239
- función convexa, 170
- función cuadrática, 25
- función de aceleración, 78, 79, 82
- función de altitud, 78, 79, 82
- función de costo, 219, 221, 270
- función de ganancia, 255
- función de ingreso, 255
- función de posición, 82
- función de producción de Cobb-Douglas, 107
- función de velocidad, 78, 79, 82
- Función Error, 55
- función eta de Riemann, 222
- función explícita, 100, 186
- función formal, 61, 63, 130, 195
- función implícita, 100, 186
- función matemática, 126, 128
- función mayor entero, 147

función menor entero, 147
 función multivariada, 105
 función multivariada, 123
 función máximo entero, 147
 función paramétrica, 108
 función piso, 147
 función sigma, 144
 función signo, 120
 función tau, 144
 función techo, 147
 función univariada, 130
 función vectorial, 110, 198, 200
 función zeta de Riemann, 174
 fundición, 75
 Fórmula Cuadrática, 36
 Fórmula Cúbica de Cardano, 41
 fórmula de Halley, 232
 fórmula de Newton, 128
 fórmula de Newton para la fuerza gravitacional, 168
 fósforo, 75

Galileo, 184
 ganacia, función de, 69
 ganancias, 67
 ganancias, función de, 67
 ganancia máxima, 69
 ganancias, 66
 gasolina, 75
 gradiente, 111, 130
 dirección de más rápido ascenso, 115
 dirección de más rápido descenso, 115
 gradiente y gráfica de contorno, 115
 grado, 136
 grados a radianes, 9
 graficación, 210, 251
 gravedad, 128
 gráfica de campo de pendientes, 188
 gráfica de conjunto de puntos, 116
 gráfica de contorno y gradiente, 115
 gráfica de contornos, 102
 valores específicos, 108
 gráfica de dispersión, 116
 gráfica de superficie, 111
 gráfica vectorial, 87, 110
 polos de una función, 114
 vectores gigantes, 114
 gráficas, 9
 gráficas de dispersión, 151
 gráficos adaptativos, 95
 guardar imagen, 11, 58

Hessiano, 195
 determinante, 196
 hidrógeno, 74
 hidróxido de calcio, 75
 hierro, 75
 hipoteca, 125, 126
 Hipótesis de Riemann, 174
 hoja de tarifas, 126

HTML, 268
 identidades, 175, 199
 incertidumbre, 52
 indentidades, 203
 infinitas soluciones, 74, 76
 infinito, 8, 55, 96, 150, 174, 213, 257
 ingreso máximo, 68, 69
 ingresos, 66
 ingresos, función de, 66, 69
 Inmersión en Python 3, 254, 260
 integración, 78, 79, 203
 integración numérica, 50, 51
 integral, 48, 203
 Integral de Fresnel, 49
 integral de laplace, 192
 integral definida, 49, 272
 integral definida exacta, 48
 integral divergente, 192
 Integral Gaussiana, 49, 55
 integral imposible, 50
 integral indefinida, 48
 integral numérica aproximada, 48
 integrales impropias, 53
 integrales múltiples, 203
 interactivo, 263, 268
 diseño, 265
 parámetros opcionales booleanos, 271
 prerrequisitos, 264
 proceso multietapa, 264
 recta tangente, 265
 interés compuesto, 1, 7, 125, 126, 210
 interés compuesto, 44
 interés simple, 1
 inversa de una matriz, 134, 135
 iteración, 212

Jacobiano, 198
 Jean-Leonard Marie Poiseuille, 70
 John Pollard, 83
 Jules Antoine Lissajous, 108

kilogramo (unidad de masa), 82

Laplace, 191
 Laplaciano, 196, 199
 lazy list, 178
 lemniscata, 99
 Leonhard Euler, 143
 Ley de Coulomb, 112
 Ley de Poiseuille, 70, 71
 ley de potencia, 116
 ley del cuadrado inverso, 112
 ligras por pulgada cuadrada (unidad de presión), 70
 linestyle (opción de comando), 167
 link temporal, 57
 lista, 7, 35, 37, 141, 144, 212, 251, 258
 construcción de, 258
 lista peresosa, 178
 listas, 252, 254

