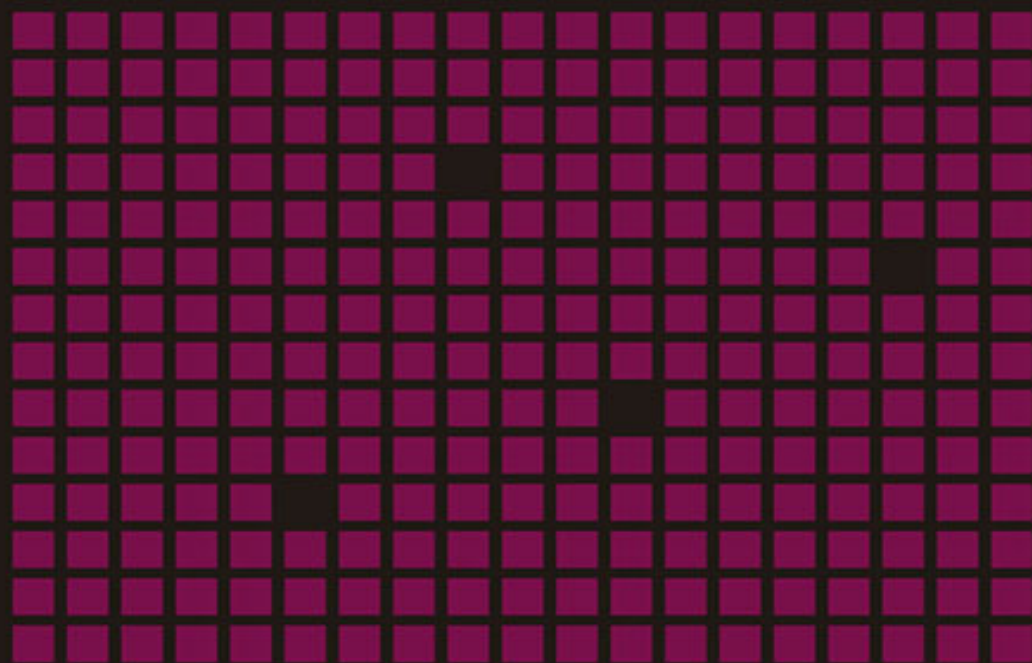


Programación científica



Técnicas y fundamentos para el desarrollo de software

PABLO ALCAIN / CECILIA JARNE
RODRIGO LUGONES
MARÍA GRACIELA MOLINA



Universidad
Nacional
de Quilmes
Editorial

Pablo Alcain / Cecilia Jarne / Rodrigo Lugones /
María Graciela Molina

Programación científica

Técnicas y fundamentos para el desarrollo de software



Bernal, 2022

UNIVERSIDAD NACIONAL DE QUILMES

Rector
Alfredo Alfonso

Vicerrectora
Alejandra Zinni

Colección Nuevos enfoques en ciencia y tecnología
Dirigida por Diego Golombek

Programación científica: técnicas y fundamentos para el desarrollo de software / Pablo Alcain... [et al.]. - 1a ed. - Bernal: Universidad Nacional de Quilmes, 2022.

210 p.; 22 x 15 cm. - (Nuevos enfoques en ciencia y tecnología / Diego Golombek)

ISBN 978-987-558-795-3

1. Ciencias Tecnológicas. 2. Nuevas Tecnologías. 3. Lenguajes de Programación. I. Alcain, Pablo
CDD 005.1

Primera edición e-book, 2022

© Pablo Alcain, Cecilia Jarne, Rodrigo Lugones, María Graciela Molina, 2022

© Universidad Nacional de Quilmes, 2022

Universidad Nacional de Quilmes

Roque Sáenz Peña 352

(B1876BXD) Bernal, Provincia de Buenos Aires

República Argentina

ediciones.unq.edu.ar

editorial@unq.edu.ar

ISBN: 978-987-558-795-3

Queda hecho el depósito que marca la ley 11.723

Impreso en Argentina

Índice

[Los autores](#)

[Prólogo](#)

[Introducción](#)

[Capítulo 1. Lenguajes interpretados y sus aplicaciones científicas, Cecilia Jarne](#)

[Introducción](#)

[Introducción a Python](#)

[Librerías científicas de Python](#)

[¿Qué más podemos hacer con los datos?](#)

[Hands-on: Python como lenguaje de *scripting*](#)

[Bibliografía](#)

[Capítulo 2. Herramientas para el desarrollo de software propio y colaborativo, Rodrigo Lugones](#)

[Sistemas de control de versiones](#)

[Flujo de trabajo \(o cómo pensar el desarrollo del software\)](#)

[Documentación](#)

[*Unit testing*](#)

[Conclusiones](#)

[Hands-on de git](#)

[Bibliografía](#)

[Capítulo 3. Desarrollo de software modular, Pablo Alcain](#)

[Introducción](#)

[Abstracción en funciones](#)

[La performance, esa palabra](#)

[Estructuras de datos](#)

[Conclusiones](#)

[Bibliografía](#)

Capítulo 4. Cómo combinar lenguaje compilado e interpretado, *Pablo Alcaín*

[Introducción](#)

[Sobre ctypes y su uso básico](#)

[Discusión sobre una forma más orientada a objetos](#)

[Estructuras](#)

[Interfaz de c/Python orientada a objetos a través de ctypes](#)

[Conclusiones](#)

[Bibliografía](#)

Capítulo 5. Procesos de optimización, *María Graciela Molina*

[Introducción](#)

[Optimización de software: algoritmos y estructuras de datos](#)

[Optimización de hardware](#)

[Debugging y profiling](#)

[Conclusión](#)

[Hands-on: debugging y profiling](#)

[Bibliografía](#)

Capítulo 6. Programación en paralelo, *Cecilia Jarne*

[Arquitectura del computador y procesamiento de datos](#)

[Procesadores escalares, vectoriales y superescalares](#)

[Programación en entornos paralelos: HPC](#)

[Programación en entornos paralelos: memoria compartida y OpenMP](#)

[Programación en entornos paralelos: memoria distribuida y MPI](#)

[GPU](#)

[Hands-on: programación en entornos paralelos](#)

[Bibliografía](#)

Capítulo 7. Ejemplos de trabajo en el ámbito del alto desempeño, *Pablo Alcain, Cecilia Jarne, Rodrigo Lugones, María Graciela Molina*

Quantum ESPRESSO

Lapack

Openblas

Bibliografía

Bibliografía general

Los autores

Pablo Alcain. Físico de la Universidad de Buenos Aires (UBA). Hizo su doctorado en Astrofísica Nuclear, estudiando la estructura interna de las estrellas de neutrones a través de técnicas de dinámica molecular computacional. Se especializó en técnicas de cómputo de alto desempeño en placas gráficas y en cálculo distribuido, compatibilizando código escrito en C, Assembler, CUDA y Python. Actualmente se dedica a la ciencia de datos y al desarrollo de software para *machine learning*.

Cecilia Jarne. Realizó su doctorado en Física en el Instituto de Física de La Plata (IFLP) y el Departamento de Física de la Universidad Nacional de La Plata. Su experiencia en investigación se basa en análisis de grandes datos, primero en física de rayos cósmicos de alta energía y luego durante el posdoctorado en el Instituto de Física de Buenos Aires (IFIBA-UBA) analizando cantos de pájaros. Posee diversas publicaciones en revistas indexadas relacionadas con ambas áreas. Actualmente es investigadora asistente en Conicet y se encuentra trabajando en redes neuronales recurrentes y sistemas complejos. También es docente de grado y posgrado en la Universidad Nacional de Quilmes (UNQ) y semanalmente da clases en una escuela técnica (que fue su escuela), la EEST 2 de Quilmes.

Rodrigo Lugones. Es doctor en ciencias físicas de la Universidad de Buenos Aires y científico de datos en Despegar. Tiene más de diez años de experiencia docente en distintos ámbitos universitarios, tanto en materias de grado como de posgrado. En su doctorado, trabajó en turbulencia en plasmas astrofísicos, aplicando computación de alto desempeño para la realización de simulaciones numéricas en clusters. Desde 2018, trabaja en inteligencia artificial, principalmente en temáticas de visión por computación y aprendizaje de máquina más tradicional. Es cofundador y CTO de Ninja IA, una empresa dedicada a proveer soluciones de inteligencia artificial.

María Graciela Molina. Es docente de grado e investigadora del Departamento de Ciencias de la Computación de la Facultad de Ciencias Exactas y Tecnología de la Universidad Nacional de Tucumán (FACET-UNT). Es docente de posgrado del Doctorado en Ciencias Exactas e Ingeniería y de la Maestría en Métodos Numéricos y Computacionales, ambos de la FACET-UNT. Actualmente, es directora del Laboratorio de Computación Científica (LabCC) y del centro de Space Weather de la FACET-UNT. Además, es docente de posgrado de la UNQ. Es Investigadora del Conicet. Desarrolla su investigación en el área de inteligencia artificial aplicada a Space Weather, sensado remoto y sistemas de radares.

Prólogo

El Abdus Salam International Centre for Theoretical Physics, también conocido por sus siglas ICTP, es un instituto de investigación científica auspiciado por la Organización de las Naciones Unidas para la Educación, la Ciencia y la Cultura (Unesco), el Organismo Internacional de Energía Atómica (IAEA) y el gobierno italiano. Sus instalaciones se ubican en la ciudad de Trieste, en Italia, a un costado del castillo de Miramar.

El ICTP es un lugar maravilloso por varias razones que exceden la geografía de sus instalaciones. La institución fue fundada en 1964 por el físico pakistaní y premio Nobel Abdus Salam, con el objetivo de promover para los científicos de los países en desarrollo la educación continua y las habilidades que necesitan para el desarrollo de disciplinas centradas en la física teórica y la matemática. Con el tiempo, estas disciplinas se fueron ampliando. El ICTP ha sido una herramienta creada con la intención de detener la fuga de cerebros científicos del mundo en desarrollo.

Estos párrafos anteriores pueden sonar a una historia poética o a un panfleto publicitario, pero afortunadamente son ciertos. El ICTP es el corazón de la historia de muchos investigadores que vivimos en países en vías de desarrollo como la Argentina. Acá comienza nuestra historia como grupo de colaboradores, luego de amigos, y de cómo surge la necesidad de escribir este libro.

El ICTP tiene varios eventos anuales internacionales de diversas disciplinas. En particular, tiene una escuela de Técnicas de Programación Científica y otra de Programación en Paralelo, ambas intensas, mentalmente muy demandantes y extremadamente satisfactorias. Pablo Alcain, Graciela Molina y yo (Cecilia Jarne) nos conocimos en el primero de estos eventos. En ese entonces, yo acababa de terminar mi doctorado en Física en la Universidad Nacional de La Plata (UNLP), y estaba por comenzar mi posdoctorado en la Universidad de Buenos Aires (UBA). Por su parte, Pablo realizaba su doctorado en la UBA y Graciela, el suyo en la Universidad Nacional de Tucumán (UNT).

El evento del ICTP, antes de la pandemia, se realizaba cada dos años en Trieste, y los años intermedios en una locación internacional. A nosotros tres

nos tocó conocernos en 2015 en la ciudad de San Pablo, Brasil. Entre líneas de código y caipiríñas, comenzamos a hablar de cuánta falta hacía promover estos conocimientos “en casa”.

En ese curso también conocimos a otro colega, Pablo Echeverría, quien trabajaba en ese momento en el Servicio Meteorológico Nacional argentino. Él nos proporcionó la invitación clave para dar un curso allí, ese mismo año, el cual dio comienzo a una serie sin pausa de *workshops*, capacitaciones y finalmente cursos de posgrado que seguimos dando, para que principalmente estudiantes de doctorado e investigadores jóvenes puedan suplir las carencias de formación en técnicas de programación en las distintas carreras de grado.

Rodrigo Lugones y Pablo Alcain son amigos desde la escuela. Ambos estudiaron juntos en la secundaria y luego fueron juntos a la UBA, donde estudiaron la Licenciatura en Física. En mi caso, a Rodrigo lo conocí cuando por azar (aunque también por decantamiento) compartimos la Escuela de Programación en Paralelo del ICTP. Cuando Pablo Echeverría migró a España, hacia fines del 2016, Rodrigo ocupó su lugar, llegando a la conformación actual del grupo.

El vínculo de colaboración y de amistad entre los cuatro ha hecho que nos encontremos frecuentemente (cuando podemos) para escribir algún trabajo, para dar una charla, para dictar un curso o, como en este caso, para escribir un libro. Nos ha unido también el objetivo de universalizar el conocimiento: hemos rotado los cursos por diversos lugares del país (Buenos Aires, Tucumán, Quilmes, Córdoba), hemos puesto énfasis en la federalización de los participantes. Hemos sentido un gran compromiso por involucrar a más mujeres en el ámbito de la programación científica. Y si bien partimos de distintas formaciones (tres físicos pero trabajando en temáticas muy distintas, y Graciela, con su origen en informática), los cuatro frecuentemente compartimos nuestras experiencias, pues los desafíos a los que nos enfrentamos suelen ser parecidos: pensar en términos de código, plantear el mejor algoritmo para analizar datos, hacer una simulación o resolver un problema son situaciones comunes en ciencia a lo largo de muchas disciplinas.

Hay muchas personas e instituciones fuertemente responsables de la existencia de este libro. En primer lugar, Ivan Girotto, del ICTP, nos ha incentivado de distintas maneras para construir y dar nuestros cursos en la Argentina, y para los científicos de Latinoamérica en general. Cada vez que

vamos por algún motivo al ICTP nos quiere ver, pregunta por los cuatro y nos acompaña moralmente. Eso nos llena de alegría.

Otra persona clave es Diego Golombek, quien estuvo allí para leer la propuesta de nuestro libro. Diego, además de hacer ciencia de calidad en su laboratorio de cronobiología en la Universidad Nacional de Quilmes (UNQ), también se dedica con gran compromiso a fomentar la formación científica y tecnológica en todos los niveles posibles; esto es un trabajo sumamente valioso.

Asimismo, la UNQ, mi lugar de trabajo, donde desarrollo mi actividad científica, nos ha brindado su apoyo incondicional a lo largo de varios años, otorgándonos un espacio formal para el desarrollo de proyectos científicos y educativos como este. Estamos completamente agradecidos por el espacio que nos dan.

Por supuesto, también queremos contarles que para que un libro quede bien, detrás de los autores hay un trabajo de edición muy extenso. A la Editorial de la UNQ también le agradecemos mucho.

Esta es una breve historia de quiénes somos, y de por qué y cómo decidimos escribir este libro. Intentamos que los contenidos sean perdurables, más que una moda, que ofrezcan una forma de pensar el programar y no simplemente cómo utilizar herramientas específicas. Deseamos que les permitan construir mejor conocimiento científico a través del uso del código de acceso abierto y eficiente. Y que al escuchar “eficiencia”, no solo piensen que el programa corra rápidamente, sino también que se desarrolle, se mantenga, se use, se modifique y se adapte al nuevo hardware rápidamente.

Esperamos que puedan disfrutar leyendo este libro tanto como nosotros al escribirlo, y que puedan implementar algunas de las ideas que aquí compartimos.

Pablo Alcain / Cecilia Jarne / María Graciela Molina / Rodrigo Lugones

Introducción

El uso de las computadoras y lenguajes de programación se han vuelto cada vez más imprescindibles para la investigación de punta en muchas áreas científicas y en el desarrollo de tecnología. La mayoría de las carreras de grado no contempla una formación específica en estos temas como el análisis computacional de datos o el desarrollo de simulaciones. Como consecuencia, el ecosistema de programación y desarrollo de software científico está alejado de las herramientas actuales que, en el mejor de los casos, se aprenden de fuentes muy diversas con calidad muy dispar y de manera informal.

Este libro tiene como objetivo ser un aglutinante de las distintas herramientas y habilidades que debe tener un científico para poder realizar las tareas que requieran el uso de una computadora, buscando transmitir una filosofía de trabajo que trascienda las herramientas concretas que se utilicen. El título de este texto no es casual, y es alrededor de la definición de *calidad* que se construye esta filosofía de trabajo. Un software de calidad tiene que tener cuatro características fundamentales: ser modular, reutilizable, verificable y documentado. Otras características deseables están relacionadas con la escalabilidad, la portabilidad y el ser amigable con el usuario.

La *modularidad* de un software refiere a separar la funcionalidad de un programa en distintas unidades, independientes entre sí, de modo de que cada una de ellas pueda realizar una y solo una tarea. La ventaja más evidente es que se puede dividir un proyecto en varios proyectos distintos, cada uno con un objetivo concreto y más rápido de alcanzar. De este modo, además, se vuelve más fácil la colaboración entre varios desarrolladores, ya que pueden trabajar en unidades separadas solo acordando cómo se comunican entre ellas.

El software es *reutilizable* si sus partes pueden ser utilizadas en otro programa sin ser escritas nuevamente. Evidentemente, un software reutilizable tiene que tener cierto grado de modularidad, pero no todo software modular es reutilizable: además, hay que tener precaución en que

las interfaces sean lo más simples posibles y dependan de estructuras de datos estandarizadas.

La *verificabilidad* es un concepto fundamental en el desarrollo de la ciencia, por lo que *debe serlo también* para el desarrollo de software científico. Para que un software sea verificable tenemos que poder estudiar cada una de sus partes por separado, analizando casos de prueba generales que pueden incluir circunstancias que inicialmente no fueron pensadas. De esta manera, cuanto más modular y reutilizable sea el software, sin duda será más verificable. Pero al igual que antes, no todo software modular y reutilizable es verificable: para que esto ocurra, es necesario tener bien claro qué tipo de resultados esperamos de cada módulo y la posibilidad de contrastarlos con datos conocidos.

Finalmente, un software de calidad tiene que ser *documentado*. Esta documentación tiene dos aristas distintas y complementarias: por un lado, la destinada al desarrollador (es decir, las personas que posteriormente van a modificar y extender el programa), y por el otro, la que está orientada al usuario (quien no tiene por qué conocer detalles del funcionamiento interno del programa, sino saber fácil y rápidamente cómo utilizarlo). Documentar adecuadamente un software no es una tarea sencilla y requiere de mucha práctica, pero es fundamental para que el código sea accesible al resto de las personas. La documentación adecuada es una de las etapas esenciales que se deben tener en cuenta tendientes a lograr una interfaz amigable.

Estas características del software refieren a un objetivo más grande: que su desarrollo sea *colaborativo*. Esta tendencia requiere pensar el desarrollo, prácticamente desde el comienzo, en la clave de calidad que mencionamos. La idea no es nueva, y es de hecho como se desarrolla en la actualidad gran parte de los programas que usamos todos los días: es el movimiento de *software libre*.

Para poder desarrollar un software de calidad, es necesario conocer y utilizar las herramientas adecuadas, pero es igualmente necesario poder *abstraerse* de ellas. El universo de la programación es muy dinámico y estas herramientas cambian muy rápidamente. La única forma de mantenerse actualizado es comprendiendo cuál es el rol de cada herramienta dentro de la filosofía de trabajo y poder suplirla o identificar herramientas superadoras. Su uso forma parte de un conjunto de habilidades que llamamos *soft skills* o *habilidades blandas* en el ámbito de la programación: no son fundamentales y no cambian directamente la funcionalidad del software. De hecho, en la

actualidad, gran parte de los programas científicos son realizados por desarrolladores que no siguen estos preceptos, y no por eso diríamos que no sirven. No obstante, sí complican su inserción a un universo de desarrollo colaborativo que permitiría extender su funcionalidad y su uso a una comunidad más amplia. Justamente, que no sean fundamentales en lo inmediato es lo que hace que muchos desarrolladores elijan no usarlas. Sin embargo, su potencial se desarrolla completamente cuando dejamos de verlas como herramientas separadas y las entendemos como parte de un engranaje más grande que lleva a desarrollar software colaborativo y de calidad. A lo largo de este libro describiremos estas herramientas, haciendo énfasis constantemente en cómo forman parte de un ecosistema para el desarrollo de software de calidad.

El libro se estructura de la siguiente manera: veremos algunas herramientas fundamentales en los primeros seis capítulos. Entre otras, estas son: Python como lenguaje de *scripting* para desarrollar un entorno amigable para los usuarios; Git como herramienta de control de versiones; Doxygen y Sphinx como herramientas de documentación; gdb, valgrind y gprof para el *debugging* y el *profiling* de software, y MPI y OpenMP como herramientas de paralelización. En el último capítulo describiremos ciertos ejemplos de la vida real en los que se utilizan estas herramientas: tanto ejemplos de personas trabajando en estos ámbitos y el uso de las herramientas, como el desarrollo paso por paso de un código que cumpla las características fundamentales del software colaborativo.

Capítulo 1

Lenguajes interpretados y sus aplicaciones científicas

Cecilia Jarne

Introducción

Los puntos clave que se deben considerar para el desarrollo de software científico tienen que ver con varias cuestiones. Por un lado, con la elección del (o los) lenguaje(s) de programación con los que trabajar. Por otro lado, con las herramientas, procesos de organización, desarrollo y mantenimiento de software.

Algunos factores que son fundamentales y a veces no son explícitamente diagramados. Estos tienen que ver con el conjunto de estrategias y buenas prácticas en programación. También con la consideración sobre si el desarrollo de software será colaborativo y sobre la posibilidad de favorecer el uso de software reutilizable.

Antes de describir los tipos de lenguaje de programación, recordemos primero cómo definirlo. Un *lenguaje de programación* es un lenguaje con reglas gramaticales bien definidas que le proporciona a una persona la capacidad de escribir (o programar) una serie de instrucciones en forma de algoritmos con el fin de controlar el comportamiento de un sistema informático. De esta manera, se pueden obtener distintas clases de datos o ejecutar determinadas tareas. Llamaremos *código* al programa escrito en un cierto lenguaje.

Podemos hacer la siguiente distinción que resulta de utilidad para comprender cómo programar con los distintos lenguajes. De un lado tenemos los lenguajes de programación compilados, y del otro los interpretados.

Compiladores e intérpretes son programas que convierten el código a lenguaje de máquina. Lenguaje de máquina son las instrucciones que entiende el procesador en código binario.

El *lenguaje compilado* requiere un paso adicional antes de ser ejecutado, la compilación, que convierte el código a lenguaje de máquina. Algunos

ejemplos son C, C++, Fortran, Java, Go y Rust y muchos otros.

En el caso del *lenguaje interpretado* o *script*, este es convertido a lenguaje de máquina a medida que es ejecutado. En este caso nuestros ejemplos son Python, R, Ruby y JavaScript y también otros lenguajes.

Este capítulo se centrará en los lenguajes interpretados y sus aplicaciones científicas, pero primero brevemente consideraremos algunas reflexiones que deberíamos hacer antes de empezar a trabajar en el código de un cierto proyecto.

Sobre los procesos de organización y desarrollo, las primeras preguntas que deberíamos realizarnos son: ¿para quién estoy programando?; ¿es para mí?; ¿es para un grupo del que formo parte?; ¿es para terceros?; ¿qué tareas se necesita realizar?; ¿cómo las vamos a implementar?

Eso es lo que permite traducir los requerimientos en subtarefas a realizar. Aquí es donde, según la cantidad de individuos que vayan a colaborar, la durabilidad del proyecto y la ponderación que realicemos, puede ser recomendable usar algún método de *project management*. Esto permite entre otras cosas ser estrictos con los tiempos de desarrollo, implementación y optimización.

Cuando se habla de *buenas prácticas*, en general existe un conjunto de sugerencias que surgen de nuestra experiencia en el desarrollo de software para investigación científica, o para *data science*.

La primera práctica la podríamos resumir vulgarmente como: ¡no inventar la pólvora! Aquí nos referimos a que es una buena práctica el uso de las mal llamadas librerías, que es una traducción de la palabra “*library*”, cuya traducción más adecuada sería “bibliotecas”. En informática, una librería es un conjunto de implementaciones funcionales, escritas en un lenguaje de programación, que ofrece una interfaz bien definida para la funcionalidad que se invoca.

Es decir que en la medida de lo posible, cuando se inicia un proyecto conviene realizar una búsqueda sobre cuáles son las librerías científicas más adecuadas existentes que se han desarrollado relacionadas con lo que se quiera implementar. Así como la búsqueda bibliográfica existente es la primera etapa para el desarrollo del conocimiento, y nadie haría una investigación en un tema sin leer lo que está escrito al respecto, con el software debería suceder lo mismo. Es decir, que es conveniente invertir

tiempo en ver si ya existe una manera de implementar la tarea que deseamos realizar y observar ejemplos antes de decidir la mejor estrategia.

La segunda práctica tiene que ver con comunicarse fluidamente con quien pide el software o desarrolla junto con nosotros si el desarrollo fuera en equipo, y con nosotros mismos en el futuro desarrollo. Es decir, documentar adecuadamente las etapas del proceso.

La tercera tiene que ver con documentar el proceso en general utilizando las herramientas existentes. Los tipos de documentación a los que nos referimos son:

- Información para los desarrolladores que quieren agregar o modificar el código.
- Documentación API (e.g. via doxygen).
- Comentarios en el código que explican las elecciones realizadas.
- Información para los usuarios.
- Manuales de referencia.
- Tutoriales, How To?

En particular, es necesario comenzar a escribir la documentación desde el comienzo del proyecto.

Existen muchas maneras de programar, y podemos destacar que la tendencia actual se centra en:

- Escribir software más modular y reusable.
- Escribir *frameworks* y librerías.
- Software modificable.
- Que permita que sus componentes puedan ser combinadas sin tener que recompilar en la medida de lo posible.
- Combinar código script y compilado.
- Intentar que las componentes puedan ser (re)testeadas y (re)validadas.

¿Qué hace distinto al software científico?

En principio, a veces los requerimientos no están del todo definidos para el software científico. Cuando se modela un problema, cualesquiera sean las limitaciones en el cálculo de punto flotante pueden perjudicarnos si no se las considera. Por otro lado, algunas aplicaciones pueden ser usadas solo una vez

o para algún análisis puntual. Y por otro lado, sucede que los científicos a veces no sabemos programar del modo más eficiente. De esta manera, empíricamente pasa a veces (muchas) que las implementaciones son hechas por gente inexperta, tales como estudiantes de posgrado.

Sobre la elección del lenguaje

Ahora discutamos brevemente la motivación para usar lenguaje de script. El primer factor para considerarlo es la portabilidad. Al no existir la necesidad de recompilar para distintas arquitecturas, se puede usar en distintas computadoras con diferentes sistemas operativos, basta que se tenga el intérprete necesario. Por otro lado, existe la motivación de tener disponibilidad de librerías en la propia plataforma. También es útil la posibilidad que brinda de adaptar múltiples extensiones de archivos. Los lenguajes de script tienen gran facilidad para el posproceso de datos.

En este punto surge la pregunta: ¿por qué usar lenguajes compilados? Los programas compilados tienden a ser más rápidos que los traducidos en tiempo de ejecución, debido a la sobrecarga del proceso de traducción. Los lenguajes de programación de bajo nivel son típicamente compilados. Para los lenguajes de bajo nivel hay más correspondencias uno a uno entre el código programado y las operaciones de hardware realizadas por el código máquina. De este modo, es más fácil controlar la CPU y el uso de memoria.

Así, podemos decir que el equilibrio está en saber combinar el lenguaje de script, el lenguaje compilado, el uso de librerías, la modularidad y favorecer el software reutilizable.

Al decir reutilizable, nos referimos a qué tareas distintas pueden requerir enfoques de cálculo o proceso análogos, o bien a que algunas tareas difieren solo en el subset de datos. Los datos pueden colocarse en archivos estructurados soportados por herramientas de análisis y visualización comunes.

En definitiva, la idea es adoptar herramientas tecnológicas útiles que pueden hacer más exitosa la tarea del desarrollo colaborativo, o que permitan hacer el software más abierto y objeto de evaluación tal como lo es el resto de la investigación científica que se plasma en los trabajos publicados o *papers*.

A partir de lo que hemos discutido como buenas prácticas, y los argumentos que hemos presentado, comenzaremos este capítulo haciendo una introducción sobre un lenguaje interpretado en particular: Python. Elegimos

este lenguaje por varios motivos. Entre ellos por su facilidad de uso y de aprendizaje. Resulta muy útil para introducirnos en la programación y se emplea en ámbitos como *data science* y *machine learning*.

Introducción a Python

Python es un lenguaje de programación interpretado, que permite tipado dinámico y es multiplataforma. Habilita además usar distintos paradigmas de programación: soporta orientación a objetos, programación imperativa y funcional. Tiene una sintaxis clara. Se puede extender su funcionalidad a través de distintas librerías. Este lenguaje de programación fue desarrollado desde 1989 por Guido van Rossum.

Existen varios motivos para estudiar Python. Entre ellos se destaca que es fácil de aprender, que posee un conjunto enorme de librerías científicas de gran calidad y que permite desarrollar software bastante rápido.

Algo importante es que posee una licencia de código abierto y también una comunidad enorme desarrollando con la cual realmente se puede contar.

Sobre las versiones de Python, existen dos grupos de versiones de Python que son la 2.x y la 3.x. Para aprender a programar con este lenguaje, probablemente es conveniente comenzar con la versión 3. Por un lado, la versión 2.7 llegó a su fin de ciclo en el año 2020, y no habrá una versión 2.8. Esto no significa que no se pueda utilizar más, sino que ya no va a tener más actualizaciones. Las librerías más populares disponibles en Python ya están adaptadas a la versión 3.0 del lenguaje. Por otro lado, hay que tener en cuenta que quizás en algún punto alguien podría tener que trabajar con un código desarrollado en Python 2 que se tenga que mantener; por este motivo, conocer algunos detalles de la versión 2.x también puede ser útil. En cualquier caso, pasar de una versión a otra no debería suponer mayor dificultad; existen varios sitios que marcan las diferencias entre ambas y que permiten traducir el código de una versión a la otra.

Cómo instalar el software necesario y su uso básico

Para producir software científico *recomendamos el uso de cualquier distribución de Linux* que crean conveniente, a lo largo de los capítulos de este libro nos referiremos a los diversos ejemplos utilizado dicho sistema operativo. Sin embargo, *la instalación de Python puede realizarse en diferentes sistemas operativos como Windows o Mac Os*. En todos los casos

es necesario ir a <https://www.python.org/downloads/> y descargar la versión actual para el sistema operativo que nos interese.

En el caso de Windows, al terminar la descarga la ejecutamos como administrador. Al ejecutar el instalador aparecerá la opción “Install launcher for all users”, la cual seleccionaremos si es que queremos que todos los usuarios puedan usarlo. Luego es necesario marcar la segunda opción, que es agregar la ruta del intérprete a la variable PATH para que podamos ejecutar `<python.exe>` desde cualquier lugar en la línea de comandos (de otra manera, habrá que poner siempre la ruta absoluta). Una vez configurado, hacemos click en “Install Now” para que el proceso comience. Para probar si todo está correctamente instalado presionamos la combinación de teclas Windows + R (ejecutar); en el menú que aparece escribimos “Python” y presionamos “enter” para iniciar el intérprete de comandos.

Para Mac Os, una vez terminada la descarga ejecutamos el archivo .pkg y se iniciará el asistente de instalación. En este caso se abrirá una ventana donde se muestra información relacionada con la versión de Python, del sistema, de la instalación. En la siguiente pantalla, el instalador muestra el contrato de licencia, que es necesario aceptar. El siguiente paso en la instalación es seleccionar el destino en el que se instalará Python. Seleccionamos en la parte derecha el disco en el que queramos instalarlo, y luego se selecciona el tipo de instalación. Antes de comenzar la instalación, se solicitan las credenciales del superusuario, puesto que este necesita privilegios para poder realizar la instalación. Luego, para probar que la instalación fue correcta, abrimos un terminal y llamamos al intérprete de comandos.

En el caso de Linux, generalmente viene con alguna versión de Python instalada. Chequeamos la versión con el comando:

```
$ python -v
```

o

```
$ python3 -v
```

es decir, para Python 3, la versión recomendada para el presente libro. Para administrar paquetes de software de Python, puede resultar útil instalar pip, una herramienta que instalará y administrará librerías o módulos que se utilizan en los proyectos.

```
sudo apt install -y python3-pip
```

Los paquetes de Python pueden instalarse escribiendo:

```
$ pip3 install <package_name>
```

Donde `package_name` se refiere a cualquier paquete o librería de Python.

Opciones de trabajo

Se puede trabajar directamente con el intérprete de comandos, lo que da la posibilidad de probar porciones de código en el modo interactivo antes de integrarlo como parte de un programa. Por ejemplo:

```
>>> 1 + 1
2
>>> a = range(10)
>>> print(a)
```

También se puede utilizar un editor de texto para escribir en un archivo `.py` que contenga las instrucciones y luego correr dicho programa en la terminal. Por ejemplo ejecutando:

```
python My-programa.py
```

Existen distintas opciones ordenadas para trabajar en los proyectos. Una posibilidad es trabajar localmente utilizando un entorno virtual (Virtual env). El objetivo principal del entorno virtual de Python es crear un entorno aislado para proyectos de Python. Cada proyecto puede tener sus propias dependencias, más allá de las dependencias que tengan los demás proyectos.

Actualmente, existe una alternativa útil que permite probar paquetes y compartir código de una manera práctica y unificada a partir de Google Colab (<https://colab.research.google.com/notebooks/intro.ipynb>). Es interesante que para su uso no requiere configuración. Da acceso gratuito a GPU. Permite compartir contenido fácilmente.

Lenguajes estáticos y dinámicos

En un lenguaje estático, el nombre de la variable está ligado a un tipo y un objeto. En Python puede tomar distintos valores en otro momento, incluso de un tipo diferente al que tenía previamente. Veamos un ejemplo:

```
x = 1  
x = "texto" # tipeo dinamico :)
```

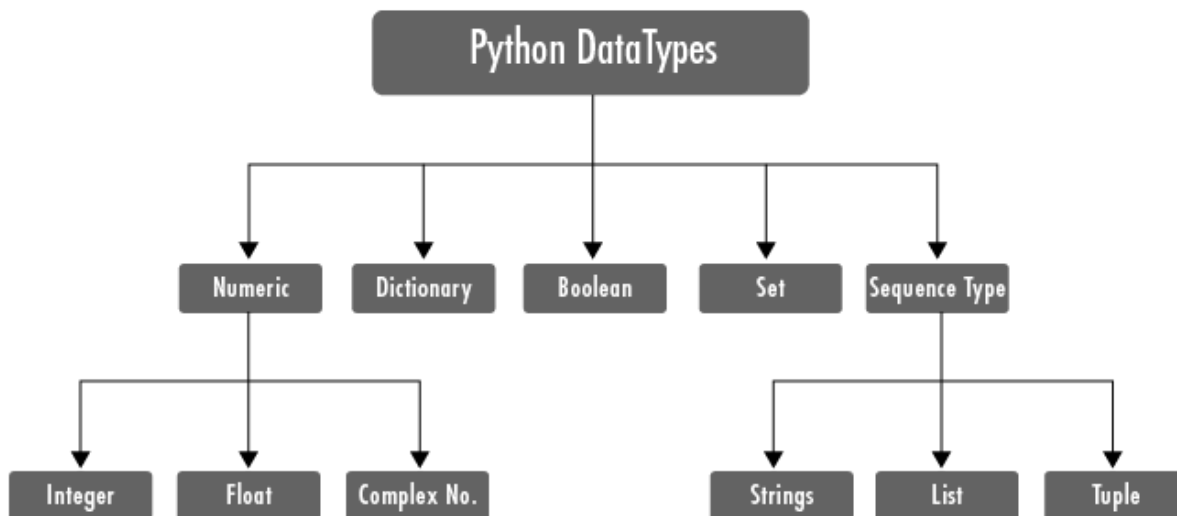
Donde en la segunda línea hemos agregado un comentario luego del símbolo #.

Tipos de datos

Los tipos de datos de las variables se pueden resumir en la figura 1.

La elección de un tipo u otro dependerá de la aplicación. Los detalles de cada tipo se pueden ver en la documentación oficial de Python.

Figura 1. Tipos de datos



Sintaxis de Python

Indentación

Para quienes estén acostumbrados a programar en C o C++, al final de cada línea en Python existe una restricción distinta, cuya implementación tiene que ver con el ordenamiento visual del código: el nivel de indentación es significativo. Se utiliza para delimitar la estructura del programa permitiendo establecer bloques de código. Son frecuentes discusiones entre programadores sobre cómo usar la indentación, si es mejor usar espacios en blanco o tabuladores. En los ejemplos que se presentarán aquí, son utilizados los espacios en blanco.

La indentación puede definirse con caracteres de espacio (se recomienda el empleo de cuatro espacios, lo que es considerado el estilo óptimo de Python) o de tabulación. Es recomendable no mezclar ambos tipos de caracteres en un mismo fragmento de código. Dependiendo de cómo esté configurado el editor de texto, algunos intérpretes pueden devolver error en este caso. La indentación debe ser la misma (igual cantidad de espacios), al menos para las líneas que componen un mismo bloque de código, y la primera sentencia de un archivo de código no debe tener indentación. Veamos un ejemplo:

```
def factorial(x):  
    if x == 0:  
        return 1  
    else:  
        return x * factorial(x - 1)
```

Operadores en Python

Otro elemento importante son los Operadores u Operators. Estos son caracteres empleados para denotar operaciones diversas tales como: aritméticas, lógicas, de asignación, etcétera.

Los operadores actuales del lenguaje son:

+	-	*	**	/	//	%	@
<<	>>	&		^	~		
<	>	<=	>=	==	!=		

Los detalles de cada operador se pueden leer también en la documentación oficial de Python.

Strings

Python permite, por supuesto, manipular cadenas de texto o strings. Dichas cadenas pueden ser expresadas de distintas formas. Pueden estar encerradas en comillas simples ('...') o dobles ("...") con el mismo resultado.

Veamos los siguientes ejemplos:

```
x= "este es un string"
```



```
y='este tambien es un string'
```

En el intérprete interactivo, la salida de cadenas está encerrada en comillas y los caracteres especiales son escritos usando barras invertidas. Aunque esto a veces luce diferente de la entrada (las comillas que encierran pueden cambiar), las dos cadenas son equivalentes. La función `print ()` produce una salida más legible, omitiendo las comillas que la encierran e imprimiendo caracteres especiales.

Veamos algunos ejemplos con la función `print`:

```
print(x)
este es un string
```

Por ejemplo, para agregar formato con números y texto, veamos otro ejemplo y como luce al aplicar la función `print`:

```
>>> print('Yo quiero comer %d %s hoy' %(5, 'chocolates'))
Yo quiero comer 5 chocolates hoy
```

Si no se desea que los caracteres antepuestos por “\” sean interpretados como caracteres especiales, se pueden usar cadenas crudas agregando una “r” antes de la primera comilla.

Estructuras de control iterativas

Las estructuras iterativas (o también llamadas cíclicas, bucles o loops) nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición.

While

Se usa para ejecutar un bloque de declaración siempre que una condición dada sea verdadera. Y cuando la condición es falsa, el control saldrá del ciclo. Por ejemplo:

```
>>> z=0
>>> while (z < 3):
    print(z)
    z = z +1
0
```

```
1  
2
```

For

```
>>> for i in range(1, 5):  
    print(i)  
1  
2  
3  
4
```

Listas

Python tiene varios tipos de datos compuestos, usados para agrupar otros valores como se mostró en la figura 1. El más versátil y una de las novedades de este lenguaje, es la lista. Esta puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes o por comprensión. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo. Las listas también soportan diverso tipo de operaciones muy útiles, tales como concatenación o extend() entre otras.

Veamos algunos ejemplos de posibles listas:

```
>>> a = [1, 'manzana', 1.2]  
>>> print(a)  
[1, 'manzana', 1.2]
```

Las listas se pueden por ejemplo definir también por comprensión, como se muestra en el siguiente ejemplo:

```
>>> b = [x**2 for x in range(1, 11)]  
>>> print(b)  
[1, 4, 9, 16, 25, , 81, 136, 49, 6400]
```

Las listas poseen varios métodos muy útiles que permiten realizar diversas operaciones. Algunos de los métodos más utilizados son los siguientes.

```
list.append(x)  
list.extend(iterable)  
list.insert(i, x)  
list.remove(x)  
list.pop([i])  
list.clear()
```

```
list.index(x[, start[, end]])  
list.count(x)  
list.sort(*, key=None, reverse=False)  
list.reverse()  
list.copy()
```

Donde para ejecutarlas hay que reemplazar list por el nombre de la lista. Veamos como funcionan:

`list.append(x)`

Permite agregar un elemento al final de la lista.

`list.extend(iterable)`

Agrega el contenido una secuencia a la lista.

`list.insert(i, x)`

Inserta un elemento en una posición determinada. El primer argumento es el índice del elemento antes del cual insertar, por lo que `a.insert(0,x)` significa que se inserta al principio de la lista, y `a.insert(len(a),x)` es equivalente a `a.append(0,x)`. Veamos un ejemplo de código.

```
>>> a = [1, 5]  
>>> x = 'pepe'  
>>> a.insert(0, x)  
>>> print(a)  
['pepe', 1, 5]
```

`list.remove(x)`

Elimina el primer elemento de la lista cuyo valor sea igual a x. Genera un código `ValueError` si no existe tal elemento.

`list.pop([i])`

Quita el ítem en la posición dada en la lista. Si no se especifica ningún índice, `list.pop()` elimina y devuelve el último elemento de la lista. Los corchetes alrededor de la “i” del método indican que el parámetro es opcional, no que se debe escribir corchetes en esa posición.

`list.clear()`

Elimina todos los elementos de la lista. Equivalente a `del a[:]`.

`list.index(x[, start[, end]])`

Devuelve el índice del primer elemento de la lista cuyo valor es igual a x. Genera un código `ValueError` si no existe tal elemento. Los argumentos *start* y *end* son opcionales y se utilizan para limitar la búsqueda a una

subsecuencia particular de la lista. El índice devuelto se calcula en relación con el comienzo de la secuencia completa.

`list.count(x)`

Devuelve el número de veces que el elemento `x` aparece en la lista.

`list.sort(*, key=None, reverse=False)`

Ordena los elementos de la lista en el lugar (los argumentos se pueden usar para personalizar la clasificación). Veamos un ejemplo:

```
>>> numeros = [7, 5, 31, 1]
>>> sorted(numeros)
[1, 5, 7, 31]
```

`list.reverse()`

Invierte los elementos de la lista.

`list.copy()`

Devuelve una copia superficial de la lista.

Cómo definir funciones en Python

Una función es un bloque de código que solo se ejecuta cuando se llama. Se le pueden pasar como argumentos datos conocidos como puede ser parámetros u otra función. Una función puede devolver datos como resultado. En Python, una función se define usando la palabra “def”. Se muestra a continuación un ejemplo:

```
def my_function():
    print("Dentro de mi funcion")
```

Para llamar a la función, se usa el nombre de la función seguido de paréntesis:

```
>>> my_function()
Dentro de mi funcion
```

Como se indicó antes, la información se pasa a funciones como argumentos. Los argumentos se especifican después del nombre de la función, entre paréntesis. Se puede agregar tantos argumentos como desee, solo separados con una coma. De forma predeterminada, se debe llamar a una función con el número correcto de argumentos. Lo que significa que si su función espera dos argumentos, debe llamar a la función con dos argumentos, ni más ni menos.

```
def my_function(primer, segundo):  
    print(primer + " y " + segundo)  
  
my_function("Raul", "Mabel")  
  
Raul y Mabel
```

Si no se sabe cuántos argumentos se pasarán a su función, se agrega un * antes del nombre del parámetro en la definición de la función. De esta forma, la función recibirá una tupla de argumentos y podrá acceder a los elementos en secuencia.

```
>>> def my_function(primer, segundo):  
        print(primer + " y " + segundo)  
>>> my_function("Raul", "Mabel")  
Raul y Mabel
```

La salida será:

```
El mas joven es Linus
```

Ya que la función devuelve el último elemento de la lista en este caso.

También es posible usar argumentos de palabras clave. De esta forma no importa el orden de los argumentos. Por ejemplo:

```
def my_function(*kids):  
    print("El mas joven es " + kids[-1])  
my_function("Emilia", "Tobias", "Linus")  
  
>>> def my_function(child3, child2, child1):  
        print("El mas joven es " + child3)  
>>> my_function(child1="Emilia", child2="Tobias",  
child3="Linus")  
El mas joven es Linus
```

Los argumentos de palabras clave a menudo se abrevian como kwargs en la documentación de Python. Del mismo modo que antes, si no sabe cuántos argumentos de palabras clave se pasarán a su función, se agregan dos asteriscos, **, antes del nombre del parámetro en la definición de la función.

De esta forma, la función recibirá un diccionario de argumentos y podrá acceder a los elementos en consecuencia.

Para permitir que una función devuelva un valor, se utiliza *return*:

```
>>> def my_function(x):  
        return 5 * x + 6  
>>> my_function(3)  
21
```

Las definiciones de función no pueden estar vacías, pero si por alguna razón es necesaria una definición de función sin contenido, lo que se hace es colocar la instrucción *pass* para evitar errores.

```
def myfunction():  
    pass
```

Es importante considerar que la medida de complejidad del tiempo brinda un valor aproximado al tiempo que tarda la función en ejecutarse. Esto se denomina tiempo de ejecución de la función. De manera similar, la medida de complejidad espacial brinda un valor aproximado a los requisitos de espacio (memoria) de una función, es decir, para una entrada determinada.

Recursividad en Python

La recursividad es un concepto muy común en matemáticas y en informática. Se llama recursividad cuando una función se llama a sí misma. Una función recursiva debe contener dos propiedades: una relación de recurrencia y una condición de terminación.

Tomemos un ejemplo para ilustrar esta definición en el cuadro siguiente. Aquí se puede ver cómo la función *fib(n)* se llama a sí misma. Este es un ejemplo de recursividad, donde *fib(n)* es la función recursiva. Por ejemplo, podemos ver cómo esta función recursiva permite definir los elementos n de la sucesión de Fibonacci, utilizando la definición matemática. En este caso construimos una función recursiva, y luego armamos una lista por comprensión usando dicha función para obtener los elementos que queremos, por ejemplo los primeros diez elementos de la sucesión:

```
def fib(n):  
    if n == 0:
```

```
    return 0
elif n == 1:
    return 1
else:
    return fib(n-1) + fib(n-2)

fibonacci_list= [fib(x) for x in range(0,10)]
print(fibonacci_list)

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Librerías científicas de python

Como ya se ha comentado, existe una enorme variedad de librerías científicas en Python. Son de una enorme utilidad para realizar distintos cálculos, análisis y procesamiento de datos. En particular comenzaremos describiendo Numpy por ser de gran importancia y además por ser la base para otras librerías de alto nivel. El motivo por el cual Numpy es rápida y eficiente es porque su core está programado en C, y Numpy es una poderosa interfaz que permite llamar rutinas rápidas y optimizadas, que en el fondo están corriendo código compilado.

Numpy

Numpy es un proyecto de código abierto que tiene como objetivo permitir la computación numérica con Python. Fue creado en 2005, basándose en el trabajo inicial de las bibliotecas Numeric y Numarray. Numpy es y será un software 100% de código abierto, de uso gratuito para todos y publicado bajo los términos liberales de la licencia BSD modificada.

Numpy se desarrolla de manera abierta en GitHub, a través del consenso de Numpy y de la comunidad científica Python en general.

Es el paquete fundamental para la computación científica en Python. Es una librería de Python que proporciona un objeto de matriz multidimensional, varios objetos derivados (como matrices y matrices enmascaradas) y una variedad de rutinas para operaciones rápidas en matrices, que incluyen manipulación matemática, lógica, de formas, clasificación, selección, entrada y salida de datos, transformadas discretas de Fourier, álgebra lineal básica, operaciones estadísticas básicas, simulación aleatoria y muchas más.

En el núcleo del paquete Numpy está el objeto `ndarray`. Esto encapsula matrices n-dimensionales de tipos de datos homogéneos (del mismo tipo), con muchas operaciones que se realizan en código compilado para el rendimiento. Hay varias diferencias importantes entre las matrices Numpy y las secuencias estándar de Python.

Las matrices Numpy tienen un tamaño fijo en el momento de la creación, a diferencia de las listas de Python (que pueden crecer dinámicamente). Cambiar el tamaño de un `ndarray` creará una nueva matriz y eliminará la original. Es interesante que a veces se puede combinar el uso de la versatilidad de las listas para la manipulación de datos, y luego dichas listas pueden ser convertidas en arrays para aprovechar toda la capacidad de procesamiento de Numpy.

Todos los elementos de una matriz Numpy deben ser del mismo tipo de datos y, por lo tanto, tendrán el mismo tamaño en la memoria. La excepción: uno puede tener matrices de objetos (Python, incluido Numpy), lo que permite matrices de elementos de diferentes tamaños.

Las matrices Numpy facilitan operaciones matemáticas avanzadas y de otro tipo en grandes cantidades de datos. Por lo general, estas operaciones se ejecutan de manera más eficiente y con menos código de lo que es posible usando las secuencias integradas de Python.

Una gran cantidad de paquetes científicos y matemáticos basados en Python utilizan matrices de Numpy; aunque estos suelen admitir la entrada de secuencia de Python, convierten dicha entrada en matrices Numpy antes del procesamiento y, a menudo, generan matrices Numpy. En otras palabras, para usar de manera eficiente gran parte (quizás incluso la mayoría) del software científico-matemático basado en Python de hoy en día, simplemente saber cómo usar los tipos de secuencia integrados de Python es insuficiente; así como es necesario saber cómo usar las matrices Numpy.

Importar librerías

Para importar las librerías científicas de Python, antes de haber instalado la librería o el paquete deseado es necesario importar el módulo al principio del código. Si es posible, siempre es deseable hacerlo utilizando un alias adecuado. En particular, existen ciertos modos estándar para hacerlo. En el caso de Numpy, la forma estándar de hacerlo es:

```
import numpy as np
```


Arrays de Numpy

Existen varios mecanismos generales para crear arrays, entre los cuales se destacan los siguientes:

- De modo directo como Numpy array (arange, ones, zeros, etc.).
- Conversion desde otras estructuras de Python (lists, tuples) hacia arrays.
- Lectura de los arrays del disco.
- Creación a partir de strings o datos en buffers.
- Usar las librerías especiales (random y otras).

Veamos algunos ejemplos de código referidos a la creación de arrays:

```
>>> x = np.array([2, 3, 1, 0])
>>> print(x)
[2 3 1 0]
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(2, 10, dtype=np.float)
array([2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(2, 3, 0.1)
array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
>>> np.linspace(1., 4., 6)
array([1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```

Para crear un array a partir de una lista:

```
>>> np_array = np.array([1,2,3,4,5,6,7,8,9])
>>> print(np_array)
[1 2 3 4 5 6 7 8 9]
```

Algunos métodos permiten cambiar la forma de los arrays, por ejemplo la función reshape. Veamos un array definido a partir de la función zeros:

```
>>> a = np.zeros((5, 2))
>>> print(a)
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
```

Aplicando el método reshape de la siguiente manera, obtenemos una nueva estructura a partir de la anterior:

```
>>> b = a.reshape((2, 5))
>>> print(b)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Existen otras varias rutinas muy útiles de álgebra lineal, como por ejemplo la transposición; veamos un ejemplo:

```
>>> x = np.array([[1.,2.,3.,4.],[1.,2.,3.,4.]])
>>> print(x)
[[1. 2. 3. 4.]
 [1. 2. 3. 4.]]
>>> b = x.T
>>> print(b)
[[1. 1.]
 [2. 2.]
 [3. 3.]
 [4. 4.]]
```

Otras, como la rutina “fill”, permiten llenar un arreglo numérico previo con un determinado elemento.

```
>>> a = np.array([1, 2])
>>> print(a)
[1 2]
>>> a.fill(0)
>>> print(a)
[0 0]
```

También hay rutinas de operaciones básicas como suma elemento a elemento o resta, que se puede definir para dos arreglos x e y como la suma, por ejemplo:

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([11, -2, 5])
>>> print(x + y)
[12 0 8]
>>> print(np.add(x, y))
[12 0 8]
```

O bien por ejemplo la resta:

```
>>> print(x - y)
[-10 4 -2]
>>> print(np.subtract(x, y))
[-10 4 -2]
```

También existen operaciones como el producto interno, y otras operaciones vectoriales o matriciales.

Por ejemplo, veamos el producto escalar o producto interno, también llamado producto punto, operación que sabemos da como resultado un escalar.

```
>>> v = np.array([-1, 2, 5])
>>> w = np.array([1, 2, 3])
>>> print(v.dot(w))
18
>>> print(np.dot(v, w))
18
```

Tengamos en cuenta que el resultado dependerá de la definición de los arrays a qué tipo de operación hará referencia el método dot. Por ejemplo, si v_1 es una matriz de dos filas, y w_1 es una matriz de una columna (o vector columna), ahora podríamos definir el producto de matrices de 2×3 columnas y de una 3×1 . Así nos da como resultado el vector de 2×1 .

```
>>> v1 = np.array([-1, 2, 5],[1, 3, 5])
>>> w1 = np.array([1, 10, 3])
>>> print(v1.dot(w1))
[34 46]
```

Existen otros métodos de manipulación de vectores, por ejemplo, el producto vectorial también. Podemos definir un ejemplo utilizando los vectores v y w que definimos antes.

```
>>> print(np.cross(v, w))
[-4 8 -4]
```

SciPy

SciPy se refiere a varias entidades relacionadas distintas. Por un lado, con el ecosistema SciPy, que es una colección de software de código abierto para computación científica en Python.

Por otro lado, SciPy hace referencia a la comunidad de personas que usan y desarrollan esta pila y las variadas conferencias dedicadas a la informática

científica en Python, tales como SciPy, EuroSciPy y SciPy.in. La librería SciPy proporciona muchas herramientas eficientes y fáciles de usar, como rutinas para integración numérica, interpolación, optimización, álgebra lineal y estadística.

Para importar funciones de SciPy

Los *namespaces* de SciPy en sí solo contienen funciones importadas de Numpy. Estas funciones aún existen por compatibilidad con versiones anteriores, pero deben importarse directamente desde Numpy.

Todo en los espacios de nombres de los submódulos SciPy es público. En general, se recomienda importar funciones desde espacios de nombres de submódulos. Por ejemplo, la función para realizar ajustes, `curve_fit` (definida en `scipy / optimizar / minpack.py`), debe importarse así:

```
from scipy import optimize
result = optimize.curve_fit(...)
```

Esta forma de importar submódulos es preferible para todos los submódulos excepto para `scipy.io`, porque “io” también es el nombre de un módulo en la biblioteca estándar de Python. De este modo:

```
from scipy import interpolate
from scipy import integrate
import scipy.io as spio
```

Sobre la base del ecosistema SciPy se han creado herramientas generales y especializadas para la gestión y cómputo de datos, para el análisis experimental y la computación de alto rendimiento. A continuación, mencionamos brevemente algunos paquetes clave, aunque también existen otros paquetes también relevantes.

Datos y cálculo

Pandas: proporciona estructuras de datos de alto rendimiento y fáciles de usar.

SymPy: para matemáticas simbólicas y álgebra informática.

NetworkX: es una colección de herramientas para analizar redes complejas.

Scikit-image: es una colección de algoritmos para el procesamiento de imágenes.

Scikit-learn: es una colección de algoritmos y herramientas para el aprendizaje automático.

Por otro lado, para datos estructurados, tanto los paquetes *h5py* como *PyTables* pueden acceder a los datos almacenados en formato HDF5.

Productividad e informática de alto rendimiento

IPython: es una interfaz interactiva que permite procesar rápidamente datos y probar ideas. El *cuaderno Jupyter* proporciona funcionalidad IPython y más en su navegador web, lo que le permite documentar su cálculo en una forma fácilmente reproducible.

Cython amplía la sintaxis de Python para que pueda crear cómodamente extensiones C, ya sea para acelerar el código crítico o para integrarlo con bibliotecas C / C++.

Dask, *Joblib* o *IpyParallel*: se usan para procesamiento distribuido con un enfoque en datos numéricos.

Matplotlib

Matplotlib es una librería para crear visualizaciones estáticas, animadas e interactivas en Python. Esta librería es interesante para aprender ya que permite crear gráficos de calidad para publicación en solo unas pocas líneas de código. También permite crear figuras interactivas donde se puede hacer zoom, desplazarse, actualizar.

Los objetos top-level de Matplotlib que gestionan todos los elementos de un gráfico determinado se denominan Figuras. Una de las tareas arquitectónicas centrales que Matplotlib debe resolver es implementar un framework para representar y manipular la Figura que está separado de la acción de representar la propia Figura en una ventana de interfaz de usuario o en una copia. Esto permite incorporar características y lógica cada vez más sofisticadas sobre las Figuras. Mientras que los “backends” o dispositivos de salida se mantienen relativamente simples, Matplotlib encapsula no solo las interfaces de dibujo para permitir la representación en múltiples dispositivos, sino también el manejo básico de eventos y la creación de ventanas de los toolkits de interfaz de usuario más populares.

Ejemplos

D) Cómo graficar un conjunto de datos.

Veamos a continuación un ejemplo ilustrativo de cómo construir una figura. En nuestro ejemplo hacemos distintos pasos: desde crear un conjunto de datos, darle formato, guardarlo en un archivo txt hasta graficarlos de manera muy sencilla. Para hacer este ejemplo, vamos a usar la librería estándar Math, Numpy y Matplotlib.

```
from math import pi
import numpy as np
import matplotlib.pyplot as pp

#Creamos el archivo de salida

f_out_max = open('tabla.txt', 'w')

#generamos un un conjunto de pseudo-datos
x = np.arange(441) #elementos en x

Sin1 = 1*np.sin(2*pi*(25/441.0)*x)
Sin2 = 0.25*np.sin(2*pi*((25./2)/441.0)*x)
Sin = Sin1+Sin2 #elementos de y
Vec = np.c_[x, Sin] #vector x-y

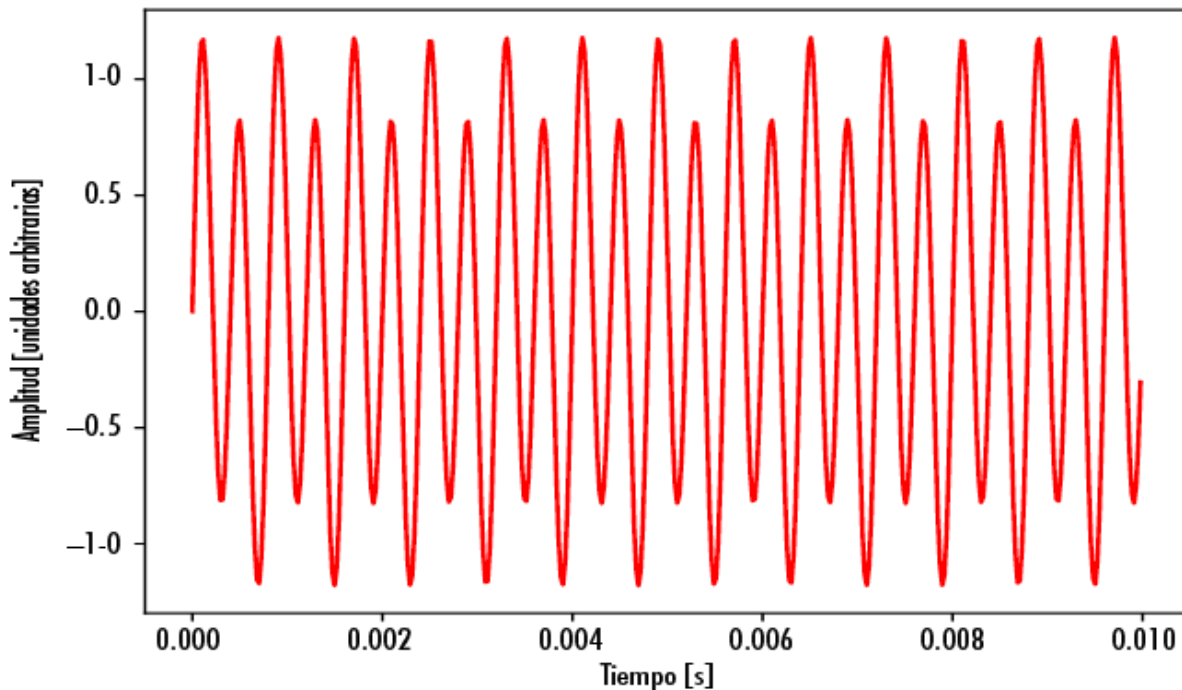
#Hacemos un print de la forma de nuestros datos
print ('Vec: ', Vec.shape)
#Salvamos en el file de salida definiendo el formato (.txt)
np.savetxt(f_out_max, Vec, fmt=' %f', delimiter='\t',
header="x #f(x)")
f_out_max.close()

#plot figure
pp.figure()
pp.plot(x*1./44100., Sin, color='r', label='Funcion vs. x')
#escaleo el vector x (tiempos)
pp.xlabel('tiempo [s]')
pp.ylabel('Amplitud [unidades arbitrarias]')
pp.savefig("fig_01.jpg",dpi=200)
```

Las primeras tres líneas permiten importar los paquetes, funciones y definiciones que usaremos luego, como lo indica el comentario, creamos el archivo de salida. A continuación generamos un conjunto de datos como par

ordenado (x, sin) y los ordenamos en un vector que llamamos vec. Lo imprimimos en pantalla y revisamos su formato. Guardamos la información en una tabla en el archivo anteriormente creado y finalmente lo graficamos para producir la figura 2.

Figura 2. Figura de salida que produce el ejemplo anterior



Veamos otros ejemplos básicos para aprender algunas de las funciones de Python.

II) Como abrir .txt o .csv

En el siguiente ejemplo, se muestra cómo abrir un archivo utilizando Numpy. En este caso, el nombre del archivo y su ubicación están dados por fname. A lo largo del código de ejemplo hay una serie de prints que van a permitir ver la orientación de las filas y columnas que tenga el archivo en cuestión que se quieran cargar, de manera de asegurarse de tener el archivo en el formato adecuado.

```
import numpy as np

file_name_you_want = np.loadtxt(fname,delimiter=" ")
```

```
print("Primer elemento de la columna 1: ",file_name_you_want[0])
#to get the full column:
Transpose_your_file = file_name_you_want.T
print("Primer columna: ", Transpose_your_file[0])
```

III) Cómo hacer un ajuste y graficarlo

Para hacer este ejemplo, vamos a utilizar las librerías Numpy, SciPy y Matplotlib. La idea es generar un pseudo-set de datos; luego definiremos una función de ajuste y utilizaremos la función `curve_fit` de SciPy para obtener el mejor ajuste posible y los parámetros de salida.

Finalmente, graficaremos los datos, la función de ajuste y la función \pm un sigma.

```
import numpy as np
import matplotlib.pyplot as pp
from scipy.optimize import curve_fit

def fitFunc(t, a, b, c):
    return a*np.exp(-b*t) + c

t = np.linspace(0,4,50)
temp = fitFunc(t, 2.5, 1.3, 0.5)
noisy = temp + 0.25*np.random.normal(size=len(temp))
fitParams, fitCovariances = curve_fit(fitFunc, t, noisy)

pp.figure(figsize=(12, 6))
pp.ylabel('Temperature (C)', fontsize = 16)
pp.xlabel('time (s)', fontsize = 16)
pp.xlim(0,4.1)
pp.errorbar(t, noisy, fmt = 'ro', yerr = 0.2)
sigma = [fitCovariances[0,0], fitCovariances[1,1],
fitCovariances[2,2] ]
pp.plot(t, fitFunc(t, fitParams[0], fitParams[1],
fitParams[2])) pp.plot(t, fitFunc(t, fitParams[0] +
sigma[0], fitParams[1] - sigma[1], fitParams[2] + sigma[2]))
pp.plot(t, fitFunc(t, fitParams[0] - sigma[0], fitParams[1]
+ sigma[1], fitParams[2] - sigma[2]))
pp.savefig('dataFitted.pdf', bbox_inches=0, dpi=600)
pp.show()
```


Nuevamente las primeras tres líneas permiten llevar los módulos y las funciones. Luego definimos las funciones de ajuste. A continuación generamos un pseudoset de datos que luego ajustaremos y graficaremos. El gráfico se muestra en la figura 3. En el gráfico se observan los datos, el ajuste y el ajuste con una desviación estandar en los parámetros.

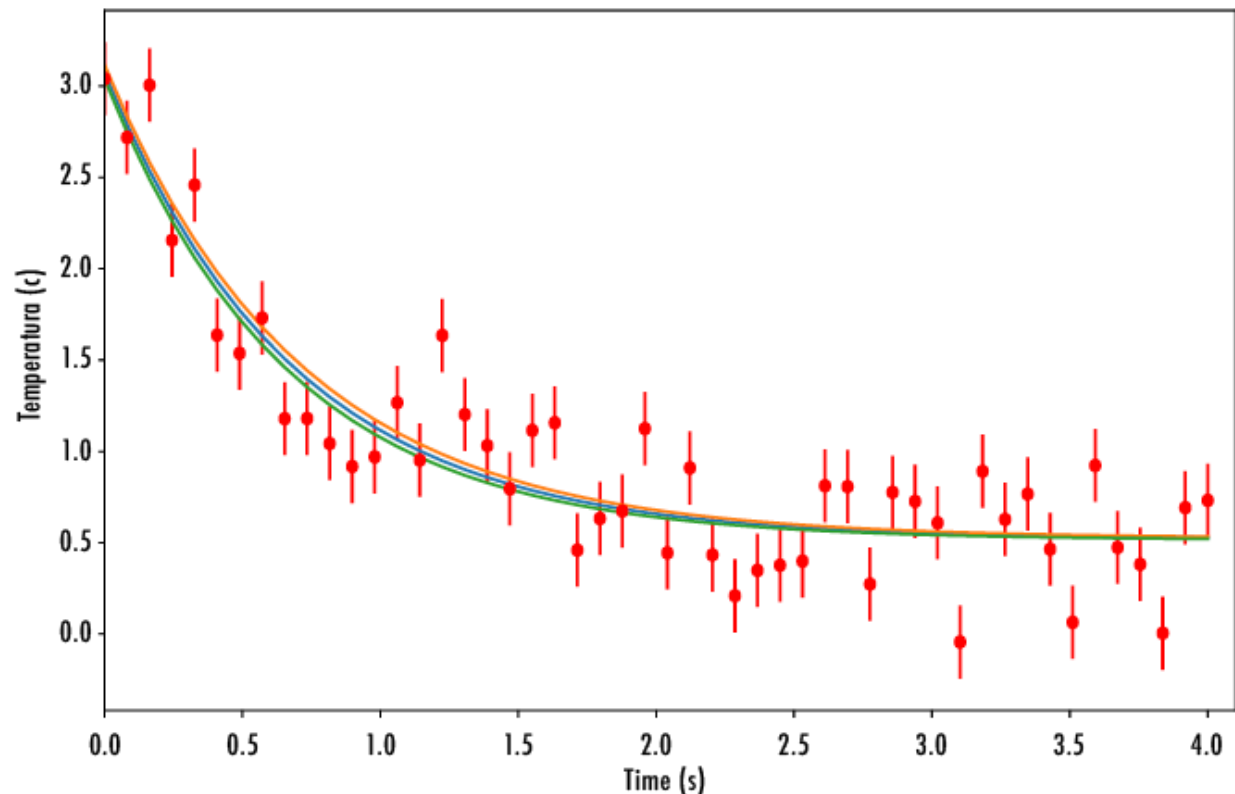
Al ejecutar el código, obtenemos los parámetros de nuestro ajuste:

```
[ 2.595658 1.74438726 0.69809511]
```

Así como la matriz de covarianza:

```
[[ 0.02506636 0.01490486 -0.00068609]  
[ 0.01490486 0.04178044 0.00641246]  
[-0.00068609 0.00641246 0.00257799]]
```

Figura 3. Archivo de salida generado por el código del ejemplo anterior



¿Qué más podemos hacer con los datos?

Antes de finalizar el capítulo, discutamos un aspecto más respecto a los datos. En general podemos estar interesados en diversos aspectos relacionados con esto. Nos puede importar la visualización, el análisis (identificar distintas características), la clasificación y muchos otros aspectos. Para todo esto a veces es útil la implementación de algoritmos inteligentes. Si bien existen varias librerías que permiten hacer esto, actualmente las más populares son Pytorch y TensorFlow. A continuación, describiremos brevemente qué es TensorFlow y cómo se puede instalar para implementar distintos algoritmos.

TensorFlow

Es una plataforma de código abierto para el aprendizaje automático. Tiene un ecosistema integral y flexible de herramientas, bibliotecas y recursos que permite a los investigadores impulsar el estado del arte en Machine learning (ML) y a los desarrolladores crear e implementar fácilmente aplicaciones impulsadas por ML.

Para instalar TensorFlow se puede actuar por un lado con el administrador de paquetes *pip* de Python:

```
# Requiere instalar la ultima version de pip
pip install --upgrade pip

# Es compatible con CPU and GPU
pip install tensorflow
```

Los paquetes oficiales están disponibles para Ubuntu y también para Windows, macOS y Raspberry Pi.

Por otro lado, se puede ejecutar también usando un contenedor de Docker en un entorno virtual y es la forma más fácil de configurar la compatibilidad con GPU. Finalmente, es posible ejecutar los [instructivos de TensorFlow](#) directamente en el navegador con Google [Collaboratory](#).

Resumen

En este capítulo recorreremos los fundamentos sobre los lenguajes interpretados. En particular, aprendemos los aspectos básicos de Python y sus librerías científicas que nos permitirán rápidamente comenzar a utilizarlo para realizar análisis de datos simples y pequeños modelos. En el camino

hemos recorrido algunos ejemplos de código para implementar. A continuación se dejan ejercicios simples propuestos enfocados para un curso introductorio.

Hands-on: Python como lenguaje de *scripting*

En esta última sección proponemos la realización de algunos ejercicios que permitirán evaluar y poner a prueba los conocimientos adquiridos en el capítulo.

1. Utilizando solo Python, proponemos elegir alguno de los siguientes problemas y solucionarlo (sin usar Numpy). Los problemas fueron extraídos de <<https://projecteuler.net/archives>>.

- a. Si hacemos una lista de todos los números naturales debajo de 10 que sean múltiplos de 3 o 5, obtendremos 3, 5, 6 y 9. La suma de los múltiplos es 23. Proponemos encontrar la suma de todos los múltiplos de 3 y 5 debajo de 1000.
- b. Cada término en la serie de Fibonacci es generado a partir de la suma de los dos términos previos; empezando por 1 y 2, los diez primeros términos serán: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89... Considerando los términos de la serie de Fibonacci que son impares y por debajo de un millón, proponemos encontrar la suma de dichos términos.
- c. Los factores primos en 13195 son 5, 7, 13 y 29. ¿Cuál es el factor primo más grande en el número 600851475143?

A partir de aquí, tengamos en cuenta que se puede utilizar Python con todas las librerías disponibles de SciPy, Numpy, Matplotlib. Proponemos visitar las páginas de referencia para intentar resolver los problemas tal como se armaría un script para el trabajo diario.

2. Dado el siguiente set de datos, proponemos obtener un gráfico tipo scatter plot para X en función de Y.

X	Y
7,5	28,66

4,48	20,37
8,60	22,33
7,73	26,35
5,28	22,29
4,25	21,74
6,99	23,11
6,31	23,13
9,15	24,68
5,06	21,89

Se trata aquí de:

- Intentar realizar un ajuste lineal o de algún polinomio utilizando este set de datos.
- Intentar colocar nombre o label para los datos.
- Graficar el siguiente polinomio, su derivada y puntos extremos:
 $f(x)=x^3+x^2-4x+4$.
- Colocar título a los ejes y agregar una grilla en ambos. Definir el rango de la función en x entre 0 y 10.
- Colocar título y colores distintos para la función y la derivada.
- Guardar los resultados de evaluar la función en el rango del punto a cada 0.1 unidades en un archivo de texto.

Bibliografía

Bhatotia, Pramod, Apuntes del curso Big Data Systems, Escuela de Ciencias Informáticas, 2017. Disponible en:

<<https://hplgit.github.io/primer.html/doc/pub/half/book.pdf>>.

Capítulo sobre Matplotlib: <<http://www.aosabook.org/en/matplotlib.html>>.

Curso básico sobre Python: <<https://www.learnpython.org/es/>>.

Ejemplos de Matplotlib: <<http://matplotlib.org/gallery.html>>.

Guía de estilo de Python: <<https://google.github.io/styleguide/pyguide.html>>.

Harris, C. R., K. J. Millman, S. J. Van der Walt *et al.*, “Array programming with Numpy”, *Nature*, N° 585, 2020, pp. 357-362. Disponible en: <<https://doi.org/10.1038/s41586-020-2649-2>>.

Haslwanter, T. A., *Introduction to Statistics with Python. With Applications in the Life Sciences*, Springer, 2016.

Langtangen, H. P., *A Primer on Scientific Programming with Python*, Springer, 2016. Disponible en: <<https://developers.google.com/machine-learning/crash-course/ml-intro>>.

Lott, S. F., *Mastering. Object-oriented Python*, Packt Publishing, 2014.

Matplotlib: <<https://matplotlib.org/>>.

Numpy: <<https://numpy.org/>>.

Pandas: <<http://pandas.pydata.org/>>.

Scikit learn: <<http://scikit-learn.org>>.

SciPy: <<https://www.scipy.org/>>.

Sobre visualización:

<<https://datavizproject.com/>>.

<<https://datavizcatalogue.com/search.html>>.

<<https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html>>.

Capítulo 2

Herramientas para el desarrollo de software propio y colaborativo

Rodrigo Lugones

Sistemas de control de versiones

Introducción. ¿Qué son?

En primer lugar, para comenzar, ¿qué es un “control de versiones”? Se llama control de versiones a la gestión de los cambios que se realizan sobre documentos, programas o cualquier colección de información. Una “versión” es el estado en el que se encuentran estos en un momento dado de su desarrollo. Ejemplos en la vida real podrían ser un cuaderno de laboratorio, o las ediciones y revisiones de un libro.

El control de versiones le da trazabilidad a nuestro trabajo y nos permite no solo ver cómo ha ido modificándose, sino también eventualmente volver a una versión anterior. Cuando se trabaja en un proyecto pequeño, saber cuál fue la historia de nuestro software es conveniente. Pero si el proyecto es grande, utilizar alguna forma de control de versiones es indispensable. Para tener cierta dimensión de lo que puede conllevar un software grande, en la tabla 1 se puede ver la cantidad de líneas de código para algunos proyectos (lista probablemente ya desactualizada al momento de la publicación del libro).

Tabla 1. Cantidad de líneas de código de algunos proyectos

Cantidad de líneas de código (aproximadas)	
Script básico propio	~ 100
Código grande propio	~ 1 mil
Aplicación de celular	~ 30 mil

Transbordador espacial	~ 400 mil
Linux Kernel 2.2.0 (1999)	~ 1,5 millones
Telescopio Hubble	~ 2 millones
Google Chrome	~ 6,5 millones
Mozilla Firefox	~ 9,5 millones
Android	~ 12 millones
Linux Kernel 3.1 (2011)	~ 15 millones
Windows 7	~ 40 millones
Large Hadron Collider	~ 50 millones
Facebook (2015)	~ 60 millones
Software de un auto (2015)	~ 100 millones
Servicios de internet de Google (2015)	~ 2.000 millones

Fuente: <<https://informationisbeautiful.net/visualizations/million-lines-of-code/>>.

Resulta bastante claro que, cuando se trabaja en un proyecto grande, no es negociable tener cierta trazabilidad de lo que se hace. Sin embargo, podría no quedar tan claro la importancia que tiene controlar versiones en proyectos pequeños. Para poner un poco de luz en el asunto, estudiemos un ejemplo concreto, que nos acompañará a lo largo del capítulo.

Supongamos que queremos simular un canal de agua con una columna en el centro. Para resolver esto en física, conocemos la ecuación de Navier-Stokes que nos dice cómo va a evolucionar un fluido a lo largo del tiempo. El problema es que, salvo en casos muy concretos, es imposible encontrar una solución cerrada (una fórmula que resuelva la ecuación). Lo que podemos hacer es escribir un programa que resuelva numérica y aproximadamente el problema. Entonces, supongamos que conseguimos escribir un programa NS_Solver que resuelve la ecuación de Navier-Stokes con el método numérico de Euler para predecir la evolución en el tiempo. El resultado arrojado por el programa es, por ejemplo, la energía a cada tiempo.

Supongamos que ahora queremos graficar la energía en función del tiempo. ¿Cómo procedemos?

Opción 0). La opción más evidente es continuar trabajando en la carpeta en la que estamos trabajando. Se podría decir, modificando la misma carpeta.

Pero si nos equivocamos y rompemos el código, ¿no querríamos poder volver fácilmente a la versión original?

Opción 1). Quizás la opción más común, con la que más nos sentimos identificados, sea guardar versiones viejas de los distintos archivos (o carpetas). El problema con esta opción queda bastante claro con el chiste [1531 de PhDComics](#): creamos un archivo *final.doc*, lo modificamos a *final_revision1.doc*, lo continuamos modificando hacia *final_revision6_comentarios3.doc*, y terminamos con 15 versiones distintas del mismo archivo. La dificultad queda resumida en la siguiente pregunta: ¿qué contiene cada versión?

Opción 2). Si el problema es no saber qué contiene cada versión, lo podríamos resolver utilizando nombres descriptivos para los archivos. Por ejemplo:

```
$ ls
NS_Solver_Euler
NS_Solver_Euler_y_grafico_E
```

Otra posibilidad similar sería guardar un archivo *versiones.txt* que registre los cambios realizados.

```
$ ls
NS_Solver_1
NS_Solver_2
versiones.txt
$ cat versiones.txt

1: Navier-Stokes con metodo de Euler
2: Navier-Stokes con metodo de Euler y grafico de Energia.
```

Ahora bien, ¿qué pasaría si en la versión 9 del programa encontráramos un bug que existía desde la versión 1? Podríamos pensar soluciones posibles, pero ¿conseguiríamos que dichas soluciones no traigan consigo más problemas?

Todas estas opciones analizadas podrían considerarse controles de versión caseros. Sin embargo, vemos que la forma de tener control sobre el proyecto comienza a ser cada vez más rebuscada, y que los problemas se multiplican rápidamente a medida que el proyecto crece. Aquí es donde entran los Sistemas de Control de Versiones (SCV), herramientas especializadas en dicha tarea.

Existen muchos SVC (Mercurial, Subversion, CVS, ClearCase, por nombrar algunos). En este capítulo nos centraremos en Git, principalmente por la facilidad de su uso y por ser el SCV más extendido.

Git: una filosofía de trabajo

Git (pronunciado “guit”) es un sistema de control de versiones pensado para poder trabajar colaborativamente entre varios programadores, desarrollando código fuente de manera colaborativa durante el desarrollo de software. Está centrado en la velocidad, la integridad de datos, y el soporte para flujos de trabajo distribuidos y no lineales (miles de ramas paralelas que se ejecutan en diferentes sistemas).

Fue creado por Linus Torvalds (entre otras personas) en 2005 para el desarrollo del kernel de Linux, y desde ese año Junio Hamano ha sido el mantenedor principal. Al igual que lo que sucede con la mayoría de los otros sistemas de control de versiones distribuidos, y a diferencia de la mayoría de los sistemas cliente-servidor, cada directorio Git en cada computadora es un repositorio completo con historial completo y capacidades completas de seguimiento de versiones, independientemente del acceso a la red o un servidor central. Además, muy importante, Git es un software gratuito y de código abierto distribuido bajo la licencia GNU General Public License Version 2. Es decir, cualquiera puede ver su código fuente y proponer mejoras.

La ventaja de utilizar las herramientas adecuadas para el objetivo que uno quiere lograr es que, de alguna forma, favorecen (o hasta imponen) una filosofía de trabajo. Con Git vamos a poder:

1. Hacer backup de estados consistentes del proyecto.
2. Documentar cambios.
3. Seguir los bugs a través de la historia del desarrollo.
4. Compartir cambios.
5. Distribuir el desarrollo a muchas personas.

¿Y cómo funciona Git? Una posibilidad es memorizar un par de comandos y utilizarlos para sincronizar todo, y si surge algún error, hacer una copia de todo, borrar el proyecto y descargar una nueva copia (tal como bien nos dice el famoso chiste de Git de [XKCD](#)).

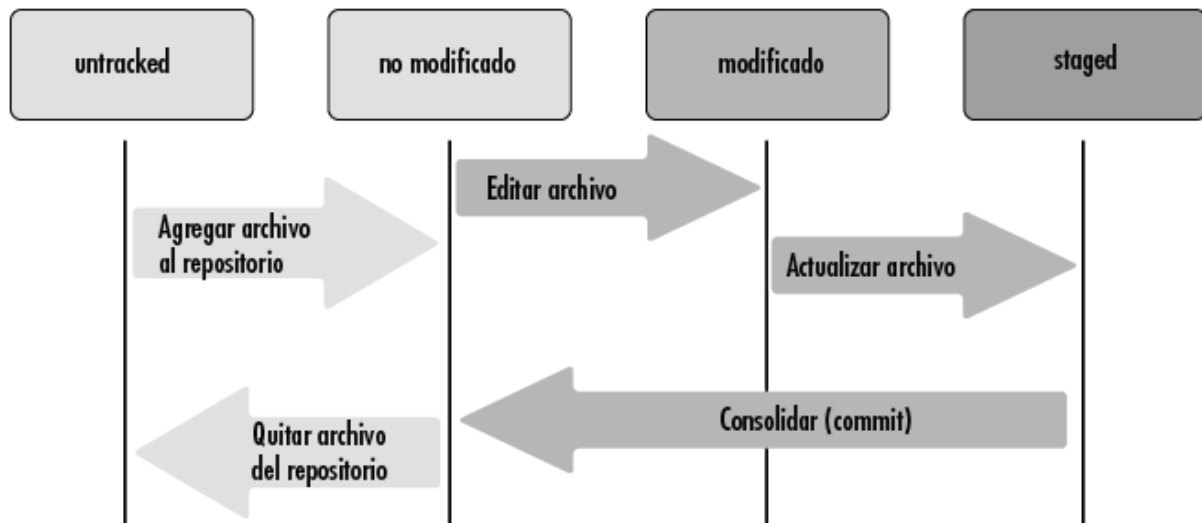
Para que no suceda esto, definamos algunos conceptos clave:

- *Estados de un repositorio*. Son *análogos* a las distintas versiones en el ejemplo del último control de versiones casero. Cuando terminamos de trabajar en un estado y lo *consolidamos* (en nuestra analogía, sería decir que terminamos de desarrollar la funcionalidad que queríamos en la carpeta y, entonces, no modificamos más esa carpeta) lo llamamos *snapshot* (porque es como una foto que le sacamos a la carpeta). El *snapshot* actual, donde estamos situados, recibe el nombre de HEAD (independientemente de que sea el último o no).
- *Ciclo de vida de los archivos*. En un repositorio de Git, cada archivo puede tener diversos estados:
 - Archivo no-modificado (*unmodified*). Un archivo está en estado no-modificado cuando es exactamente igual al archivo que está guardado en el último *snapshot*.
 - Archivo modificado (*modified*). Modificar un archivo (por ejemplo, cambiar el nombre de una variable) lo transforma, evidentemente, en un archivo modificado.
 - Archivo actualizado (*staged*). La diferencia con un archivo modificado es que Git no hace seguimiento a un archivo solo porque cambió modificado. Para que Git se haga cargo del archivo modificado lo tenemos que *actualizar*, poner en escena (o, el término en inglés, *stage*). Git solo utiliza los archivos actualizados para consolidar el cambio y, en consecuencia, tomar un nuevo *snapshot*. Al hacer esto, los archivos que estaban actualizados ahora forman parte del nuevo *snapshot*, que pasa a ser el nuevo HEAD del repositorio. Es decir que consolidar cambios actualiza automáticamente el HEAD del repositorio, y de esta manera los archivos que se encontraban en el estado actualizado pasan al estado no-modificado.
 - Archivos no seguidos (*untracked*). Finalmente, si creamos un archivo nuevo y le queremos hacer seguimiento, tenemos que agregarlo al repositorio. Es decir, Git no agrega automáticamente nada al repositorio. Todo debe ser especificado: desde los archivos modificados (que deben ser *actualizados* para formar parte de la consolidación) hasta los archivos *no seguidos*. De la misma manera, podemos remover

un archivo del repositorio para dejar de seguirlo, sin que ello signifique borrar el archivo.

- Esto se puede ver esquematizado en la figura 1.

Figura 1. Ciclo de vida de los archivos en un repositorio



Comandos básicos de Git

Retomemos nuestro proyecto original, NS_Solver, pero desde el comienzo vamos a construir un repositorio de Git.

En primer lugar, abramos una terminal, donde escribiremos todos los comandos. A lo largo de este capítulo supondremos que estamos trabajando en una computadora con Linux, pero todo lo que vamos a ver también puede hacerse en cualquier otro sistema operativo.

En caso de tener Windows o Mac, los comandos de Git serán los mismos, aunque pueden cambiar los nombres de algunas instrucciones propias del lenguaje de la terminal, o la forma de escribir los *paths* de los archivos y las carpetas.

En Windows, trabajaremos con el *Command Prompt*, que se puede encontrar en Inicio > Windows System, o presionando la tecla Windows + R y ejecutando el comando cmd.

En el caso de Mac, es posible acceder a la terminal entrando en Aplicaciones > Utilidades.

Una vez abierta la terminal, debemos crear la carpeta donde guardaremos nuestro código y posicionarnos dentro de ella: [\[1\]](#)

```
$ mkdir carpeta_proyecto
$ cd carpeta_proyecto
$ pwd
/home/usuario/carpeta_proyecto
```

Una vez hecho esto, daremos el primer paso en la utilización de Git. Para crear el repositorio, ejecutamos la instrucción `git init <path>`. Es decir,

```
$ git init .
Iniciado repositorio Git vacío en /home/usuario/proyecto_nuevo/.git/
```

El `.` significa “acá mismo”. Es decir, habríamos obtenido el mismo resultado ejecutando `git init /home/usuario/proyecto_nuevo`. En esta carpeta estará guardado el repositorio de Git del proyecto. ¿Dónde se guarda la historia del proyecto? [\[2\]](#)

```
$ ls -a
.git
```

En la subcarpeta oculta `.git` se encontrará toda la información correspondiente al repositorio, a la historia del proyecto. Si bien es educativo entrar en la carpeta e investigar qué hay, es *fuertemente recomendable* no editar ningún archivo contenido allí. Todos los *snapshots*, todas las versiones que hemos creado, estarán en dicha carpeta. Si la información de esa carpeta se corrompe (por edición voluntaria o involuntaria de los archivos), se perderá toda la historia del proyecto. En consecuencia, amerita la insistencia: no editar ningún archivo contenido en la carpeta `.git`.

Para saber en qué estado se encuentra un repositorio, utilizamos el comando `git status`.

```
$ git status
En la rama master
No hay commits todavía
No hay nada para confirmar (crea/copia archivos y usa “git add” para hacerles seguimiento)
```

En este caso, tanto el repositorio como el directorio están vacíos (salvo por la carpeta oculta `.git`). Creemos ahora el archivo `navierstokes.py`, que

resuelve el problema con el método de Euler, y veamos cuál es el estado del repositorio.

```
$ ls
navierstokes.py
$ git status
En la rama master
No hay commits todavía
Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que será confirmado)
  navierstokes.py
no hay nada agregado al commit pero hay archivos sin seguimiento presentes (usa
"git add" para hacerles seguimiento)
```

En el directorio aparece ese archivo, pero Git no lo reconoce como parte del repositorio, pues nunca le dijimos que lo *siguiera*. En otras palabras, el archivo `navierstokes.py` se encuentra en el estado *untracked*, no seguido (figura 1). Si lo agregamos al repositorio, mediante el comando `git add`, pasa a estar *actualizado* (directamente a la *stage area*):

```
$ git add navierstokes.py
$ git status
En la rama master
No hay commits todavía
Cambios a ser confirmados:
  (usa "git rm --cached <archivo>..." para sacar del área de stage)
  nuevo archivo: navierstokes.py
```

Ahora *consolidamos* los archivos *actualizados* mediante `git commit`.

```
$ git commit -m "Agregado método de Euler"
[master (root-raíz) 13aa40c] Agregado método de Euler
 1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 navierstokes.py
```

Es posible que cuando se intente hacer el primer *commit* en una computadora, Git devuelva el siguiente mensaje de error, y no haga el *commit* que se pidió:

```
*** Por favor cuéntame quién eres.
Corre
  git config --global user.email "you@example.com"
  git config --global user.name "Tu Nombre"
para configurar la identidad por defecto de tu cuenta.
Omite --global para configurar tu identidad solo en este repositorio.
```

fatal: no es posible auto-detectar la dirección de correo (se obtuvo
'<usuario>@<hostname>.(none)')

Este mensaje aparece porque Git no sabe de quién se trata, por lo que no puede configurar quién hizo los cambios en el repositorio. Git es muy informativo cuando surgen errores. Siempre intenta adivinar qué quiso hacer el usuario (tal como cuando se escribe mal un comando, por ejemplo, **git ad** con una sola “d”). Para resolver el conflicto, se deben seguir las instrucciones y configurar nombre y correo electrónico. Vale aclarar que estos datos no necesitan ser ciertos, pero se verán reflejados al subir el repositorio a la nube (veremos cómo más adelante). Por lo tanto, es conveniente, de manera que si alguien se quiere comunicar por el código, pueda hacerlo. Una vez arreglado este problema, se puede proceder y volver a consolidar los cambios, con **git commit**.

Notemos dos detalles. Al correr el comando de *commit*, utilizamos el *flag* -m, seguido del *mensaje de commit*. Este mensaje es *obligatorio*, y se utiliza para informar qué cambios se realizaron entre el *snapshot* anterior y este que acabamos de consolidar. Si no se utiliza el *flag* -m, es decir si se ejecuta directamente **git commit**, lo que sucede es que se abre automáticamente el editor de texto plano por defecto para escribir el mensaje de *commit*.

El segundo detalle es el *commit hash*, ese número hexadecimal (13aa40c) con el que podemos referirnos a este *snapshot*.

Si ahora miramos el estado del repositorio, vemos que el mensaje impreso en consola cambió.

```
$ git status
En la rama master
nada para hacer commit, el arbol de trabajo esta limpio
```

Con el mensaje “el árbol de trabajo está limpio”, Git nos está diciendo que no hay archivos en estado *modificado*, ya que el archivo navierstokes.py (que no volvimos a tocar) es idéntico al que está guardado en HEAD (que ahora representa el último y único *commit*).

A partir de acá, la tarea para generar un *snapshot* es siempre la misma:

1. Modificamos/creamos uno o varios archivos. Esos archivos dejan de estar *no-trackeados* o *no-modificados* y pasan a estar *modificados*.
2. Agregamos los archivos *modificados* a la *stage area* con **git add**.

3. Los consolidamos en un *snapshot* con git commit.

Por ejemplo, creamos el archivo graficar_E.py para graficar la energía y modificamos algunas líneas de código del archivo navierstokes.py:

```
$ git status
En la rama master
Cambios no rastreados para el commit:
(usa "git add <archivo>..." para actualizar lo que sera confirmado)
(usa "git checkout -- <archivo>..." para descartar los cambios en el directorio de
trabajo)
    modificado: navierstokes.py
Archivos sin seguimiento:
(usa "git add <archivo>..." para incluirlo a lo que se sera confirmado)
    graficar_E.py
sin cambios agregados al commit (usa "git add" y/o "git commit -a")
```

Este mensaje de consola nos dice lo que ya sabemos: el archivo navierstokes.py se encuentra en estado *modificado*, mientras que el archivo graficar_E.py se encuentra en estado *no-tracked*.

Ahora bien, sabemos que navierstokes.py se encuentra modificado, pero no sabemos qué cambios se realizaron. Si quisiéramos ver las diferencias entre la versión modificada de navierstokes.py y la que se consolidó en último *snapshot* (o sea, la versión presente en HEAD), simplemente escribimos:

```
$ git diff navierstokes.py
diff --git a/navierstokes.py b/navierstokes.py
index 219e905..eb3f118 100644
--- a/navierstokes.py
+++ b/navierstokes.py
@@ -1,1 @@
-codigo viejo
+codigo nuevo
```

Y si ahora queremos consolidar los nuevos cambios en un nuevo estado consolidado del repositorio (*snapshot*), utilizamos git add seguido de git commit.

```
$ git add graficar_E.py navierstokes.py
$ git commit -m "Graficar energia"
[master 9b72f8b] Graficar energia
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 graficar_E.py
```

Antes de continuar, el comando `git add` amerita dos comentarios. Primero, tenemos la posibilidad de agregar a la *stage area* todos los archivos que modificamos y todos los archivos no seguidos escribiendo `git add .`, donde el “.” le indica a Git que agregue a la *stage area* todo lo modificado en ese lugar (es decir, “aquí”). El segundo comentario a hacer es que `git add` es muy versátil: es posible agregar a la *stage area* solo parte de las modificaciones realizadas en un archivo. La forma de hacerlo excede los alcances de este libro, pero es importante saber que se puede. Prosiguiendo, si queremos ver todos los *commits* que hemos hecho en el repositorio, usamos `git log`.

```
$ git log
commit 9b72f8bf4e05dd88ab8d365e712a3d6dfe372ca7 (HEAD -> master)
Author: Mariela Carboni <marielacarboni@mail.com>
Date: Sun Feb 25 12:28:43 2021 -0300
    Graficar energia
commit 13aa40cbad74d2ecd8bb58452471cd79afb20e83
Author: Mariela Carboni <marielacarboni@mail.com>
Date: Sun Feb 25 12:26:19 2021 -0300
    Agregado metodo de Euler
```

En este mensaje vemos varias cosas interesantes. En primer lugar, notemos que fácilmente podemos releer los mensajes de commit viejos. Además, podemos saber cuándo se efectuó el *commit* y quién lo hizo (útil para saber a quién culpar cuando las cosas fallan en un trabajo colaborativo). Por último, vemos que el *commit hash* es un número mucho más grande de lo que nos mostraron anteriormente (para el primer *commit*, nos había aparecido el número 13aa40c, que corresponde solo a los primeros siete dígitos). Esto se debe a que suele no ser necesario utilizar muchos dígitos para identificar unívocamente un *snapshot*, y por esa razón cuando hicimos el commit se informaron solo algunos números.