- acceso a elementos, 253, 255
- aleatorización de orden, 253
- añadir un elemento, 252
- concatenación, 251, 252
- indexación, 253
 - índices negativos, 253
- longitud, 144, 252, 255
- pertenencia, 253
- producto de elementos, 253
- remoción de elementos, 253
- suma, 251, 252
- suma de elementos, 253
- vs. conjuntos, 252
- lisura, 83
- log-log, 116
- log-log-log, 117
- logaritmo, 6
 - base 42, 6
 - binario, 6
 - común, 6, 230
 - natural, 6
- logaritmo binario, 6
- logaritmo común, 6
- logaritmo en base 42, 6
- logaritmo natural, 6
- longitud arterial, 70
- longitud de un entero, 158
- longitud de un vector, 139
- Lorentz, 215
- límite, 29, 96, 213
 - evaluación numérica, 214
- límite inferior de integración, 49
- límite por derecha, 96
- límite por izquierda, 96
- límite superior de integración, 49
- límites, 149
- límites infinitos, 150
- límites laterales, 96, 150
- manos de Póker
 - orden relevante, 148
- Manual de Referencia de Sage, 45
- mapeo, 62, 130
- Marie-Sophie Germain, 86
- masa, 82
- masa de la Tierra, 128
- matemáticas finitas, 65
- matrices de Vandermonde, 26
- matrix identidad, 24
- matriz, 19, 74, 130
- matriz aumentada de un sistema, 76
- matriz Hessiana, 112, 195
 - determinante, 196
- matriz identidad, 131, 135
- matriz inversa, 134, 135
- matriz invertible, 134, 135
- matriz Jacobiana, 198
- matriz singular, 135, 161
- Maxima (software), 56, 192
- mayúsculas, 5
- media evaluación, 105, 124
- meganeutons (unidad de empuje), 82
- mensaje de error, 241
- microeconomía, 66, 107
- miliPascal segundo (unidad de viscosidad), 71
- milímetros de mercurio (unidad de presión), 70
- minas, 72
- minimización, 270
- minúsculas, 5
- mmHg (unidades de presión), 70
- momento de impacto, 81
- momento de lanzamiento, 81
- monedas, 217, 250
- monóxido de carbono, 75
- movimiento de proyectiles, 82
- multiplicación, 1
- multiplicación de polinomios, 32
- multiplicación matricial, 131
- multiplicadores de Lagrange, 171
- multivariada, función, 32
- máxima altura, 82
- máximo, 147
- máximo común divisor, 33, 140
- máximo global, 196
- máximo local, 112
- máximo local único, 170, 196
- máximo precio factible, 66
- método de Halley, 232
- método de Newton, 222, 232, 237
 - convergencia, 226
 - falla, 235
 - forma tabular, 224
 - multiplicidad de las raíces, 246
- método de newton, 245
- método de Newton-Raphson, véase método de Newton 232
- método, véase subrutina 216
- Métodos Cuantitativos, 65
- métodos numéricos, 155
- métodos simbólicos vs. numéricos, 173
- mínimo, 147
- mínimo común múltiplo, 76, 141
- mínimo costo, 69
- mínimo global, 196
- mínimo local, 112
- mínimo local único, 170, 196
- mínimos cuadrados, 165
- mód, 84, 146
- mútuamente primos, 141
- Nathaniel Bowditch, 108
- Newton segundo / metro² (unidad de viscosidad), 71
- Newtons por metro cuadrado (unidad de presión), 70
- norma de un vector, 139
- notación
 - funciones vectoriales, 194
- notación de evaluación, 63
- notación de punto, 131
- notación vectorial, 131