Entonces, hasta ahora hemos aprendido a crear un repositorio (`git init`), agregar archivos y modificaciones (`git add`), consolidar cambios (`git commit`) y revisar la historia (`git log`). El siguiente paso es aprender cómo ir a una versión anterior del código, cómo ir a un *snapshot* viejo. O, lo que es lo mismo, cómo cambiar dónde está el HEAD. Para ello, el comando es `git checkout <commit hash>`, donde el *commit hash* no necesita ser todo el número completo, sino tan solo unos pocos dígitos que permitan identificar unívocamente el *commit*.


```
$ git checkout 13aa
```

Nota: actualizando el árbol de trabajo '13aa'.

Te encuentras en estado 'detached HEAD'. Puedes revisar por aquí, hacer cambios experimentales y confirmarlos, y puedes descartar cualquier commit que hayas hecho en este estado sin impactar a tu rama realizando otro checkout.

Si quieres crear una nueva rama para mantener los commits que has creado, puedes hacerlo (ahora o después) usando `-b` con el comando checkout. Ejemplo:
`git checkout -b <nombre-de-nueva-rama>`

HEAD está ahora en 13aa40c Agregado metodo de Euler

Veamos qué hay en la carpeta...

```
$ ls
```

```
navierstokes.py
```

... y antes de entrar en pánico, recordemos que en este *snapshot*, el archivo `graficar_E.py` no existía. El HEAD dejó de ser `9b72f8b` y pasó a ser `13aa40c`. Si ahora queremos volver al *snapshot* donde está implementado el gráfico de la energía, ejecutamos:

```
$ git checkout 9b72
```

La posición previa de HEAD era 13aa40c Agregado metodo de Euler

HEAD está ahora en 9b72f8b Graficar energia

```
$ ls
```

```
navierstokes.py graficar_E.py
```

Toda la información necesaria para pasar de un estado del repositorio a otro (es decir, la información necesaria para recrear los cambios entre dos *snapshots* distintas) está contenida en el repositorio, en la historia del proyecto, y más específicamente en la carpeta oculta `.git`.

Evidentemente, tener una lista de todos los *commit hashes* que nos interesan no es cómodo. Usualmente vamos a tener ciertos *commits* que queremos destacar (por ejemplo, nueva funcionalidad totalmente implementada). Para acceder a ellos fácilmente sin necesidad de recordar el *hash*, podemos *etiquetarlos* mediante el comando `tag`, e indicando a qué *commit* nos referimos y qué nombre queremos ponerle:

```
$ git tag v1.0 9b72
```

De esta forma, podemos acceder fácilmente a ese *commit* utilizando el comando `git checkout v1.0`.

Con todo esto, vemos que los *commits* son análogos al último CV casero (*consolidar* una nueva carpeta en el CV casero), pero mucho más fáciles de

hacer. Por esta razón, al utilizar un SCV tiene mucho sentido realizar muchos *commit* intermedios, y no solamente cuando una funcionalidad está totalmente desarrollada. Y justamente por eso, es muy importante ser descriptivos con los *commit messages* y evitar las malas prácticas. Es muy común comenzar escribiendo detalladamente qué cambios se realizaron en cada *commit*, para prontamente aburrirse y comenzar a escribir cualquier cosa (como nos muestra, nuevamente, [XKCD](#)).

Antes de pasar al siguiente nivel de entendimiento de Git, es necesario un último comentario. Si bien se podría pensar que Git es un buen programa para que la computadora sea un gran repositorio y así poder recuperar cualquier archivo borrado por error, esa no es una buena idea. El procedimiento mediante el cual Git graba la historia del proyecto es mediante *Delta compression*. Si bien la explicación qué es y por qué funciona excede los alcances del presente libro, sí tiene sentido simplificar la explicación diciendo que Git guarda solo diferencias entre distintos *snapshots*. Para ello, Git debe poder analizar en forma simple los archivos. Es por esta razón que no debe hacerse un seguimiento de *todos* los archivos, sino solo de los archivos *fuentes* (el código, algún archivo de configuración). Git tiene dificultades para analizar archivos más complejos, como binarios (o documentos en texto enriquecido, o música, etc.), por lo que al incluirlos en el repositorio, este empieza a ser cada vez más pesado. En consecuencia, los archivos que se generan a través de la compilación o ejecución del programa, y que deberán volver a generarse repetidas veces a partir del código fuente, es mejor no incluirlos en el repositorio, puesto que solo ocuparán espacio en el disco (y en el repositorio), y no nos servirán cuando los traslademos a otra computadora.

La pregunta evidente es cómo podemos ignorar archivos. Es incómodo que Git nos recuerde permanentemente que hay archivos dentro de la carpeta donde se encuentra el repositorio que no están siendo seguidos. La forma de decirle al repositorio que debe ignorar ciertos archivos es mediante la creación de un archivo, `.gitignore`, que se encontrará en la carpeta raíz del repositorio (donde está la carpeta `.git`). En dicho archivo, podremos especificar reglas para que el repositorio ignore los archivos que queramos.

Algunas reglas para poner dentro de `.gitignore` son:

```
$ cat .gitignore
# Un comentario. Esta línea es ignorada.
# Ningun archivo *.a
*.a
```

```
# Pero trackear lib.a, aunque este ignorando los archivos *.a
!lib.a

# Ignorar archivo TODO, pero no los subdir/TODO
/TODO

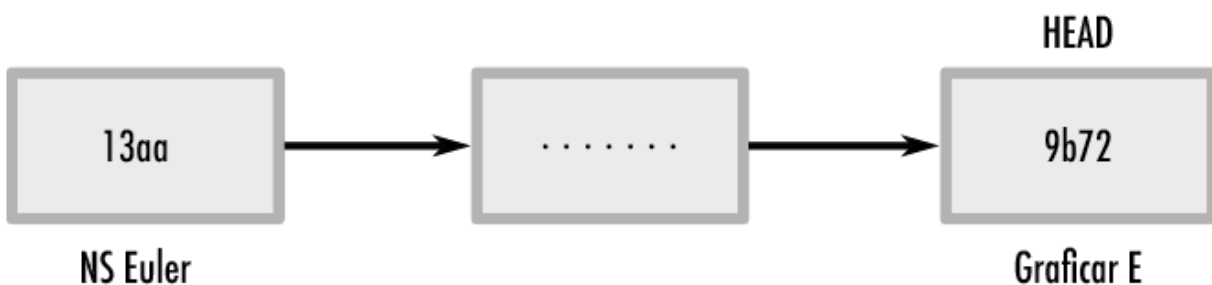
# Ignorar todos los archivos en el directorio build/
build/
# Ignorar archivos doc/*.txt, pero no los doc/server/arch.txt
doc/*.txt
# Ignorar archivos .txt en el directorio doc/ y los subdirectorios
doc/**/*.txt
```

Hay que recordar que este archivo, *a priori*, no es distinto a cualquier otro. Entonces, para que Git tome en consideración lo que dice, hay que incluirlo en el repositorio (con `git add .gitignore` seguido de un `git commit`).

La historia como un grafo

Con estas herramientas y conceptos fundamentales, podemos avanzar un poco más en el entendimiento y la visualización de la historia de un repositorio. Usualmente, para esta etapa de entendimiento se utilizan grafos. En nuestro ejemplo, tendríamos la situación de la figura 2, una historia lineal donde cada *snapshot* es solo una variación del anterior.

Figura 2. Grafo de una historia lineal de un repositorio



Cada rectángulo corresponde a un estado consolidado (o versión, o *snapshot*) del repositorio. El número hexadecimal representa el *hash commit*, debajo vemos un texto similar al *commit message* (para poder orientarnos más fácilmente) e indicamos dónde estamos parados mediante el HEAD.

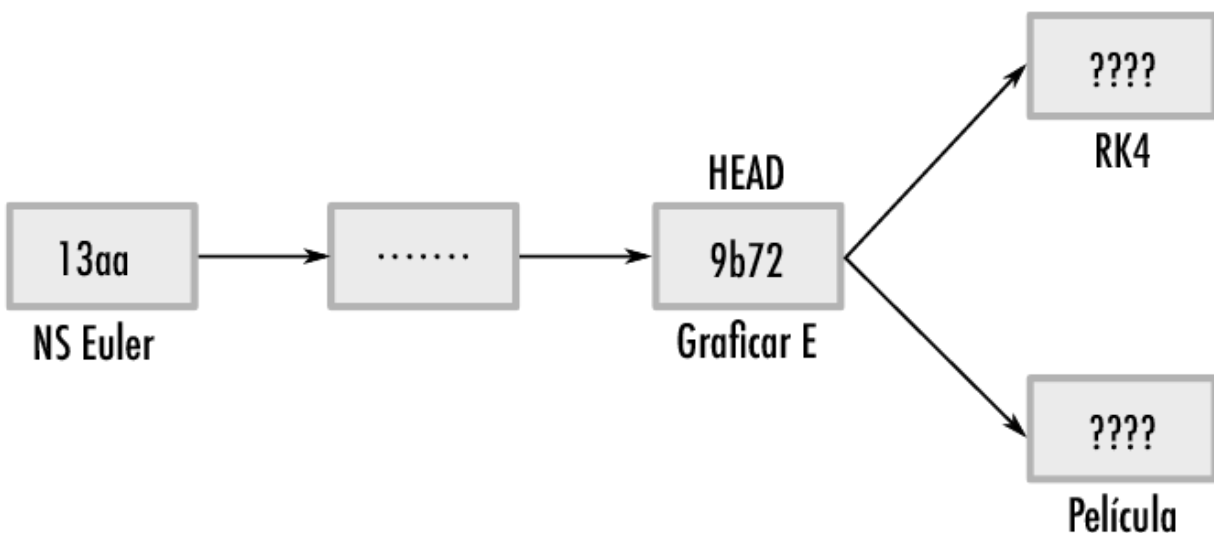
Esta visualización del repositorio nos da una mejor idea del desarrollo del código. Para mostrarla en la consola, podemos utilizar el *flag* `--graph` del comando `git log`: `git log --graph`.

Ahora compliquemos las cosas. Supongamos que queremos implementar dos funcionalidades a la vez:

1. Crear una película que grafique el paso del agua.
2. Implementar el método numérico Runge-Kutta de orden 4 (en lugar de Euler).

En el repositorio, ya tenemos una versión funcional (la **9b72** ya mostrada), que si bien no genera la película ni implementa Runge-Kutta, funciona. Como estamos trabajando en un repositorio, sabemos que podemos acceder fácilmente a dicha versión con un **checkout**, por lo que nos quedamos tranquilos al implementar una nueva funcionalidad. Pero si queremos implementar dos funcionalidades distintas, lo mejor sería codificarlas por separado, de manera que las dificultades de una no afecten al desarrollo de la otra. Gráficamente, podríamos pensar que el desarrollo convendría que fuese como se muestra en el grafo de la figura 3, donde ambas funcionalidades se desarrollarían por separado, empezando de un punto en común.

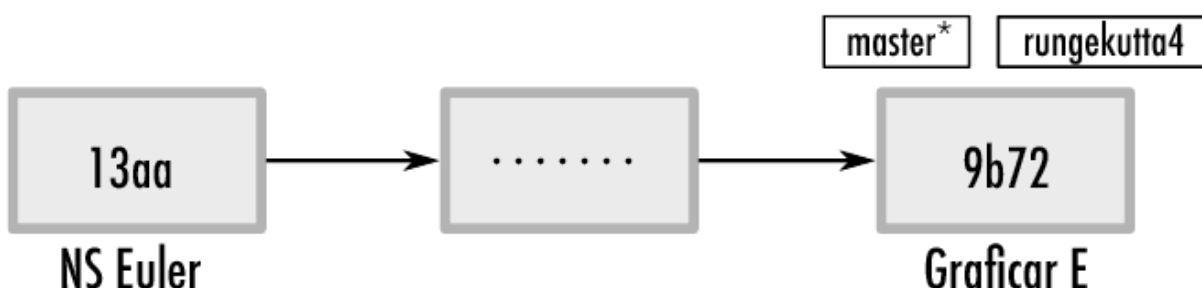
Figura 3. Repositorio con ramificaciones, donde cada nueva funcionalidad se realizará por separado



Ahora bien, ¿cómo se puede implementar esto? Mediante la utilización de *ramas* (*branch*), mediante el comando **branch**. Entonces, para crear una rama llamada “rungekutta4” y obtener así el caso mostrado en la figura 4, ejecutamos:

```
$ git branch rungekutta4
$ git branch #este comando imprime las ramas existentes
* master
rungekutta4
```

Figura 4. Creando una nueva rama

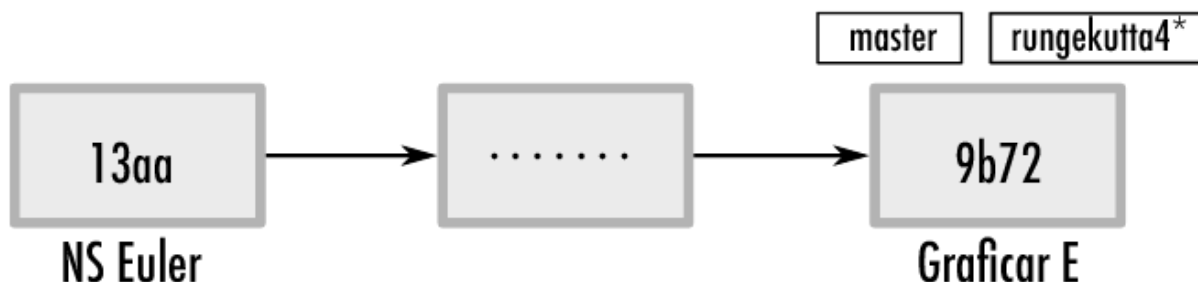


El asterisco nos indica que estamos en la rama **master**, por lo que los nuevos *commits* que hagamos irán a dicha rama. Para posicionarnos en la nueva rama, simplemente usamos el comando **checkout** (figura 5):

```
$ git checkout rungekutta4
$ git branch

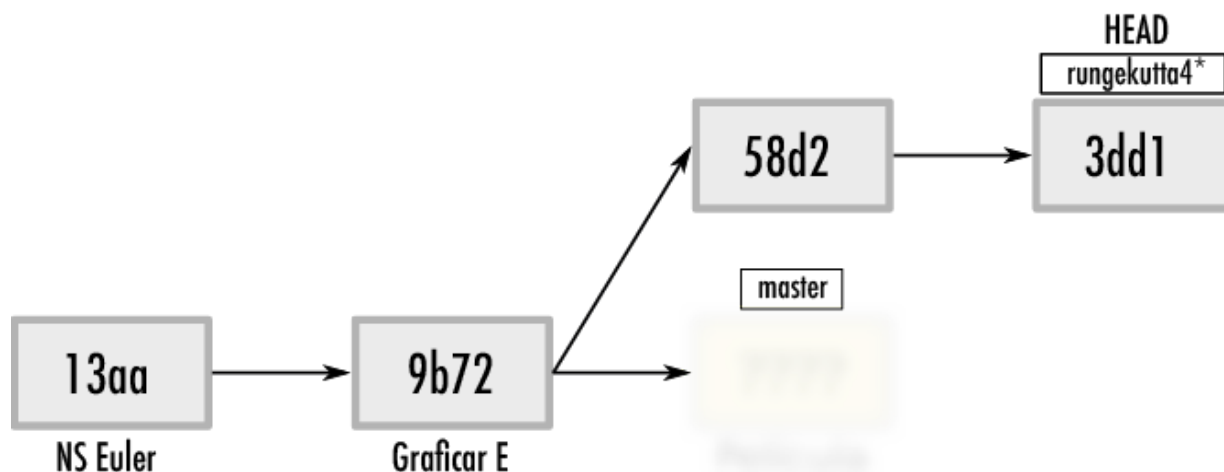
master
* rungekutta4
```

Figura 5. *Checkout* a la nueva rama del repositorio



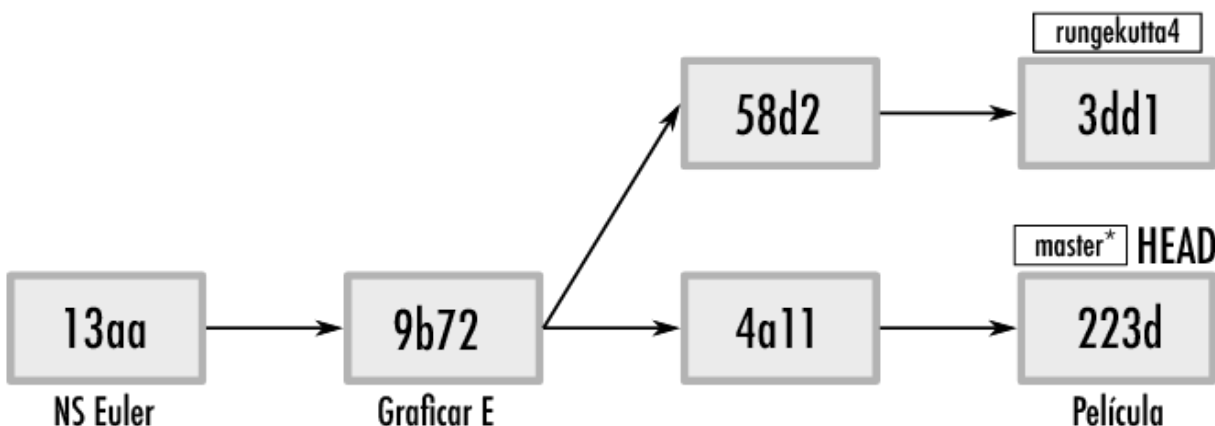
Entonces, ahora sí, continuemos con el desarrollo de nuestro código. Creamos el archivo `rk4.py` y empezamos a implementar el método. Observemos en la figura 6 que todos los *commits* los hemos hecho parados en la rama `rungekutta4`, mientras que la rama `master` no ha tenido *commits*.

Figura 6. Desarrollando en la rama `rungekutta4`



Ahora volvemos a la rama `master`, con `git checkout master`, y desarrollamos el código para que genere la película [\[3\]](#) (figura 7). A lo largo del desarrollo hacemos varios *commits*, y al final del día nos encontramos con la siguiente situación:

Figura 7. Código desarrollado y consolidado paralelamente en las ramas `rungekutta4` y `master`

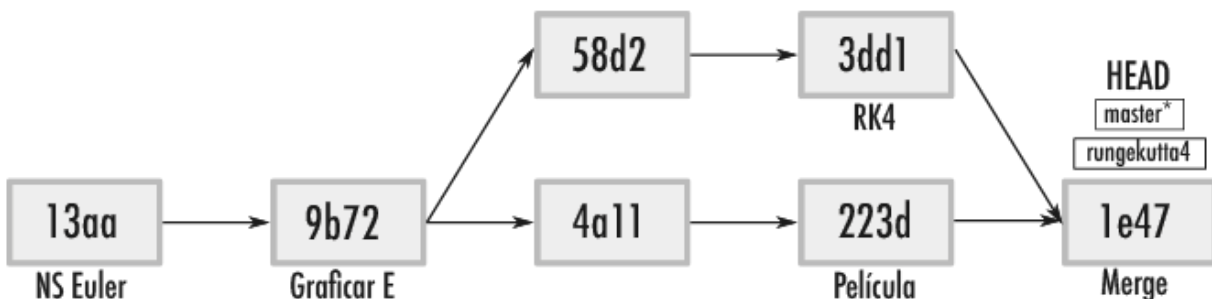


De esta forma, conseguimos desarrollar la implementación con Runge-Kutta y la película en dos desarrollos completamente separados, sin interferir entre ellos. El siguiente paso será juntar estas dos implementaciones, ya que al fin y al cabo lo que queremos es un código que se ocupe de todo. Para eso, debemos *unir* las dos ramas, mediante el comando `merge`. Este comando unirá sobre la rama en la que nos encontramos.

```
$ git merge master rungekutta4
Merge made by the 'recursive' strategy.
rk4.py | 0
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 rk4.py
```

Unir las ramas genera un nuevo estado del repositorio, por lo que se genera un nuevo *snapshot*. O sea, cuando hacemos un *merge*, Git nos pide un *commit message* para consolidar los cambios (figura 8).

Figura 8. Unir ramas



¿Cómo supo Git cómo debía unirlos? Las líneas de código en las que no hay duda acerca de cómo juntarlas, las une sin preguntar. Si en cambio existe la posibilidad de que haya algún conflicto, Git pregunta. En pocas palabras, es imposible (o casi) que Git arruine las líneas de código, pues es sumamente conservador en todos sus comandos. Eso *no* quiere decir que no se pueda romper el código. Git no hace magia. Siempre después de mergear ramas es necesario revisar que el código funcione adecuadamente. Por esa razón, es conveniente que las ramas que se van a unir sean lo más parecidas posible. Pero seguiremos con esto más adelante.

En caso de que haya conflictos al intentar unir ramas, Git nos avisará que tenemos que arreglar el problema antes de hacer el *commit*. Por ejemplo, podría suceder lo siguiente:

```
$ git merge master rungekutta4
Auto-fusionando navierstokes.py
CONFLICTO (contenido): Conflicto de fusión en navierstokes.py
Fusión automática falló; arregle los conflictos y luego realice un commit con el
resultado.
```

Aquí nos está informando que tiene un problema al unir el archivo `navierstokes.py` de las dos ramas. Si miramos el archivo, veremos (entre el código útil) un par de líneas similares a lo siguiente:

```
$ cat navierstokes.py
...
<<<<<< HEAD
Código proveniente de la rama master
=====
Código proveniente de la rama rungekutta4
>>>>>> exp
...
```

Si vemos en qué estado se encuentra el repositorio, obtendremos un mensaje similar al siguiente:

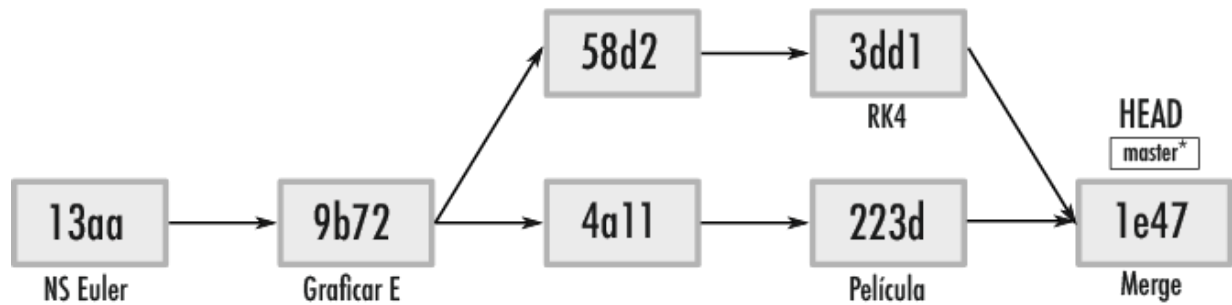
```
En la rama master
Tienes rutas no fusionadas.
(arregla los conflictos y corre "git commit"
(usa "git merge --abort" para abortar la fusion)
Rutas no fusionadas:
(usa "git add <archivo>..." para marcar una resolucion)
  ambos modificados: navierstokes.py
sin cambios agregados al commit (usa "git add" y/o "git commit -a")
```

Para proceder, debemos entrar en los archivos que están generando conflicto, arreglarlos a mano, ponerlos en la *stage area* (con **git add**) y, finalmente, hacer el **git commit** para terminar de unir las ramas.

Luego de *mergear* ramas, cuando la rama secundaria (en este caso, `rungekutta4`) dejó de utilizarse, es buena costumbre borrarlas. Es importante mencionar que las ramas no son más que una etiqueta que se les pone a las *snapshots*. Por lo tanto, “borrar” una rama no significa borrar los *snapshots* de esa rama guardados en el repositorio, sino solo que no nos aparezca el nombre de la rama cuando ejecutamos **git branch** (figura 9). Para borrar una rama ejecutamos:

```
$ git branch -d rungekutta4
```


Figura 9. Unir ramas y borrar rama secundaria



Resumen de comandos básicos en un repositorio local

Tabla 2. Resumen de comandos básicos utilizados cuando se trabaja en un repositorio local

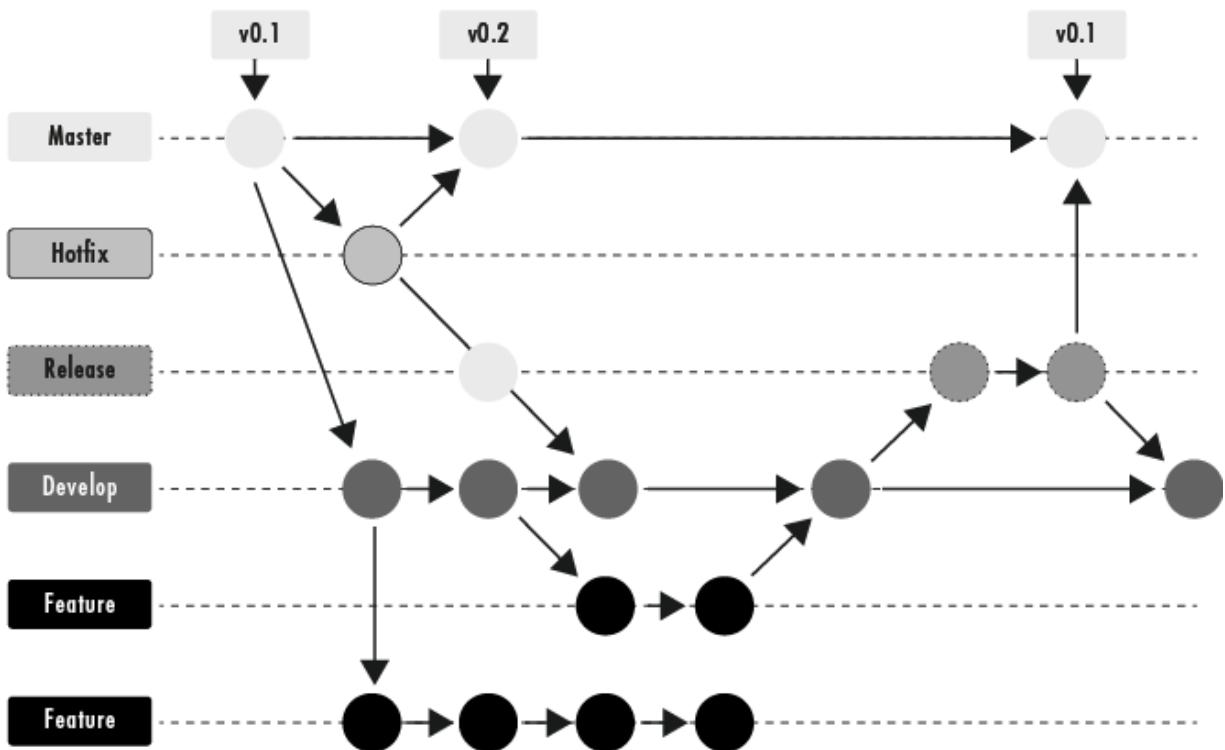
Comando	Descripción
git init	Crea repositorio
git status	Estado del repositorio
git add	Agrega a <i>stage</i>
git commit	Guarda en repositorio lo que está en stage
git branch <rama>	Crea nueva rama
git branch	Lista las ramas existentes
git checkout <id>	Visita un determinado estado
git diff	Indica diferencias
git log	Muestra los <i>commits</i> hechos
git tag <new_tag> <hash>	Permite cambiar tags de los <i>snapshots</i>

Flujo de trabajo (o cómo pensar el desarrollo del software)

Flujo de trabajo de un proyecto

El ciclo de vida de un programa se asemeja bastante al grafo de la figura 10. Analicémoslo un poco.

Figura 10. Flujo de trabajo de un proyecto



Vemos que hay una rama *Master*, que corresponde a las distintas versiones del programa a las que puede acceder el público en general. También hay una rama *Hotfix* utilizada para arreglar pequeños errores de la rama *Master*, y ramas *Release*, con versiones casi listas para poner en *Master*. Y también, ya adentrándonos en el desarrollo de software, están las ramas *Develop* y varias ramas *Feature* (que, a la hora de implementarlas en Git, deben llevar distintos nombres).

Entonces, con esta lógica, desarrollamos el código en la rama *Develop*, vamos añadiendo nuevas funcionalidades mediante ramas *Feature*. Y a medida que vamos testeando el código, lo vamos pasando a las ramas *Release* y *Master*.

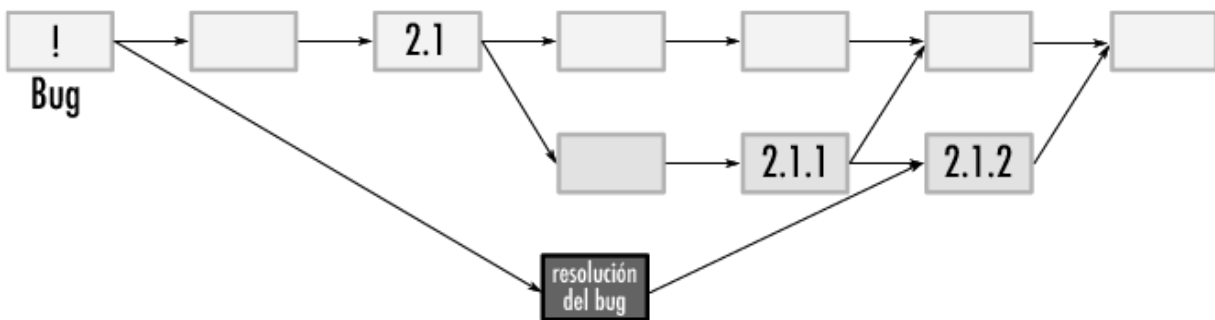
¿Y por qué se trabaja de esa manera? Además de las razones más obvias (como, por ejemplo, que el código que hay en *Master* debería tener la menor

cantidad de bugs posible), hay una razón concreta para tratar el desarrollo de software de manera tan atómica. Esa razón es que los *merge* son complicados: al unir dos ramas pueden surgir bugs. Y es mucho más sencillo encontrarlos y arreglarlos si las dos ramas que unimos son parecidas.

Cómo resolver los bugs

Antes de estudiar Git y los SCV, cuando todavía estábamos viendo las dificultades que podían surgir con los controles de versiones “caseros”, mencionamos un problema al que no le propusimos ninguna solución. Concretamente, ¿qué podemos hacer si en la versión 19 del programa nos damos cuenta de que arrastramos un bug desde la versión 1? La forma correcta de proceder es arreglar el bug donde se generó. Es decir, si el bug apareció en un determinado estado, la forma correcta de proceder es hacer un *checkout* a ese estado, arreglar el bug en una nueva rama, y mergear esa nueva rama con nuestra rama de desarrollo (figura 11).

Figura 11. Cómo solucionar un bug antiguo



De esta forma, cuando uno ve la historia del repositorio, sabe entre qué versiones estaba presente el bug. Si quisiéramos que, por ejemplo, en la versión 2.1 del programa ese bug estuviese resuelto, podríamos hacer un *Hotfix* y sacar una nueva versión 2.1. Pero, en última instancia, el bug debe ser resuelto donde se generó.

Trabajando a distancia. Repositorios remotos

Cada copia de un repositorio de Git es un par. O sea, en Git, no existe un repositorio Principal central y repositorios Secundarios, que deben comunicarse con el central. Cada copia del repositorio tiene el mismo nivel de importancia. Esta dinámica hace posible que una red de repositorios

remotos no esté conectada a internet, sino que alcance con que distintas computadoras estén conectadas entre sí.

Hecha esta aclaración, también es posible *utilizar* la idea de repositorio central y repositorios secundarios, donde todos los repositorios secundarios actualizan *hacia* el principal y se actualizan *desde* el principal, pero sin conexión entre ellos. Esa dinámica es la que suele generarse cuando se utiliza una plataforma en la nube para alojar repositorios (Github, Bitbucket, GitLab, SourceForge, por nombrar algunas). Es común tomar como repositorio principal el que está en la nube, y usarlo como repositorio central para actualizar los repositorios presentes en las computadoras del trabajo, laboratorio, casa, etc. Para ejemplos específicos, utilizaremos Github (<<http://www.github.com>>), pero los comandos son válidos para cualquiera de las plataformas.

Hay dos direcciones en las que se pueden intercambiar información entre un repositorio local (en la computadora en la que se está trabajando) y uno remoto (ya sea en la nube o en otra computadora). La información puede ir del repositorio local al remoto (*push*), o del remoto al local (*fetch*, *pull*, *clone*). Veamos qué hace cada uno de estos comandos.

En primer lugar, el comando *clone* sirve para clonar un repositorio remoto en una computadora local (copiar toda la historia). Es un comando que se utiliza una única vez, cuando localmente no existe repositorio. Por ejemplo, cuando queremos descargar un repositorio que tenemos en la nube hacia una computadora en la que no habíamos estado trabajando. Es, básicamente, una forma elegante de copiar un repositorio, en la que quedan configuradas algunas variables para poner conectarse con el repositorio remoto más fácilmente.

La forma de ejecutarlo es mediante el siguiente comando:

```
$ git clone https://github.com/wtpc/HOgit.git
Clonando en 'HOgit'...
remote: Enumerating objects: 21, done.
remote: Total 21 (delta 0), reused 0 (delta 0), pack-reused 21
Desempaquetando objetos: 100% (21/21), listo.
```

De esta forma, hemos hecho una copia local del repositorio remoto localizado en <<https://github.com/wtpc/HOgit.git>>, en la carpeta HOgit (por defecto, tiene el mismo nombre que el repositorio) creada en el directorio donde nos encontramos.

Utilizaremos ahora el resto de los comandos para tener actualizados tanto el repositorio local como el remoto, y Git sabrá cuál es el repositorio remoto (dónde está localizado), porque todo eso fue configurado durante la clonación.

El comando `git push` se utiliza para “empujar” los cambios nuevos efectuados en el repositorio local hacia el remoto. Por su parte, los comandos `git pull` y `git fetch` se utilizan para traer la información del repositorio remoto al local.

Hay una diferencia entre los dos comandos. Por un lado, `git fetch` realmente solo descarga nuevos datos desde un repositorio remoto, pero no integra ninguno de estos nuevos datos en el repositorio local. Este comando es excelente para obtener una vista nueva de todas las cosas que sucedieron en un repositorio remoto, y debido a su naturaleza “inofensiva”, se puede estar seguro de que no se manipulará, destruirá ni estropeará nada. Sin embargo, se suele utilizar para usos más avanzados, ya que, como se mencionó, no actualiza realmente el repositorio local.

Por el contrario, `git pull` se usa con un objetivo diferente en mente: actualizar el repositorio local (en verdad, la rama HEAD actual) con los últimos cambios del servidor remoto. Esto significa que no solo se descargan nuevos datos, sino que también se integran directamente en el repositorio local.

La forma en la que Git logra identificar el repositorio remoto es creando un *alias* (**origin**) con el que llamará la localización remota. La forma de ver hacia dónde está apuntando dicho alias es ejecutando `git remote -v`.

```
$ git remote -v
origin https://github.com/wtpc/HOgit.git (fetch)
origin https://github.com/wtpc/HOgit.git (push)
```

Por último, cuando hacemos un *push*, tenemos la opción de configurar su *upstream*, que en este contexto significa el repositorio remoto por defecto. La sintaxis es:

```
$ git push -u origin <branch>
```

Donde **<branch>** será la rama que estamos subiendo. Git entenderá que estamos subiendo desde la rama **<branch>** hacia la rama **origin/<branch>**. Posteriormente, para *pushear* alcanzará con ejecutar:

```
$ git push
```

Por ejemplo, para subir los nuevos *commit* hechos en la rama *master*, basta con ejecutar:

```
$ git push -u origin master
```

De la misma manera, para hacer un *pull*, ejecutamos:

```
$ git pull -u origin <branch>
```

Cuando hacemos *push* y *pull* nos estaremos comunicando con el repositorio remoto en, por ejemplo, Github. Entonces, cada vez que lo hagamos, deberemos poner nuestro usuario y contraseña.

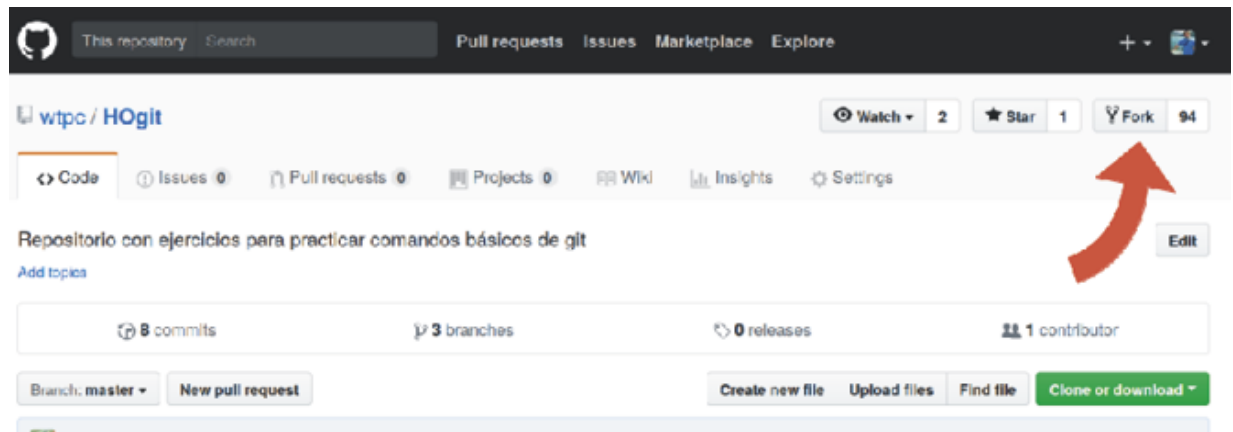
Cómo modificar repositorios ajenos

Si quisiéramos modificar un repositorio de otra persona, clonado directamente desde (por ejemplo) su Github, nos encontraríamos con que cuando queremos *pushear* las modificaciones que hicimos, no tenemos permisos para hacerlo. Esto se debe a que, efectivamente, la otra persona no nos permitió modificar su repositorio.

Ahora bien, si copiamos este repositorio (de otra persona) a nuestra cuenta de Github, y recién ahí clonamos el repositorio en una computadora local, cuando hagamos un *push* lo estaremos haciendo al repositorio remoto alojado *en nuestra cuenta*. Para hacer eso, podríamos descargar el repositorio ajeno a nuestra computadora y recién ahí subirlo a nuestra cuenta. O, más sencillo, directamente hacer la copia dentro de Github.

Para ello, realizamos un *fork*, que no es más que copiar el repositorio. *Forkear* un repositorio nos permite experimentar libremente con él sin afectar el proyecto original. Generalmente, los *forks* se utilizan para proponer cambios en el repositorio de otra persona o para utilizar el proyecto de otro como punto de partida para una idea propia. La razón del nombre (*fork* en inglés significa “tenedor”, el utensilio de cocina) es que el desarrollo de un código permite abrir varias ramas de desarrollo (como el cuerpo del tenedor, que se abre en los dientes). Independientemente de si es un buen nombre o no, la forma de realizarlo se ve en la figura 12.

Figura 12. Cómo forkear un repositorio ajeno en Github

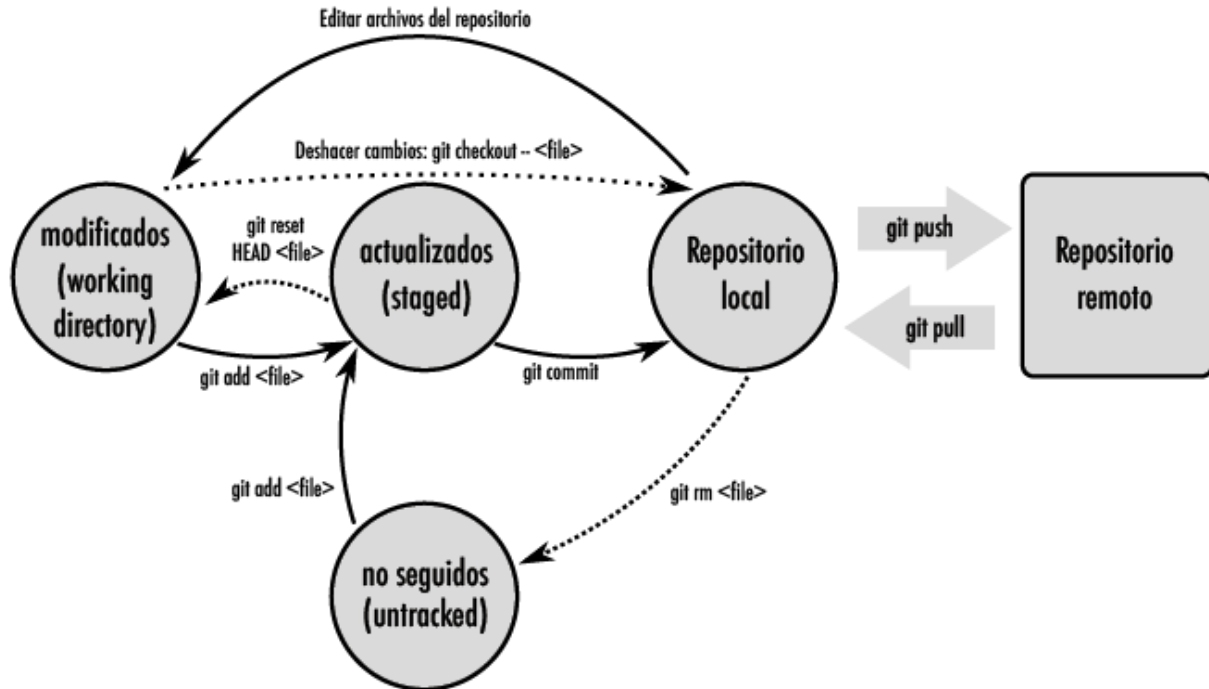


Comentarios finales

La forma usual de trabajar con Git es iniciar el día de trabajo con un **git pull** para traer los últimos cambios del repositorio remoto, trabajar todo el día (con múltiples *commit*), y finalizar el día con un **git pull** primero (para resolver localmente cualquier conflicto que pueda suceder con los *merge*), y luego con un **git push**, subiendo todas las nuevas *snapshots* a la nube.

Con todo lo visto, también podemos completar un poco más precisamente el ciclo de vida de un archivo, tal como puede observarse en la figura 13. Notemos que en dicha figura aparecen nuevos comandos, que no hemos visto en el capítulo. Concretamente, **git rm <file>** (para quitar un archivo del repositorio, sin por ello borrarlo), **git checkout -- <file>** (para deshacer los cambios de un archivo y llevarlo a como estaba en el último *commit* realizado) y **git reset HEAD <file>** (para quitar un archivo de la *stage area*). Además de estos comandos, hay muchos otros que han quedado por fuera (pueden consultar el libro “oficial” de Git, Pro Git de Chacon y Straub, 2014, para muchos más detalles). El objetivo de estas dos secciones no fue escribir un manual exhaustivo con todos los comandos y todas las posibilidades de utilización de Git, sino simplemente introducir las ideas básicas e instarlos a ustedes, lectores, a empezar a utilizar esta maravillosa herramienta, a incorporar esta hermosa filosofía de trabajo y a explorar con mucha curiosidad las amplias posibilidades que da Git.

Figura 13. Ciclo de vida de un archivo



Documentación

El siguiente paso importante para tener trazabilidad en un proyecto es la (ya conocida) documentación. Y si bien la forma de documentar puede llegar a ser muy personal, es bueno tener ciertas pautas generales.

Es muy común la situación de leer un código escrito por otra persona y no entenderlo, por la simple razón de que no está claro ni qué hace ni por qué. Y si bien esta situación es frustrante, es aún peor leer un código propio y no entenderlo.

Entonces, debemos recordar que *documentar significa comunicar*. Por esta razón, es necesario hacerlo en todos los niveles:

- Anotaciones en el código: de formato, comentarios, de estructuración de funciones de clases.
- Manual de usuario o de referencia.
- Introducción para nuevos usuarios o desarrolladores.

El código que escribimos puede ser reutilizado y leído varias veces. Entonces, en general la claridad es más importante que la astucia. De la misma forma, es conveniente elegir un estilo, respetarlo y ser consistentes: cómo estructuramos bloques, qué indentación utilizamos, cómo nombramos

las variables (nombres que tengan algún significado), cuál es la máxima longitud de línea que vamos a permitir (en general, 80 o 100 caracteres). Además, resulta más relevante explicar la intención más que el trabajo hecho. Por ejemplo, en la tabla 3, el código a la izquierda resulta mucho menos claro que el de la derecha. El primero describe pasos evidentes, pero falla en explicar el porqué de dichos pasos. El segundo, a la derecha, utiliza los comentarios entre líneas para exponer la razón por la que se realizan dichos cálculos.

Tabla 3. Documentar la intención de lo hecho. El código a la izquierda aclara lo evidente (qué se está haciendo), pero no la razón por la cual se realizan esos cálculos

<pre>def Tripletupla(x): # igualar y a x y= x # igualar z a x z= x # duplicar y y *= 2 # triplicar z z *= 3 # crear tupla t = (x, y, z) # retornar tupla return t</pre>	<pre>def Tripletupla(x): y =x z =x y *= 2 z *= 3 # Aplicado para escalar. Ver [34], eq. (2.3) t = (x, y, z) return t</pre>
---	--

La comunidad de Python tiene una serie de convenciones para trabajar: el PEP 8. En dicho documento se definen cómo deben llamarse las variables, las funciones, las clases, qué convención utilizar para las mayúsculas y minúsculas, qué indentación se debe usar, cómo documentar una función, etc. Es muy recomendable leerlo alguna vez.

Existen algunos programas que se encargan de revisar si el código escrito por uno sigue ciertas normativas sintácticas y estilísticas. Estos programas reciben el nombre de *linters*. El nombre *linter* se deriva del nombre de los pequeños trozos de fibra y pelusa que desprende la ropa, ya que el programa debe actuar como una trampa de pelusa de la secadora, detectando pequeños errores con grandes efectos. En el caso de Python, algunos de ellos son pyflakes, flake8, pep8, pylint.

Por último, existen herramientas que extraen automáticamente la documentación y los comentarios escritos en el código, y crean un documento

que queda como manual para los desarrolladores y los usuarios. Dos de estas herramientas son Doxygen y Sphinx.

Doxygen es muy versátil. Tiene soporte para códigos escritos en muchos lenguajes (C++, C, ObjC, C#, PHP, Java, Python, IDL, Fortran, etc.). Además, permite visualizar gráficamente las dependencias y generar páginas generales.

Por su parte, Sphinx tiene un soporte muy bueno para principiantes, y está especialmente desarrollado para generar documentación en Python.

Ambas herramientas generan páginas de documentación muy bellas. Por ejemplo, las páginas de documentación de SciPy, Numpy y Matplotlib están generadas con Sphinx, basándose en la documentación escrita dentro del código (tal como puede comprobarse en la figura 14).

Figura 14. La página de documentación de NumPy generada por Sphinx



Unit testing

El *unit testing* es una metodología de testeo de software mediante el cual se prueban unidades individuales de código fuente (uno o más módulos de programas de computadora junto con los datos de control asociados, los procedimientos de uso y los procedimientos operativos) para determinar si son aptos para su uso.

Este procedimiento sirve para asegurar que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, que si el estado inicial es válido, entonces el estado final es válido también.

Existen muchas herramientas de Python que permiten la automatización de pruebas, el intercambio de códigos de configuración y cierre para pruebas, la agregación de pruebas en colecciones y la independencia de las pruebas del marco de informes. Quizás la más fundamental sea *unittest*, pero una más sencilla (y de todas formas muy poderosa) es el paquete *pytest* (instalable vía *pip*, o vía los repositorios del sistema operativo). Lo más importante es que conceptualmente cumplen las mismas funciones.

El *framework* de *pytest* facilita la escritura de pequeñas pruebas, pero escala para admitir pruebas funcionales complejas para aplicaciones y bibliotecas.

Un ejemplo de una prueba simple podría ser el siguiente código en Python:

```
# contenido de test_redondear.py

import numpy as np
def redondear(x):
    return np.floor(x)

def test_respuesta():
    assert redondear(3.4) == 3
    assert redondear(0.9) == 1
    assert redondear(-1.1) == -1
```

Vemos que la función *redondear* está mal. Esta función está calculando el piso. Es decir, *floor*(3.4)=3, *floor*(0.9)=1 y *floor*(-1.1)=-2, mientras que los resultados esperables son 3, 1 y -1.

Ejecutando el test, obtenemos lo siguiente:

```
$ pytest
===== test session starts =====
platform linux -- Python 3.7.8, pytest-6.2.4,
py-1.10.0, pluggy-0.13.1
rootdir: /home/<usuario>/proyecto_nuevo
collected 1 item

Test_redondear.py F [100%]
```

```

===== FAILURES =====
def test_respuesta():
    assert redondear(3.4) == 3
>    assert redondear(0.9) == 1
E
    assert 0 == 1
E
    + where 0 = redondear(0.9)

test_redondear.py:9: AssertionError
===== short test summary info =====
FAILED test_redondear.py::test_respuesta -
assert 0 == 1 ===== 1 failed in 0.13s =====

```

Claramente, el test falló. Si corregimos el error, poniendo que la función `redondear` es `np.round(x)`, y corremos nuevamente el test, el test debería pasar todas las aseveraciones.

```

===== test session starts =====
platform linux -- Python 3.7.8, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/<usuario>/proyecto_nuevo
collected 1 item
Test_redondear.py . [100%]
===== 1 passed in 0.14s =====

```

Habiendo visto un caso muy simple y la funcionalidad básica del *unit testing*, podemos pensar por qué amerita que tengamos una buena parte de las líneas de nuestro código testeadas. Todos estos test pueden realizarse automáticamente, buscando errores en nuestro código. No veremos *unit testing* con más profundidad, ya que es un tema para escribir un libro entero. Sin embargo, es importante transmitir que estas herramientas existen y ayudan.

Entonces, mencionemos algunas de las ventajas de utilizar *unit testing*:

1. *Cualquier error se encuentra fácil y rápidamente.* Un código cubierto con pruebas es más confiable. Si un cambio futuro rompe algo en el código, podremos identificar la raíz del problema de inmediato en lugar de recurrir a una base de código difícil de manejar.
2. *Unit testing ahorra tiempo.* Cuando se escriben las pruebas unitarias, se encuentran muchos errores en la etapa de construcción del software, lo que evita la transición de estos errores a las siguientes etapas. Esto ahorra los costos de corregir los errores más adelante en el ciclo de vida del desarrollo.

3. *Las pruebas unitarias son una parte integral de la “programación extrema”* (esencialmente una estrategia de “probar todo lo que posiblemente pueda romperse”). Escribir pruebas unitarias con esta metodología simplifica el desarrollo y la refactorización de código facilita la integración y crea documentación viva. Lo que nos lleva al siguiente punto...
4. *Las pruebas unitarias proporcionan documentación.* Las pruebas unitarias son una especie de documentación viva del producto. Para saber qué funcionalidad proporciona un módulo u otro, un desarrollador puede consultar las pruebas unitarias para obtener una imagen básica de la lógica del módulo y del sistema en su conjunto. Los casos de prueba unitaria representan indicadores que contienen información sobre el uso apropiado o inapropiado de un componente de software. Por tanto, estos casos proporcionan la documentación perfecta para estos indicadores.
5. *Reutilizable y confiable.* Dentro de los entornos de pruebas unitarias, los módulos individuales de un producto quedan aislados entre sí y tienen su propia área de responsabilidad. Eso significa que el código es más confiable, se ha probado en un entorno contenido y, por lo tanto, reutilizable. El código reutilizable es muy beneficioso: es limpio, eficiente y coherente. Todo esto se acelera con las pruebas unitarias.
6. *Las pruebas unitarias ayudan a medir el rendimiento.* Unit testing nos puede brindar la oportunidad de descubrir posibles puntos de ruptura.
7. *Las pruebas unitarias reducen la complejidad del código.* La complejidad ciclomática es una medida cuantitativa que puede utilizar para comprender exactamente qué tan complejo es el programa y su código. Cuantas más rutas estén implícitas en un solo bloque de código, mayor será la complejidad. Es decir, cuando no hay una declaración de flujo de control en el código fuente, la tasa de complejidad es de uno, y con las declaraciones “if” aumenta gradualmente a dos o más. Aquí es donde lograr una cobertura de prueba unitaria perfecta (cobertura del 100% del código) se convierte en una tarea difícil. Cuantas más declaraciones condicionales tenga el código, más complejo será. Si la creación de pruebas unitarias se vuelve engorrosa, podría ser un indicador de que el código también

podría ser demasiado complicado. Pero sin pruebas unitarias que respondan objetivamente a la pregunta de si nuestro código funciona o no, todo lo que tenemos es nuestra propia suposición. Con las pruebas unitarias, tiene pruebas concretas.

En resumen, las pruebas unitarias ayudan a encontrar *bugs*, presentes y futuros, y reducen drásticamente su presencia. Entonces, ¿por qué no hay pruebas unitarias en todos los códigos? La principal razón es que hacer las pruebas unitarias conlleva tiempo, y en general estamos más preocupados por obtener los resultados rápidamente antes que tener un código limpio, reutilizable, etc. Sin embargo, no debemos olvidar el término *eficiencia*: poder hacer crecer el código sin *bugs*, también es eficiente.

Conclusiones

A lo largo del capítulo hemos introducido distintas formas de tener trazabilidad en un proyecto: cómo guardar y acceder a su historia (SCV), cómo entenderla (documentación) y cómo corregirla y prevenir errores (*unit testing*). Si bien algunas de estas herramientas son más fáciles de implementar que otras, y requieren solamente un poco de práctica, todas hacen a las buenas prácticas de programación, y es recomendable incorporarlas de a poco. La versatilidad de todas estas herramientas es muchísimo mayor que lo mostrado, pero las instrucciones básicas que hemos estudiado deberían permitirles empezar a utilizarlas y a ganar experiencia.

Para cerrar el capítulo, es importante recordar que el software que uno escribe probablemente sea utilizado por otra persona (como mínimo, por el yo del futuro). Por esta razón resulta muy relevante todo lo visto, buscando mejorar la calidad de la comunicación.

Resumen

Este capítulo trata acerca de cómo tener trazabilidad sobre un proyecto, en distintos niveles. En las dos primeras secciones, explicamos qué es un sistema de control de versiones, cómo lo podemos utilizar para conocer la historia de un proyecto, cómo podemos pensar el flujo de desarrollo de un código y cómo podemos distribuirlo remotamente. En particular, aprendemos los comandos básicos de Git, el sistema de control de versiones más

extendido. Posteriormente, en la tercera sección, hablamos un poco acerca de cómo documentar un proyecto. Por último, en la cuarta sección, introducimos cómo hacer tests automáticos del código. La unión de estos tres temas nos permitirá trazar la historia de un proyecto, y además desarrollar software disminuyendo los tiempos de resolución de errores y los tiempos requeridos para aprender (o reaprender) a utilizar y modificar el código.

Hands-on de git

En primer lugar, creamos una cuenta en Github. Hecho esto, forkeamos el repositorio que se encuentra en [<https://github.com/wtpc/HOgit>](https://github.com/wtpc/HOgit) a su cuenta (así pueden utilizar los comandos `push` y `pull` sin problemas).

Ahora sí, comenzamos. Entrando al repositorio propio, en [<https://github.com/<usuario>/HOgit>](https://github.com/<usuario>/HOgit), vemos que hay solo dos archivos, README y ejercicios.md. En cualquier lugar se pueden (¡se deberían!) correr `git status` y `git branch` -a para chequear en qué estado está el repositorio y en qué branch están.

Clonamos el repositorio de Github en nuestra computadora local. Para ello, abrimos una terminal y ejecutamos:

```
$ git clone https://github.com/wtpc/HOgit.git
$ cd HOgit
```

Editamos el archivo de README (con el editor de texto plano que prefiramos; en este caso usamos Vi) y hacemos un nuevo *commit*:

```
$ vi README.md
.....
$ git add README.md
$ git commit
```

Ya hay un nuevo *snapshot*. Ahora creamos una rama y nos movemos a ella.

```
$ git branch charlas
$ git checkout charlas
```

En esta *branch*, editamos README.md de nuevo y creamos un archivo llamado charlas.md:

```
$ vi README.md
...
$ vi charlas.md
```

```
$ git add .  
$ git commit
```

Ahora vamos a *master* (recuerden que no tiene estos cambios, porque es otra rama):

```
$ git checkout master
```

Y a partir de *master* creamos una nueva branch:

```
$ git branch ejercicios  
$ git checkout ejercicios
```

Editamos el archivo ejercicios.md:

```
$ vi ejercicios.md  
...  
$ git add ejercicios.md  
$ git commit
```

Volvemos a *master*, y ahora hacemos un *merge* de ambas *branches* por separado. No importa que el orden sea el mismo que en el que las modificamos. Esto es sensato porque las *branches* no se comunican entre sí:

```
$ git merge --no-ff ejercicios  
$ git merge --no-ff charlas
```

La opción `--no-ff` sirve para no “mezclar” las dos *branches*, y queda más prolijo el grafo. Es recomendable usarla siempre, pero no es fundamental. Podemos buscar en internet qué sucedería si no se utilizara.

Si queremos ver cómo quedó la historia del repo:

```
$ git log --oneline --graph
```

Finalmente, hacemos un *push* de todas las *branches* al repositorio remoto:

```
$ git push -u origin master  
$ git push -u origin ejercicios  
$ git push -u origin charlas
```

¡Y listo! En nuestra cuenta de Github ya tiene que estar subido. Podemos ver el *network* de Github que nos va a mostrar la historia.

Luego también podemos editar el Readme para agregar los comandos con los que hicimos el repositorio, y los subimos al repositorio remoto:


```
$ vi README.md
...
$ git add README.md
$ git commit
$ git push
```

Bibliografía

Chacon, S. y B. Straub, *Pro Git*, Apress, 2014. Disponible en: <<https://git-scm.com/book/en/v2>>.

Comparación entre distintos sistemas de control de versiones:

<https://en.wikipedia.org/wiki/Comparison_of_version-control_software>.

Gitflow Workflow: <<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>>.

Kolawa, A. y D. Huizinga, *Automated Defect Prevention: Best Practices in Software Management*, Wiley-IEEE Computer Society Press. 2007, p. 75.

Página de Git de Wikipedia: <<https://es.wikipedia.org/wiki/Git>>.

Página oficial de Git: <<https://git-scm.com/>>.

Página oficial de Doxygen: <<https://www.doxygen.nl/index.html>>.

Página oficial de Github: <<https://github.com/>>.

Página oficial de Pytest: <<https://pytest.org>>.

Página oficial de Sphinx: <<https://www.sphinx-doc.org/>>.

PEP 8, *Style Guide for Python Code*,
<<https://www.python.org/dev/peps/pep-0008/>>.

Unittest-Unit testing framework:
<<https://docs.python.org/3/library/unittest.html>>.

Wikipedia, “GNU General Public License”,
<https://es.wikipedia.org/wiki/GNU_General_Public_License#Versi%C3%B3n_2>.

Notas

- ¹ En Linux, el comando *mkdir* crea una carpeta, *cd* cambia el directorio en el que nos encontramos y *pwd* imprime en pantalla el path completo donde nos encontramos. El símbolo \$ indica que lo escrito a continuación es un comando introducido por el usuario en la terminal.

- 2 En Linux, el comando *ls* imprime en pantalla todos los archivos y carpetas no ocultos. Al agregar el *flag -a*, muestra en pantalla todo, lo oculto y lo no oculto.
- 3 Lo más correcto sería crear una rama “Película” y desarrollar el código allí.

Capítulo 3

Desarrollo de software modular

Pablo Alcain

Introducción

La programación para desarrollar ciencia tiene algunas características de uso que difieren mucho de las técnicas de *ingeniería de software*. En general, la búsqueda de velocidad de cómputo y el foco del programa como una *herramienta* para obtener un resultado hace que acostumbremos dejar de lado ideas fundamentales de ingeniería. No es el objetivo que nos volvamos *ingenieros de software*, en absoluto. Pero es necesario entender los conceptos y búsquedas detrás de estas ideas. No solo porque nos puede resultar muy útil para nuestro propio desarrollo, sino porque muchas veces tenemos que poder leer códigos más grandes, en los que indefectiblemente hay una proliferación de funciones y clases.

Como ejemplo vamos a ver estos dos códigos de Python, que integran la función $-e^{-t}$ con el método de Euler. Y agreguémosles una pequeña dificultad: ninguno tiene comentarios.

El primer código será:

```
import numpy as np
from matplotlib import pyplot as plt
h = 0.1
t = np.arange(0, 1.1, h)

s = np.zeros(len(t))
s[0] = -1

for i in range(len(t) - 1):
    s[i + 1] = s[i] + h*np.exp(-t[i])

plt.figure()
plt.plot(t, -np.exp(-t), label='Exacta')
plt.plot(t, s, '--', label='Aproximación')
```

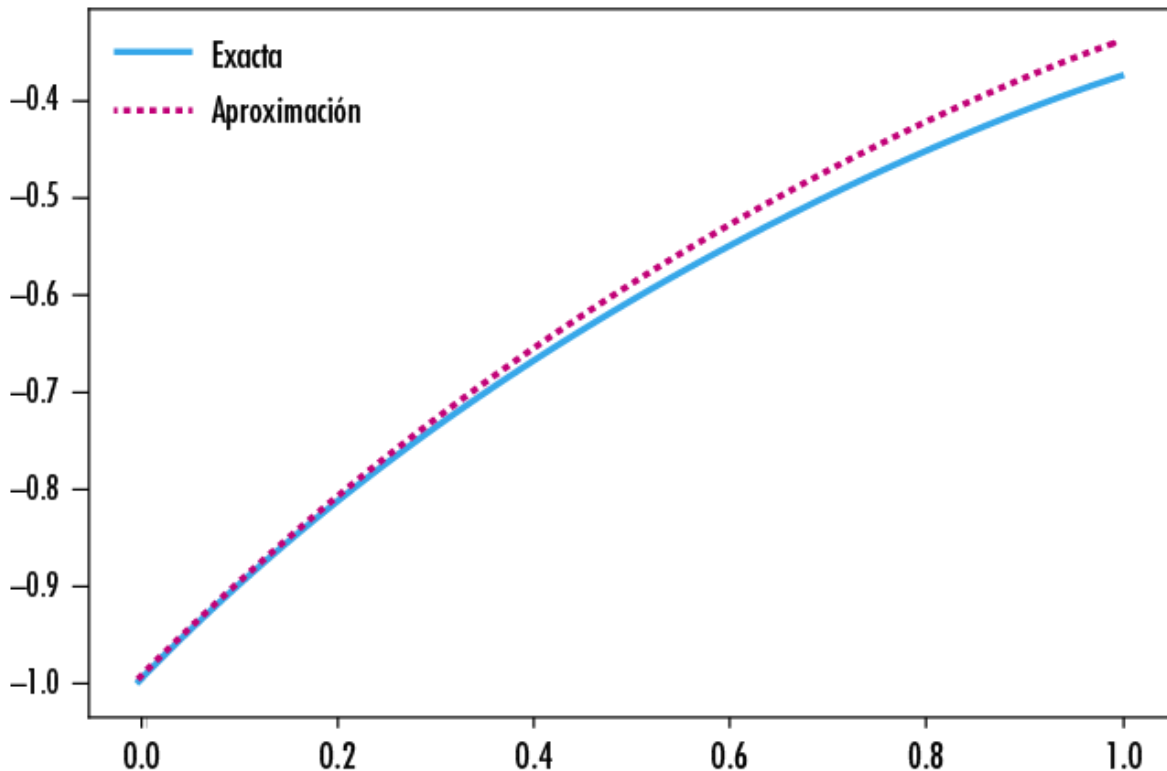
```
plt.legend()  
plt.show()
```

El segundo código será:

```
import numpy as np  
from matplotlib import pyplot as plt  
  
def integrar_euler(derivada, condicion_inicial, paso,  
tiempo_final):  
    tiempo = np.arange(0, tiempo_final, paso)  
    cantidad_de_pasos = len(tiempo)  
    solucion = np.zeros(cantidad_de_pasos)  
    solucion[0] = condicion_inicial  
    for i, t in enumerate(tiempo[:-1]):  
        solucion[i + 1] = solucion[i]  
        + paso * derivada(t)  
    return tiempo, solucion  
  
def integral_exacta(x):  
    return -np.exp(-x)  
  
def funcion(x):  
    return np.exp(-x)  
  
tiempo, solucion_numerica = integrar_euler  
(funcion,  
-1, 0.1, 1.0)  
plt.figure()  
plt.plot(tiempo, integral_exacta(tiempo),  
label='Exacta')  
plt.plot(tiempo, solucion_numerica, '--',  
label='Aproximación')  
plt.legend()  
plt.show()
```

Ambos devuelven como resultado la figura 1.

Figura 1. Salida producida por ambos códigos presentados en los ejemplos anteriores



¿Cómo se categorizarían? ¿Cuáles son las diferencias entre uno y otro? ¿Cuál es mejor usar y cuál es mejor escribir?

Si los ejecutamos, nos vamos a dar cuenta de que ambos códigos hacen *exactamente* lo mismo: integran, con el método de Euler, la función $-e^{-t}$ entre 0 y 1, con paso 0.1 y partiendo de la condición inicial a $t=0$. Sin embargo, es evidente que son distintos. Esto se debe a que el software que escribimos tiene dos objetivos diferentes: 1) ejecutarse y dar resultados; 2) ser modificado.

Y si bien en el punto 1 todos hacen lo mismo, en el punto 2 es donde quedan a las claras las diferencias. Es cierto, la cantidad de líneas para hacer la cuenta (es decir, descontando el gráfico y los *imports*) pasa de 6 en el primer caso a 13 en el segundo. Si, por ejemplo, quisiéramos cambiar la ecuación que queremos integrar, de $-e^{-t}$ a $(1 + t)$: ¿cuál de los dos códigos preferirían tener? El segundo código es mucho más claro en las intenciones de lo que hace: si quisiéramos cambiar a integrar $(1 + t)$, con condición inicial 1, el código es evidente sobre dónde tenemos que tocar: la condición inicial pasa a ser 1, la función es $(1 + t)$ y, si queremos comparar, la integral exacta es $(t + t^2)/2$.

Por lo tanto el código quedará como:

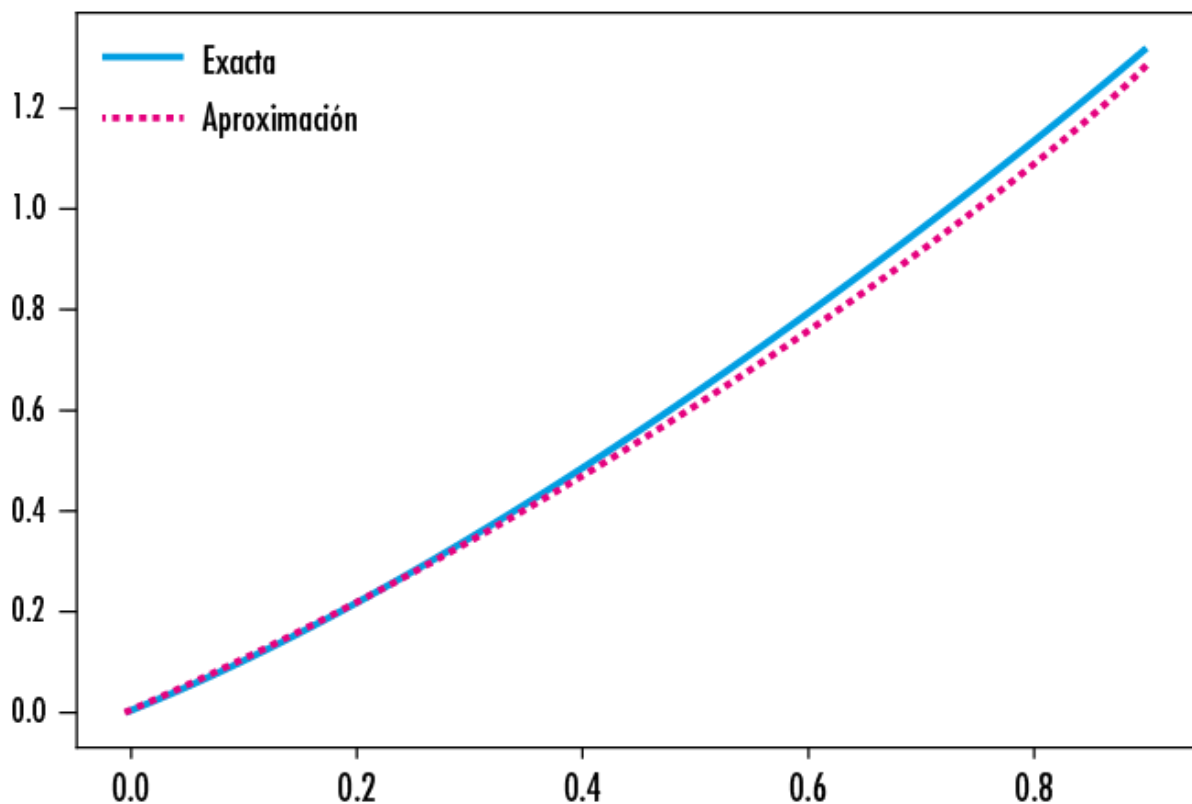
```
def integral_exacta(x): return x + x**2/2

def funcion(x): return 1 + x

tiempo, solucion_numerica = integrar_euler(funcion, 0, 0.1,
1.0)
plt.figure()
plt.plot(tiempo, integral_exacta(tiempo), label='Exacta')
plt.plot(tiempo, solucion_numerica, '--',
label='Aproximacion') plt.legend()
plt.show()
```

Y producirá como resultado la figura 2.

Figura 2. Salida producida por el código del cuadro 3



Incluso si quisiéramos implementar varias técnicas de integración y compararlas todas, sería mucho más fácil utilizar la segunda versión. Las modificaciones son dirigidas por el código. Es justamente para poder

escribir un código que sea más fácil de modificar que usamos técnicas de software development. La diferencia fundamental entre estos dos códigos es la *abstracción* del código en funciones.

Abstracción en funciones

¿Por qué las llamamos abstracciones? Porque el código principal se desentiende de cómo se integran las ecuaciones. Confía, podríamos decir, en que si a la función *integrar_euler* le pasamos los parámetros que pide, va a encargarse de hacer la cuenta correcta. Es decir, la función *integrar_euler* es una referencia *abstracta* al mecanismo de integración.

Escribir funciones en el código permite, además, que sea fácil de entender. [1] En el caso del segundo código, una vez que sabemos que *integrar_euler* realiza la tarea que necesitamos.

La performance, esa palabra

Muchas veces a la hora de implementar soluciones en las que le prestamos atención a la ingeniería del código surge la incómoda pregunta: ¿esto va a impactar en la velocidad de ejecución? Primero que nada, vamos con algunas ideas básicas. Casi todo el código que escribimos pasa por una especie de compilador o intérprete. Podemos pensar al compilador como el encargado de armar un “plan de ejecución” a partir de las instrucciones que escribimos en el código.

Por ejemplo, si en el código escribimos:

```
horas_por_semana = 7 * 24
```

El compilador tiene una *oportunidad* para darse cuenta de que es lo mismo escribir $7 * 24$ que escribir 168 directamente, de modo que automáticamente transforma esa expresión a `horas_por_semana = 168`. Esta técnica es conocida como *constant folding*. Y $7 * 24$ puede ser mucho más expresivo que 168 para la variable que estamos calculando.

Este pequeño ejemplo funciona como un buen botón de muestra respecto de las cosas que podemos hacer “gratis” al escribir código. Las optimizaciones más sencillas las puede hacer el compilador, lo que nos

permite a nosotros escribir código más claro... ¡gratis! La regla general es: si la optimización parece “demasiado simple”, se la dejamos al compilador. Y nosotros aprovechamos para escribir un código claro, con variables descriptivas y con funciones.

Como era de esperar, esta regla general no funciona siempre. Y en los casos en los que no funciona, hay un compromiso entre claridad del código y performance en la ejecución. Lo mejor que podemos hacer en este caso es tener claro cuál es el compromiso: qué ganamos y qué perdemos ante cada decisión. No es fácil medir todas estas variables (seguro que “claridad de código” sería una medida difícil de cuantificar), pero aprovechemos las que sí: cuando estamos en esta situación, siempre es recomendable medir cuánto más rápido resulta el código. Si escribir la función de forma óptima va a lograr que el código mejore un 1% y hacerlo ilegible y difícil de mantener, entonces quizás no valga la pena. Si ese código forma parte de una función que se ejecuta rara vez, entonces probablemente tampoco.

Pero hay casos en que sí, en los que la mejor decisión es escribir el código para que se pueda ejecutar lo más rápido posible. Incluso en esos casos, todavía hay herramientas de diseño que nos pueden servir: aislar lo más posible el código en cuestión (encapsularlo en una función, por ejemplo) y documentar en el código las razones detrás de esa decisión permiten que más adelante sepamos dónde debemos tocar y dónde no.

Estructuras de datos

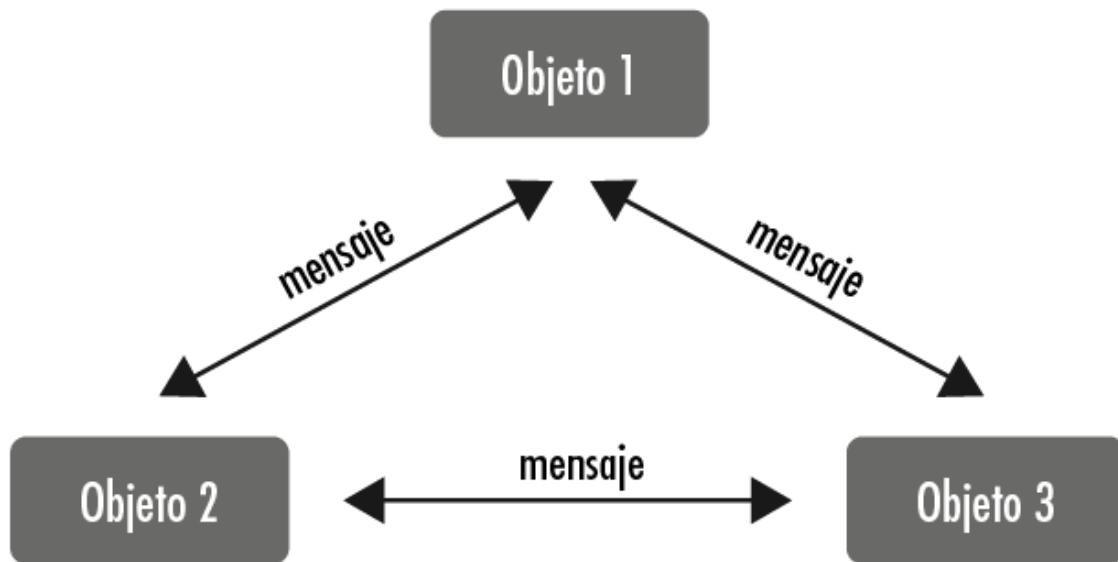
Programación orientada a objetos en Python

En general, gran parte de los códigos y scripts sencillos que comenzamos a escribir a la hora de escribir programas para simulaciones científicas son *estructurados*. Esto quiere decir que a la computadora le damos órdenes, que se ejecutan de forma secuencial. Así, el código queda en general segmentado en lo que llamamos funciones. Ya hablamos un poco respecto de cómo podemos usar las funciones para que el código sea más legible y más fácil de modificar, y dimos unas pautas generales respecto de cuándo usarlas. Ahora vamos a hablar de otro paradigma de programación: la programación orientada a objetos.

En la programación orientada a objetos, la idea es radicalmente distinta. En vez de enfocarnos en las instrucciones que le queremos dar a la

computadora, nos enfocamos en el diseño de objetos: entidades con cierto comportamiento, que pueden interactuar entre sí. Y, si quisiéramos ponerlo en esos términos, la “instrucción” que le damos a la computadora es decirle cuándo queremos que realicen cada una de las interacciones mostradas en la figura 3.

Figura 3. Interacciones entre los objetos



Estos objetos son conceptualmente similares a los objetos de la “vida real”. Por ejemplo, podemos decir que *una flauta* es un objeto. Este objeto tiene que tener:

- Estado: *cómo está el objeto*, por ejemplo: si la flauta está rota, afinada, etcétera.
- Comportamiento: *qué hace el objeto y cómo*, por ejemplo cómo la flauta hace sonar la nota *do bemol*.
- Identidad: *quién es el objeto*. En este caso, permite diferenciar entre esta flauta y cualquier otra flauta, aunque tengan el mismo estado y el mismo comportamiento.

El *estado* y el *comportamiento* de un objeto se asocian usualmente a *atributos* y *métodos* respectivamente. Los atributos de un objeto son los *datos* que este objeto posee; los métodos son los *procedimientos* que puede

hacer. Estos procedimientos pueden, y *en general lo hacen*, modificar el estado del objeto. Por ejemplo, en el caso de la flauta, puede ser que al tocar muchas notas, el estado pase de *afinada* a *desafinada*. Incluso más, podría pasar hasta a estar *rota*. Ese cambio de estado se debe únicamente al comportamiento propio de la flauta.

Además de los objetos, existe el concepto de *clase*. Las clases son *el concepto abstracto* que le da soporte conceptual a un objeto. Tomemos como ejemplo a una persona, llamémosla Alejandra. Podríamos decir que el concepto abstracto detrás de Alejandra, o de cualquier otra persona, es el de *ser humano*. Alejandra, Nicolás o Paula son todos “ejemplos” de qué es ser *humano*. En este caso, decimos que *humano* es la clase, mientras que cada una de las instancias en particular de esa clase (Alejandra, Nicolás, Paula) es el *objeto*. Una confusión usual en este escenario es pensar que la relación clase -> objeto es “compartir características”. Cuando lo pensamos así, estamos tentados a decir que *humano*, por ejemplo, puede ser el objeto de la clase *animal*, que es más general. Pero recordemos que lo fundamental para diferenciar *clase* y *objeto* no es la “generalidad”, sino si es un concepto abstracto o concreto. La relación entre *animal* -> *humano* se llama *herencia*, y forma parte de los cuatro principios de programación orientada a objetos que vamos a ver a continuación.

Cuatro principios de POO

En este apartado vamos a describir cuatro principios básicos de la programación orientada a objetos: encapsulación, abstracción, herencia y polimorfismo.

Encapsulación

La encapsulación trata de mantener el *estado* de un objeto como *privado*. Esto quiere decir que otros objetos no pueden acceder ni a leer ni a modificar ese estado si no pueden *pedirle* al objeto en cuestión que lo modifique. Una de sus aplicaciones más usuales son los conocidos como métodos *getter* y *setter*. Esto garantiza que el objeto pueda ser responsable de mantenerse consistente *todo el tiempo*. Retomemos el ejemplo de la flauta, que tiene un atributo que es *cantidad de notas tocadas*, y otro atributo que es *si está rota o no*. Y supongamos que la flauta que tenemos se rompe cuando se tocaron más de 10 notas. Si uno, desde afuera, pudiera modificar

la cantidad de notas tocadas sin más, podría generarse un estado en el que la cantidad de notas tocadas sea 30 (modificada por el exterior), mientras que el estado sea *sana*. Sin embargo, si nos aseguramos que *la única forma* de cambiar la cantidad de notas tocadas sea a través del método *tocar nota*, la flauta como objeto puede encargarse de fijarse si, al tocar esta nota, la flauta se rompió.^[2] La encapsulación permite que solo el objeto A pueda informar o modificar el estado del objeto A; todos los otros objetos deben hacerlo *a través* del objeto A.

También, como en el estado, podemos tener *métodos* privados. Es decir, acciones que solo el objeto puede decidir si toma o no. Por ejemplo, podemos decir que, en nuestro modelo, nadie puede forzar a un humano a hablar.^[3] Eso quiere decir que *hablar* es un método privado.

Abstracción

¿Qué pasa dentro de una flauta cuando tocamos una nota? La respuesta a esta pregunta es fundamental para describir y construir una flauta, pero (en general) irrelevante para un músico que la interpreta. Más allá del complicado movimiento de aire, resonancias y condiciones de presión en cada uno de los extremos, a la hora de tocar una nota nos importa tener una forma de decir qué nota queremos (por ejemplo, a través de la colocación de los dedos) y que efectivamente suene esa nota. Así, tocar una nota puede ser una *abstracción* sobre la compleja mecánica de fluidos que sucede detrás. En esencia está muy emparentada con la encapsulación, pero la idea de fondo es muy distinta. Las abstracciones son las que deciden qué tipo de comportamiento del objeto pueden ser *relevantes* o útiles para su comunicación con el exterior.

Herencia

En el apartado anterior mencionamos como ejemplo que un *humano* es una clase particular de *animal*. Esa relación se llama herencia y está muy relacionada con el concepto de *subtipado*. Usualmente decimos que *humano* es una *subclase* de *animal*. Hay muchos tipos de herencia: múltiple, jerárquica, multinivel, entre otras. Lo importante de entender en la herencia es que se comparten ciertos métodos e implementaciones. Por ejemplo, todos los instrumentos que mencionamos tienen la posibilidad de estar *afinados* o *desafinados*. A su vez, los instrumentos pueden ser de varios tipos (cuerdas,

viento, etc.). Podemos decir que la *Flauta* es una subclase de *Instrumento de Viento* que, a su vez, es una subclase de *Instrumento*.

Polimorfismo

El polimorfismo trata de dar una única interfaz y múltiples implementaciones para varios objetos de clases distintas. Está muy relacionado con el concepto de herencia. Por ejemplo, en el caso de los instrumentos: cualquier instrumento puede tocar una nota.^[4] Pero seguro que una *Flauta* y un *Violín* suenan muy distinto. Si bien la interfaz es la misma, las implementaciones difieren. Esto se puede expresar en que tanto *Flauta* como *Violín* son subtipos de *Instrumento*. El *Instrumento* tiene un contrato para poder tocar algunas notas, pero cada *Instrumento* en particular tiene que implementarlo. Incluso hasta podríamos implementar una forma *genérica* de tocar un instrumento cualquiera y que, después, las clases que hereden de él decidan cómo implementarlo si quieren sobrescribirlo.

El ejemplo de *tocar una nota* en una *Flauta* tiene entonces contacto con los cuatro conceptos fundamentales: es una *abstracción* de la compleja mecánica de fluidos detrás, que a su vez puede modificar atributos privados *encapsulados*; una flauta *hereda* de un *Instrumento* el atributo de estar rota o sana; ser un *Instrumento* define que puede tocar una nota, pero es su propia implementación (*polimorfismo*) la que decide cómo.

POO en Python

Comenzar programando clases en Python es muy sencillo. Por ejemplo, el siguiente código define la clase *Flauta*, e *instancia* un objeto *una_flauta* a partir de ella:

```
class Flauta:
    pass

una_flauta = Flauta()
print(una_flauta)
```

Que producirá como salida:

```
<__main__.Flauta object at 0x7f0f98a1ae50>
```

La línea `una_flauta = Flauta()` se conoce como “construcción” o instanciación: es el momento en el que creamos un objeto a partir de la clase. En general, se utiliza un método especial, el inicializador (llamado `__init__`) para poder *instanciar* una clase con ciertos valores desde el comienzo. Por ejemplo, podemos decir que una flauta tiene que tener, por lo menos, un nombre (para poder identificarla), un color y una vida media. Y esos atributos se deciden en el momento en el que se instancia la clase. Eso se escribiría así:

```
class Flauta:
    def __init__(self, nombre, color, vida_media):
        self.nombre = nombre
        self.color = color
        self.vida_media = vida_media
        self._notas_tocadas = 0

una_flauta = Flauta('primera', 'bronce', 10)
print(una_flauta)
```

Vale la pena notar dos cosas: el primer argumento del método es *self*. En Python, siempre el primer elemento de un método tiene que hacer referencia al objeto en cuestión. Es costumbre llamarlo *self*. Así, *self.nombre = nombre* dice, por ejemplo, que el *nombre de ese objeto* va a ser el parámetro *nombre* que le pasamos al `__init__`. Pero notemos también que este *self* no se usa cuando llamamos al constructor, sino que directamente se omite. Esta figura, si bien es un poco confusa al principio, termina siendo parte usual de un Python idiomático.

Además, `self._notas_tocadas` comienza con un guión bajo. En Python, a diferencia de otros lenguajes, *no existen los atributos ni los métodos privados*. Esto es una decisión de diseño explícita. El guión bajo al comienzo quiere decir que *esperamos* que ese atributo no sea modificado ni accedido por otros objetos. Sin embargo, recuerden que nadie nos prohíbe hacerlo. Es solo una convención de nombres: si quisiéramos modificarlo, podríamos.

Vamos a agregarle un par de métodos a esta flauta: que tenga la capacidad de tocar una nota y que pueda decirnos el estado (si el instrumento está roto o no):

```
class Flauta:
    def __init__(self, nombre, color, vida_media):
```

```
        self.nombre = nombre
        self.color = color
        self.vida_media = vida_media
        self._notas_tocadas = 0

    def tocar(self, nota):
        print(f"Flauta toca nota {nota}")
        self._notas_tocadas += 1
    def imprimir_estado(self):
        if self._notas_tocadas <= self.vida_media:
            print("La flauta esta sana")
        else:
            print("La flauta esta rota")
```

Y ahora que tenemos la flauta lista, ¡toquemos algunas notas!:

```
una_flauta = Flauta('primera', 'bronce', 10)
una_flauta.tocar("la")
una_flauta.tocar("sol")
una_flauta.tocar("mi")
```

Y producirá como salida:

```
Flauta toca nota la
Flauta toca nota sol
Flauta toca nota mi
```

Podemos aprovechar para averiguar si la flauta está rota o no, e incluso acceder al atributo de `_notas_tocadas`, a pesar de su convención:

```
una_flauta.imprimir_estado()
una_flauta._notas_tocadas
```

Que producirá:

```
La flauta está sana
3
```

Aprovechemos para ver la diferencia entre este paradigma de programación y la programación secuencial: en este caso, toda la lógica está puesta dentro desde un objeto. Ni siquiera el cálculo de si el estado es *sano* o *roto* lo hacemos “nosotros”. Todo lo hace la *clase* Flauta.

Herencia

Queremos seguir armando nuestra pequeña orquesta y, claro está, con flautas solo no nos alcanza. Así que vamos a hacer un segundo instrumento, el Cello:

```
class Cello:
    def __init__(self, nombre, color, vida_media):
        self.nombre = nombre
        self.color = color
        self.vida_media = vida_media
        self._notas_tocadas = 0
    def tocar(self, nota):
        print(f"Cello toca nota {nota}")
        self._notas_tocadas += 1
    def imprimir_estado(self):
        if self._notas_tocadas <= self.vida_media:
            print("El cello esta sano")
        else:
            print("El cello esta roto")

un_cello = Cello("primer", "madera", 20)
un_cello.tocar("la")
un_cello.tocar("re") un_cello.imprimir_estado()
```

Que producirá:

Cello toca nota la
Cello toca nota re
El cello está sano

Notamos que, si bien el Cello se comporta distinto a la Flauta (no suenan igual), algunos comportamientos los comparten: ambos tienen las mismas características al instanciarse y ambos imprimen el estado de la misma manera. Cuando estas cosas suceden es que se empiezan a producir fenómenos de herencia. Claro está que ni el Cello es una clase particular de Flauta, ni viceversa. En realidad, ambos son una clase particular de Instrumento. Y en este Instrumento vamos a implementar el comportamiento que el Cello y la Flauta tienen en común. Luego, tanto el Cello como la Flauta van a heredar de Instrumento e implementar los métodos propios (en este caso, tocar una nota).

```
class Instrumento:
```

```

def __init__(self, nombre, color, vida_media):
    self.nombre = nombre
    self.color = color
    self.vida_media = vida_media
    self._notas_tocadas = 0

def imprimir_estado(self):
    if self._notas_tocadas <= self.vida_media:
        print("Sano")
    else:
        print("Roto")

class Flauta(Instrumento):
    def tocar(self, nota):
        print(f"Flauta toca nota {nota}")
        self._notas_tocadas += 1

class Cello(Instrumento):
    def tocar(self, nota):
        print(f"Cello toca nota {nota}")
        self._notas_tocadas += 1

un_cello = Cello("primer", "madera", 20)
una_flauta = Flauta("primera", "bronce", 10)
un_cello.tocar("la")
una_flauta.tocar("si")
un_cello.imprimir_estado()
una_flauta.imprimir_estado()

```

Cello toca nota la
 Flauta toca nota si
 Sano
 Sano

No solo esto nos ahorra líneas de códigos, sino que además expresa en el código *por qué* esas líneas eran “repetidas”: formaban parte de una relación más profunda. Se repetían porque tanto Flauta como Cello son instrumentos.

Podríamos ir un paso más allá si quisiéramos. Observemos que ambos instrumentos, al tocar una nota, comparten un comportamiento: suben en 1 la cantidad de notas tocadas. Si lo pensáramos en la clave de la herencia, quiere decir que tanto la Flauta como el Cello primero realizan su acción específica y luego *delegan* un comportamiento común a la clase *Instrumento*. En Python, la keyword *super* se utiliza para acceder a la clase *padre*. Así, este comportamiento lo podemos escribir como:


```

class Instrumento:
    def __init__(self, nombre, color, vida_media):
        self.nombre = nombre
        self.color = color
        self.vida_media = vida_media
        self._notas_tocadas = 0

    def tocar(self, nota):
        self._notas_tocadas += 1
    def imprimir_estado(self):
        if self._notas_tocadas <= self.vida_media:
            print("Sano")
        else:
            print("Roto")

class Flauta(Instrumento):
    def tocar(self, nota):
        print(f"Flauta toca nota {nota}")
        super().tocar(nota)

class Cello(Instrumento):
    def tocar(self, nota):
        print(f"Cello toca nota {nota}")
        super().tocar(nota)

un_cello = Cello("primer", "madera", 20)
una_flauta = Flauta("primera", "bronce", 10)
un_cello.tocar("la")
una_flauta.tocar("si")
una_flauta.tocar("do")
una_flauta.tocar("re")
print(un_cello._notas_tocadas)
print(una_flauta._notas_tocadas)

```

Con su salida:

Cello toca nota la
 Flauta toca nota si
 Flauta toca nota do
 Flauta toca nota re
 1
 3

Con bastante criterio, podemos argumentar que esto que acabamos de hacer no ahorra líneas de código. Es cierto, pero la ventaja aquí es otra: el código es mucho más claro. Ahora es evidente que la flauta no sube la cantidad de “notas tocadas” por ser una flauta, sino por ser un instrumento. Además le estamos dando la pista, a cualquier persona que quiera programar un nuevo instrumento, de que seguramente tendrá que usar esa funcionalidad.

También estamos cayendo en un pequeño problema: ¿cómo extenderíamos este comportamiento si quisiéramos agregar un *Piano* por ejemplo, que puede tocar tanto notas como acordes? El diseño de un sistema de objetos está siempre cambiando, a medida que se agrega o se quita funcionalidad. Recuerden que es un *modelo* de la realidad y, como tal, tiene que cumplir su propósito: exponer las características más relevantes.

Composición

Ahora podemos armar una pequeña orquesta, compuesta de dos flautas y tres cellos. La única funcionalidad de esta orquesta (son músicos sin mucha experiencia... aún) es pedir que todos los instrumentos que la componen toquen la misma nota. Para poder diferenciarlos, vamos a modificar un poco el método *tocar*, para que escriba el nombre del instrumento.

```
class Flauta(Instrumento):
    def tocar(self, nota):
        print(f"Flauta {self.nombre} toca nota {nota}")
        super().tocar(nota)

class Cello(Instrumento):
    def tocar(self, nota):
        print(f"Cello {self.nombre} toca nota {nota}")
        super().tocar(nota)
```

Otra característica fundamental de los objetos es que pueden ser compuestos, a su vez, de otros objetos. En este ejemplo, una orquesta es un objeto que se *compone* de varios instrumentos. Y, a su vez, esta orquesta tiene la funcionalidad de tocar una nota con todos sus instrumentos:

```
class Orquesta:
    def __init__(self, instrumentos):
        self.instrumentos=instrumentos
```

```
def tocar(self, nota):
    for instrumento in self.instrumentos:
        instrumento.tocar(nota)

def imprimir_estado():
    for instrumento in self.instrumentos:
        instrumento.imprimir_estado()

una_flauta=("primera","bronce","10")
otra_flauta=("segundo","plata","10")
un_cello=("primera","madera","20")
otro_cello=("segundo","madera","15")
otro_cello_mas=Cello("tercer","madera","5")
instrumentos=[una_flauta,otra_flauta,
un_cello,otro_cello,otro_cello_mas]
una_orquesta=Orquesta(instrumentos)
una_orquesta.tocar("la")

print(una_flauta._notas_tocadas)
```

Cuya salida producirá:

Flauta primera toca nota la

Flauta segunda toca nota la

Cello primer toca nota la

Cello segundo toca nota la

Cello tercer toca nota la

1

No solo cada uno de los instrumentos tocó la nota adecuada, sino que además se actualizó su estado como esperábamos. Nuevamente, fíjense que toda la lógica, la comunicación y la orquestación de clases sucede dentro de estas.

Conclusiones

Como decíamos al comienzo del capítulo no es necesario implementar todas estas soluciones y estrategias para cada programa que hagamos. Pero van como disparador un par de preguntas que nos podemos hacer a la hora de escribirlo:

- ¿Podemos aprovechar esta oportunidad para dejar claro, en el código, que *entendemos* un poco más del problema?
- ¿Cuántas veces vamos a tener que modificar esta parte del código? (Tendemos a *subestimar* esto. Hay que ser consciente de que al escribir algo aún hay cosas que uno no sabe que no sabe.)
- ¿Estamos escribiendo varias líneas repetidas? Si la respuesta es positiva, ¿vale la pena poner todas las líneas repetidas dentro de la misma función/método/clase?
- Si alguien tiene que leer lo que programamos, ¿el código lo *ayuda* a entender o se lo *complica*?