- transpuesta, 132
- nulidad, 136
- nulidad de una matriz, 136
- nulidad derecha, 136
- nulidad izquierda, 136
- numeración de columnas de una matriz, 23
- numeración de filas de una matriz, 23
- numeros amigos, 145
- numérica, resolución, 35
- núcleo, 135
 - degree, 136
 - grado, 136
 - rango, 136
 - rank, 136
- núcleo de una matriz, 135
- núcleo derecho, 135
- núcleo derecho de una matriz, 135
- núcleo izquierdo, 135
- núcleo izquierdo de una matriz, 135
- números B -lisos, 83
- números complejos, 7, 117, 257
- números enteros, 257
- números gigantes, 158
- números lisos, 83
- números primos, 82, 140, 249
- números primos en un intervalo, 140, 239
- números racionales, 257
- números reales, 257
- octano, 75, 77
- OMS, 70
- ondas sinusoidales, 270
 - amplitud, 270
 - frecuencia, 270
- open-source, 45
- operadores vectoriales, 194
- optimización, 72, 179, 180, 196, 270
 - multivariada, 169
 - restringida, 171
 - sin restricciones, 170
- Organización Mundial de la Salud, 70
- oro, 75
- overfitting, 25
- oxígeno, 74, 75
- papel log-log, 116
- par ordenado, 251
- pares de números amigos, 145
- parábola, 68
- parábola de vuelo, 81
- parábola tangente, 166
- paréntesis, 2, 127
- Pascal (unidad de Presión), 70
- permalink, 57
- permitividad del vacío, 112
- permutaciones de un conjunto, 148
- permutación, 147, 148
- peso, 189
- phi de Euler, 143
- pies/segundo² (unidad de aceleración), 102
- piso, 147
- pivote, 76
- plano complejo, 117
- PNG, 58
- polihedro de un programa lineal, 179, 183
- polinomio, 32
- polinomio característico, 161, 163
- polinomio cúbico, 234
- polinomio cúbico mónico deprimido, 41
- polinomio de grado cuatro, 26, 41
- polinomio de MacLaurin, 166
- polinomio de Taylor, 166, 215
 - quinto grado, 167
- polinomio mínimo, 164
- polos de una función, 114
- Portable Network Graphics, 58
- potenciación, 1, 2, 118
- potenciación matricial, 132
- prescio de transporte, 73
- presión, 70
- preuba de escritorio, 261
- primer factorial mayor que, 244
- primos de Sophie Germain, 86
- primos seguros, 86
- procedimiento, véase subrutina 216
- procesamiento de calificaciones, 243, 255, 257
- producto, 1
- producto cruz, 139
- producto de polinomios, 32
- producto escalar, 139
- producto interior, 139
- producto matricial, 131
- producto por un escalar, 139
- producto punto, 139
- producto vectorial, 139
- programa lineal, 73
- programación, 209, 216, 220, 229, 234, 237, 241, 260, 266
- Programación Lineal, 72
- programación lineal, 179
 - familia de variables, 180, 182
 - función objetivo, 180
 - restricciones, 180
 - variables binarias, 183
 - variables booleanas, 183
 - variables enteras, 183
- programación por objetivos, 181
- programadores reales, 167
- progresión aritmética, 213
 - diferencia común, 213
- promedio, 255
- proyectil, 78
- proyectil balístico, 78
- préstamo, 126
- PTF (Pequeño Teorema de Fermat), 84
- punto, 251
- punto de equilibrio, 68, 69
- punto de equilibrio atractivo, 112
- punto de equilibrio repulsivo, 112