Resumen

En este capítulo se realiza un recorrido por los conceptos básicos sobre modularización en funciones, el uso de estructuras de datos y la programación orientada a objetos. Estos conceptos resultan fundamentales para desarrollar proyectos de código colaborativo, así como para poder entender cómo usar y modificar proyectos existentes. Estas son tareas que cada día se vuelven más frecuentes en distintas disciplinas científicas.

Bibliografía

- Beck, K., *Extreme Programming Explained*, Massachusetts, Addison-Wesley, 1999.
- Evans, E., *Domain-Driven Design*, Massachusetts, Addison-Wesley, 2003.
- Fowler, M., *Refactoring*, Massachusetts, Addison-Wesley, 1999.
- Knuth, D., *Literate Programming*, Stanford, Stanford University Center for the Study of Language and Information, 1992.
- Naur, P., *Computing: A Human Activity*, ACM Press, 1992.
- Videos sobre algunas técnicas de desarrollo de software con y sin notebooks: <<https://www.youtube.com/watch?v=9Q6sLbz37gk>> y <<https://www.youtube.com/watch?v=7jiPeIFXb6U>>.

Notas

- 1 Esta idea de “escribir código que se pueda entender al leer” es llevada al extremo en un paradigma de computación (no del todo llevado a la práctica) llamado Literate Programming, en el que se espera que un programador escriba código simplemente describiendo sus ideas en lenguaje humano (por ejemplo, castellano).
- 2 Esta no es la única manera en la que esto se puede lograr, ¡se nos pueden ocurrir muchas más!
- 3 Esto, si bien dicho al pasar, es muy importante: nuestro diseño de objetos es un *modelo* de la realidad. No necesita comportarse tal cual con respecto a la realidad. Lo que importa es que el modelo capture, de la realidad, las características más relevantes para el uso que le vamos a dar.
- 4 Aquí es interesante el modelo que elegimos: los tambores, por ejemplo, ¿pueden *tocar una nota específica*? ¿O solo pueden afinarse para producir una nota y luego simplemente “suenan”? El modelo que queramos elegir va a depender mucho del uso que le vamos a dar y del contexto.

Capítulo 4

Cómo combinar lenguaje compilado e interpretado

Pablo Alcain

Introducción

Entre los aspectos de Python se suele decir que es un lenguaje que permite [pegar diversos lenguajes de programación](#). A pesar de que varios usuarios y defensores de Python esgrimen este argumento para valorarlo por sobre otros lenguajes, la realidad es que cualquier lenguaje interpretado (como Perl, Ruby, o el incipiente Julia) es capaz de conseguir la misma funcionalidad. Sin embargo, y a pesar del Bloqueo Global del Intérprete (GIL por sus siglas en inglés), el [problema más difícil de Python](#) (especialmente v2.7+) es el lenguaje interpretado principal en aplicaciones científicas. Su amplia difusión se debe sobre todo a la alta disponibilidad de librerías científicas tales como NumPy, SciPy, SciKitLearn o TensorFlow entre otras. Una de las ventajas de los lenguajes interpretados por sobre los compilados se vuelve evidente al comparar un código simple para calcular valores medios en Python (figura 1) y (figura 2). Consideremos que lo sintético del código no es una ventaja *per se* de los lenguajes interpretados por sobre los compilados, sino más bien es debido a las características usuales de estos lenguajes.

Figura 1. Ejemplo de un código simple para calcular valores medios en Python

```
# file: add_numbers.py
total = 10000000
for i in range(10):
    avg = 0.0
    for j in range(total):
        avg += j
    avg = avg/total
print("Average is {0}".format(avg))
```

Figura 2. Ejemplo de un código simple para calcular valores medios en Python

```
/* file: add_numbers.c */
#include <stdio.h>
int main(int argc, char **argv) {
    int i, j, total;
    double avg;
    total = 10000000;
    for (i = 0; i < 10; i++) {
        avg = 0;
        for (j = 0; j < total; j++) {
            avg += j;
        }
        avg = avg/total;
    }
    printf("Average is %f\n", avg);
}
```

No solo la sintaxis de Python es mucho más limpia, sino que además el código de C hay que compilarlo antes de ejecutarlo. La pregunta es, entonces, ¿por qué usamos C? La respuesta puede resultar obvia: ambos códigos realizan exactamente lo mismo, pero mientras que el código de Python lleva 8.047 s, el código en C tarda 0.284 s: 28 veces más rápido. Entonces, ¿cómo podemos resolver el problema de la velocidad? Cualquiera que haya utilizado NumPy sabe que un problema como este encaja justo con el uso de esta librería. El código de Python utilizando NumPy quedaría como se muestra en la figura 3.

Figura 3. Código del ejemplo para el cálculo del promedio utilizando Python y NumPy

```
# file: add_numbers_fast.py
from numpy import mean, arange
total = 10000000
a = arange(total)
for i in range(10):
    avg = mean(a)
print("Average is {0}".format(avg))
```

Este nuevo código tarda 0.266 s, comparable con el de C. Sin embargo, aunque para algunos esta resulte ser la solución adecuada, NumPy es bastante estricto: tenemos que usar vectores y escribir nuestra

implementación utilizando solo funciones de NumPy para aprovechar su velocidad. En consecuencia, perdimos gran parte de la versatilidad de Python. La cuestión que analizaremos en el presente capítulo es la siguiente: ¿es posible tener la versatilidad del lenguaje interpretado y la velocidad de uno compilado?

Para resolver esta cuestión, en principio tenemos dos métodos:

1. *Implementar un lenguaje interpretado en C.* Así como Python está escrito en C, podemos escribir nuestro propio programa en C y también crear un lenguaje interpretado “alrededor” de nuestro programa principal. El problema principal es que tendríamos que re-implementar muchas cosas en C que no son críticas en tiempo y podría resultar, además, en un lenguaje no muy sólido y con reglas no muy claras. Un buen ejemplo de la escritura de un lenguaje interpretado desde cero, con los problemas ya mencionados, a partir de código en C es la librería LAMMPS.
2. *Escribimos código en C y en Python por separado y los conectamos.* Como todos los lenguajes, en el fondo, tienen que ser código de máquina, entonces existe una forma de comunicar los lenguajes que queramos entre sí. En este caso, podemos escribir:
 - a. Código de Python.
 - b. Código de C.
 - c. Una API C/Python que se encargue de la comunicación.

Solo escribimos la parte que consume tiempo en C y podemos utilizar la flexibilidad de Python. La ya mencionada librería Numpy hace esto de hecho para conseguir su gran velocidad. Como estamos ejecutando en realidad una librería, liberamos el Bloque Global del Intérprete y, en la librería, podemos hacer *threading*.

En la siguiente sección, vamos a focalizarnos en la segunda opción. Pero, a su vez consideremos que la API C/Python tiene muchas formas de ser implementada.

1. *Escribimos el módulo completo en C, al estilo del lenguaje Python:*
Si Python fue escrito en C, podemos escribir cualquier módulo de Python en C. Esto es a través de, por ejemplo, la [API Python.h](#). Esto se puede tornar bastante engorroso, ya que hay que escribir mucho del parseo de Python en C (por ejemplo, cómo acceder a los elementos de las listas), y también considerar cuidadosamente el manejo de memoria. La mayor ventaja de esta opción, sin embargo, es que escribimos el código de C que va a ser ejecutado y, en consecuencia, podemos tomar decisiones de sintonía fina que pueden ser críticas en la velocidad de ejecución.
2. *Escribimos en módulo en Python y generamos código en C a partir de él.* Esto es lo que hace Cython. Como se puede ver de los ejemplos en la página oficial escribimos puro código de Python y luego se convierte a C automáticamente. Esta traducción de un lenguaje a otro de similar nivel se conoce como transpilación (yuxtaposición de *translation* y *compilation*). Cython es, efectivamente, un transpilador. Aunque esto parece muy bueno *a priori*, la verdad es que, como en cualquier compilador, se requiere muchísimo trabajo en el desarrollo para obtener buenas optimizaciones. Lo mismo vale al escribir código en C. Sin importar cuán bueno sea el compilador, no vamos a obtener mejor rendimiento que con un código bueno, escrito a mano en *assembler*. La sintonía fina que antes podíamos hacer con la API C/Python ya no se puede realizar.
3. *Escribimos el módulo en C y lo comunicamos explícitamente con Python.* Si pudiéramos exponer en Python los tipos de datos de bajo nivel de C (int, float...), podríamos utilizarlos para llamar a librerías de C. Esta es la idea detrás de [ctypes](#). Escribimos una función en C como lo haríamos usualmente, pero luego la llamamos desde Python con los tipos de datos adecuados. De esta forma podemos escribir código en C como en el caso 1, pero con la ventaja de que la interfaz y el manejo de memoria lo hacemos en Python.

Discutiremos brevemente la opción 2. Cython y otros similares como Numba pueden resultar en mejoras de tiempo relativamente (y a veces sorprendentemente) buenas con muy poco esfuerzo a partir de código en Python. Son, obviamente, la opción a seguir si no tuviéramos conocimientos de C. O, si el objetivo es que ese código de Python que se arrastra para

tardar una hora pase a tardar 10 minutos, seguramente valga la pena intentar este tipo de alternativas. Pero si escribimos código desde el comienzo mentalizados en el Cómputo de Alto Desempeño, definitivamente no va a alcanzar, justamente porque no podemos hacer optimizaciones “a mano” sobre el código de C, que es generado automáticamente. Existe una alternativa similar, llamada [F2PY](#), que implementa la misma idea, pero invertida. A partir de código de Fortran puro, genera automáticamente la interfaz de Python; en consecuencia, tenemos la sintonía fina en el lenguaje compilado y el código generado automáticamente y eventualmente no-optimizado en el ya lento Python (y no crítico en tiempo).

Respecto de las opciones 1 y 3 son muy similares y, en nuestra opinión, la opción 3 supera en varios aspectos la opción 1. Este planteo nos permite hacer una introducción general del problema, y el objetivo del presente capítulo es dar una mirada amplia de las posibilidades para llevar a cabo la comunicación entre Python y C. El resto del capítulo está organizado de la siguiente manera: en la siguiente sección vamos a mostrar algunos ejemplos de ctypes y su uso básico. Luego veremos un uso avanzado de ctypes que no es discutido usualmente y resulta muy útil para hacer completamente transparente la implementación en C para un eventual usuario de la librería. Finalmente, presentamos algunas conclusiones.

Sobre ctypes y su uso básico

En la introducción discutimos brevemente acerca de la validez de ctypes para acelerar el código de Python. Vale mencionar que, a decir verdad, no estamos acelerando el código en Python: llamamos, desde Python, a un código en C que es más rápido. Esta diferencia que parece casi trivial es fundamental para comprender este enfoque. ¿Qué tenemos que hacer siempre que queramos ejecutar código ya escrito? Linkeamos a una librería. Eso es lo que vamos a hacer en este caso: en vez de ejecutar una rutina en Python, vamos a darle ese trabajo a una librería de C. Recordemos que hay dos tipos distintos de librerías: estáticas y dinámicas. Para poder ejecutar una librería desde Python es evidente que vamos a necesitar que esta sea dinámica, ya que las llamadas a funciones van a tener que ser resueltas en tiempo de ejecución (recordemos que Python no se compila, por lo que no hay “tiempo de compilación”).

Para entender el procedimiento, a continuación construimos una librería muy pequeña que hace algunas operaciones matemáticas en escalares y vectores, separadas en dos archivos llamados `add_two.c` (figura 4) y `arrays.c` (figura 5).

Figura 4. Librería `add_two.c`

```
/* file: add_two.c */

float add_float(float a, float b) {
    return a + b;
}

int add_int(int a, int b) {
    return a + b;
}

int add_float_ref(float *a, float *b, float *c) {
    *c = *a + *b;
    return 0;
}

int add_int_ref(int *a, int *b, int *c) {
    *c = *a + *b;
    return 0;
}
```

Figura 5. Librería `arrays.c`

```
/* file: arrays.c */

int add_int_array(int *a, int *b, int *c, int n) {
    int i;
    for (i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
    return 0;
}

float dot_product(float *a, float *b, int n) {
    float res;
    int i;
```

```
res = 0;
for (i = 0; i < n; i++) {
    res = res + a[i] * b[i];
}
return res;
}
```

Construimos la librería dinámica con el compilador como se muestra en la figura 6.

Figura 6. Cómo construir la librería dinámica

```
$ gcc -c -fPIC arrays.c
$ gcc -c -fPIC add_two.c
$ gcc -shared arrays.o add_two.o -o libmymath.so
```

De este modo, chequeamos que la librería efectivamente tenga todos los símbolos definidos. Obtenemos como resultado los símbolos que se observan en la figura 7.

Figura 7. Símbolos que se obtienen

```
$ nm -n libmymath.so
...
000000000000010f5 T add_int_array
00000000000001168 T dot_product
000000000000011de T add_float
000000000000011f8 T add_int
0000000000000120c T add_float_ref
0000000000000123f T add_int_ref
```

Trabajando con escalares

Enteros

Con la librería dinámica ya compilada, ahora realizamos la comunicación con Python. Esa es la tarea de ctypes. La librería ctypes es simplemente la definición de los tipos usuales de C (int, float, double...) y un *loader* dinámico de librerías. ¿Cómo usamos estas funciones en Python, entonces? Tomemos, por ejemplo, la función add_int como se muestra en la figura 8.

Solo es necesario cargar con ctypes la librería que compilamos y llamamos a la función.

Figura 8. Función add_int

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> math.add_int(3, 4)
7
```

Con este ejemplo simple mostramos cómo sumar dos enteros utilizando esta librería como ejemplo pedagógico.

Punto flotante

Es válido preguntarse ahora: ¿qué pasa si tratamos de sumar dos números de punto flotante? Retomando el mismo ejemplo pero ahora con valores en punto flotante como se muestra en la figura 9. El valor que se obtiene como resultado es absurdo.

Figura 9. Función add aplicada ahora a valores de punto flotante

```
>>> math.add_float(3, 4)
0
```

En el caso de la figura 9, la función de la librería interpreta las entradas como float, pero nunca le informamos a Python que estamos pasándole números en punto flotante. Una solución ingenua sería tratar de pasar simplemente 3.0 y 4.0 como parámetros, pero eso falla catastróficamente como se observa en la figura 10.

Figura 10. Fallo. Al intentar pasar los valores como 3.0 y 4.0

```
>>> math.add_float(3.0, 4.0)
-----
-----
ArgumentError Traceback (most recent call last)
<ipython-input-6-c49cc7dd9937> in <module>
----> 1 math.add_float(3.0, 4.0)
ArgumentError: argument 1: <class 'TypeError'>: Don't know how to convert
parameter 1
```

Notemos que no es posible pasar cualquier parámetro bajo la confianza de que Python puede resolverlo. Recordemos que estamos llamando a una función de C, así que todas las ventajas del llamado *duck typing* de Python no están disponibles. Tenemos que, explícitamente, decir que estamos pasando float de C. Es decir, tenemos que usar los tipos definidos en ctypes como muestra la figura 11.

Figura 11. Aplicamos la función ahora con tipos definidos en ctypes

```
>>> math.add_float(C.c_float(3.0), C.c_float(4.0))  
2
```

Aún no obtenemos el resultado correcto pero estamos cerca de resolver la cuestión. Lo único que falta es que necesitamos que Python interprete el resultado como un float como se muestra en la figura 12.

Figura 12. Le indicamos a Python que interprete el resultado como un float

```
>>> math.add_float.restype = C.c_float  
>>> math.add_float(C.c_float(3.0), C.c_float(4.0))  
7.0
```

Escribir C.c_float cada vez que queremos ejecutar esta función no parece ser la solución más limpia de todas. Hay de hecho una forma mucho mejor para que Python sepa que la función va a tomar siempre C.c_float como argumentos, a través del método argtypes, como se muestra en la implementación de la figura 13.

Figura 13. Método argtypes

```
>>> math.add_float.restype = C.c_float  
>>> math.add_float.argtypes = [C.c_float, C.c_float]  
>>> math.add_float(3, 4)  
7.0
```

De este modo, la función puede ser llamada de forma completamente transparente, como llamaríamos a cualquier otra función de Python, pero en el fondo ejecuta código de C.

Por referencia

Cuando pasamos los argumentos por referencia (aunque puede ser una forma rara para implementarla en escalares, al menos al principio), la notación es mucho más engorrosa, ya que necesitamos pasar una posición de memoria. Podemos pedir la posición de memoria de una variable con la función `byref`.

Hay, sin embargo, una ventaja: como los argumentos son siempre posiciones de memoria (y, en consecuencia, enteros). Existe, de todos modos, el puntero a void, `c_void_p`, dentro de `ctypes`. Además podemos crear el puntero a cualquier otro tipo con la función `POINTER(type)`.

Esto funciona inmediatamente para cualquier tipo de variable, como vemos en la figura 14.

Figura 14. Un mismo ejemplo, pero los argumentos por referencia

```
>>> three = C.c_int(3)
>>> four = C.c_int(4)
>>> res = C.c_int()
>>> math.add_int_ref(C.byref(three),
                     C.byref(four),
                     C.byref(res))
0
>>> res.value
7
```

Existe sin embargo una ventaja: como los argumentos son siempre posiciones de memoria (y, en consecuencia, enteros), funciona inmediatamente para cualquier tipo de variable, como se observa en la figura 15.

Figura 15. El mismo ejemplo, pero los argumentos pasados por referencia, con float types

```
>>> three = C.c_float(3)
>>> four = C.c_float(4)
>>> res = C.c_float()
>>> math.add_float_ref(C.byref(three),
                       C.byref(four),
                       C.byref(res))
0
>>> res.value
7.0
```

En este caso la notación no se puede limpiar fácilmente como en el caso anterior, pero podemos escribir un wrapper de la función como se muestra en la figura 16.

Figura 16. Wrapper de la función

```
def add_float_ref_python(a, b):
    a_c = C.c_float(a)
    b_c = C.c_float(b)
    res_c = C.c_float()
    math.add_float_ref(C.byref(a_c), C.byref(b_c), C.byref(res_c))
    return res_c.value
```

De este modo, tenemos una llamada a C que suma por referencia y completamente transparente para el usuario final.

Trabajando con arrays

Sabiendo manejar argumentos por referencia en llamadas a funciones de C para escalares, manejar arrays se vuelve más sencillo: es simplemente una variable por referencia apuntando al primer elemento del array. Es, en general, buena práctica manejar la memoria en Python, pero esto lleva a la siguiente pregunta: ¿cómo alocamos memoria en Python? La forma más inmediata es como se muestra en la figura 17.

Figura 17. Cómo alocamos memoria en Python

```
>>> in1 = (C.c_int * 3) (1, 2, -5)
>>> in2 = (C.c_int * 3) (-1, 3, 3)
>>> out = (C.c_int * 3) (0, 0, 0)
>>> math.add_int_array(C.byref(in1),
                        C.byref(in2),
                        C.byref(out),
                        C.c_int(3))
>>> out[0], out[1], out[2]
(0, 5, -2)
```

NumPy arrays

Existe un enfoque distinto. Vamos a aprovechar la ventaja de tener todo un ecosistema desarrollado para trabajar con arrays: NumPy. Un array de

NumPy, como vimos en el capítulo 1, es un montón de metadatos (como tamaño, forma, tipo) y un puntero a la primera posición en memoria. Podemos acceder a esa posición de memoria de un NumPy array a través de `data_as` o `data` en el array como se observa en la figura 18.

Figura 18. Cómo acceder a esa posición de memoria de un NumPy array a través de `data_as` o `data` en el array

```
>>> import numpy as np
>>> intp = C.POINTER(C.c_int)
>>> in1 = np.array([1, 2, -5], dtype=C.c_int)
>>> in2 = np.array([-1, 3, 3], dtype=C.c_int)
>>> out = np.zeros(3, dtype=C.c_int)
>>> math.add_int_array(in1.ctypes.data_as(intp),
                        in2.ctypes.data_as(intp),
                        out.ctypes.data_as(intp),
                        C.c_int(3))
>>> out
array([0, 5, -2], dtype=int32)
```

Dos cuestiones son importantes a considerar aquí:

- No necesitamos definir el tipo como puntero a entero, podemos usar directamente el tipo `c_void_p` de `ctypes`.
- La salida es un array de NumPy que podemos usar inmediatamente en cualquier función de NumPy. Podemos, al igual que en el caso anterior, crear un wrapper para esta función para usarlo transparentemente como si fuera una función de Python.

Discusión sobre una forma más orientada a objetos

El presente capítulo cubre la mayor parte de la comunicación entre C y Python a través de funciones, que puede resultar útil si programamos en Python en un estilo parecido a C (con una programación más estructurada). La siguiente parte de esta serie estará dedicada a una forma de usar `ctypes` orientada a objetos. Vamos a discutir ahora una forma completamente “pythonica” de comunicar C y Python y hacer que el resultado final sea completamente orientado a objetos.

En la sección anterior discutimos cómo conectar Python y C a través de `ctypes` para acelerar código de Python. El objetivo principal era no solo

poder ejecutar código de C, sino también poder hacerlo de forma transparente para el usuario final. De esta forma, podíamos llamar código en C de modo que el usuario de la librería nunca sabría si está realmente ejecutando código de C. Tanto pasando los argumentos por valor o por referencia, logramos la transparencia de funciones de C. Sabemos, sin embargo, que a pesar de que Python es multiparadigma, su uso está muy orientado a la Programación de Objetos. De modo que sería provechoso si pudiéramos emular el comportamiento orientado a objetos con ctypes.

Estructuras

Entre los muchos tipos que ctypes expone a Python, está la estructura struct. Supongamos entonces que tenemos una estructura llamada Rectangle en C y una función que toma dicha estructura como argumento del modo en que se muestra en la figura 19.

Figura 19. Ejemplo de estructura que hemos llamado Rectangle en C

```
/* file: rectangle.c */

struct _rect {
    float height, width;
};

typedef struct _rect Rectangle;

float area(Rectangle rect) {
    return rect.height * rect.width;
}
```

La compilamos como una librería dinámica como se muestra en la figura 20.

Figura 20. Cómo compilamos la librería que hemos creado

```
$ gcc -c -fPIC rectangle.c
$ gcc -shared rectangle.o -o libgeometry.so
```

Una estructura de C en Python es un objeto que hereda de Structure en ctypes, y las variables de la estructura (en este caso height y width) son llamadas

`_fields_` en la estructura de ctypes. Así, una librería mínima de Python podría ser como se muestra en la figura 21.

Figura 21. Ejemplo de una librería mínima de Python

```
# file: geometry_minimal.py

import ctypes as C

CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

def area(rect):
    return CLIB.area(rect)
```

Interfaz de c/Python orientada a objetos a través de ctypes

Sobre la disposición de memoria

Cuando utilizamos los `_fields_` de la estructura de ctypes tenemos que ser extremadamente cuidadosos: el orden tiene que ser el mismo que en la estructura original de C. Estudiando la razón detrás de esto vamos a responder también una pregunta más importante: ¿por qué funciona lo que hicimos recién? Primero que nada: una estructura de C es en realidad una manera inteligente de nombrar posiciones de memoria relativas. En este caso, `height` es un float (o sea que tiene 4 bytes) que está localizado 0 bytes relativo a la posición de memoria `Rectangle`. De forma análoga, `width` es un float que está localizado a 4 bytes de la posición de memoria de `Rectangle` (ya que los primeros 4 bytes están tomados por `height`). Así que la función de C `area` simplemente toma los primeros 4 bytes comenzando por `rect` y los segundos 4 bytes comenzando por `rect`, interpreta los datos como float y los multiplica. Los primeros bytes van a estar ocupados por `_fields_`. Cualquier

otro método o atributo que sumemos va a ser agregado a continuación. En conclusión, la función área de C, cuando vaya a las posiciones de memoria que mencionamos recién, va a encontrar height y width, los valores que pretendíamos. Esto funciona, obviamente, siempre que pongamos en el mismo orden los atributos en la estructura de C y los `_fields_` en la clase de Python.

Implementación de la librería

Si modificamos la clase de Python agregando métodos o atributos nuevos, los primeros bytes van a mantenerse iguales. Esto significa que, para las funciones de C, agregar métodos y atributos no va a cambiar la estructura (al menos en las posiciones de memoria que estaban originalmente permitidas en la estructura de C). Esto es lo que vamos a usar para poder encapsular por completo la estructura de C como un objeto de Python. Podríamos, por ejemplo, agregar un constructor simple `__init__`. Pero el ejemplo más interesante es agregar métodos que originalmente eran funciones de C. A partir de la explicación de más arriba, se vuelve claro que tiene que haber alguna forma de encapsular la función área de C como un método del objeto de Python. El argumento que tenemos que usar para llamar la función área de C será la estructura en sí misma. Ahora podemos crear una librería de Python más avanzada, como se muestra en la figura 22.

Figura 22. Ejemplo de cómo se implementa una librería de Python más avanzada

```
>>> import geometry_minimal
>>> r = geometry_minimal.Rectangle()
>>> r.width = 10
>>> r.height = 30
>>> geometry_minimal.area(r)
300.0
```

De este modo, al utilizar esta librería desde Python se observa lo que vemos en la figura 23.

Figura 23. Cómo implementar la librería avanzada que creamos

```
# file: geometry.py
```

```
import ctypes as C

CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

    def __init__(self, height, width):
        self.height = height
        self.width = width

    def area(self):
        return CLIB.area(self)
```

Finalmente, la estructura de C y sus funciones están completamente encapsuladas. Para el usuario final no hay diferencia entre esta librería y una hecha completamente en Python, pero aquí estamos ejecutando, en realidad, código de C.

De este modo pudimos encapsular completamente el comportamiento de funciones de C que actúan sobre estructuras como métodos de un objeto de Python. Así obtuvimos un código completamente orientado a objetos, que a simple vista es puramente pythonico pero, en realidad, realiza sus cálculos en C.

Conclusiones

En el presente capítulo hemos explorado cómo tener la versatilidad del lenguaje interpretado y la velocidad de uno compilado, y hemos presentado y ponderado algunos ejemplos de aplicación. Hemos mostrado estrategias para comunicar los lenguajes que queramos entre sí, del modo que más se ajuste a las necesidades de quien desee implementar cómputo de alto desempeño. En particular, hemos analizado cuidadosamente el uso de ctypes. Realizamos un recorrido por las distintas opciones existentes y presentamos una comparación y análisis de cada caso de manera novedosa. Esperamos que pueda resultar útil en el camino particular que constituye el desarrollo de software científico.

Resumen

En el desarrollo de software científico, una de las habilidades que pueden resultar cruciales es la posibilidad de combinar e identificar cuál es el lenguaje de programación más adecuado para desarrollar la aplicación que se necesita. Consideramos que tener la versatilidad del lenguaje interpretado y la velocidad de uno compilado es una opción deseable que nos permite conservar dos aspectos. Por un lado, realizar las partes que tengan mayor requerimiento de cálculo aprovechando al máximo las capacidades del hardware mediante optimizaciones. Por otro lado, conservar la versatilidad de un lenguaje interpretado o de alto nivel, cuestión que puede ser muy conveniente a la hora de mejorar la interfaz con el usuario. En general, como ya hemos indicado, dicho usuario suele ser otro científico y no necesariamente experto en programación. En este capítulo hicimos un recorrido con ejemplos de implementación claros y comparaciones para poder conectar C y Python del modo más adecuado para desarrollar una librería para el cálculo.

Bibliografía

- Abadi, M. *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015, <<https://www.tensorflow.org/>>.
- Behnel, S. *et al.*, “Cython: The Best of Both Worlds”, *Computing in Science & Engineering*, vol. 13, Nº 2, 2011.
- Bezanson, J. *et al.*, *Julia*, 2012, <<https://julialang.org/>>.
- Duck-typing, <<https://docs.python.org/3/glossary.html#term-duck-typing>>.
- Matsumoto, Y., *Ruby*, 1995, <<https://www.ruby-lang.org/es/>>.
- Oliphant, T., *Guide to NumPy*, Trelgol Publishing, 2006.
- Pedregosa, F. *et al.*, *Scikit-learn: Machine Learning in Python*, 2011, <<https://scikit-learn.org/stable/>>.
- Plimpton, S., “Fast Parallel Algorithms for Short-Range Molecular Dynamics”, *Journal of Computational Physics*, vol. 117, 1995.
- Scipy Library Documentation, <<https://docs.scipy.org/doc/scipy-0.16.0>>.
- Van Rossum, G., *Python tutorial. Technical Report CS-R9526*, Ámsterdam, Centrum voor Wiskunde en Informatica, 1995.

Wall, L., Perl, 1987, <<https://www.perl.org/>>.

Capítulo 5

Procesos de optimización

María Graciela Molina

Introducción

A menudo, cuando se escribe software científico se requiere resolver problemas complejos que hacen un uso intensivo de los recursos de cómputos. Algunos ejemplos incluyen simulaciones Montecarlo, entrenamiento de modelos de aprendizaje automático, cómputo de un gran volumen de datos, resolución de complejos sistemas de ecuaciones diferenciales, entre tantos otros ejemplos.

En el proceso de resolución de un problema desde el punto de vista computacional la primera etapa suele ser la de probar nuestras soluciones en modo “*toy model*” (modelo de juguete) con pocos datos de entrada y/o condiciones iniciales. Esta etapa lleva a la escritura de una primera versión o prototipo de nuestra solución computacional. Cuando este primer programa resuelve el modelo de juguete nos sentimos satisfechos sin revisar más a fondo el algoritmo, las estructuras de datos o la lógica empleados. Recién comenzamos a pensar en optimizar el programa cuando necesitamos ejecutarlo con condiciones más realistas, con un mayor número de datos o en definitiva cuando el problema escala y requiere más recursos de los que disponemos.

Mostraremos que comprar más hardware no conlleva a una mejor performance del programa sino que, en algunos casos, poniendo esfuerzo en repensar la solución se puede alcanzar un desempeño más eficiente.

Optimizar código no es fácil y requiere del conocimiento del dominio pero también experiencia en desarrollo de software para conocer técnicas algorítmicas apropiadas, seguimiento del proyecto, herramientas para detectar errores y zonas críticas (cuellos de botella) en el uso de recursos de cómputo (almacenamiento y tiempo de ejecución).

Una buena estrategia es la de aprovechar el gran número de librerías científicas muy optimizadas y de libre acceso que la comunidad científica

utiliza. Es una buena práctica reusar código, más aún cuando se encuentra tan probado y optimizado como las librerías mencionadas. Ejemplos de estas librerías para Python son aquellas como Matplotlib, NumPy, SciPy, Keras, scikit-learn, entre tantas otras. Existen también librerías robustas y muy utilizadas en la comunidad científica para otros lenguajes (incluso multilenguajes), como por ejemplo: LAPACK - Linear Algebra PACKage, GSL - GNU Scientific Library, C++ Boost, Open MPI, entre tantas otras. El punto es no inventar la rueda de nuevo sino aprovechar las funciones que tienen estas librerías tan probadas.

Aparejado al uso de librerías y reuso del código está el punto de desarrollar habilidades para la búsqueda o “googleo”. Así como nos preocupamos por estar actualizados respecto al estado del arte en nuestra área de investigación, es sumamente importante aprender a encontrar y evaluar lo que existe en cuanto a software científico de calidad que podamos aprovechar.

Cuando comenzamos a plantear una solución computacional para nuestro problema, una disyuntiva común es la elección del lenguaje de programación. ¿Elijo el lenguaje que usa el grupo o aprendo uno nuevo? Heredar el lenguaje de programación suele ser un tema complejo. Suele suceder que muchas veces se hereda software, o se suele programar en el lenguaje que el supervisor o el grupo conoce. En el camino de entender ese código heredado, solemos preguntarnos si no es mejor mudarnos al lenguaje de programación más difundido en la comunidad o el que tenga mejor proyección futura. Todas estas respuestas posibles son parcialmente válidas y es un punto que es preciso analizar con detenimiento evaluando las fortalezas y debilidades de cada caso. Si es necesario aprender un nuevo lenguaje de programación, Python es uno de los lenguajes que poseen, entre otras características deseables, una curva de aprendizaje más acelerada. En los últimos años, además, se mantiene consistentemente al tope de los lenguajes más usados por la comunidad científica. La comunidad de Stack Overflow es una de las más usadas por programadores en el mundo y entre otras cosas realiza encuestas anuales sobre los lenguajes más utilizados en la actualidad y suele ser un buen proxy sobre la evolución en el uso de los lenguajes.

Al momento de evaluar qué podemos optimizar, hay que tratar de lograr un equilibrio entre el tiempo que lleva el desarrollo, el debugging, la validación, portabilidad, escalabilidad, tiempo de ejecución, entre otros puntos.

En líneas generales, al momento de pensar en optimizar nuestro programa podemos seguir el siguiente mapa de ruta:

- Usar aproximaciones cuando sea posible. Este punto hace hincapié en que en ocasiones al evaluar las necesidades de precisión de nuestros cálculos podemos resignar los resultados exactos por aproximaciones que involucren menos tiempo de ejecución o uso de menos recursos de almacenamiento (por ejemplo, uso de simple precisión).
- Una vez más hacemos énfasis en el uso de librerías ya optimizadas, como por ejemplo BLAS, LAPACK, etc. No es recomendable escribir nuestra propia rutina para descomposición de matrices si ya hay una versión optimizada de esta que podemos usar.
- Desarrollar algoritmos más eficientes. Esto involucra repensar la lógica del programa evaluando técnicas algorítmicas apropiadas y en caso de ser necesario aplicarlas en particular en el caso de las secciones críticas del programa.
- Utilizar estructuras de datos apropiadas. Muchas veces estructuras de datos simples pueden tener mejor performance que estructuras de datos más complejas y abstractas. Posiblemente esto signifique programar más cerca de la representación real de los datos y tener mayor conocimiento del hardware subyacente como por ejemplo el uso de memoria.
- Usar o escribir software optimizado para el hardware que se posee. Es imprescindible conocer el hardware que se tiene disponible ya que, a diferencia del desarrollo de software tradicional de propósitos generales, el software científico suele requerir muchos recursos de cómputo y al momento de optimizarlo conocer, por ejemplo, la jerarquía de memoria y cómo la estamos usando (entre otros recursos) es imprescindible para mejorar el desempeño de nuestro programa. En este punto también es importante analizar cuáles son los requisitos respecto a la portabilidad del código: ¿es necesario que sea ejecutado en otras computadoras?, ¿será ejecutado de manera intensiva o pocas veces? Cuanto más optimizado y adaptado a un determinado hardware, más resignamos en portabilidad. En contraposición, si los requisitos de portabilidad no son altos (por ejemplo, solo corremos la simulación en el sistema de cómputo de nuestra universidad),

entonces es aún más importante conocer cómo administrar eficientemente los recursos de hardware disponibles.

- Obtener hardware más veloz. Cuando se han implementado todas las soluciones en los puntos anteriores y aun así requerimos mayor cantidad de recursos (más velocidad, más memoria, etc.), entonces se debe analizar cómo escalar el hardware. En este punto, es muy importante entender nuestro problema, cómo escala a medida que crece y tener un software eficiente. De este modo, la compra de nuevo hardware será más óptima.
- Paralelizar. Cuando aun siguiendo las recomendaciones anteriores no alcanzamos un tiempo de ejecución aceptable, entonces se puede paralelizar el programa. Es importante saber que el proceso de paralelizar un código secuencial no es una mera traducción o simple división. Requiere un gran esfuerzo del programador para alcanzar resultados correctos y que realmente mejoren los tiempos. Aquí también se debe evaluar el tiempo que le insume al programador llegar a un resultado aceptable (y mejor que la versión secuencial). Generalmente, el programador es responsable de la comunicación y/o sincronización entre las secciones paralelizadas, es difícil hacer una evaluación de la performance (no se puede anticipar teóricamente la eficiencia), requiere conocer mucho más el hardware, y no suele tener resultados para cualquier tipo de problema (cuanto más grande o complejo es el problema suelen obtenerse mejores resultados al paralelizar).

En líneas generales, en este capítulo presentamos una serie de técnicas para la optimización de software científico, tanto a nivel algorítmico como a nivel de arquitectura, y compartimos algunos ejemplos y ejercicios.

Optimización de software: algoritmos y estructuras de datos

En esta sección, el objetivo es poner en foco la elección que hacemos de los algoritmos para resolver un problema determinado. ¿Cómo evaluamos a priori que la lógica que elegimos para la solución será la más eficiente? ¿Cuáles son los límites de mi algoritmo? ¿Cómo escala la complejidad cuando mi problema crece?

Una vez elegida una solución algorítmica, ¿qué estructuras de datos me permiten un acceso eficiente a mis datos? ¿Cuáles se adaptarán mejor si el problema escala?

El espíritu de esta sección no es hacer una revisión acabada sobre algoritmos y estructuras de datos, temas que requieren más de una materia en carreras de ciencias de la computación. El objetivo en nuestro caso es el de presentar algunas técnicas y compartir ejemplos pensados específicamente para el desarrollo de software científico.

Cómo elegir un algoritmo

Dado un problema cualquiera, existen infinitas maneras de resolverlo algorítmicamente. Al momento de seleccionar cuál es el *mejor* algoritmo que resuelve nuestro problema es importante tener en cuenta que: a) es deseable que sea fácil de entender, codificar y depurar (debuggear) y b) es deseable que haga un uso eficiente de los recursos de computación, especialmente en lo que se refiere a una ejecución rápida.

Para alcanzar los puntos en a), se recurre a la programación estructurada, a técnicas y herramientas para diseño como el lenguaje unificado de modelado o UML, a guías de estilo como en Python la conocida PEP8, a la adecuada documentación, a herramientas para el versionado del software, entre muchas otras herramientas; pero sobre todo requiere de práctica en algoritmia.

Respecto al punto b), para garantizar el uso adecuado de los recursos de computación disponibles, hay dos maneras de evaluar si un algoritmo es óptimo: 1) realizar una evaluación teórica, 2) realizar una evaluación práctica.

La evaluación teórica se enfoca en analizar cómo será el comportamiento del algoritmo *a priori*. Esto significa analizar el tiempo de ejecución respecto a las entradas y cómo este tiempo escala a medida que el tamaño de la entrada crece. Este análisis se realiza sin tener en cuenta las características particulares de la computadora donde correrá. Existe una definición formal del tiempo de ejecución.

Se denomina $T(n)$ al tiempo de ejecución de un programa de tamaño n . Una manera de acotar cómo puede crecer el tiempo de ejecución en el peor caso es usando la notación O-grande, notación asintótica o simplemente $O(n)$. Por ejemplo, si decimos que el tiempo de ejecución $T(n)$ es $O(n)$, significa que existen dos constantes positivas c y n_0 tal que para $n \geq n_0$, $T(n) < cn$.

$O(n)$ representa el peor tiempo de ejecución para una entrada de tamaño n . También se suelen usar otras notaciones para indicar el mejor caso y el caso promedio, aunque el más usado es el peor caso u $O(n)$.

Para ilustrar cómo entender y analizar el tiempo de ejecución de un algoritmo, veamos un ejemplo. Supongamos que con una computadora cualquiera deseamos ejecutar el siguiente código:

```
import time
start = time.time()
n=12
print(n)
end = time.time()
print("t=",end - start)
```

El tiempo resultante en este caso fue de $t = 0.0008983612060546875$.

¿Qué sucede si el tamaño de la entrada crece? supongamos que ejecutamos el código con $n=120000000$, el tiempo resultante en este caso fue de $t = 0.00010991096496582031$.

Una aclaración importante es que estos tiempos de ejecución obtenidos son válidos para una computadora en particular y bajo ciertas condiciones particulares, si se ejecuta el código en otras computadoras y/o instancias el valor será diferente.

Está claro que el tiempo de ejecución no cambió sustancialmente si el valor de la entrada es mayor. Esto se debe a que lo que estamos contando es el tiempo que insumen las instrucción en ejecutarse y en este caso cualquiera sea el valor de n se ejecutan el mismo número de operaciones. Sea cual sea el valor de n , el tiempo será prácticamente constante a lo que se denomina $O(1)$.

Veamos el siguiente ejemplo ilustrativo:

```
import time
start = time.time()
n=12
for i in range(n):
    print(i)
end = time.time()
print("t = ",end - start)
```

En este caso, el tiempo que se obtiene es de $t = 0.0020668506622314453$.

Modificando el valor de la entrada para diferentes valores de n se obtuvieron, para una computadora determinada, los siguientes tiempos:

$n=120$

$t=0.015095233917236328$

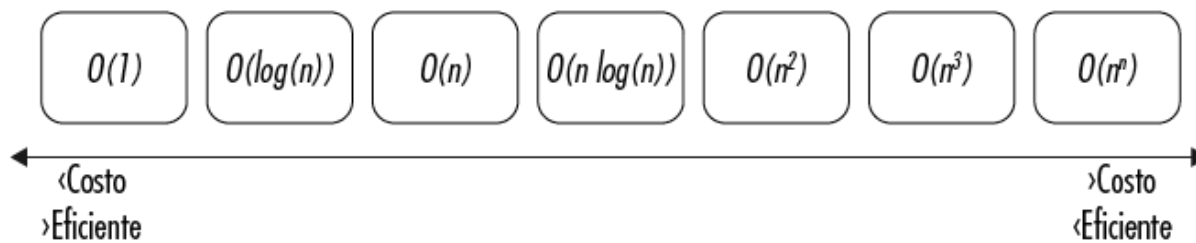
$n=12000$

$t=2.403151750564575$

En este caso, claramente el tiempo de ejecución crece con la entrada. Cada vez que n aumenta, aumenta la cantidad de iteraciones y en consecuencia aumenta la cantidad de instrucciones que se ejecutan. En este caso, el costo de ejecución es $O(n)$ porque teóricamente se puede probar que el tiempo de ejecución crece linealmente con el valor de n . Para más detalles de cómo estimar *a priori* el costo de ejecución, se puede consultar bibliografía especializada en algoritmia.

Este tipo de análisis nos permite elegir cuál será el algoritmo más eficiente para resolver un problema determinado. Si por ejemplo, para un problema particular, existen dos algoritmos que lo resuelven con costos $O(n)$ uno de ellos y el otro con costo $O(1)$, la mejor elección en términos de eficiencia será optar por el de menor costo.

Figura 1. Comparativa entre diferentes tiempos de ejecución



En la figura 1, se ilustra la ordenación de algoritmos de acuerdo a su eficiencia. Si nuestro algoritmo tiene un $T(n)$ cercano a $O(1)$, sabemos *a priori* que este es sumamente eficiente mientras que la situación inversa se encuentra para algoritmos con $T(n)$ cercanos a la parte derecha de la gráfica, siendo el peor caso aquí el tiempo exponencial $O(Cn)$.

¿Cómo usamos la información que brinda la notación O-grande al momento de seleccionar un algoritmo?

Supongamos que queremos ordenar un arreglo $A1 = [9 \ 0 \ 0 \ 4 \ 3 \ 4 \ 1]$ de manera ascendente de modo que $Aord = [0 \ 0 \ 1 \ 3 \ 4 \ 4 \ 9]$. Comencemos con un primer algoritmo naíf. En primer término comparamos el primer elemento $A[0]$ con cada uno del resto del arreglo. Si $A[0]$ es mayor que el elemento j

que se compara, entonces se intercambia $A[0]$ con el elemento $A[j]$. En el ejemplo (en negrita los valores que se están comparando):

$[9\ 0\ 0\ 8\ 3\ 4\ 1] \rightarrow [0\ 9\ 0\ 8\ 3\ 4\ 1] \rightarrow$ compara e intercambia $A[0]$ con $A[1]$

Así los siguientes pasos serían:

$[0\ 0\ 9\ 8\ 3\ 4\ 1]$

$[0\ 0\ 8\ 9\ 3\ 4\ 1]$

$[0\ 0\ 8\ 3\ 9\ 4\ 1]$

$[0\ 0\ 8\ 3\ 4\ 9\ 1]$

$[0\ 0\ 8\ 3\ 4\ 1\ 9]$

De esta forma, al cabo de 6 (n-1) comparaciones y de 6 (n-1) intercambios el primer valor está ubicado correctamente. Este es el peor de los casos ya que el número de intercambios es el mismo que de comparaciones. Ahora se debe repetir este procedimiento nuevamente por cada uno de los elementos restantes.

Podemos analizar la complejidad de este algoritmo según el arreglo original esté más o menos ordenado. Así podemos ver lo siguiente.

Mejor caso: el arreglo de entrada está ordenado.

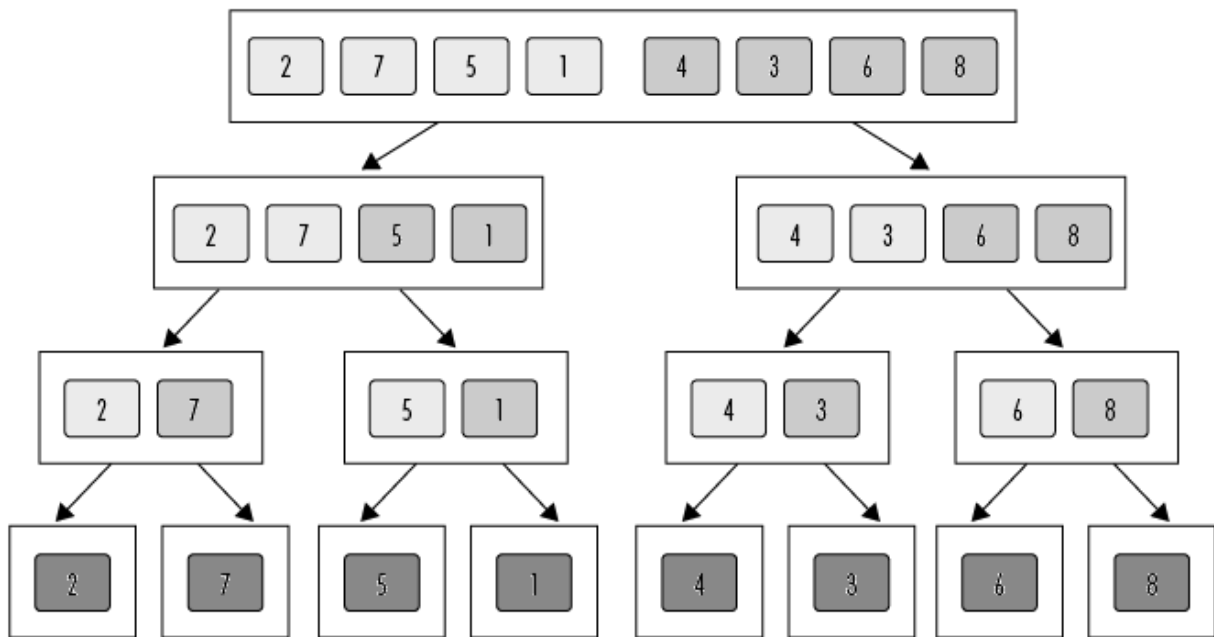
Peor caso: el arreglo está en el orden inverso.

En cualquier caso, el algoritmo requiere n comparaciones x n elementos del arreglo, lo cual nos lleva a un tiempo $T(n) \in O(n^2)$. Esto significa que el tiempo crece cuadráticamente a medida que el número de elementos crece.

Analicemos otro algoritmo de ordenación llamado Merge Sort que se basa en una técnica algorítmica muy conocida denominada “divide and conquer” (D&C). El algoritmo se compone de dos partes: a) se divide el arreglo desordenado en n subarreglos donde cada uno de ellos se compone por un único elemento y que por lo tanto se encuentra ordenado, y b) sucesivamente se van mezclando los arreglos ordenados obtenidos al finalizar la etapa a) para construir subarreglos de mayor número de elementos, también ordenados. El proceso se repite hasta mezclar todos los elementos en un único arreglo final completamente ordenado.

Para realizar la primera etapa a), se subdivide sucesivamente el arreglo en dos, como se puede observar en la figura 2.

Figura 2. Se muestran los diferentes niveles de divisiones sucesivas hasta llegar a tener n arreglos de un solo elemento ordenados



Siguiendo el ejemplo de la figura 2, el arreglo tiene ocho elementos. Se puede ver claramente que se realizó $\log_2^{(8)}$ veces el proceso de división (tres niveles de división en dos) o más genéricamente hablando es del orden de $O(\log_2^{(n)})$ donde n es el número de elementos del arreglo. Siendo este último el costo algorítmico de esta parte del ordenamiento.

La segunda etapa (b) del algoritmo consiste en la combinación de arreglos ordenados. En primera instancia se mezclan los arreglos que tienen un solo elemento. Para ello se compara resolviendo de izquierda a derecha y de abajo hacia arriba (partiendo de los subarreglos ordenados de un solo elemento), como se muestra en los siguientes pasos:

Figura 3. Se combinan los arreglos de un elemento, para formar un nuevo subarreglo ordenado [2,7]

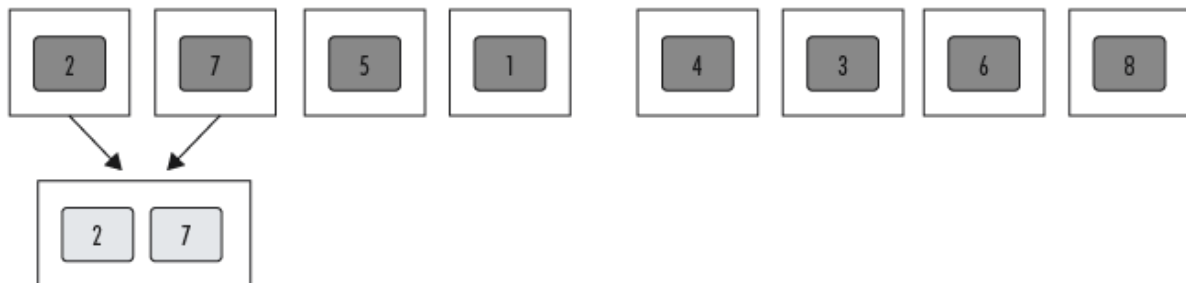
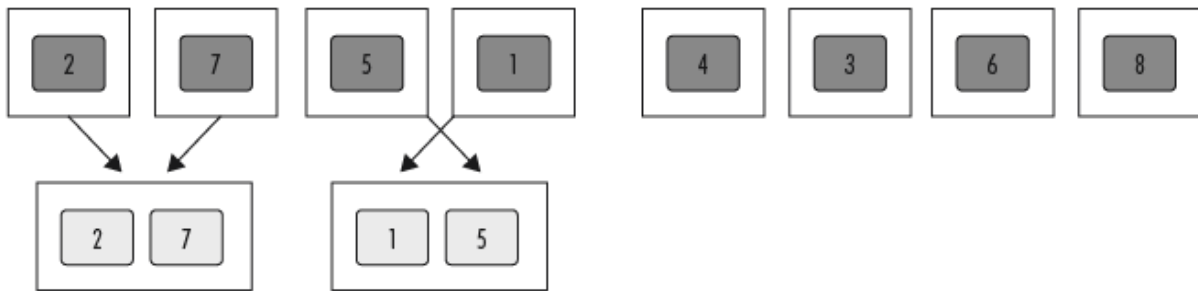


Figura 4. Se repite el proceso en este nivel y en la rama izquierda, se obtienen al final dos subarreglos ordenados [2,7] y [1,5]



A continuación se sube de nivel (completando la rama izquierda). Para ello: se compara el 2 con el 1, y se mezcla el 1 en el arreglo resultante. Luego se compara el 2 con el 5 y por ser menor se combina el 2. Quedan entonces para comparar 1 combinar el 5 y el 7 que se combinan ordenadamente al final como sigue en la figura 5:

Figura 5. Los subarreglos ordenados de dos elementos de la rama izquierda se combinan ordenadamente para completar el siguiente nivel de la rama derecha

Ahora es necesario resolver la recursión del lado derecho como se muestra en la figura 6:

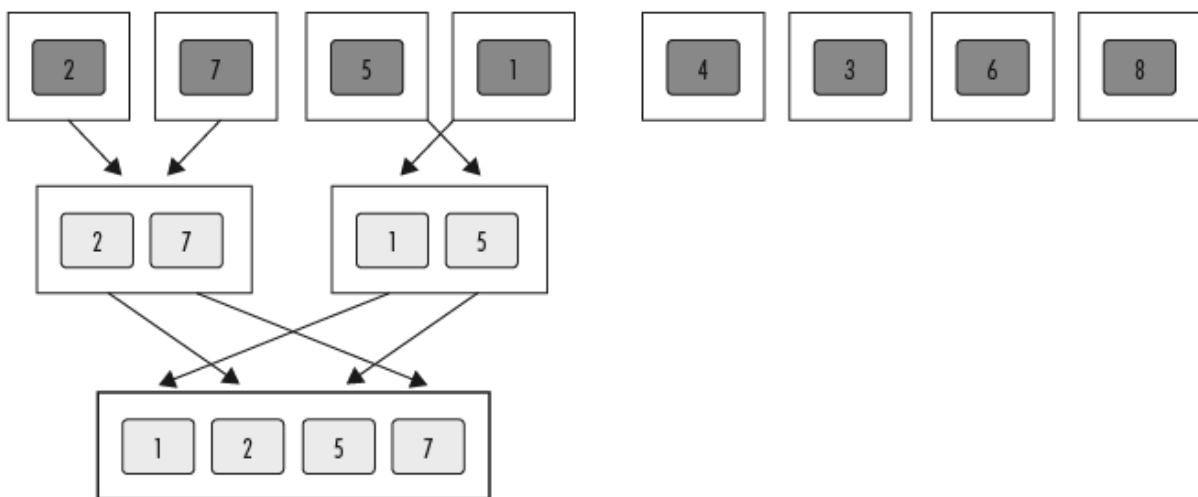


Figura 6. Para poder combinar un nivel más alto es necesario primero resolver la rama derecha y por lo tanto los subarreglos ordenados [4] y [3] se combinan en el subarreglo [3,4]

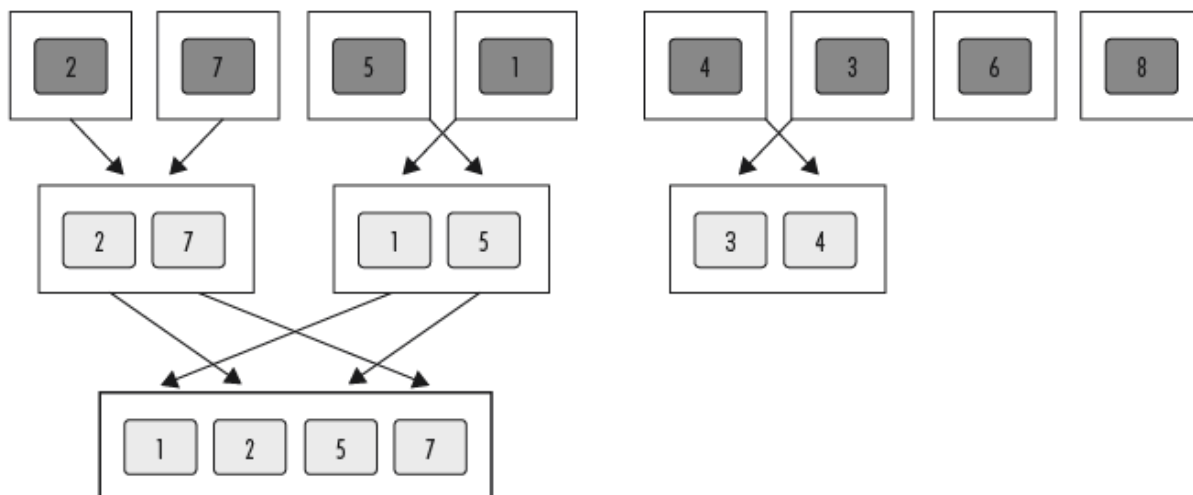


Figura 7. Se repite el trabajo con los últimos elementos del nivel más bajo (subarreglos ordenados con un solo elemento)

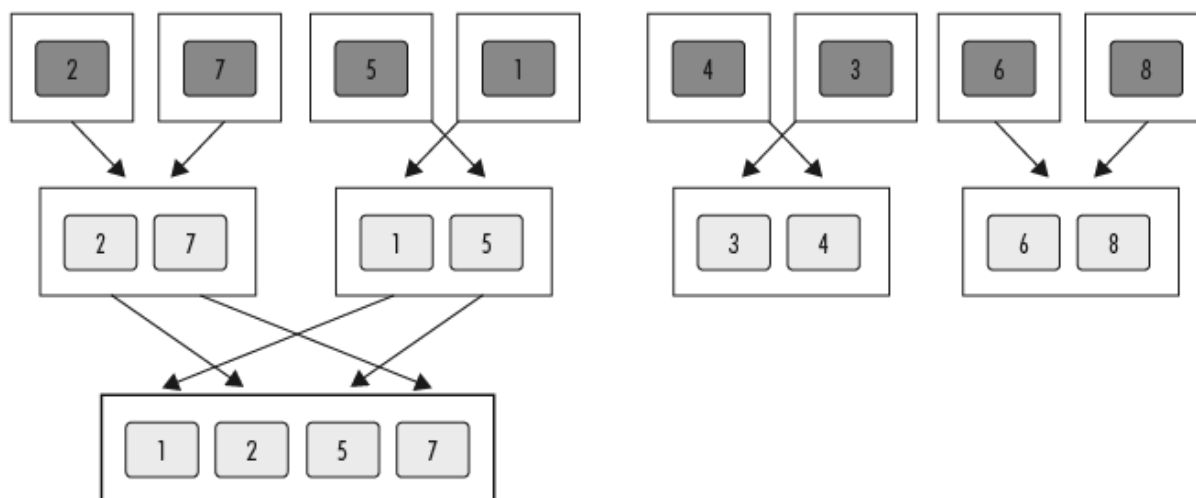


Figura 8. En esta etapa ya es posible completar la rama derecha del segundo nivel en un subarreglo ordenado de 4 elementos [3,4,6,8]

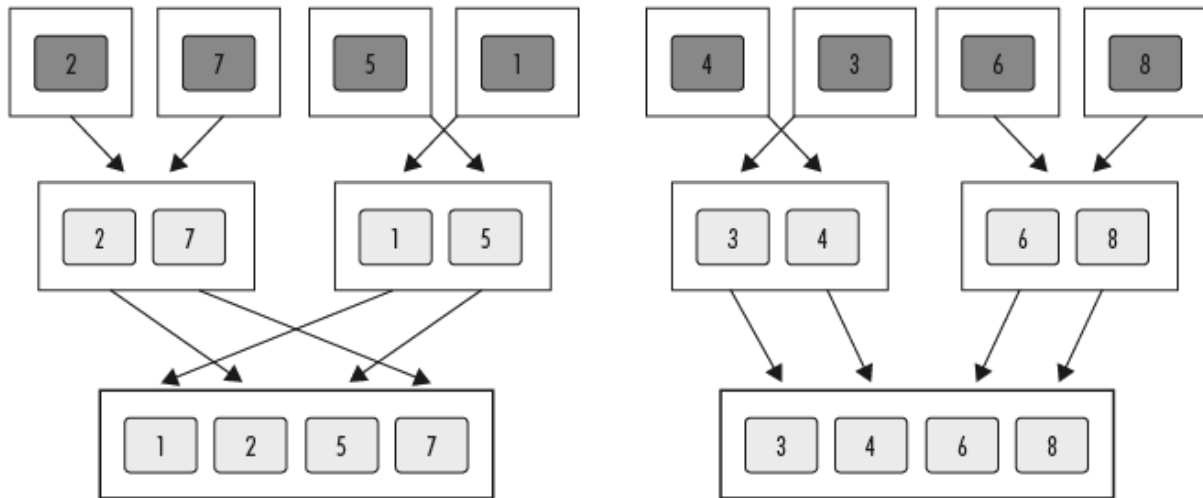
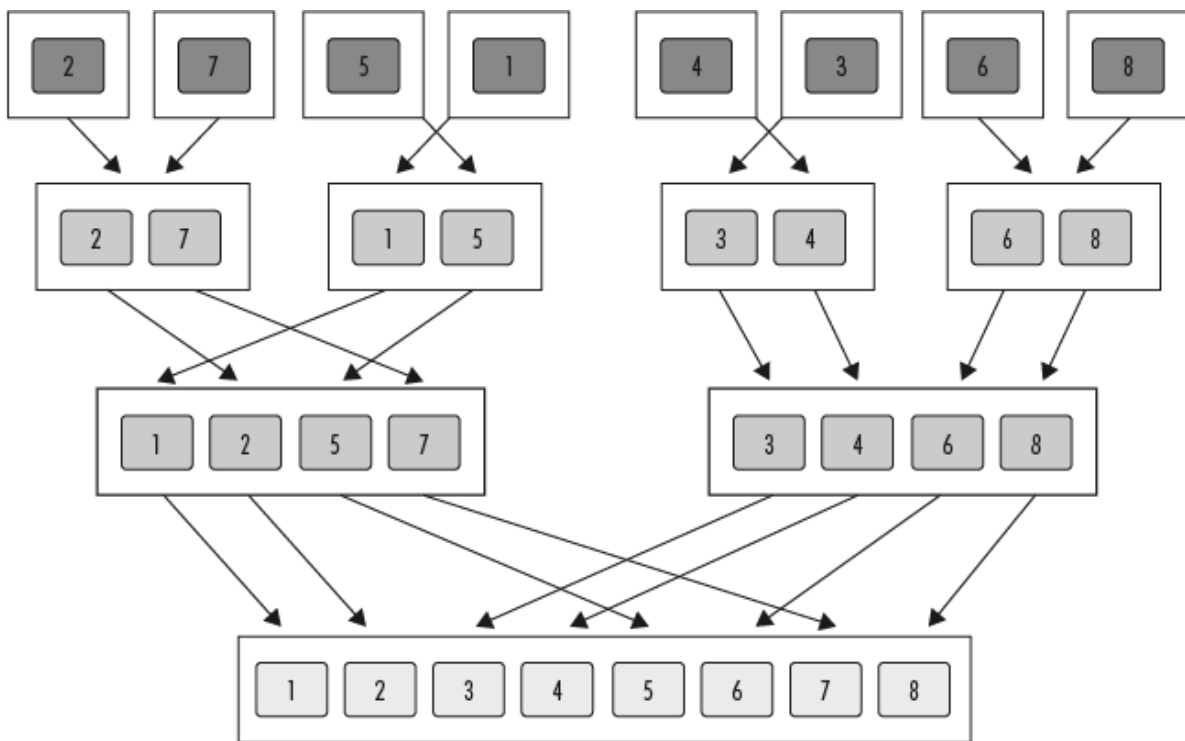


Figura 9. En esta etapa ya se resolvieron las recursiones izquierda y derecha y solo resta combinar los subarreglos resultantes en un único arreglo final completamente ordenado



Por cada nivel se puede observar que se requieren n combinaciones, donde n es la cantidad de elementos de cada nivel. Es decir que el costo total del algoritmo es de $O(n \log_2^{(n)})$ aun cuando el arreglo inicial se encuentre completamente desordenado (pero caso).

Si comparamos con el desempeño del primer algoritmo naïf, el costo bajó de $O(n^2)$ a $O(n \log_2 n)$.

Existen numerosos algoritmos de ordenación y que se adaptan a diferentes tipos de problemas. Por ejemplo, si sabemos que el problema de aplicación puede resultar en un arreglo parcialmente ordenado es importante seleccionar algún algoritmo sensible a este ordenamiento inicial y que no realice movimientos innecesarios. Para ahondar en los diferentes métodos de ordenación, es recomendable consultar libros sobre algoritmia o incluso diferentes recursos de la web donde incluso se muestran animaciones para diferentes casos de estudio.

En tabla 1 se muestra un resumen comparativo entre algunos métodos de ordenación, se ponen en relevancia los casos de arreglos completamente desordenados (peor caso), arreglos parcialmente ordenados (caso promedio) o arreglos ordenados (mejor caso).

Tabla 1. Comparación entre los costos de ejecución de algunos de los algoritmos de ordenación más conocidos. La escala de colores muestra desde los peores costos a los mejores

Algoritmo	Mejor caso	Caso promedio	Peor caso
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Timsort	$O(n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Tree sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Shell sort	$O(n \log n)$	$O(n^2)$	$O(n^2)$
Bucket sort	$O(n+k)$	$O(n+k)$	$O(n^2)$
Radix sort	$O(nk)$	$O(nk)$	$O(nk)$
Counting sort	$O(n+k)$	$O(n+k)$	$O(n+k)$
Cube sort	$O(n)$	$O(n \log n)$	$O(n \log n)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Muy bueno
Bueno
Razonable
Malo
Muy malo

También es importante señalar que, en la mayoría de los casos, encontrar algoritmos eficientes requiere de una inversión sustancialmente mayor por parte del programador en la etapa de diseño algorítmico; esta es una variable más al momento de balancear el rendimiento del software.

En algunos casos, invertir tiempo en el diseño del algoritmo puede resultar crítico ya que las versiones simples suelen ser tan costosas que pueden resultar inviables en la práctica.

Un ejemplo muy destacado de cómo la elección de un algoritmo más eficiente puede cambiar la potencialidad de un método son los algoritmos para el cálculo de la transformada discreta de Fourier (TDF). La implementación tradicional de este método (que es útil en muchísimas aplicaciones en las ciencias) tiene, para el cálculo de coeficientes, un costo computacional de $O(n^2)$. En 1965, J. W. Cooley y J. W. Tukey presentan el famoso algoritmo rápido de la transformada de Fourier o Fast Fourier Transform (FFT) con un costo de $O(n \log n)$ usando la técnica D&C. Si se

tienen $n=1.000$ elementos, el costo de TDF tradicional es de 10^6 mientras que el costo de la FFT sería de 10^4 . Hasta aquí ambos algoritmos parecen razonables pero cuando el problema escala, por ejemplo $n=10^9$, entonces el costo de TDF tradicional sería 10^{18} y FFT 30×10^9 . Para tener una idea de cómo impactó esta mejora en el costo del algoritmo, supongamos que cada operación que se realiza demora 1 ns. Entonces, evaluando estos costos en tiempo: 10^{18} ns equivalen a 31.2 años (tiempo totalmente prohibitivo), mientras que 30×10^9 ns equivalen aproximadamente a 30 s.

Diferentes técnicas algorítmicas y estudios más formales sobre el cálculo de costos para algoritmos se suelen enseñar en cursos de algoritmia en las carreras de ciencias de la computación.

Cómo elegir una estructura de datos apropiada

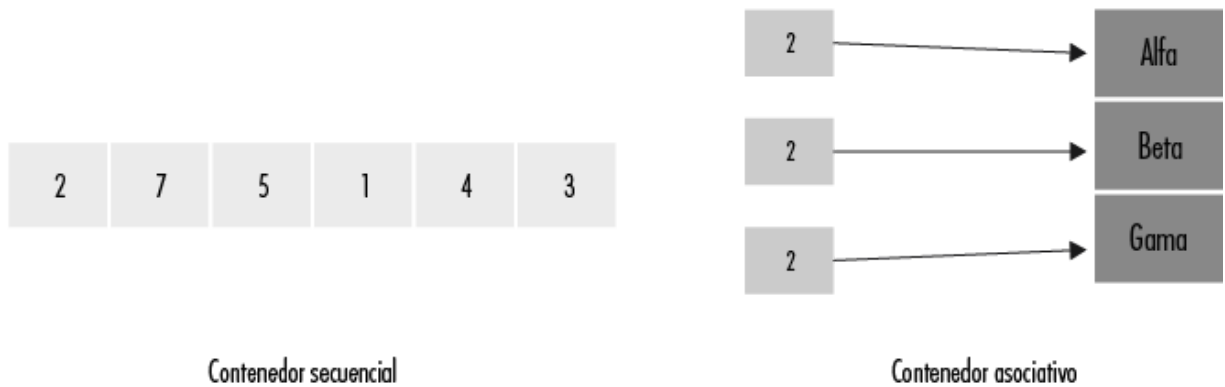
Las estructuras de datos abstractas nacen como una manera de almacenar información de una manera organizada y estructurada. En contraste con los datos simples, permiten ver la información como una colección de datos que se caracterizan por cómo están organizados y qué operaciones se pueden realizar sobre ellos.

El objetivo de crear tipos o estructuras de datos abstractos es la de poder tener un fácil y eficiente acceso y manejo de los datos.

Existen contenedores de datos secuenciales y asociativos. Los primeros se tratan de datos organizados de manera consecutiva, mientras que los contenedores asociativos hacen referencia a datos que no necesitan situarse de manera consecutiva sino que se localizan mediante una clave.

La figura 10 muestra de manera gráfica estos dos tipos de contenedores.

Figura 10. Organización de datos en contenedores secuenciales y asociativos



Algunas de las estructuras de datos más usadas son:

- *Arreglos*: compuestos por elementos almacenados de manera contigua. Pueden ser implementados tanto en contenedores secuenciales como asociativos. En el primer caso se trata de datos almacenados de manera contigua en memoria y con ello se establece el orden del arreglo. En el segundo caso, la clave resulta indicar el orden de aparición de los elementos en el arreglo, funcionando como un índice de los elementos.
- *Listas enlazadas*: son estructuras de datos compuestas por nodos que se encuentran enlazados indicando el orden que tienen en la mencionada lista. Los nodos están compuestos de manera genérica por una parte correspondiente al dato y otra correspondiente al enlace con los otros elementos. Cada nodo puede estar enlazado con su nodo siguiente (listas enlazadas simples) o pueden estar enlazados tanto con el nodo anterior como con el siguiente (listas enlazadas dobles). Las listas enlazadas corresponden a contenedores secuenciales.
- *Cola, pila o LIFO (last in first out)*: corresponden a contenedores secuenciales con operaciones especiales para el acceso a los datos. Los elementos o datos que ingresan primero a la pila son los últimos en poder ser accedidos. Tienen operaciones especiales para el acceso y manipulación de los datos. Las dos operaciones más importantes y características son las operaciones denominadas *push* y *pop*. La operación *push* equivale a insertar un elemento a la pila al final de esta. El último elemento agregado o “apilado” es el elemento

- accesible. La operación pop elimina un elemento de la pila, solo es posible eliminar el último elemento ingresado (*last in first out*).
- *Fila o FIFO (first in first out)*: corresponde a una estructura de datos de elementos secuenciales con la característica de que los nuevos elementos o datos se van agregando al final. De este modo se tiene una fila (como la fila de un banco) donde el primer elemento que ingresó es el que puede ser accedido (atendido). Las operaciones básicas para el manejo de elementos en una fila son la operación enfila o enqueue (agregar un elemento a la fila) y la operación desenfila o dequeue (eliminar el primer elemento en la fila).
 - *Mapas ordenados o hash maps*: es una estructura de datos que almacena los elementos de manera asociativa, para ello existe una clave (única) que mediante una función (función *hash*) permite indexar la posición del valor asociado a la clave.

En la figura 11 se muestran de manera gráfica algunas de las estructuras de datos más usadas. Existen muchas otras estructuras de datos como diccionarios, conjuntos, árboles, etcétera.

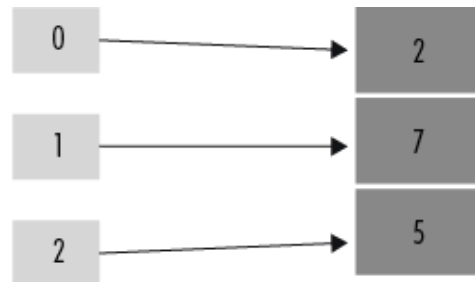
Al momento de elegir una estructura de datos para nuestro programa, es importante cómo esta elección puede impactar en el desempeño general del software. Podemos ver un ejemplo para entender qué tan importante es elegir una estructura de datos adecuada.

Supongamos que queremos programar un juego denominado “Follow Me” que tiene las siguientes reglas:

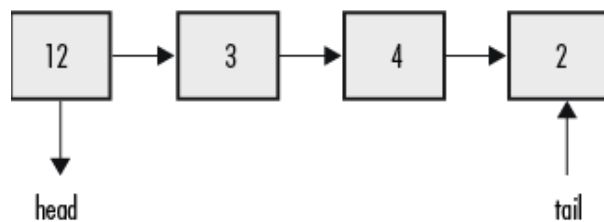
Figura 11. Descripción de las principales estructuras de datos



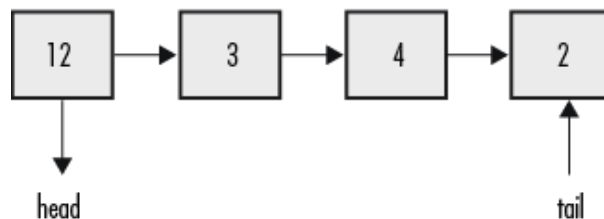
Arreglo (contenedor secuencial): los elementos se pueden acceder mediante $A[\text{índice}]$



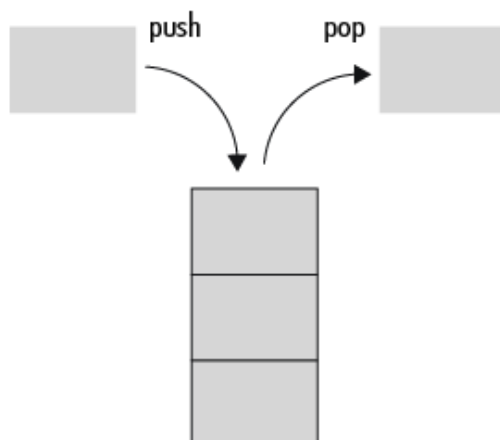
Arreglo (contenedor asociativo): los elementos se acceden mediante la clave (índice del elemento). En este ejemplo la clave 0 apunta al primer elemento del arreglo



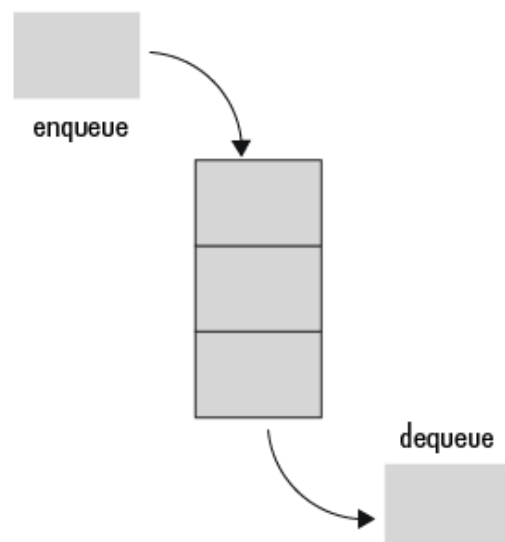
Lista simplemente anlazada (contenedor secuencial): Cada uno de los nodos contiene el dato y el enlace al próximo nodo. La manera de acceder a cada uno de los elementos es mediante un puntero al inicio de la lista (head) o un puntero al final (tail). No siempre es necesario o eficiente tener ambos punteros.



Lista doblemente anlazada (contenedor secuencial): La diferencia con la anterior es que cada nodo se enlaza con el nodo anterior y con el siguiente



Cola, LIFO o Pila: secuencia de elementos. Mediante push se insertan elementos al final y con la operación pop se elimina el último ingresado



Fila o FIFO: secuencia de elementos tal que los mismos ingresan después del último elemento (enqueue) y se elimina el primer elemento de la fila (dequeue)

- Se genera una secuencia aleatoria de N números que se van mostrando por pantalla uno a continuación del otro.
- Después de que se muestran todos los números generados, el jugador debe reproducir la secuencia.
- Cada coincidencia en el orden de aparición es un punto.
- Gana el jugador con mayor puntaje.

El algoritmo debe realizar las siguientes tareas:

- Generar, almacenar y mostrar por pantalla una secuencia.
- Capturar y almacenar la secuencia ingresada por el jugador.
- Comparar las dos secuencias y establecer el puntaje.

Vamos a implementar la solución usando dos estructuras de datos diferentes.

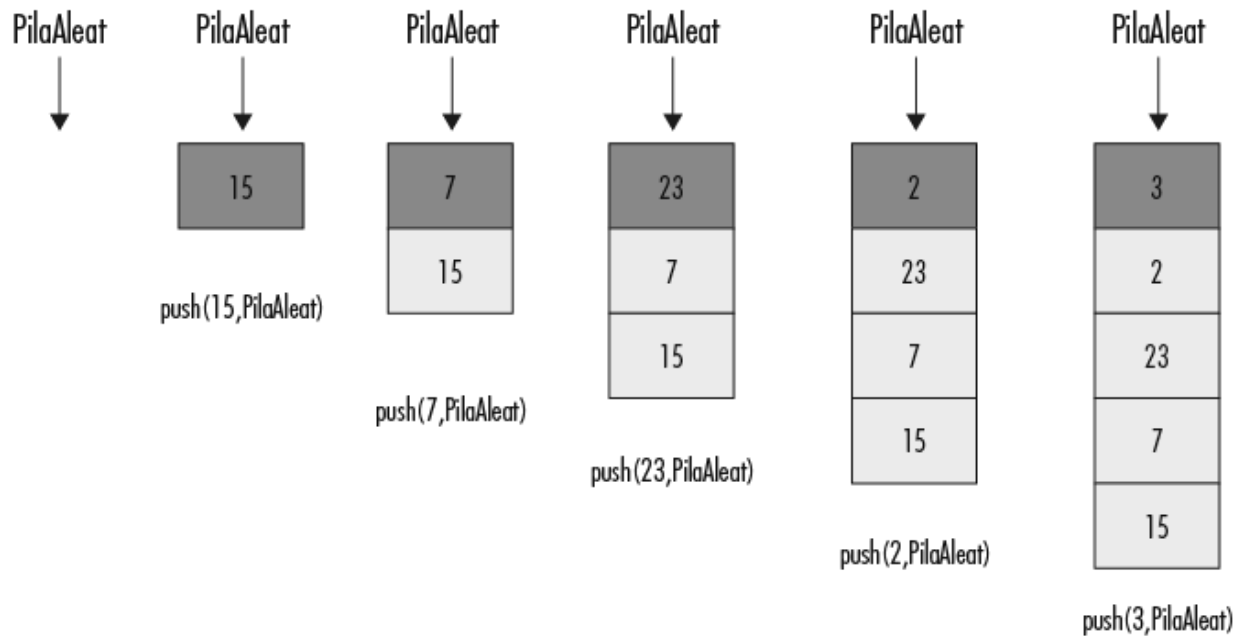
Primer enfoque: Usar una estructura pila (LIFO). Recordemos que la estructura de datos tiene las siguientes operaciones: *push* para agregar un elemento al final de la pila y *pop* para eliminar el último elemento agregado. A estas operaciones vamos a agregar dos más: *espilavacia*, que devuelve true si una pila no tiene elementos y false en caso contrario; y la operación *top*, que imprime el valor del elemento visible de la pila (que es el último ingresado).

Usando estas operaciones, podemos escribir una solución para el juego. Primero vamos a llenar la estructura de datos (una pila) con la secuencia de valores aleatorios:

```
WHILE not EndOfSequence
elemento=GENERAR_ALEAT()
push(elemento, PilaAleat)
MOSTRAR(elemento)
END
```

Supongamos $N=5$, y siguiendo el algoritmo después de terminar el lazo, la pila queda como se muestra en la figura 12.

Figura 12. Construcción de la pila PilaAleat. Iterativamente se van agregando elementos al final de la pila



En este ejemplo, el orden de aparición aleatoria es 15, 7, 23, 2, y finalmente el 3. Por la manera de acceso a la estructura solo es visible o accesible el último elemento, en este caso el 3. Este último elemento es el único que se puede mostrar usando la operación `top(PilaAleat)`.

Hasta aquí solo resolvimos la generación y almacenaje de la secuencia de valores aleatoria.

Veamos cómo podemos hacer para almacenar la respuesta del jugador en otra variable con la misma estructura (`PilaJugador`).

```

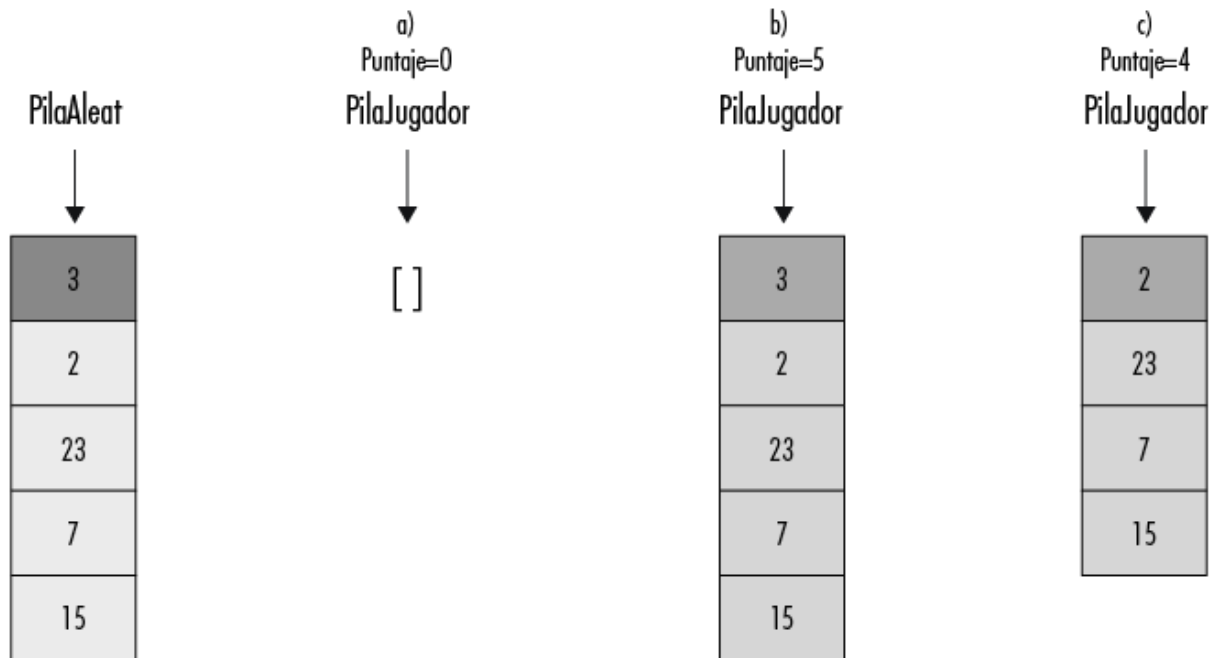
WHILE not EndOfSequence
elemento=LEER()
push(elemento, PilaJugador)
MOSTRAR (elemento)
END

```

Esta porción de código tiene también un costo de $O(n)$.

Supongamos cuatro opciones de respuestas del jugador (figura 13).

Figura 13. Puntajes resultantes ante tres posibles respuestas del jugador



En este punto, la clave en el análisis está en la regla que indica que un punto es asignado si coinciden los valores aleatorios con los del jugador en el orden de aparición. Para poder asignar el puntaje ahora hay que comparar las dos pilas.

En el caso a) es necesario chequear que la PilaAleat tiene valores y la PilaJugador es vacía. El caso b) corresponde al mejor resultado posible porque son dos pilas iguales y se le asigna el puntaje máximo. Pero ¿qué pasa en el caso c)? Claramente las pilas no son iguales pero el puntaje se asigna comparando los valores en el orden de aparición. Es decir, habría que invertir el orden de los elementos (invertir las pilas), lo que implica recorrer todos los valores.

Segundo enfoque: Usar una estructura fila (FIFO). Las operaciones enfila (agregar un elemento al final) y defila (elimina el elemento al principio de la fila) son las principales para el manejo de esta estructura de datos.

Repitamos el trabajo con los mismos ejemplos pero ahora usando filas.

Para generar la secuencia y mostrar por pantalla ahora:

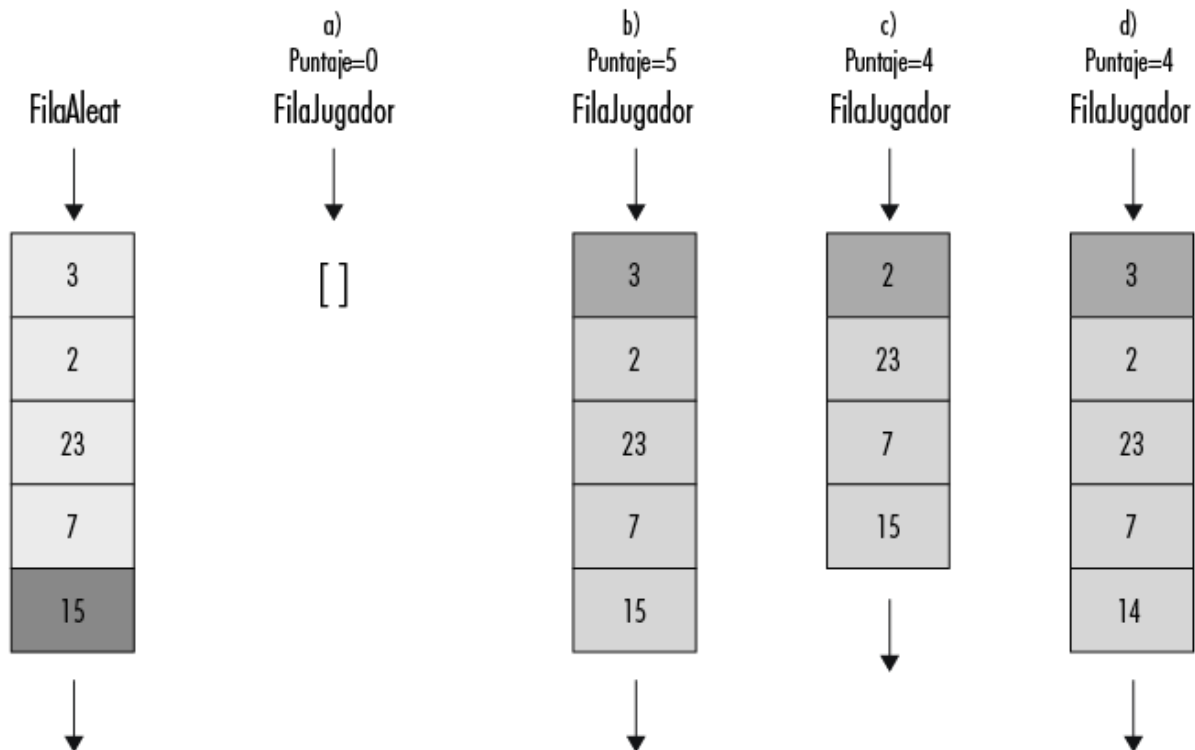
```
WHILE not EndOfSequence
elemento=GENERAR_ALEAT()
enfila(elemento, FilaAleat)
MOSTRAR(elemento)
END
```

Y para almacenar las respuestas del jugador:

```
WHILE not EndOfSequence  
elemento=LEER()  
enfila(elemento, FilaJugador)  
MOSTRAR (elemento)  
END
```

Al finalizar esta parte del algoritmo, y usando los mismos casos de respuestas podemos tener lo que se ve en la figura 14.

Figura 14. El primer valor de la secuencia aleatoria está directamente accesible en FilaAleat (marcado en color más oscuro); lo mismo ocurre en los diferentes casos para FilaJugador. Marcados en celeste se encuentran los elementos accesibles al FilaJugador en cada caso.



Claramente la comparación entre las secuencias y calcular el puntaje final es mucho más directa y eficiente. Solamente hay que ir aplicando una comparación entre el elemento accesible (el primero que ingresó a cada fila) y la operación defila de manera simultánea. El caso a) se resuelve mediante una nueva operación esfilavacia (que devuelve true en caso de no tener elementos la fila, o false en caso contrario). Para realizar la comparación

entre los elementos se utiliza la operación frente que simplemente muestra el valor del elemento visible (el primer elemento que llegó o frente de la fila).

Este es un ejercicio práctico para entender cómo puede cambiar sustancialmente una solución de acuerdo a la estructura de datos que se elija. Se deja como ejercicio para el lector la implementación completa de cada una de estas soluciones.

Optimización de hardware

En líneas generales, cuando pensamos en optimización (especialmente para el desarrollo software científico) pensamos en cómo hacer más rápido nuestros cálculos. Para esto podemos pensar que:


- Puedo realizar las operaciones de escritura y lectura de manera más rápida. Por ejemplo limitando el acceso a memoria no-uniforme y multi-core. La memoria no-uniforme es aquella que sigue un esquema de memoria distribuida. Se puede pensar como un grupo de nodos de cómputo, cada uno de ellos con una memoria local, interconectados y que permite que cada nodo puede acceder a la memoria del resto de nodos. Esta arquitectura por ejemplo es la que se encuentran en los sistemas multi-core, en clusters, etc. El acceso a memoria no-uniforme es eficiente si logramos acceder a la porción local de la misma, pero cambia mucho el desempeño del programa si debe recurrir (operaciones de entrada/salida) a espacios de memoria que se encuentran en otro nodo de cómputo.
- Podemos acelerar los cálculos si podemos lograr que en un ciclo de reloj de la CPU se ejecuten un mayor número de instrucciones. Esto se puede lograr con arquitecturas multi-core y con paralelización. En el capítulo VI se hace una revisión sobre las diferentes arquitecturas actuales y sobre programación en entornos paralelos.
- Aprovechar la tecnología subyacente (más específicamente la CPU). En este caso es necesario conocer las capacidades de los compiladores para aprovechar la arquitectura del computador, por ejemplo aprovechando unidades vectoriales, superescalares, etc. La mayoría de los compiladores tienen la posibilidad de activar diferentes flags de optimización.

- El costo en tiempo de cómputo de las operaciones aritméticas no es igual para cada una de ellas y puede cambiar significativamente. Las operaciones menos costosas son las correspondientes a suma, diferencia y multiplicación. Mientras que hay operaciones como, por ejemplo, la potencia de números reales $\text{pow}(x,y)$, x e y reales es sumamente costosa y hay que evitarlas en lo posible. Por ejemplo, $\text{pow}(x,2)$ es equivalente a $x*x$, siendo la primera opción hasta 100 veces más costosa que la segunda opción. Ciertamente $\text{pow}(x,2)$ puede resultar un poco más elegante al momento de programar pero, si hacemos foco en la eficiencia, no es la mejor opción. La tabla 2 muestra un comparativo en velocidades entre algunas de las operaciones más usadas.

Tabla 2. Comparación en velocidad de diferentes operaciones aritméticas

Operaciones	Costo
$+ - *$	0.5 a 1x
$/ \% \text{sqrt} ()$	5x a 10x
Funciones trascendentes	20x a 50x
$\text{pow}(x,y)$ x e y reales	+ 100x

Más veloces,
menos costosas



Más lentas,
más costosas

Las operaciones simples permiten aprovechar características del CPU como el *pipelining* (que se discute más adelante en este mismo capítulo). También es importante recalcar una vez más el uso de librerías muy optimizadas para cálculos, por ejemplo, de álgebra lineal.

Técnicas de optimización del compilador

La mayoría de los compiladores modernos poseen una serie de optimizaciones que permiten hacer un uso más eficiente del hardware disponible del computador donde se corre el programa. En muchos casos, estas optimizaciones están disponibles de manera automática o por defecto, pero en otros casos es necesario activar parámetros de optimización de manera explícita.

La tabla 3 resume las principales optimizaciones que realizan los compiladores respecto del hardware.

Tabla 3. Principales optimizaciones que realizan los compiladores: optimización escalar, sobre lazos y sobre funciones

Escalar	Lazos	Funciones
<ul style="list-style-type: none"> - Copy propagation - Const folding - Strength reduction - Eliminación de subexpresiones comunes - Renombrado de variables 	<ul style="list-style-type: none"> - Loop invariante - Loop unrolling - Intercambio de orden - Fusión/fisión 	<ul style="list-style-type: none"> - Inlining

Copy propagation: El compilador elimina las dependencias en caso de ser posible. La siguiente porción de código muestra un ejemplo:

```
x = y
z = 1+x
```

La variable z depende de que se realice antes la asignación x=y. El compilador es capaz de detectar esta dependencia y cambiar el código de una manera más eficiente sin cambiar el resultado final de las operaciones (para aprovechar el paralelismo a nivel de instrucción por ejemplo).

El código cambiado quedaría como:

```
x = y
z = 1+y
```

De esta manera, se elimina la dependencia permitiendo por ejemplo una ejecución en paralelo.

Const folding: En algunos casos, y cuando hay cálculos basados únicamente en constantes, el compilador pre-calcula el resultado una única vez en tiempo de compilación. De este modo se elimina el código redundante. Por ejemplo, la porción de código,


```
a = 100  
b = 200  
sum = a+b
```

es reemplazado simplemente por:

```
sum = 300
```

Desde el punto de vista del programador, la primera versión no optimizada es más clara y puede tener algún tipo de semántica particular para el problema. Pero a nivel del compilador, es preferible la segunda manera con el valor pre-calculado. Esta optimización suele ser automática sin necesidad de la intervención del programador.

Strength reduction: En casos donde se usan operaciones costosas como $\text{pow}(x,y)$ o $/$, el compilador puede analizar el cálculo y reemplazar estas operaciones con otras equivalentes pero de menor costo. Por ejemplo:

```
x = pow(y, 2.0)  
a = b/2.0
```

En la primera línea se calcula y elevado a 2, pero la función pow para números reales es muy costosa y el compilador puede automáticamente detectar que hay una manera más eficiente y reemplaza esta línea por:

```
x = y*y
```

Respecto a la división, el compilador es capaz de detectar que el denominador es una constante y entonces realiza el siguiente reemplazo (ya que el producto es una operación menos costosa):

```
a = b * 0.5
```

Eliminación de subexpresiones comunes: La idea es eliminar aquellos cálculos que aparezcan más de una vez (subexpresión) cuando sea posible. Esto sucede solo si la subexpresión es un cálculo costoso o si resulta en la reducción de registros de memoria a utilizar. En el siguiente cálculo, la subexpresión que se puede eliminar es la $/$, que está duplicada:

```
d = c*(a/b)  
e = (a/b)*2.0
```

Luego del reemplazo el código quedaría como:

```
x = a/b  
d = c*x  
e = x*2.0
```

Loop invariante: hace referencia a cálculos dentro de lazos que no cambian y por lo tanto no es necesario que se ejecute iterativamente. En estos casos, el compilador detecta estos cálculos invariantes en el loop y los resuelve afuera del lazo.

En el siguiente ejemplo:

```
DO i = 1,n  
  a(i) = b(i)+c*d  
  e = g(n)  
END DO
```

El cálculo $c*d$ y la llamada a la función $e = g(n)$ son invariantes (no dependen de i), y por lo tanto el compilador es capaz de optimizar este código como sigue:

```
temp = c*d  
DO i = 1,n  
  a(i) = b(i)+temp  
END DO  
e = g(n)
```

Loop unrolling: Está asociado a la capacidad de ejecutar cuentas en paralelo a nivel de instrucción. El compilador evalúa si es posible desdoblar o hacer unroll del lazo para que la cantidad de iteraciones sea menor y para que algunas operaciones se ejecuten en paralelo. Así el siguiente ejemplo,

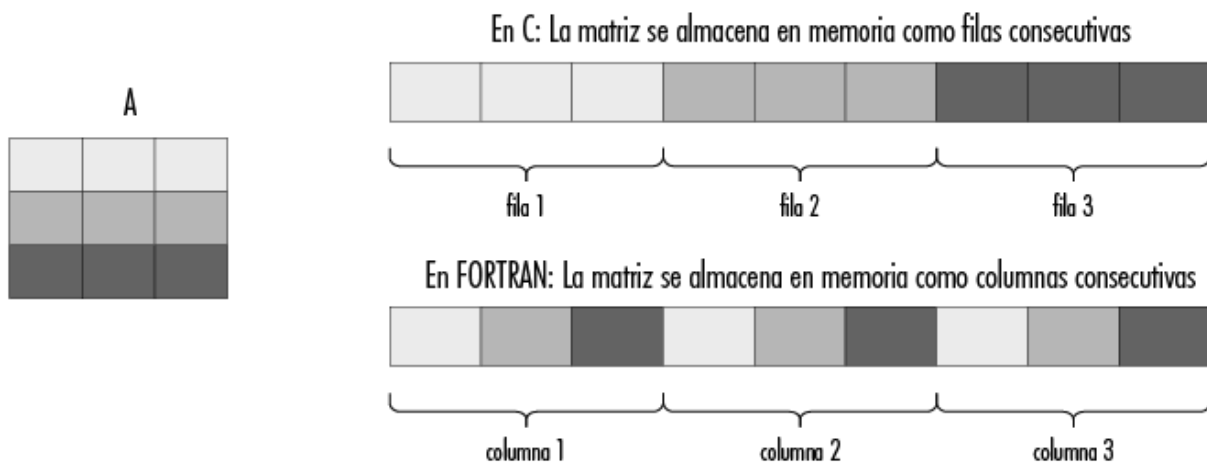
```
DO i = 1,n  
  a(i) = a(i)+b(i)  
END DO
```

Podría ser reemplazado por,

```
DO i = 1,n,4  
  a(i) = a(i)+b(i)  
  a(i+1) = a(i+1)+b(i+1)  
  a(i+2) = a(i+2)+b(i+2)  
  a(i+3) = a(i+3)+b(i+3)  
END DO
```

Intercambio en el orden de loops: Esta optimización está muy asociada a la manera en que se almacenan en memoria los datos. Supongamos que tenemos una matriz A de tamaño $n \times n$. Por ser un arreglo, los datos se almacenan consecutivamente, dependiendo de los lenguajes y de cómo estos acceden a memoria, la matriz puede ser almacenada de dos maneras diferentes (figura 15).

Figura 15. Entender cómo se almacena una matriz en memoria, para diferentes lenguajes, es importante al momento de programar. En este esquema se muestra cómo se almacenan los elementos de una matriz en C y en Fortran



Veamos dos ejemplos de código:

A)

```
DO i=1,n
    DO j=1,
        a(i,j)=b(i,j)
    END DO
END DO
```

B)

```
DO j=1,n
    DO i=1,n
        a(i,j)=b(i,j)
    END DO
END DO
```

El primer código A) accede primero a todos los elementos de una fila de la matriz (loop que se mueve sobre j) para luego cambiar a la siguiente fila (loop que se mueve en i). Mientras que el código en B) accede a los elementos por columnas.

Se puede ver que el código A) es más eficiente si el programa está escrito en C mientras que el código B) es más eficiente en el caso de Fortran.

Existen compiladores como gcc que permiten agregar una opción de optimización para loop unrolling. En tal caso, el compilador es capaz de ejecutar los lazos anidados de la forma más conveniente.

Fusión/Fisión de loops: En términos de optimización, pueden ocurrir casos donde sea más eficiente que los cálculos que se realizan dentro de un loop se fusionen en más de un lazo. O por el contrario, en otros casos puede ser más eficiente fusionar múltiples cálculos en un mismo lazo. La evaluación de cuál es la mejor manera la realiza el compilador.

Así, un caso de fisión sería:

```
DO i =1,n
    a(i) = b(i)+1
END DO
DO i =1,n
    b(i) =b(i)/2
END DO
```

```
DO i =1,n
    c(i) = 1/c(i)
END DOO
```

Mientras que la versión fusionada del mismo ejemplo sería:

```
DO i =1,n
    a(i) = b(i)+1
    b(i) =b(i)/2
    c(i) = 1/c(i)
END DO
```

Inlining: Refiere a la posibilidad de eliminar el overhead que involucra el llamado a una función. Si la función corresponde a un cálculo simple, el llamado a la función es reemplazado por el cálculo en línea. En el siguiente ejemplo:

```
REAL FUNCTION f(x)
      f = x*3
END
...
DO i = 1,n
      a(i) = f(i)
END DO
```

La función $f(x)$ es muy simple y realizar el llamado es mucho más costoso que simplemente hacer el siguiente reemplazo:

```
DO i = 1,n
      a(i) = i*3
END DO
```

En general, el compilador tiene esta funcionalidad para altos niveles de optimización. Por ejemplo el compilador gcc tiene la opción de activar la flag -O 0, -O 1, -O 2 y -O 3 (la más agresiva), entre otras. Por defecto, el nivel de optimización con este compilador es 1, así que si se desea ejecutar el código sin ningún tipo de optimización se debe agregar explícitamente con -O 0.

Debugging y profiling

Debugging hace referencia al proceso de eliminar los errores de nuestro programa. Quitar errores es un trabajo difícil y no existen técnicas específicas para ello, más bien resulta ser una habilidad blanda (soft skill) que requiere de mucha práctica. Cuanto mejor sea el desarrollador, mejor sabrá debuggear.

Para quitar errores, es necesario antes saber que nuestro programa tiene errores, dónde están y cómo quitarlos.

Claro que también hay categorías de dificultad para los errores: fáciles, medios y difíciles.

Los errores fáciles son aquellos que identificamos de manera relativamente sencilla: sabemos que están y sabemos dónde están. Lo difícil puede ser quitarlos. Ejemplos de este tipo de errores ocurren por ejemplo cuando nuestro programa se rompe (segmentation fault o segfault), o cuando el programa se cuelga (lazos infinitos).

Aquellos errores o bugs de nivel medio de dificultad son los que notamos, como por ejemplo cuando el programa arroja resultados erróneos, o cuando arroja NaN o inf como resultado (generalmente asociados a floating point exceptions). En estos casos, nos damos cuenta de que están pero es más complejo saber dónde y cómo quitarlos.

Los errores difíciles lo son debido a que, en primera instancia, es difícil saber que están. Por ejemplo: el programa da resultados erróneos en pocos casos (hacer pruebas exhaustivas ayuda), solo aparecen en ciertos niveles de optimización, solo aparecen en condiciones extremas, etc. Si encontramos estos errores, no es trivial saber dónde ocurren, porque además suele ser difícil poder reproducirlos. Y por supuesto, quitar estos errores suele ser mucho más complejo.

Podemos clasificar en tres las opciones para debugging:

1) Imprimir mensajes a lo largo del programa para entender qué sucede. Por ejemplo:

```
printf("I'm here!\n");  
buggy_code();  
printf("Now I'm there!\n");  
...
```

Esta es la manera más simple de buscar un error pero no la más eficiente.

2) Usar un debugger: es una herramienta (programa) que nos permite realizar un monitoreo de nuestro programa. Generalmente corre sobre la ejecución del programa. Permite entre otras cosas realizar un monitoreo paso a paso de cada función, hacer un seguimiento de variables e incluso analizar cada sentencia de assembler. Al final, se puede realizar un análisis post-mortem.

3) Pedir ayuda: existen numerosas comunidades y foros donde se puede consultar. Un ejemplo de comunidad muy útil para consultar es stack overflow [véase bibliografía]. Es importante saber cómo pedir ayuda, y es

necesario ayudar a que nos ayuden. Es importante, antes de pedir ayuda a otra persona, que hayamos hecho el esfuerzo de resolver el problema, y que hayamos buscado en otros lados posibles soluciones. Si aun así no podemos resolver solos el problema, entonces pedir ayuda indicando cómo reproducir el problema, qué versión de lenguaje se usó, qué compilador, etcétera.

Un debugger obtiene información sobre cada objeto de nuestro programa. Para el caso del compilador gcc, para activar o “prender” el debugger es necesario compilar nuestro programa con la opción -g. En este caso, la recomendación es compilar con opción -O0 para que no se compile con ningún tipo de optimización. La razón es que, cuando hay algún tipo de optimización, el compilador puede modificar el código.

Una vez compilado con esta opción, es posible correr el programa usando el debugger. En el caso de GNU, el debugger se llama gdb y se utiliza de la siguiente manera.

En primera instancia, se ejecuta el programa con gdb (figura 16).

Figura 16. Ejemplo de ejecución de un programa con gdb

```
$ gdb ./my_program.  
...  
(gdb)
```

Una vez que estamos dentro del debugger, se pueden usar cualquiera de las siguientes sentencias que permiten monitorear cada paso de nuestro programa, seguir el contenido de una variable, detener la ejecución en un determinado punto, etcétera.

La figura 17 muestra el listado de comandos de gdb (se recomienda consultar el tutorial oficial de esta herramienta).

Figura 17. Listado de comandos de gdb

(gdb) break	<ident>	se detiene la ejecución si llega a ident
(gdb) watch	<ident>	se detiene la ejecución si ident cambia
(gdb) run		ejecuta el programa
(gdb) continue		continúa la ejecución

```
(gdb) next          siguiente paso (sin entrar a
subrutinas)
(gdb) step          siguiente paso (entrando en
subrutinas)
(gdb) print <ident> imprime el valor de ident
(gdb) delete <bpn>  borra el breakpoint
```

Otro debugger es Valgrind, que en realidad es una máquina virtual sobre la que corre nuestro programa; es muy exhaustivo permitiendo que podamos analizar absolutamente todo. Por ejemplo, permite encontrar casos de cache miss que permiten corregir el código para aprovechar mejor la memoria, entre muchas otras ventajas. Como se trata de una máquina virtual, su ejecución es sumamente lenta. Valgrind permite también hacer profiling.

Se puede correr Valgrind como se observa en la figura 18.

Figura 18. Ejemplo de uso de Valgrind para el programa my_program.e

```
$ valgrind -leak-check=full -track-origins=yes ./my_program.e
```

Profiling refiere a la acción de evaluar o medir cuán rápido es nuestro código. Hay diferentes maneras de hacer un profiling:

- Usar un cronómetro externo.
- Usar un cronómetro interno. Por ejemplo, usando el comando time del shell de Linux.
- Usar un timer dentro del programa. Por ejemplo, usando la función time de la librería time de Python, como se vio en un ejemplo anterior en este mismo capítulo:

```
import time
start = time.time()
n=12
print(n)
end = time.time()
print(end - start)
```

- Usar un profiler.

Un profiler es una herramienta (programa) que permite recopilar estadísticas respecto al rendimiento de un programa. A diferencia de los otros métodos de medición de tiempos, permite conocer los tiempos de diferentes secciones del código independientemente del contexto donde corre, e incluso pueden tener acceso a contadores internos de los registros del computador. De este modo, es posible reconocer cuáles son las secciones de código que insumen mayor tiempo de cómputo y así enfocar en estas secciones los esfuerzos de optimización. Esta herramienta puede incluso detectar bugs que antes no fueron encontrados, observando por ejemplo la cantidad de veces que una función es ejecutada o si una sección de código no se ejecuta de manera inesperada.

Gprof es un profiler de GNU que utiliza información que recolecta durante la ejecución de un programa y devuelve al finalizar las estadísticas sobre este.

Los pasos a seguir para realizar el profiling con gprof son los siguientes y en la figura 19 se puede ver en términos de las instrucciones:

- Compilar y linkear el programa preniendo la opción del profiler usando la opción -pg (no es recomendable usar optimización).
- Ejecutar el programa para generar el archivo con información del profiler (estadísticas que consiguió el profiler).
- Correr gprof y analizar los resultados obtenidos por el profiler.

Figura 19. Ejemplo de uso de gprof

```
$ gcc code.c -pg -o program.e
$ ls -tr
code.c program.e
$ ./program.e
$ ls -tr
code.c program.e gmount.out
$ gprof program.e gmon.out > profile.info
$ ls -tr
code.c program.e gmon.out profile.info
```

En primera instancia, se compila el programa code.c con la opción -pg (se prende el profiler), se ejecuta el programa normalmente y luego con el comando gprof se generan las estadísticas que en este ejemplo son enviadas al archivo profile.info.

Otro profiler es Perf, que tiene como ventaja adicional sobre gprof la capacidad de acceder a contadores internos en los registros de la CPU. Dado que Linux tiene acceso a estos registros, el overhead es muy poco. No es necesario compilar y linkear con una flag especial, el profiler puede extraer la información directamente de los registros en el momento de la ejecución del programa.

La figura 20 muestra un ejemplo usando el comando `perf stat` para presentar las estadísticas recopiladas por perf durante la ejecución del programa `program.e`.

Figura 20. Modo de uso del comando `perf stat`

```
$ gcc code.c -o program.e
$ perf stat ./program.e
performance counter stats for './calc.e'
.....
```

El comando `report` de perf permite coleccionar los datos de performance en un archivo de salida (`perf.data`) que luego puede ser analizado y mostrado por pantalla usando `perf report` (figura 21).

Figura 21. Ejemplo de uso de `perf record` y análisis post-mortem usando `perf report`

```
$ gcc code.c -o program.e
$ perf record ./program.e
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.0014 MB perf.data (~586 samples) ]
$ perf report -i perf.data
```

Conclusión

En este capítulo nos enfocamos en la optimización como una herramienta para mejorar la performance de nuestros programas, desde prestar atención y analizar los algoritmos que usamos, pensar en las estructuras de datos que se ajusten mejor a nuestros problemas, hasta las consideraciones de optimización respecto al hardware.

Además, presentamos herramientas para debugging y profiling que en conjunto permiten detectar y eliminar errores pero, por sobre todo, tener una

manera de evaluar o medir qué tan rápido es nuestro código. En conjunto, estas dos herramientas ayudan a optimizar nuestros programas.

Resumen

En general, al momento de escribir código científico, no es común hacer un estudio y/o diseño previo pensando en cómo puede escalar el problema o cómo podemos evaluar el desempeño de nuestra solución *a priori*. Tampoco se suele pensar, en una primera etapa, en las estructuras de datos abstractas que se usan y cómo estas pueden influir en los tiempos de ejecución. Menos aún solemos tener en cuenta cómo el compilador o nuestra manera de programar pueden hacer uso eficiente de nuestro hardware. Cabe en este análisis también la elección de herramientas que nos permitan medir qué tan rápida es nuestra solución, detectar y eliminar bugs (incluso aquellos que solo aparecen en casos extremos).

El proceso de optimización de un programa involucra todos estos aspectos y en este capítulo abordaremos cada uno de ellos.

Hands-on: debugging y profiling

En esta última sección del capítulo, proponemos la realización de algunos ejercicios que tienen por finalidad poner a prueba el uso de las herramientas de debugging y profiling mostradas en este capítulo. Para ello, se debe acceder al repositorio que se encuentra en <https://github.com/wtpc/HOdebug-profile>. Es posible clonar el repositorio o simplemente bajarlo en formato comprimido (zip). En el repositorio se pueden encontrar dos carpetas: debug y profile.

-*Debug*: carpeta que contiene los ejercicios que numeramos a continuación.

1. *Varios bugs*: consiste en varios errores ya resueltos para analizar. La carpeta contiene un makefile que compila todos los casos. Se recomienda editar el makefile, en particular la variable `$CFLAGS` para probar diferentes flags de compilación (por ejemplo, de optimización). Algunas cosas que se pueden hacer son:
 - Corramos el programa con un debugger, sin agregar flags de debug. ¿Se tiene toda la información que se requería?

- ¿Qué pasa si se pone el flag de debug? ¿Qué flag de optimización es la mejor para debuggear?
- Agreguemos algún flag para que informe todos los warnings en la compilación, como -wall. ¿Alguno da alguna pista de por qué el programa se rompe?

2. *Floating point exception*: En la carpeta fpe/ hay tres códigos de C, independientes, para compilar. Cada uno de estos códigos genera un ejecutable. Hay además una carpeta que define la función set_fpe_x87_sse_. Una vez compilados los tres ejecutables sin la opción -DTRAPFPE, proponemos responder las siguientes preguntas:

- ¿Qué función requiere agregar -DTRAPFPE? ¿Cómo se puede hacer que el programa linkee adecuadamente?
- Para cada uno de los ejecutables, ¿qué hace agregar la opción -DTRAPFPE al compilar? ¿En qué se diferencian los mensajes de salida de los programas con y sin esa opción cuando tratan de hacer una operación matemática prohibida, como dividir por 0 o calcular la raíz cuadrada de un número negativo?

Nota: Si se agrega -DTRAPFPE, el programa va a tratar de incluir, en la compilación, un archivo .h que está en la carpeta fpe_x87_sse. Para pedirle al compilador que busque archivos .h en una carpeta, se debe usar el flag -Icarpeta (sí, sin espacio en el medio).

Otra nota: Para poder linkear fpe_x87_sse.c se tiene que agregar la librería matemática libm, con -lm.

3. *Segmentation Fault*: En la carpeta sigsegv/ hay códigos de C y de Fortran con su Makefile. Compilemos y corramos small.e y big.e. Identifiquemos los errores que devuelven (¡si devuelven alguno!) los ejecutables. Ahora, ejecutemos ulimit -s unlimited en la terminal y volvamos a correrlo. Luego, respondamos las siguientes preguntas:

- ¿Devuelven el mismo error que antes?
- Averigüemos qué se hizo al ejecutar la sentencia ulimit -s unlimited. Algunas pistas son: abramos otra terminal distinta y fijémonos si vuelve al mismo error, veamos la diferencia entre ulimit -a antes y después de ejecutar ulimit -s unlimited, googleemos, etcétera.

- La “solución” anterior, ¿es una solución en el sentido de debugging?
 - ¿Cómo haríamos una solución más robusta que la anterior, que no requiera tocar los ulimits?
4. *Valgrind*: En la carpeta valgrind/ hay ejemplos en C y Fortran que se pueden ejecutar con valgrind. La propuesta es describir el error y por qué sucede.
5. *Funny*: En la carpeta funny/ hay un código de C. Proponemos describir las diferencias de los ejecutables al compilar con y sin el flag -DDEBUG. ¿De dónde vienen esas diferencias?

-*Profiling*: Este ejercicio es muy libre. Compilemos los códigos de C o Fortran con distintas opciones de optimización (-O0, -O1, -O3) y ejecutemos con distintas formas de hacer un profiling: desde `time ./programa.e` hasta `perf`, pasando por `gprof` y cualquier otra cosa que se nos ocurra. Escribamos un pequeño reporte en `reporte.md` respecto de qué cosas hicieron y cómo mejoraron/empeoraron los tiempos.

Bibliografía

Aho, A. V., M. Hill, J. E. Hopcroft, J. D. Ullman, *Data Structures and Algorithms*, Nueva York, Addison-Wesley, 1987.

Algoritmos de ordenación, <<https://www.toptal.com/developers/sorting-algorithms>>.

Compilador para C de GNU, <<https://gcc.gnu.org/>>.

Cooley, J. W., J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series”, *Mathematics of Computation*, vol. 19, N° 90, 1965.

Documentación debugger gdb, <<https://www.gnu.org/software/gdb/>>.

Documentación sobre opciones de optimización para GCC, <<https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/Optimize-Options.html>>.

Foro de consultas para programadores Stack Overflow, <<https://es.stackoverflow.com/>>.

Guía de estilo de Python PEP8, <<https://www.python.org/dev/peps/pep-0008/>>.

LAPACK: paquete de software de álgebra lineal,
<<http://www.netlib.org/lapack/>>.

Librería BLAS-LAPCK, <<http://www.netlib.org/lapack/>>.

Librería Boost para C++, <<http://www.boost.org/>>.

Librería científica y numérica GSL-GNU,
<<https://www.gnu.org/software/gsl/>>.

Librería Open MPI, <<https://www.open-mpi.org/>>.

Manuales de gprof, <https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_1.html>.

Valgrind, <<https://www.valgrind.org/>>.

Capítulo 6

Programación en paralelo

Cecilia Jarne

Arquitectura del computador y procesamiento de datos

Para poder comprender cómo utilizar los entornos de programación en paralelo es necesario comenzar por describir los fundamentos sobre arquitectura del computador. En ese caso, en particular describiremos el Modelo de J. von Neumann.

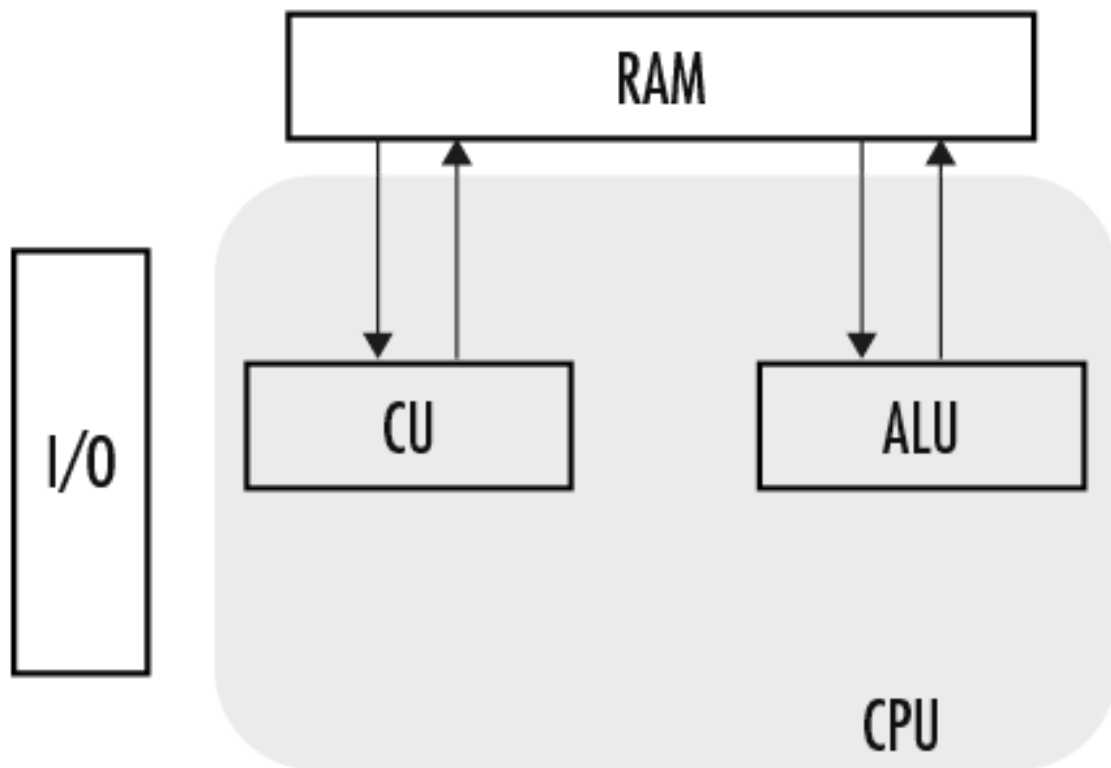
La arquitectura de Von Neumann, también conocida como arquitectura de Princeton, es una arquitectura basada en la descrita en 1945 por el matemático y físico John von Neumann. Él describió la arquitectura para una computadora electrónica digital con sus partes. Estas consisten en una unidad de procesamiento (CPU) que contiene una unidad aritmético-lógica (ALU) y registros de procesador, una unidad de control (CU) que contiene un registro de instrucciones y un contador de programas (PC), una memoria para almacenar datos e instrucciones, elementos externos, almacenamiento masivo y mecanismos de entrada y salida (I/O). Un esquema de esta arquitectura se puede ver en la figura 1.

La arquitectura de Von Neumann es, en definitiva, la arquitectura fundamental en la que se han basado casi todas las computadoras digitales. Tiene una serie de características que han tenido un inmenso impacto en los lenguajes de programación más extendidos. Estas características incluyen un único control centralizado, alojado en la unidad central de procesamiento, y un área de almacenamiento separada, memoria primaria, que puede contener tanto instrucciones como datos.

Las instrucciones son ejecutadas por la CPU, por lo que deben llevarse a la CPU desde la memoria primaria. La CPU también alberga, como hemos mencionado, la unidad que realiza operaciones en operandos, la unidad aritmética y lógica (ALU), por lo que los datos deben obtenerse de la

memoria primaria y llevarse a la CPU para poder actuar sobre ellos. La memoria primaria tiene un mecanismo de direccionamiento incorporado, de modo que la CPU puede referirse a las direcciones de instrucciones y operandos. Finalmente, la CPU contiene un banco de registros que constituye una especie de “scratch pad” donde los resultados intermedios pueden ser almacenados y consultados con mayor rapidez que la memoria primaria.

Figura 1. Arquitectura de Von Neumann integrada por la memoria externa (RAM), la CPU conteniendo la unidad de control (CU) y la unidad aritmético-lógica, y los dispositivos de entrada y salida para interactuar con el sistema (I/O)



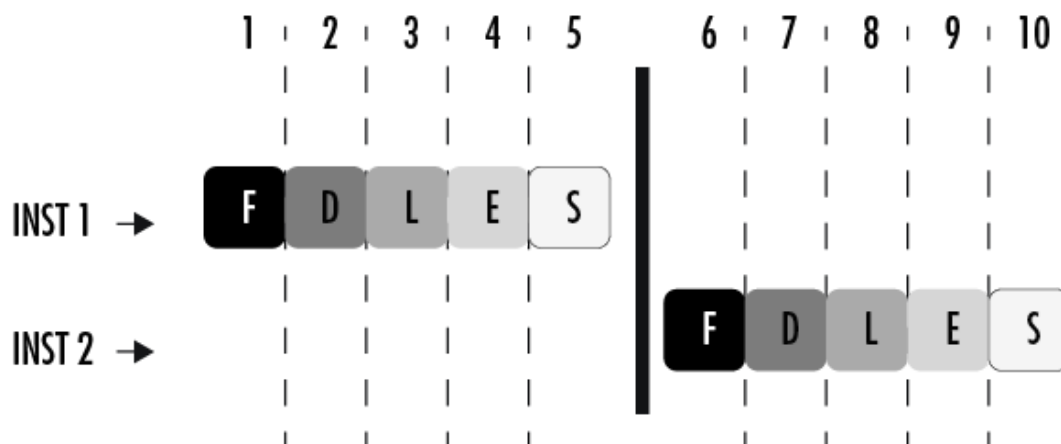
Podemos esquematizar el procesamiento de las instrucciones a través de los siguientes pasos:

1. **FETCH:** obtener instrucciones de la memoria del programa almacenado en la RAM.
2. **DECODE:** leer registros y decodificar la instrucción utilizando la unidad de control.
3. **LOAD:** acceder a un operando en la memoria de datos.

4. EXECUTE: ejecutar la instrucción o calcular una dirección.
5. STORE: escribe el resultado en un registro.

De modo que el procesamiento secuencial consistirá en repetir la secuencia de pasos por cada instrucción que procesa la computadora. Esta situación está representada por la figura 2. Cada etapa se realiza durante un ciclo de reloj.

Figura 2. Procesamiento secuencial de instrucciones. Los números en la secuencia superior representan los ciclos de reloj



Un tipo muy extendido de procesadores se conoce como RISC (Reduced Instruction Set Computer). En este tipo de arquitectura de microprocesador, se utiliza un conjunto de instrucciones pequeño y altamente optimizado, en lugar de un conjunto de instrucciones más especializadas que a menudo se encuentran en otros tipos de arquitecturas.

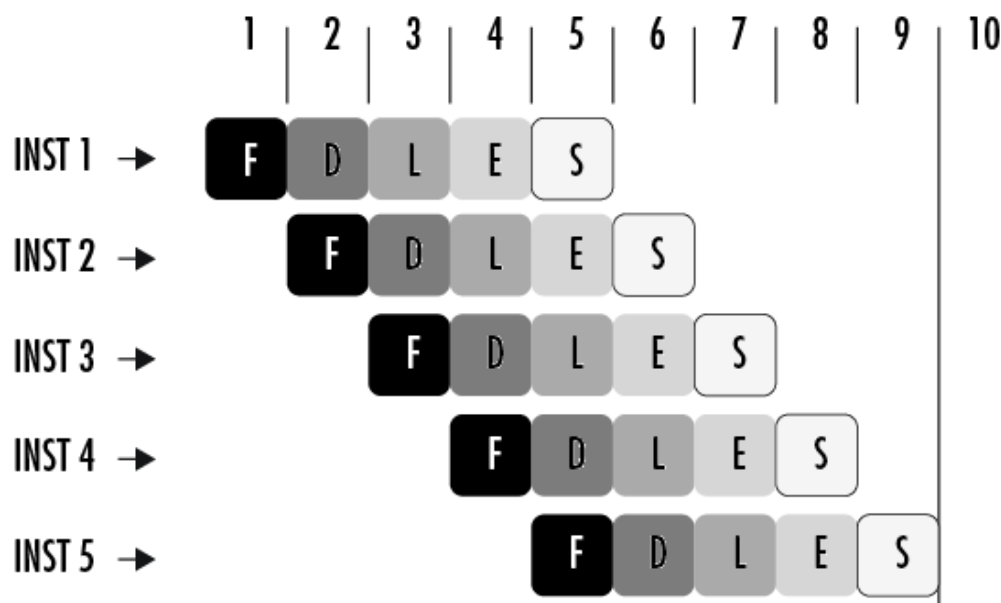
Los primeros proyectos de RISC vinieron de IBM, Stanford y UC-Berkeley a finales de la década de 1970 y principios de la de 1980. Ciertas características de diseño generales han sido características de la mayoría de los procesadores RISC que siguieron:

One cycle execution time: los procesadores RISC tienen un CPI (reloj por instrucción) de un ciclo. Esto se debe a la optimización de cada instrucción

en la CPU.

Pipelining: una técnica que permite la ejecución simultánea de partes, o etapas, de instrucciones para procesar de manera más eficiente. La figura 3 presenta un esquema de pipeline donde se muestra cómo se solapa el conjunto de instrucciones. Debido a que el procesador trabaja en diferentes pasos de la instrucción al mismo tiempo, se pueden ejecutar más instrucciones en un período de tiempo más corto.

Figura 3. Estructura de pipeline ejemplificada para un conjunto de instrucciones. Los números en la secuencia superior representan los ciclos de reloj



Gran número de registros: la filosofía de diseño RISC generalmente incorpora un mayor número de registros para evitar grandes cantidades de interacciones con la memoria.

Procesadores escalares, vectoriales y superescalares

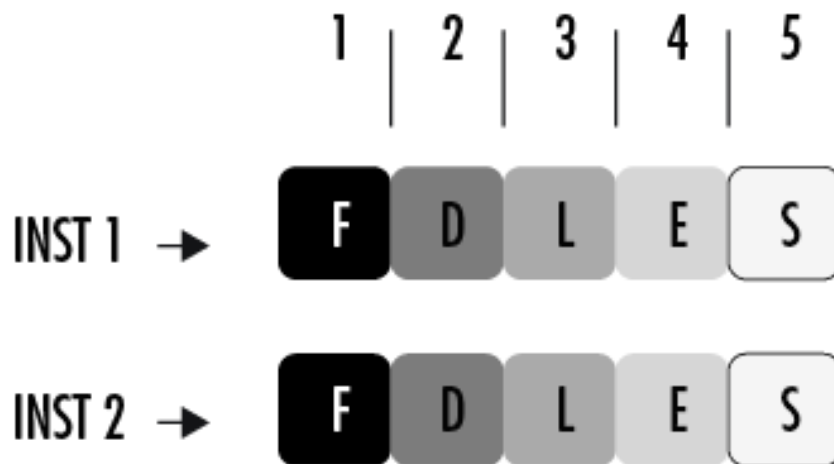
Un procesador escalar trabaja sobre una pieza de datos por vez. Un procesador vectorial actúa sobre varios datos con una sola instrucción. Un procesador superescalar emite varias instrucciones a la vez, cada una de las cuales opera en un solo dato.

Los procesadores vectoriales fueron populares para las supercomputadoras en las décadas de 1980 y 1990 porque manejaban de manera eficiente los vectores largos de datos comunes en los cálculos científicos, y ahora se utilizan mucho en las unidades de procesamiento de gráficos (GPU). Los microprocesadores modernos de alto rendimiento son superescalares, porque la emisión de varias instrucciones independientes es más flexible que el procesamiento de vectores.

Sin embargo, los procesadores modernos también incluyen hardware para manejar vectores cortos de datos que son comunes en aplicaciones de gráficos y multimedia. Se denominan unidades de datos múltiples de instrucción única (SIMD).

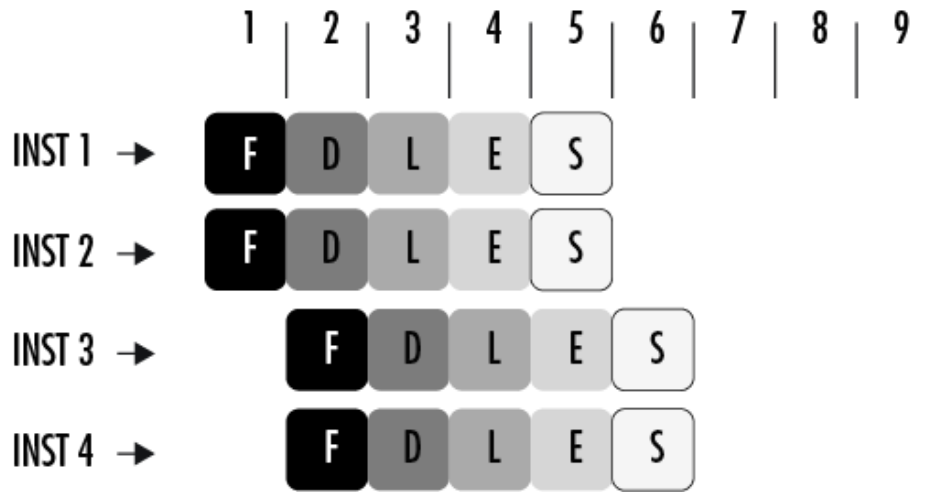
Un procesador superescalar contiene múltiples copias del hardware de la ruta de datos para ejecutar múltiples instrucciones simultáneamente. Esquemáticamente esto puede observarse en la figura 4. Hay que tener en cuenta que sigue siendo escalar y el paralelismo se da a nivel de instrucción. Para que esto sea posible, las instrucciones deben ser independientes.

Figura 4. Estructura de superscaling ejemplificada para un conjunto de instrucciones. Los números en la secuencia superior representan los ciclos de reloj. En este caso, el paralelismo es a nivel instrucción



Esto permite que muchas instrucciones se ejecuten simultáneamente, generalmente puede ser combinado con pipelining, como se muestra en la figura 5.

Figura 5. Estructura de superscaling y pipeline ejemplificada para un conjunto de instrucciones



Lo que constituye un procesador vectorial ha cambiado mucho a lo largo de los años, pero su característica principal es que estos procesadores pueden operar con matrices o vectores de datos, mientras que las CPU convencionales operan en elementos de datos individuales o escalares. Los sistemas recientes típicos tienen las siguientes características:

Registros vectoriales. Son registros capaces de almacenar un vector de operandos y operar simultáneamente sobre su contenido. La longitud del vector la fija el sistema y puede oscilar entre 4 y 128 elementos de 64 bits.

Unidades funcionales vectorizadas y que admiten pipeline. Se debe tener en cuenta que se aplica la misma operación a cada elemento del vector o, en el caso de operaciones como la suma, se aplica la misma operación a cada par de elementos correspondientes en los dos vectores. Por tanto, las operaciones vectoriales son SIMD.

Instrucciones vectoriales. Estas son instrucciones que operan en vectores en lugar de escalares.

Memoria intercalada. El sistema de memoria consta de múltiples “bancos” de memoria, a los que se puede acceder de forma más o menos independiente. Después de acceder a un banco, habrá un retraso antes de que

se pueda volver a acceder, pero se puede acceder a un banco diferente en un tiempo menor. Entonces, si los elementos de un vector se distribuyen en varios bancos, puede haber poco o ningún retraso en la carga o almacenamiento de elementos sucesivos.

Acceso rápido a la memoria y dispersión / recopilación de hardware. En el acceso escalonado a la memoria, el programa accede a elementos de un vector ubicados a intervalos fijos. Por ejemplo, acceder al primer elemento, al quinto elemento, al noveno elemento, etc., sería un acceso escalonado con un paso de cuatro. Dispersar / reunir (en este contexto) es escribir (dispersar) o leer (reunir) elementos de un vector ubicado a intervalos irregulares, por ejemplo, acceder al primer elemento, al segundo elemento, al cuarto elemento, al octavo elemento, etc. Los sistemas vectoriales típicos proporcionan hardware especial para acelerar el acceso escalonado y la dispersión / recopilación.

Los procesadores vectoriales tienen la virtud de que para muchas aplicaciones son muy rápidos y muy fáciles de usar. Los compiladores de vectorización son bastante buenos para identificar código que se puede vectorizar. Además pueden identificar loops que no se pueden vectorizar y, a menudo, proporcionan información sobre por qué no se puede vectorizar. De este modo, el usuario puede tomar decisiones informadas sobre si es posible reescribir dicho loop para que se vectorice. Los sistemas vectoriales tienen un ancho de banda de memoria muy alto y todos los elementos de datos que se cargan se utilizan realmente, a diferencia de los sistemas basados en caché que pueden no hacer uso de todos los elementos de una línea de caché.

Por otro lado, no manejan estructuras de datos irregulares así como otras arquitecturas paralelas, y parece haber un límite muy finito para su escalabilidad. En definitiva, un procesador vectorial es capaz de ejecutar operaciones matemáticas sobre múltiples datos de forma simultánea gracias a sus registros vectoriales. En general, esta característica es adicional al pipelining superescalar. Permiten usar un conjunto de instrucciones vectoriales especiales (SSE, AVX, etc). Este tipo de instrucciones, por ejemplo, son bien aprovechadas por *frameworks* de cálculo tales como tensorflow.

Veamos un breve ejemplo de código en C de cómo podemos aprovechar los registros vectoriales. Supongamos que tenemos el siguiente código para realizar la operación dada por la variable *z* y que debemos repetirla dentro de un cierto loop:

```
for (i=0; i<length; i++){  
  z[i]=a[i]*b[i]+c[i]*d[i];  
}
```

Luego, si escribimos el loop de manera inteligente, el compilador podrá aprovechar la estructura vectorial de nuestro procesador:

```
for (i=0; i<length; i+=2){  
  z[i] = a[i]*b[i] + c[i]*d[i];  
  z[i+1] = a[i+1]*b[i+1] + c[i+1]*d[i+1];  
}
```

Este ejemplo pone de manifiesto que al comprender las características del hardware sobre el que estamos programando, podemos tener un mejor control sobre los procesos de optimización.

Cómo medir la performance de un sistema de cómputo

Es necesario primero definir las distintas métricas utilizadas para medir la performance. Entre estas tenemos: latencia y rendimiento. Por otro lado tenemos que discutir brevemente cómo definir la performance de la CPU, la performance pitfalls y lo que se conoce como benchmarking.

Se define latencia (o tiempo de ejecución) como el tiempo para finalizar una tarea dada. Por otro lado, definimos el rendimiento como el número de tareas en tiempo fijo. Un usuario puede estar interesado en reducir el tiempo de respuesta (latencia), es decir, el tiempo entre el inicio y la finalización de un evento, o tiempo de ejecución. Al comparar alternativas de diseño, cuando queremos relacionar el rendimiento de dos computadoras diferentes, digamos, *X* e *Y*. La frase “*X* es más rápido que *Y*” se usa para decir que el tiempo de respuesta o el tiempo de ejecución es menor en *X* que en *Y* para la tarea dada.

El tiempo no siempre es la métrica más adecuada al comparar el rendimiento de las computadoras. Una medida consistente y confiable de la performance es el tiempo de ejecución de programas reales. Incluso el tiempo de ejecución se puede definir de diferentes formas dependiendo de lo

que contamos. La definición más sencilla de tiempo se denomina tiempo wall-clock, tiempo de respuesta o tiempo transcurrido, que es la latencia para completar una tarea, incluidos los accesos al disco, los accesos a la memoria, las actividades de entrada y salida, la sobrecarga del sistema operativo, todo. Con la multiprogramación, el procesador trabaja en otro programa mientras espera la E/S y no necesariamente minimiza el tiempo transcurrido de un programa. Por lo tanto, necesitamos un término para considerar esta actividad. El tiempo de CPU reconoce esta distinción y significa el tiempo que el procesador está calculando, sin incluir el tiempo de espera para E/S o ejecutar otros programas. (Claramente, el tiempo de respuesta que ve el usuario es el tiempo transcurrido del programa, no el tiempo de la CPU.)

Se define como *benchmark* a una *prueba de rendimiento o comparativa*. Es una técnica utilizada para medir el rendimiento de un sistema de cómputo, o alguno de sus componentes. Formalmente puede entenderse que una prueba de rendimiento es el resultado de la ejecución de un programa o un conjunto de programas en una máquina, con el objetivo de estimar el rendimiento de un elemento concreto, y poder comparar los resultados con máquinas similares.

A nivel desktop, los benchmarks se dividen en dos grandes clases: benchmarks de procesador intensivo y las evaluaciones comparativas intensivas en gráficos. En sistemas más grandes como servidores que cuentan con múltiples funciones, existen varios tipos de benchmarks definidos para las distintas funciones. Entendiendo estos aspectos, podemos enfocarnos ahora en la performance de la CPU y cómo medirla.

Una forma de medir la performance es considerar las operaciones de punto flotante por segundo (FLOPS, flops o flop / s) que se pueden realizar. Esta puede ser una medida del rendimiento de la computadora, útil en campos de la programación científica donde se requieren operaciones de punto flotante. Para tales casos, es una medida más precisa que medir el número de instrucciones por segundo.

El rendimiento teórico máximo en términos de operaciones de punto flotante de un procesador (*peak theoretical floating-point performance*) es generalmente de al menos $2 \times \text{núcleos} \times \text{frecuencia} \times n$, donde n es el número de operaciones de punto flotante que el procesador puede realizar por ciclo y asumiendo que el procesador admite operaciones de acumulación múltiple.

En la práctica, es imposible que cualquier procesador logre su máximo rendimiento de punto flotante, ya que requeriría que el procesador proporcione entradas a todas sus unidades de punto flotante en cada ciclo y oculte todas las fuentes de latencia. Una fuente importante de latencia es el tiempo medio de acceso a la memoria, lo que significa que las unidades de punto flotante deberían mantenerse ocupadas durante todo el tiempo necesario para acceder a los datos necesarios para el cálculo. Dado que las unidades de punto flotante pueden lograr un rendimiento mucho mayor que la memoria fuera del chip, su rendimiento efectivo depende del grado en que el kernel reutilice los datos durante su ejecución. Esta tasa de reutilización generalmente se caracteriza mediante una medida llamada *intensidad aritmética*, que expresa el número promedio de operaciones de punto flotante realizadas por byte de datos.

En resumen:

Peak Performance (teórico): estimación del desempeño de la CPU cuando trabaja a máxima velocidad.

Benchmark Performance: se utilizan herramientas específicas para medir el pico de performance real.

Real Performance: medición realizada con el programa que quiero correr.

Jerarquía de memoria

Una de las claves en la programación de alto rendimiento es conocer y ser capaces de realizar el mejor manejo posible de memoria. Para esto primero es necesario conocer la jerarquía de memoria.

En el nivel más alto en la jerarquía de memoria nos encontramos con *los registros*. Estos están directamente cableados dentro del procesador. Los registros que forman parte de la propia CPU tienen tiempos de acceso muy bajos, del orden de algunos cientos de picosegundos y el espacio de almacenamiento es de unos pocos miles de bytes. La transferencia de datos entre registros y memoria es administrada por el compilador y el programador.

Antes de definir la *memoria caché*, definiremos el principio de localidad. El mismo establece que un programa accede a una parte relativamente

pequeña del espacio de direcciones en cualquier momento. Esta localidad se puede referir al tiempo o al espacio. Consideremos por ejemplo el principio de localidad espacial; en este caso nos referimos a partes que pertenecen a un mismo bloque, como por ejemplo recorrer un vector dentro de un loop. En cuanto al principio de localidad temporal, se refiere a la reutilización de un bloque, por ejemplo un llamado a una función dentro de un loop.

Respecto de la *memoria caché*, corresponde a uno de los niveles más altos o al primer nivel de la jerarquía de memoria que se encuentra una vez que la dirección sale del procesador. Dado que el principio de localidad se aplica en muchos niveles, y es popular aprovechar la localidad para mejorar el rendimiento, el término “caché” se aplica ahora siempre que se emplea el almacenamiento en búfer para reutilizar elementos comunes. Los ejemplos incluyen cachés de archivos, cachés de nombres, etcétera.

La *memoria caché* tiene un mayor costo, menor capacidad pero mayor velocidad. Los datos se transfieren a caché en bloques de un determinado tamaño *caché lines*.

Cuando el procesador encuentra un elemento de datos solicitado en la caché (en una operación de load/store), se denomina *caché hit*. Cuando el procesador no encuentra un elemento de datos que necesita en el caché, se produce un *caché miss*. Una colección de datos de tamaño fijo que contiene la palabra solicitada, llamada bloque o ejecución de línea, se recupera de la memoria principal y se coloca en la caché. La localidad temporal nos dice que es probable que necesitemos esta palabra nuevamente en un futuro cercano, por lo que es útil colocarla en la caché donde se puede acceder rápidamente. Debido a la localidad espacial, existe una alta probabilidad de que pronto se necesiten los demás datos del bloque.

El tiempo necesario para la *caché miss* depende tanto de la latencia como del ancho de banda de la memoria. La latencia determina el tiempo para recuperar la primera palabra del bloque y el ancho de banda determina el tiempo para recuperar el resto de este bloque. Un *caché miss* es manejado por hardware y hace que los procesadores que utilizan la ejecución en orden se detengan o se esperen hasta que los datos estén disponibles.

A su vez, la memoria caché está dividida en tres niveles. El primer nivel tiene unos pocos kilobytes y los tiempos de acceso son solo unos pocos nanosegundos. La caché de segundo nivel tiene unos cientos de kilobytes y los tiempos de acceso aumentan a unos 10 nanosegundos. El almacenamiento

aumenta a unos pocos megabytes en el caso del tercer nivel de caché y los tiempos de acceso aumentan a unas pocas decenas de nanosegundos.

La transferencia de datos entre la memoria caché y la memoria la gestiona el hardware.

Si queremos ver las características del caché de nuestro CPU, en Linux podemos utilizar el comando `lscpu`.

Por ejemplo, ejecutando este comando en google colab (`!lscpu`), me devuelve las características del procesador particular que ha sido asignado, incluida la información sobre memoria caché:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    2
Core(s) per socket:    1
Socket(s):             1
NUMA node(s):         1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU @ 2.20GHz
Stepping:              0
CPU MHz:               2199.998
BogoMIPS:              4399.99
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              56320K
NUMA node0 CPU(s):     0,1
```

En el siguiente nivel de memoria nos encontramos con *la memoria principal*. Hay diferentes tipos de memoria disponibles. Una clasificación posible se basa en los modos de acceso. Una memoria de acceso aleatorio (RAM) tiene el mismo tiempo de acceso para todas las ubicaciones. Hay dos tipos de RAM: dinámica y estática. La memoria dinámica de acceso aleatorio tiene alta densidad, consume menos energía, es barata y lenta. Se le llama dinámico, porque necesita ser “actualizado” con regularidad. Una SRAM –

memoria estática de acceso aleatorio— tiene baja densidad, consume mucha energía, es cara y rápida. Aquí, el contenido durará “siempre” (hasta que se pierda la energía). Dependiendo del nivel del lenguaje de programación, podemos tener distinto grado de control al acceso a memoria.