- punto de saturación, 66, 68
- puntos radialmente tangentes, 100
- Python (lenguaje de programación), 209, 216
- página web interactiva, 268
 - plantilla, 268
 - llenado, 268, 269
- página web interactiva, *véase también* interactivo 263
- Póker
 - orden irrelevante, 148
- quebrado invertido (símbolo), 132, 133
- química, 74
- radianes, 8, 96
- radianes a grado, 9
- radianes a grados, 139
- radio arterial, 70, 71
- ramificación y poda, método de, 183
- rango, 136
- rango de una matriz, 136
- rango de variación del eje x , 11, 120
- rango de variación del eje y , 14
- rango del proyectil, 80
- rank, 136
- razón de flujo de la sangre, 71
- razón del flujo del volumen, 70
- raíces de orden superior, 4, 117, 119
- raíces de un polinomio, 257
- raíces de una ecuación, 257
- raíz
 - de una ecuación
 - multiplicidad, 247
 - raíz (de una función), 41
- raíz cuadrada, 3, 7
- raíz cúbica, 117
- raíz cúbica real, 119
- reacción química, 74
- reacción ácido-base, 75
- recta de mejor ajuste, 152, 166
- recta tangente, 16, 166, 231, 265
 - ecuación pendiente-ordenada en el origen, 266
 - ecuación punto-pendiente, 266
- redondeo hacia abajo, 147
- redondeo hacia arriba, 147
- REF, 134
- regresión, 154
- regresión cuadrática, 153, 155
- regresión exponencial, 153, 155
- regresión lineal, 153, 154
- relación precio-demanda, 69
- relatividad especial
 - transformada de Lorentz, 215
- resistencia del aire, 81
- resolución de ecuaciones con una variable, 35
- resolución de ecuaciones con varias variables, 36, 37
- resolución de varias ecuaciones, 37
- resolución numérica de ecuaciones, 41
- resta, 1
- resta de matrices, 131
- resta de polinomios, 32
- resta de vectores, 139
- restricciones, 72
- Riemann, 222
- rosa, 96, 98
- rosetón, 96, 98
- rotacional, 200
 - interpretación, 201
- rotacional del gradiente, 203
- RREF, 19, 22, 76, 77, 135
- rref, 132
- Sage Cell Server, 1
- salida de una subrutina, 229, 237
- saltos de línea, 18
- sangrado (del texto), 210
- sangrado (texto), 217, 233
- satélite, 128
- secante, 8
- secante hiperbólica, 149
- secante hiperbólica inversa, 149
- segunda derivada, 48
- segundo mmHg (unidad de viscosidad), 71
- selector, 271
- seno, 8
- seno hiperbólico, 149
- seno hiperbólico inverso, 149
- serie, 174, 176, 214
- serie aritmética, 174
- serie de Taylor, 215
- serie geométrica, 176
 - convergencia, 176
- Servidor de Celdas de Sage, 1
- Share (botón), 57
- SHIFT+ENTER (combinación de teclas), 2
- silo cilíndrico, 221
- simbólica, resolución, 35
- simulación numérica, 82
- Sir Isaac Newton, 222, 232
- sistema binario, 156
- sistema de ecuaciones sin soluciones, 26
- sistema de ecuaciones con infinitas soluciones, 27
- sistema de ecuaciones con infinitas soluciones, 137
- sistema de ecuaciones lineales, 19, 75, 133
- sistema de ecuaciones sin soluciones, 137
- sistema de numeración
 - Babilonio, 146
 - base 20, 146
 - base 60, 146
 - binario, 146
 - decimal, 146
 - Maya, 146
- sistema hexadecimal, 156
- sistema octal, 156
- sistema rectangular de ecuaciones lineales, 164
 - solución aproximada, 165
- sistemas de ecuaciones con infinitas soluciones, 137
- sistemas de ecuaciones lineales, 74
- sobreajuste, 25