En el último nivel, nos encontramos con la memoria donde el tiempo de acceso varía de un lugar a otro y de vez en cuando. Los ejemplos de este tipo de memoria incluyen discos rígidos principalmente y dispositivos de almacenamiento. La transferencia entre la memoria y los discos es administrada por el hardware y el sistema operativo (memoria virtual) y por el programador (archivos).

La figura 6 muestra los distintos niveles de la jerarquía de memoria.

Figura 6. Jerarquía de memoria



Programación en entornos paralelos: HPC

Comencemos por discutir la computación de alta performance también llamada HPC o High Performance Computing. Esta es utilizada fundamentalmente cuando la clave del problema se relaciona con la eficiencia, es decir, cuando nos importa tener una respuesta rápida o una alta productividad.

Se puede hacer computación de alto rendimiento en distintos dispositivos, desde una computadora de escritorio, notebook, smartphone hasta una

supercomputadora, un clúster o una infraestructura grid o cloud, y también en combinación de todas las opciones anteriores. La idea es aprovechar el hardware que tenemos disponible al máximo. También nos permite hacer consideraciones sobre cuál es el hardware que se necesita para una determinada tarea. En particular, actualmente es muy utilizada para grandes simulaciones. Hay que tener en cuenta que muchas veces al pensar en este enfoque se sacrifican algunos aspectos tales como la posibilidad de tener interfaces amigables, software reutilizable, o incluso la portabilidad.

La programación en entornos paralelos es una manera de obtener un excelente rendimiento en distintos tipos de problemas. En general se aplica a problemas con gran complejidad, ya sea por ser de gran escala y/o que impliquen el manejo de grandes volúmenes de datos:

- Modelos complejos.
- Problemas complicados.
- Grandes volúmenes de datos.
- Capacidad de respuesta en tiempo limitado (sistemas de tiempo real, por ejemplo).

Existen cuatro formas en las que el hardware soporta la paralelización a nivel datos y a nivel tareas. Michael Flynn encontró una clasificación simple cuyas abreviaturas todavía son usadas hoy. Observó el paralelismo en las instrucciones y los flujos de datos requeridos por las instrucciones en el componente más restringido del multiprocesador, y colocó todos los sistemas en una de cuatro categorías:

1. *Single instruction, single data (SISD)*. Esta categoría está referida a un único procesador. El programador lo considera como la computadora secuencial estándar que vimos en la sección donde se describe la arquitectura del computador básica, pero puede explotar el paralelismo a nivel de instrucción, como hemos discutido con los procesadores superescalares y vectoriales.

2. *Single instruction, multiple data (SIMD)*. En esta categoría, la misma instrucción es ejecutada por múltiples procesadores usando diferentes flujos de datos. Las computadoras SIMD explotan el paralelismo a nivel de datos

aplicando el mismo conjunto de operaciones a varios elementos de datos en paralelo. Cada procesador tiene su propia memoria de datos, pero hay una única memoria de instrucciones y un procesador de control, que obtiene y envía instrucciones. Tres arquitecturas diferentes que lo explotan son vectoriales, extensiones multimedia para conjuntos de instrucciones estándar y GPU.

3. *Multiple instruction, single data (MISD)*. Cada unidad de procesamiento opera sobre los datos de forma independiente a través de flujos de instrucciones separados. Un único flujo de datos es alimentado a múltiples unidades de procesamiento. Existen pocos ejemplos reales (si los hay) de esta clase de paralelo, la computadora alguna vez ha existido pero completa esta clasificación simple.

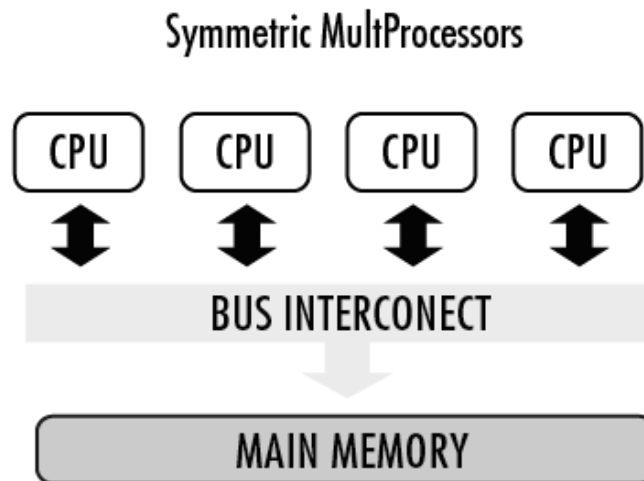
4. *Multiple instruction, multiple data (MIMD)*. Cada procesador obtiene sus propias instrucciones y opera con sus propios datos, en este caso tiene como objetivo el paralelismo a nivel de tarea. En general, MIMD es más flexible que SIMD y, por lo tanto, de aplicación más general, pero es inherentemente más caro que SIMD. Por ejemplo, las computadoras MIMD también pueden explotar el paralelismo a nivel de datos, aunque es probable que el *overhead* sea mayor de lo que se vería en una computadora SIMD.

Esta taxonomía es un modelo simplificado, ya que muchos procesadores paralelos son híbridos respecto a las clases que hemos definido. No obstante, es útil poner un marco en el diseño para poder seleccionar la mejor opción que se ajuste a nuestro tipo de problema de interés o al sistema que tenemos disponible.

En términos de la distribución de memoria, existen diversos paradigmas cuando nos referimos a programación en entornos paralelos. Una opción es cuando hablamos *Symetric Multi-processors*, donde la idea es que se tiene una memoria principal interconectada (o *memoria compartida*) por un bus que accede a varios CPU. Un ejemplo de esta topología se puede ver en la figura 7. Las ventajas de este tipo de sistemas son que por un lado son fáciles de programar y por otro permiten compartir datos de manera rápida y

directa. Entre las desventajas, se destaca que presentan una escalabilidad pobre.

Figura 7. Modelo de memoria compartida

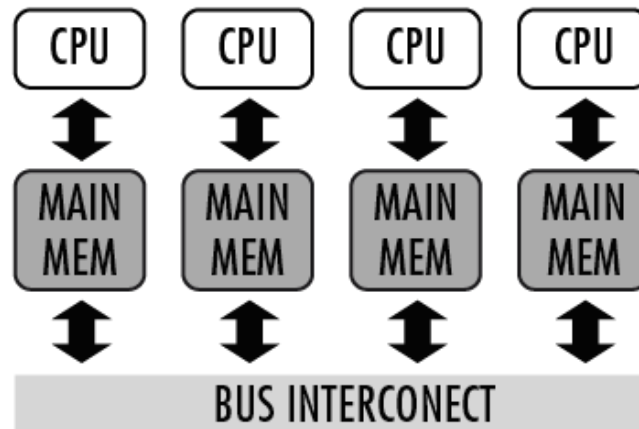


Otro modelo de sistema consiste en el *NonUniform Memory Access* (o *memoria distribuida*). En este caso, para cada CPU tenemos una memoria asociada. Esta topología se muestra en la figura 8. Entre las ventajas de estos sistemas se destacan primero la escalabilidad organizada: la memoria escala con el número de procesadores. Aquí, cada procesador accede rápidamente a su propia memoria local sin interferencia y sin overhead. El overhead se define como la combinación de tiempo de cálculo, memoria, ancho de banda u otros recursos en exceso o indirectos que se requieren para realizar una tarea específica.

Entre las desventajas de estos sistemas podemos destacar principalmente que la comunicación queda a cargo del programador. Por otro lado, puede ser difícil adaptar el código que se tiene ya hecho para aprovechar estos sistemas. Finalmente, hay que tener en cuenta que el acceso a los datos no es uniforme.

Figura 8. Modelo de memoria distribuida

NonUniform Memory Acces



Una pregunta que podemos hacer es cómo impacta el diseño de estos sistemas en el desarrollo de software. Cuando se aplican técnicas de programación en paralelo hay que pensar en el hardware durante el desarrollo del código. En este sentido, podemos pensar primero, antes de pasar al aspecto de los CPU múltiples, en una serie de pasos previos para que primero el código esté optimizado y luego tenga sentido y se puedan aprovechar realmente las características de estos sistemas. Estos pasos son:

- Usar aproximaciones cuando sea posible.
- Desarrollar algoritmos más eficientes.
- Utilizar estructuras de datos apropiadas.
- Obtener hardware más veloz.
- Usar/escribir software optimizado para el hardware que tenemos disponible. Aprovechar librerías ya optimizadas como BLAS, LAPACK u otras.

Cuando se han realizado todas estas consideraciones, es el momento de pensar en paralelizar las tareas. Hay que tener en cuenta que la programación paralela no siempre es la solución para todos los problemas y que no es fácil obtener resultados óptimos.

Ley de Amdahl

Establece que la mejora en la performance que se puede obtener mediante el uso de un modo de ejecución más rápido está limitada. Este límite está dado

por la parte serial de un programa. Esta determina una cota inferior para el tiempo de ejecución, aun cuando se utilicen al máximo técnicas de paralelismo.

El tiempo total (T) para la ejecución depende del tiempo del procesador (T_{proc}), del tiempo de comunicación (T_{com}) y del tiempo ocioso (idle).

$$T = T_{\text{proc}} + T_{\text{com}} + T_{\text{idle}}$$

El tiempo del procesador depende de varios factores, tales como la complejidad del problema, el número de tareas requeridas o las características del hardware. El tiempo de comunicación depende de la localización de los procesos y los datos (si la comunicación es inter- e intraprocador, cuántos canales de comunicación existen, etc.). En el caso del tiempo ocioso, no es un valor determinístico y depende de varios factores. En general, minimizar este tiempo suele ser un objetivo en el diseño de software.

Cómo empezamos a programar en paralelo

Primero hay que lograr un compromiso entre el grado de paralelismo, la sincronización y la comunicación. Cuando hablamos de comunicación, nos referimos al hecho de que las tareas realizadas en paralelo generalmente necesitan intercambiar datos, y existen en general varias formas en las que esto se pueda realizar.

Segundo, hay que elegir el modelo de programación de memoria compartida o memoria distribuida acorde con el sistema que se tenga. Hay que tener en cuenta que los recursos computacionales pueden ser o bien una única computadora con múltiples procesadores o cores, o bien un arbitrario número de computadoras conectadas en red.

Luego hay que pensar cómo dividir el problema, es decir, cómo particionar los datos y las funciones. El problema computacional tiene que tener ciertas características:

- Debe poder ser dividido en partes discretas que se pueden resolver simultáneamente.
- Debe permitir ejecutar múltiples instrucciones del programa en cualquier momento.

- Debe poder resolverse en menos tiempo con múltiples recursos informáticos que con un recurso informático único.

Por otro lado, hay que considerar distintos aspectos también relacionados con el problema computacional de interés como la *granularidad*, que se define como una medida cualitativa de la relación entre el cómputo y la comunicación. También considerar si el problema que estamos abordando es o no del tipo “*embarazosamente paralelo*”. En este tipo de problemas son llamados así porque consisten en resolver muchas tareas similares, pero independientes de modo simultáneo con poca o ninguna necesidad de coordinación entre las tareas. Estos son muy fáciles de paralelizar. Finalmente, considerar la *escalabilidad* del problema, es decir, la capacidad para demostrar un aumento proporcional de la velocidad paralela con la adición de más recursos.

Finalmente, como último paso, hay que elegir el lenguaje de programación, de manera acorde al conocimiento del grupo de investigadores, la disponibilidad de librerías de cálculo específicas u otros aspectos de interés.

Podemos considerar dos ejemplos de librerías para implementar la programación en paralelo y describiremos algunos aspectos de aquellas que permitirán al lector realizar al menos implementaciones simples de práctica: *MPI* (interfaz para el pasaje de mensajes) para memoria compartida y *OpenMP* para memoria distribuida.

El modelo de comunicación deseado se puede crear en software sobre un modelo de hardware que admita cualquiera de estos mecanismos. Basar el software de pasaje de mensajes por encima del sistema de memoria compartida es considerablemente más fácil debido a que los mensajes esencialmente envían datos de una memoria a otra. El envío de un mensaje se puede implementar haciendo una copia de una parte del espacio de direcciones a otra. Las principales dificultades surgen al tratar con mensajes que pueden estar no alineados y ser de longitud arbitraria en un sistema de memoria que normalmente está orientado hacia la transferencia de bloques alineados de datos organizados como bloques de caché. Estas dificultades se pueden superar con pequeñas penalizaciones de rendimiento en el software o prácticamente sin penalizaciones, utilizando una pequeña cantidad de soporte de hardware.

Basar el sistema de memoria compartida de software de manera eficiente sobre el hardware para el pasaje de mensajes es mucho más difícil. Sin soporte de hardware explícito para la memoria compartida, todas las referencias de memoria compartida deben involucrar al sistema operativo para proporcionar traducción de direcciones y protección de memoria, así como para traducir referencias de memoria en mensaje envía y recibe. Las cargas de datos y el almacenaje generalmente mueven pequeñas cantidades de datos, por lo que el gran overhead de manejar estas comunicaciones en software limita severamente la gama de aplicaciones para las que el rendimiento de la memoria compartida basada en software es aceptable.

Con estas salvedades, consideremos a continuación las distintas características de cada uno de los sistemas.

Programación en entornos paralelos: memoria compartida y OpenMP

OpenMP no es un lenguaje de programación, sino que funciona en conjunto con lenguajes existentes como Fortran estándar o C y C++. Es una interfaz de programación de aplicaciones (API) que proporciona un modelo portátil para paralelizar aplicaciones. Sus tres componentes principales:

- Las directivas del compilador.
- Un conjunto de rutinas runtime.
- Variables de entorno.

En el esquema de comunicación en memoria compartida, las ventajas incluyen:

- Compatibilidad con los mecanismos bien entendidos que se emplean en los multiprocesadores centralizados que utilizan la comunicación de memoria compartida. El consorcio OpenMP (www.openmp.org) ha propuesto una interfaz de programación para multiprocesadores de memoria compartida.
- Facilidad para la programación cuando los patrones de comunicación entre procesadores son complejos o varían dinámicamente durante la ejecución. También se presentan ventajas similares que simplifican el diseño del compilador.

- La capacidad de desarrollar aplicaciones centrando la atención solo en aquellos accesos que son críticos para el rendimiento.
- Permite reducir el *overhead* para la comunicación y hacer un mejor uso del ancho de banda al comunicar elementos pequeños. Esto surge de la naturaleza implícita de la comunicación y el uso de la asignación de memoria para implementar la protección en el hardware, en lugar de a través del sistema de E/S.
- La capacidad de utilizar el almacenamiento en caché controlado por hardware para reducir la frecuencia de la comunicación remota al admitir el almacenamiento en caché automático de todos los datos, tanto compartidos como privados. El almacenamiento en caché reduce tanto la latencia como la contención para acceder a los datos compartidos.

Para poder comprender mejor su funcionamiento, definamos lo que es un *thread*. Este se define como un proceso (liviano) –una instancia de un programa más sus propios datos–. Cada thread puede seguir su propio flujo de control a través de un programa. Los thread pueden compartir datos con otros subprocesos, pero también tienen datos privados. Los thread se comunican entre sí a través de los datos compartidos. Un thread maestro es responsable de coordinar los grupos de threads. OpenMP permite controlar el manejo de threads, pero esconde los llamados a threads en una librería, esto lo hace menos flexible, pero se necesita menos programación.

OpenMP se basa en directivas. Una directiva es una línea especial de código fuente con solo significado a ciertos compiladores. Pueden ser agregadas también de manera incremental al código. El modelo que utiliza es Fork-Join. El programa OpenMP comienza como un solo proceso: el *master thread*. El proceso maestro se ejecuta secuencialmente hasta que la primera región paralela es encontrada. Cuando se encuentra una región paralela, el master thread crea un grupo de threads con FORK. De este modo, ese thread se convierte en el maestro de este grupo de subprocesos y se le asigna el ID de subproceso 0 dentro del grupo. Las instrucciones en el programa están incluidas en la construcción de la región paralela y se ejecutan en paralelo entre estos subprocesos. Cuando los subprocesos completan la ejecución de la instrucción en la construcción de la región paralela, se sincronizan y terminan, dejando solo el subproceso maestro (JOIN).

OpenMP no paraleliza automáticamente la entrada y salida de datos. Depende del programador asegurarse de que la entrada y salida se realice correctamente dentro del contexto de un programa multiproceso. Los subprocesos pueden “almacenarse en caché” y no es necesario que mantengan una consistencia exacta con la memoria real todo el tiempo. Cuando sea crítico que todos los subprocesos vean una variable compartida de forma idéntica, el programador es responsable de asegurarse de que todos los subprocesos actualicen la variable según sea necesario.

Formato de las directivas para el compilador

Como ya hemos mencionado en la sección anterior, una directiva es una línea especial de código fuente con solo significado para el compilador. Estas directivas se distinguen con un símbolo al principio. Por ejemplo, en Fortran:

```
!$OMP (or C$OMP or *$OMP)
```

En C y en C++:

```
#pragma omp
```

Por ejemplo, consideremos un bloque de código C. Para crear una región paralela de cuatro threads, donde cada thread llama a la función foo(ID,A), de manera que cada thread de forma redundante ejecuta el código dentro el bloque estructurado, usamos el siguiente código:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID =omp_get_thread_num();
    foo(ID,A);
}
printf("Todo finalizado\n");
```

Donde la variable ID corre desde 0 hasta 3. Lo que determina el número de procesos en una región paralela depende de varios factores:

- Del uso de la función de la librería `omp_set_num_threads ()`.

- De la configuración de la variable de entorno `OMP_NUM_THREADS`.
- Del valor de la implementación predeterminada.

Compilando con openMP

Usando el compilador gcc:

```
$ gcc -fopenmp -c my_openmp.c  
$ gcc -fopenmp -o my_openmp.x my_openmp.o
```

Usando el compilador icc:

```
$ icc -openmp -c my_openmp.c  
$ icc -openmp -o my_openmp.x my_openmp.o
```

Algunos métodos de la librería Runtime

```
OMP_GET_NUM_THREADS() # - devuelve el número de threads.  
OMP_GET_THREAD_NUM() #- devuelve el id de ese thread.  
OMP_SET_NUM_THREADS(n)# - setea el numero de threads.  
OMP_IN_PARALLEL() #- devuelve .true. si está dentro de la región paralela.  
OMP_GET_MAX_THREADS() # - devuelve el número de posibles threads.
```

Ejemplo simple en C sobre un programa utilizando OpenMP

En el siguiente ejemplo se muestra cómo llamar la librería, y como pararse dentro de un conjunto de procesos para ejecutar una instrucción determinada, mostrando las diferencias entre la región serial y la paralela del código.

```
#include <omp.h>  
#include <stdio.h>  
int main ( ) {  
    printf("Comenzando de modo secuencial.\n");  
    #pragma omp parallel  
    {  
        printf("Hola desde thread o proceso numero %d\n", omp_get_thread_num() );  
    }  
    printf("Regresando al modo secuencial.\n");  
    return 0;  
}
```

Alcance de las variables

Todas las variables todavía existen dentro de una región paralela y, por defecto, están compartidas entre todos los threads. Pero trabajar compartiendo variables requiere tener variables privadas. Hay que ser consciente también de que las variables estáticas son compartidas.

Directivas para paralelizar loops

Estas se pueden hacer en Fortran como:

```
!$omp do
```

O bien, para C,

```
#pragma omp for
```

Estas directivas no crean un conjunto de threads, pero asumen que ya se ha bifurcado (forked). Si no se está dentro de una región paralela, se pueden usar atajos:

```
!$omp parallel do
```

O bien para C.

```
#pragma omp parallel for
```

En la sección Hands-on se proponen algunos ejercicios simples e introductorios para poder implementar las directivas estudiadas en esta sección. Para más detalles, se debe acudir a la guía de referencia que se actualiza periódicamente: <<https://www.openmp.org/resources/refguides/>>.

A continuación mostraremos una instrucción para el uso del paradigma de memoria distribuida.

Programación en entornos paralelos: memoria distribuida y MPI

MPI significa Message Passing Interface. Permite ejecutar un mismo código en múltiples procesadores. La clave de este paradigma es la comunicación entre nodos. Proporciona una topología virtual esencial, sincronización y funcionalidad de comunicación entre un conjunto de procesos que se han asignado a los distintos nodos/servidores/cores. Los programas MPI siempre

funcionan con procesos. Los programadores comúnmente se refieren a los procesos como procesadores. Normalmente, para rendimiento máximo, cada CPU (o núcleo en una máquina multinúcleo) será asignado a un solo proceso. Esta asignación ocurre en tiempo de ejecución a través del agente que inicia el programa MPI, normalmente llamado `mpirun` o `mpiexec`.

Las ventajas de este paradigma incluyen:

- El hardware puede ser más simple, especialmente en comparación con una implementación de memoria compartida escalable que soporte el almacenamiento en caché coherente de datos remotos.
- La comunicación es explícita, lo que significa que es más fácil de entender. En los modelos de memoria compartida, puede ser difícil saber cuándo se está produciendo una comunicación y cuándo no, así como qué tan costosa es la comunicación que se realiza.
- La comunicación explícita centra la atención del programador en este costoso aspecto de la computación en paralelo, lo que a veces conduce a una estructura mejorada en un programa multiprocesador.
- La sincronización está naturalmente asociada con el envío de mensajes, y esto reduce la posibilidad de errores introducidos por una sincronización incorrecta.
- Hace más fácil el uso de la comunicación iniciada por el remitente, lo que puede tener algunas ventajas en el rendimiento.
- Si la comunicación es menos frecuente y más estructurada, es más fácil mejorar la tolerancia a fallas utilizando una estructura similar a una transacción. Además, el acoplamiento menos estrecho de los nodos y la comunicación explícita simplifican el aislamiento de fallas.
- Los multiprocesadores más grandes utilizan una estructura de clúster, que se basa inherentemente en el pasaje de mensajes. El uso de dos modelos de comunicación diferentes puede introducir más complejidad de la que se justifica.

Existen diferentes implementaciones de MPI, entre ellas: MPICH, MSPI y OpenMPI. Por ejemplo para instalar OpenMPI:

```
$ sudo apt-get install -y autotools-dev g++ buildessential  
openmpi-bin openmpi-doc libopenmpi-dev
```

Los conceptos que describimos a continuación ayudan a comprender y proporcionar contexto y permitirán al programador decidir qué funcionalidad utilizar en sus programas de aplicación.

- Comunicador.
- Conceptos básicos de la comunicación punto a punto.
- Conceptos básicos de la comunicación colectiva.
- Tipos de datos derivados.

Para llamar a la librería, dependiendo del lenguaje de programación que se use, tenemos las siguientes opciones.

Por ejemplo para C:

```
#include <mpi.h>
```

Para Fortran:

```
include 'mpif.h'
```

Para Python (que también tiene una implementación):

```
from mpi4py import MPI
```

Sobre la inicialización, MPI controla sus propias estructuras de datos internas. Veamos un ejemplo de inicialización con C:

```
int MPI_Init(int *argc, char ***argv)
```

La estructura de código de un programa que tiene implementado MPI contiene primero la llamada a las librerías necesarias, incluida mpi (`#include mpi.h`), luego vendrá la porción serial del código, como ya estamos acostumbrados. En la sección que vamos a paralelizar, se inicializa la región como indicamos según el lenguaje de programación. Luego escribimos la porción de código paralelizada recordando finalizarla, y finalmente cerramos la parte serial del programa.

Comunicadores

La clave en este paradigma es administrar la comunicación, es decir, el pasaje de mensajes. Para esto se define un comunicador dentro del cual

nuestros procesos están identificados. Veamos un ejemplo simple de un programa implementado en C utilizando MPI para poder definir los distintos aspectos necesarios básicos. En particular, haciendo foco en el tamaño del comunicador, es decir, la cantidad de procesos que tenga.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
int main (int argc, char* argv[])
{
    int my_rank, size;
    MPI_Init (&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Soy el proceso %d de %d !!\n", my_rank, size );
    MPI_Finalize();
    return 0;
}
```

MPI_Init, como hemos mencionado, indica el comienzo de la región paralela del código. MPI_Comm_size indica el tamaño del comunicador. La instrucción MPI_Comm_rank permite identificar diferentes procesos en un comunicador, va desde 0 hasta size-1. Así, para finalizar la región paralela del código usamos MPI_Finalize(). En este caso, la función printf se ejecutará tantas veces como sea el tamaño del comunicador. Para compilar el código:

```
$ mpicc -o my_parallel_application_c my_parallel_application.c
```

Y para ejecutarlo:

```
$ mpirun -np N my_parallel_application
```

Comunicación punto a punto

Un mensaje contiene varios elementos de algún tipo de datos en particular. Los tipos de datos MPI pueden ser:

- Tipos básicos.
- Tipos derivados.

Los tipos derivados se pueden construir a partir de tipos básicos. Los tipos C son diferentes de los tipos Fortran. En el caso de Python MPI no es

necesario especificar. El tipo de datos derivado más simple consta de una serie de elementos contiguos del mismo tipo de datos. Por ejemplo en C:

```
int MPI_Type_contiguous(int count,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Otro tipo de datos puede ser vectorial:

```
int MPI_Type_vector (int count,  
int blocklength, int stride,  
MPI_Datatype oldtype,  
MPI_Datatype *newtype)
```

Existen muchos otros tipos de datos que pueden consultarse en el manual de referencia, según la aplicación para la cual se los necesite: <https://www.mpich.org/static/docs/latest/www3/Constants.html>.

Una lista muy resumida de estos para C es:

<i>MPI Datatype</i>	<i>C datatype</i>
MPI CHAR	signed char
MPI SHORT	signed short int
MPI INT	signed int
MPI LONG	signed long int
MPI UNSIGNED CHAR	unsigned char
MPI UNSIGNED SHORT	unsigned short int
MPI UNSIGNED	unsigned int
MPI UNSIGNED LONG	unsigned long int
MPI FLOAT	float
MPI DOUBLE	double
MPI LONG DOUBLE	long double
MPI BYTE	
MPI PACKED	

Comunicación entre dos procesos (Blocking communication)

El proceso de origen envía un mensaje al proceso de destino. La comunicación tiene lugar dentro de un comunicador. El proceso destino está identificado por su rank dentro del comunicador. Para enviar un mensaje se usa:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

Y para recibirlo:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Estas funciones no regresan (es decir, se bloquean) hasta que finaliza la comunicación. Simplificando un poco, esto significa que el búfer pasado a MPI_Send () se puede reutilizar, ya sea porque MPI lo guardó en algún lugar o porque ha sido recibido por el destino. De manera similar, MPI_Recv () regresa cuando el búfer de recepción se ha llenado con datos válidos.

De este modo, los procesos se sincronizan. El bloqueo de ambos procesos espera hasta que se complete la transacción.

Para que una comunicación sea exitosa

El remitente debe especificar un rango de destino válido.

El receptor debe especificar un rango de fuente válido.

El comunicador debe ser el mismo.

Las etiquetas deben coincidir.

Los tipos de mensajes deben coincidir.

El búfer del receptor debe ser lo suficientemente grande.

Los mensajes no se superan entre sí.

Esto es cierto incluso para envíos no síncronos.

Comunicación No bloqueante

Separa la comunicación en tres fases:

- Inicia la comunicación sin bloqueo.
- Trabaja (quizás involucrando otras comunicaciones).

- Espera a que se complete la comunicación sin bloqueo.

Para enviar un mensaje:

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Para recibirlo:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype
datatype, int src, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Comunicación colectiva

Todas las operaciones colectivas son bloqueantes. No hay etiquetas. Los búferes que reciben deben ser exactamente del mismo tamaño. Son comunicaciones que involucran un grupo de procesos que son llamados por todos los procesos en un comunicador. Algunos ejemplos de comunicación colectiva incluyen los siguientes:

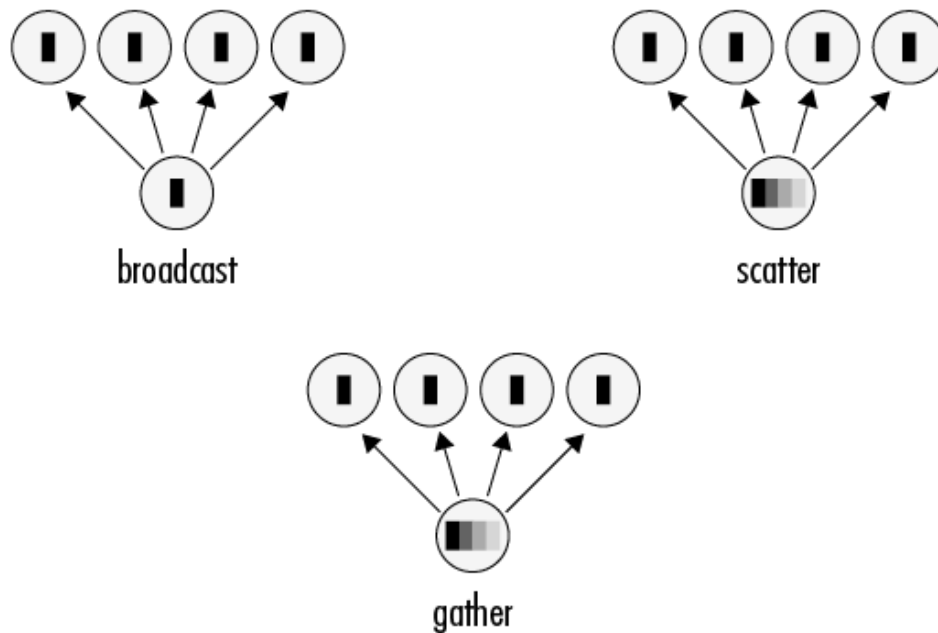
```
#BROADCAST
int MPI_Bcast(void *buffer, int count, MPI_Datatype
datatype,
int root, MPI_Comm comm)

#SCATTER
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf,
int recvcnt, MPI_Datatype recvtype,
int root, MPI_Comm comm)

#GATHER
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf,
int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm
comm)
```

La figura 9 permite ejemplificar qué se obtiene del resultado de cada una de estas operaciones.

Figura 9. Distribución de los mensajes para las funciones colectivas (*broadcast* significa transmitir el mismo mensaje a varios receptores; *scatter*, repartir el contenido de los mensajes y *gather*, agrupar en uno los mensajes de varios emisores)



Por otro lado, para sincronizar

Existe la función `MPI_Barrier`; esta función bloquea la función que llama hasta que todos los miembros del grupo se han comunicado.

```
int MPI_Barrier(MPI_Comm comm)
```

Operaciones de Reducción Global

Este tipo de operaciones se utilizan para calcular un resultado que involucra datos distribuidos en un grupo de procesos. Algunos ejemplos son:

- Suma global o producto.
- Máximo o mínimo global.
- Operación global preestablecida por el usuario.

Por ejemplo, en términos de código podemos escribir la función como:

```
MPI_Reduce(&x, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

Suma todos los valores de *x*, y almacena el resultado en la variable *result* que se encuentra en el proceso 0. Existe un listado enorme de funciones como `MPI_SUM`, entre ellas:

MPI Name	Function
<code>MPI_MAX</code>	Máximo
<code>MPI_MIN</code>	Mínimo
<code>MPI_SUM</code>	Sum
<code>MPI_PROD</code>	Product
<code>MPI LAND</code>	Logical AND
<code>MPI_LOR</code>	Logical OR
<code>MPI_LXOR</code>	Logical exclusive OR
<code>MPI_MAXLOC</code>	Maximum and location
<code>MPI_MINLOC</code>	Minimum and location

Comentarios finales

MPI es una librería de transmisión de mensajes estandarizada y portable. Fue diseñado por un grupo de investigadores de la academia y la industria que funcionan en una amplia variedad de arquitecturas de computación paralela. El estándar define la sintaxis y la semántica de un núcleo de librerías *runtime*. Es útil para una amplia gama de usuarios que escriben programas en C, C++ y Fortran. Hay varias implementaciones recientes y bien probadas de MPI, en Python también, muchas de las cuales son de código abierto o de dominio público como MPI4PY, por ejemplo.

GPU

Inicialmente, estas placas fueron diseñadas para tareas gráficas. Las GPU aportan más transistores en la unidad de cálculo en lugar de controlar y almacenar en caché. Por este motivo, las GPU son más potentes que la CPU y los dispositivos basados en CPU con respecto a la capacidad computacional. Mientras tanto, para satisfacer el alto rendimiento de datos que exigen los gráficos, el ancho de banda de la GPU es mucho mayor que el de la CPU desde sus inicios.

Como la GPU está diseñada con muchos núcleos desde su inicio, es bastante eficaz para utilizar el paralelismo y el *pipeline*. Tiene muchas ventajas sobre las CPU de un solo núcleo y las de varios núcleos.

La *peak theoretical floating-point performance* de la GPU es un orden de magnitud mayor que la de las CPU. Se ha utilizado ampliamente la enorme potencia informática de las GPU para cómputo de alto rendimiento (HPC). Entre las supercomputadoras más rápidas del mundo, se encuentran la supercomputadora Tianhe-1A de China y la supercomputadora TITAN de Oak Ridge Laboratory que dependen en gran medida de las GPU para mejorar el rendimiento. La GPU proporciona un nuevo y enorme potencial de desarrollo HPC.

Existen distintas formas de implementar la programación de placas GPU. Una muy conocida es CUDA, cuyas siglas significan Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo). Es una plataforma de [computación en paralelo](#) incluyendo un [compilador](#) y un conjunto de herramientas de desarrollo creadas que permiten usar una variante del [lenguaje de programación C](#) para escribir algoritmos en [GPU](#) de la empresa nVidia. Por medio de [wrappers](#), se pueden usar [Python](#), [Fortran](#) y [Java](#) en vez de C/C++. Funciona en todas las GPU nVidia de la serie G8X hacia adelante, incluyendo GeForce, Quadro, ION y la línea Tesla.

La programación directa en estos dispositivos excede el alcance del presente texto, pero no queríamos dejar de mencionar que esta es una opción posible más que puede resultar muy conveniente. Nuevamente, frameworks como Tensorflow pueden aprovechar las capacidades de las GPU a distintos niveles, y ser aprovechados de manera indirecta.

Resumen

Comprender los detalles de arquitectura del computador permiten programar de manera más eficiente y comenzar a hacer un uso real de las características

y capacidades disponibles de los sistemas para poder hacer realmente cómputo de alto rendimiento. Así, este capítulo recorre desde los fundamentos básicos dentro de una computadora hasta las características de sistemas grandes con el objetivo de poder aprovechar realmente las características disponibles en cada escala.

Hands-on: programación en entornos paralelos

1. Utilizando OpenMP y las directivas de compilador adecuadas:
 - Proponemos escribir un programa de “hola mundo” que imprima en la salida estándar cuántos procesos se ejecutan y el `thread_ID` para cada uno. Se pueden utilizar C o Fortran.
 - Crear un programa para transponer matrices de 3x3. Optimizar los loops de manera apropiada.
2. Utilizando MPI y una adecuada elección en la comunicación, proponemos implementar:
 - Una tarea tipo ping pong entre dos ranks distintos.
 - Un intercambio en forma de topología de anillo entre distintos ranks.

Bibliografía

- Bakos, J. D., *Embedded Systems*, Morgan Kaufmann, 2016, “Multicore and data-level optimization: OpenMP and SIMD”.
- Giroto, I., *Shared Memory Programming Paradigm*, ICTS / ICTP, 2016, <<http://indico.ictp.it/event/7659/>>.
- Gropp, W., E. Lusk y A. Skjellum, *Using MPI-2. Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1999.
- Guía de Compiladores para OpenMP, <<https://www.openmp.org/resources/openmp-compilers-tools/>>.
- Guía de referencia para las directivas, <<https://www.openmp.org/resources/refguides/>>.
- Harris, S. y D. Harris, *Digital Design and Computer Architecture: ARM Edition*, San Francisco, Morgan Kaufmann Publishers Inc., 2015.

Hennessey, J. L. y D. A. Patterson, *Computer Architecture. A Quantitative Approach*, San Francisco, Morgan Kaufmann Publishers Inc., 2011.

Pacheco, P. S., *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011, “Parallel Hardware and Parallel Software”.

Parthasarathi, R., *Computer Architecture*, INFLIBNET Centre, s/f, <<http://www.cs.umd.edu/~meesh/411/CA-online/index.html#main>>.

Referencia para las funciones de OpenMP, <<https://www.mpich.org/static/docs/latest/www3/Constants.html>>.

Rosenberg, J., “Security in embedded systems”, en Vega, A., P. Bose y A. Buyuktosunoglu (eds.), *Rugged Embedded Systems*, Morgan Kaufmann, 2017.

Shipley, C., S. Jodis, “Programming Languages Classification”, en H. Bidgoli (ed.), *Encyclopedia of Information Systems*, Elsevier, 2003.

Stanford University, <<https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html>>.

Tan, Y., *GPGPU: General-Purpose Computing on the GPU*, Morgan Kaufmann, 2016, “Gpu-Based Parallel Implementation of Swarm Intelligence Algorithms”.

Capítulo 7

Ejemplos de trabajo en el ámbito del alto desempeño

Pablo Alcain, Cecilia Jarne, Rodrigo Lugones y María Graciela Molina

En este breve capítulo final, mostraremos tres ejemplos de los diversos que existen de software desarrollado teniendo en cuenta los distintos aspectos del cómputo de alto rendimiento. En el primer capítulo hemos visto cómo las distintas librerías científicas con interfaces en Python en su core están hechas con código compilado y cómo esto permite aprovechar las características de los procesadores o sistemas multicore.

Presentaremos aquí algunos ejemplos de software, entonces, para observar cómo los distintos aspectos del diseño, así como la filosofía del uso del código abierto, pueden ser utilizados exitosamente en aplicaciones científicas específicas. Esta selección es simplemente un recorte, pero en cada ejemplo elegido queremos resaltar los distintos aspectos vistos a lo largo de los capítulos de este libro.

Quantum ESPRESSO

Es un framework para los primeros principios de cálculo de estructuras electrónicas y modelado de materiales. Es un software libre bajo la Licencia Pública General GNU. Se basa en la teoría funcional de la densidad, conjuntos de bases de ondas planas y pseudopotenciales (tanto conservadores de normas como ultrablandos). ESPRESSO es un acrónimo para opEn-Source Package for Research in Electronic Structure, Simulation and Optimization.

El núcleo del proyecto es un software llamado PWscf, que existía anteriormente como independiente. PWscf (Plane-Wave Self-Consistent Field) es un conjunto de programas para el cálculo de estructuras electrónicas dentro de la teoría funcional de densidad y la teoría de

perturbación funcional de densidad, que utiliza conjuntos de bases de ondas planas y pseudopotenciales.

Quantum ESPRESSO es una iniciativa abierta, del Centro Nacional de Simulación CNR-IOM DEMOCRITOS en Trieste (Italia) y en colaboración con diferentes centros internacionales como el MIT, la Universidad de Princeton, la Universidad de Minnesota o la École Polytechnique Fédérale de Lausana. El proyecto está coordinado por la fundación Quantum ESPRESSO, formada por numerosos centros y grupos de investigación de todo el mundo. La primera versión, llamada pw.1.0.0, fue lanzada el 15 de junio de 2001.

El programa, escrito principalmente en Fortran-90 con algunas partes en C o en Fortran-77, fue construido a partir de la fusión y reingeniería de diferentes paquetes desarrollados independientemente, además de un conjunto de paquetes diseñados para ser interoperables con los componentes centrales, que permiten la realización de tareas más avanzadas.

Los componentes principales de Quantum ESPRESSO están diseñados para explotar la arquitectura de las supercomputadoras actuales, que están caracterizadas por múltiples niveles y capas de comunicación entre procesadores. La paralelización se logra utilizando la de MPI y OpenMP (descriptas en el capítulo 6), lo que permite que los códigos principales de la distribución se ejecuten en paralelo en la mayoría o en todas las máquinas paralelas con muy buen rendimiento.

Las diferentes tareas que se pueden realizar con este software incluyen:

- Cálculos del estado fundamental.
- Optimización estructural.
- Estados de transición y trayectorias de energía mínima.
- Propiedades de respuesta (DFPT), como frecuencias de fonones, interacciones electrón-fonón y desplazamientos químicos de EPR y RMN.
- Propiedades espectroscópicas.
- Generación de pseudopotenciales.

La documentación general puede encontrarse en el sitio del proyecto y cubre la instalación y el uso de la versión estable actual de Quantum ESPRESSO.

Funciona en casi todas las arquitecturas actuales: desde grandes máquinas paralelas (IBM SP y BlueGene, Cray XT, Altix, Nec SX) hasta estaciones de

trabajo (HP, IBM, SUN, Intel, AMD) y PC con Linux, Windows, Mac OS-X, incluidos los clústeres de procesadores Intel o AMD de 32 o 64 bits con varias conectividades (Gigabit Ethernet, Myrinet, Infiniband...). Aprovecha al máximo las librerías matemáticas como MKL para CPU Intel, ACML para CPU AMD, ESSL para máquinas IBM.

LAPACK

LAPACK (Linear Algebra Package) es una biblioteca de software estándar para los cálculos de álgebra lineal numérica. Proporciona rutinas optimizadas para resolver sistemas de ecuaciones lineales y cuadrados mínimos lineales, problemas de autovalores y descomposición en valores singulares. También incluye rutinas para implementar las factorizaciones matriciales asociadas como la descomposición LU, QR, Cholesky y Schur. LAPACK se escribió originalmente en Fortran 77, pero se trasladó a Fortran 90 en la versión 3.2 (2008). Las rutinas manejan matrices tanto reales como complejas con precisión simple y doble.

LAPACK fue diseñado como el sucesor de las ecuaciones lineales y las rutinas lineales de cuadrados mínimos de LINPACK y las rutinas de valores propios de EISPACK. LINPACK, escrito en las décadas de 1970 y 1980, fue diseñado para ejecutarse en las entonces modernas computadoras vectoriales con memoria compartida. LAPACK, por el contrario, fue diseñado para explotar eficazmente los cachés en arquitecturas modernas basadas en caché y, por lo tanto, puede ejecutar órdenes de magnitud más rápido que LINPACK en tales máquinas, dada una implementación BLAS bien ajustada. LAPACK también se ha ampliado para ejecutarse en sistemas de memoria distribuida en paquetes posteriores como ScaLAPACK y PLAPACK.

Netlib LAPACK tiene una licencia de estilo BSD de tres cláusulas, una licencia de software libre permisiva con pocas restricciones.

OpenBLAS

OpenBLAS es una implementación de código abierto de BLAS (subprogramas de álgebra lineal básica) y las API de LAPACK con muchas optimizaciones hechas manualmente para tipos de procesadores específicos. Se desarrolla en el Laboratorio de Software Paralelo y Ciencias Computacionales (ISCAS).

OpenBLAS agrega implementaciones optimizadas de kernels de álgebra lineal para varias arquitecturas de procesador. Además, logra un rendimiento comparable al Intel MKL; esto se cumple principalmente en la parte BLAS, mientras que la parte LAPACK se queda atrás.

OpenBLAS es una bifurcación de GotoBLAS2, que fue creada por Kazushige Goto en el Texas Advanced Computing Center.

Bibliografía

BSD, <<http://www.linfo.org/bsdlicense.html>>.

Giannozzi, P. *et al.*, “QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials”, *Journal of Physics: Condensed Matter*, vol 21, N° 39, 2009.

ISCAS,

<https://en.wikipedia.org/wiki/Institute_of_Software,_Chinese_Academy_of_Sciences>.

LAPACK, <<http://www.netlib.org/lapack/>>.

Quantum ESPRESSO Foundation, <<http://www.quantum-espresso.org/project/what-can-qe-do>>.

Spiga, F. e I. Girotto, “phiGEMM: a CPU-GPU library for porting Quantum ESPRESSO on hybrid systems”, *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, IEEE, 2012.

Bibliografía general

- Alcain, P. N., C. G. Jarne, R. Lugones, M. G. Molina, “Workshop en Técnicas de Programación Científica: Enseñando a desarrollar software colaborativo para la ciencia”, *CET. Revista de ciencias exactas e ingeniería*, N° 39.
- Chapman, B., *Using OpenMP. Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, 2007.
- Friedl, J. E. F., *Mastering Regular Expressions*, O’Reilly, 2006.
- Harbison, S. P. y G. R. Steele, C. *A Reference Manual*, Londres, Pearson, 2002.
- Kernighan, B. W. y D. M. Ritchie, *The C Programming Language*, Londres, Metcalf, M. et al., *Modern Fortran Explained*, Oxford, 2018.
- Pearson, 1988.
- Pacheco, P., *Parallel Programming with MPI*, Morgan Kaufmann, 1986.
- Sedgewick, R., *Algorithms in C. Parts 1-4. Fundamentals, Data Structures, Sorting, Searching*, Londres, Pearson, 1998.
- , *Algorithms in C. Part 5. Graph Algorithms*, Londres, Pearson, 2002.
- Stevens, W. R., *Advanced Programming in the UNIX Environment*, Addison-Wesley Professional, 2013.

Ejercicios extra

Ejercicios basados en nuestro curso de posgrado y nuestros workshops:

<<http://wp.df.uba.ar/wtpc/clases/>>.

Ejercicios de ejemplo: <<https://projecteuler.net/>>.

Conversión digital: Juan I. Siwak

El cuidado de edición estuvo a cargo de Anna Mónica Aguilar, Rafael Centeno, Germán Conde, Hernán Morfese y Mariana Nemitz