- soluciones complejas, 40
- solución asistida por computadora, 7, 177, 184
- sopa de letras, 125
- SSL (Secure Socket Layer), 82
- STEM, 58
- subprograma, *véase* subrutina 216
- subrutina, 216, 225
 - argumentos, 217, 225
 - nombrados, 218
 - orden, 218
 - documentación, 217
 - parámetros, 227
 - nombrados, 228
 - opcionales, 228, 232–234, 237
 - valor por defect, 237
 - valor por defecto, 228
 - parámetros, *véase* subrutina
 - argumentos 217
 - terminación, 237, 245, 249
- subrutinas
 - anidadas, 230
 - salida, 229
- sucesión, 221, 223
- Sudoku, 157
- suma, 1
- suma de fuerzas, 78, 114
- suma de imágenes, 16
- suma de los cuadrados de los divisores de un entero, 144
- suma de los cubos de los divisores de un entero, 144
- suma de los divisores de un entero, 144
- suma de matrices, 131
- suma de polinomios, 32
- suma de vectores, 139
- suma finita, 174, 175
- suma infinita, 174
- superposición de gráficas, 16
- sustitución, 190
- sustracción, 1
- sustracción de polinomios, 32

- TAB (tecla), 5, 44, 210, 218
- tabla, 210, 222, 224
- tablas, 71
- taconita, 72
- tangente, 8
- tangente hiperbólica, 149
- tangente hiperbólica inversa, 149
- tansformada de Lorentz, 215
- Taylor, 215
- techo, 147
- temperatura en función del tiempo y la posición, 104
- Teorema de Cayley-Hamilton, 162
- Teorema de las Raíces Enteras, 34
- Teoría de Números, 83, 140, 147
- termodinámica, 104
- test de primalidad, 140, 142, 143
- texto de documentación, 44, 217
- Thebit, 145
- Thebith, 145

- tiempo de vuelo, 80, 81
- tipografía de camello, 127
- TLS (Transport Layer Security), 82
- transformada de Laplace, 191
- transformada inversa de Laplace, 193
- transposición, 165
- tratamiento de aguas residuales, 75
- trayectoria de vuelo, 82
- trayectorias balísticas, 78
- traza de una matriz, 196, 198
- trigonometría, 8
 - grados a radianes, 9
 - radianes a grados, 9
- tropedo, 189
- trébol, 98
- Tsu Ch'ung-Chih, 31
- tutorial, 45

- unidad imaginaria, 7
- utilidad, 67
- utilidad máxima, 69
- utilidad, función de, 67, 69

- valor absoluto, 10, 120, 238
- valor de una función, 61, 130, 195
- valor futuro de una anualidad, 125
- valor futuro de una anualidad creciente, 125
- valor presente de una anualidad, 2, 125
- valor presente de una anualidad decreciente, 59, 125
- vapor de agua, 74
- variabel
 - binaria, 234
- variable, 220
 - booleana, 234
 - constante, 220
 - global, 220
 - local, 220
- variable constante, 30, 126, 128
- variable intermedia, 30, 126, 128
- variables determinadas, 76
- varias variables, función de, 32
- vector
 - longitud, 112
 - magnitud, 112
 - norma, 112
- vector columna, 132
- vector fila, 132
- vectores, 137
- velocidad, 82
- velocidad de cálculo, 158
- velocidad de flujo de la sangre, 71
- velocidad del flujo del volumen, 70
- velocidad horizontal, 80
- velocidad inicial, 78
- velocidad terminal, 191
- velocidad vertical, 79
- verbosidad, 232, 251
- viscosidad, 71
- viscosidad de la sangre entera, 71

viscosidades comunes, 71

Warning: Desired error not necessarily achieved due to
precision loss, 172

wavelets, 96

WHO, 70

Zu Chongzhi, 31

Álgebra Lineal, 130

Álgebra Matricial, 130

ácido clorhídrico, 75

ácido fosfórico, 75

ácido hipocloroso, 75

álgebra matricial, 74

álgebra modular

 inverso multiplicativo, 143

 recíproco, 143

ángulo entre dos vectores, 139

ángulo inicial, 78

área de visualización de un gráfico, 120

área de visualización de un gráfico, 13

óptimo de la parábola, 68

órbita, 128

óxido ferroso, 75