Practical scientific computing

Antoine Levitt

May 4, 2021

Contents

1 Introduction

Scientific computing is the science of using computers to answer scientific questions. It is distinct from but strongly linked to modeling (reality to model), mathematical analysis (mathematical statements about models), numerical analysis (mathematical statements about numerical methods) and computer science (general computing, not necessarily applied to science). Scientific computing is used in virtually all domains of science, from its traditional roots in physics and mechanics to applications in economics, biology, and humanities.

These notes are not intended to be a definite resource, but rather a short introduction to some of the practical aspects of scientific computing. For each class of problem (linear systems, nonlinear systems, optimization, differential equations...), I aim to concisely present the most basic methods (Richardson iteration, Newton method, gradient descent, explicit Euler...), some elements of their analysis, and pointers to the more sophisticated methods that should be used in practice.

Recommended reading to go further. Generic resources

- The short paper at `https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745` on best practices for scientific computing
- Wikipedia (English version)
- The "Numerical recipes" book series

Specialized in-depth textbooks:

- For linear systems and numerical linear algebra, Iterative Methods for Sparse Linear Systems by Saad, or Numerical Linear Algebra by Trefethen & Bau
- For nonlinear systems, Solving Nonlinear Equations with Iterative Methods by Kelley
- For optimization, Numerical Optimization by Nocedal & Wright
- For time integration, Solving Ordinary Differential Equations by Hairer & Wanner

2 Hardware

At the very basic level is the transistor. Information is moved around electronic circuits as voltage, a positive voltage corresponding to 1 and a zero voltage to 0. Transistors allow switchable electronics and therefore logic gates (AND, OR, NOT...), which take some input voltages and yield output voltages. Logic gates can be combined to form more complex circuits (for instance, adding numbers). Processors (central processing unit, CPU) have a number of fixed circuits which they use to read, decode and execute programs written in executable files (machine code). As a first approximation, processors execute an instruction each cycle of a clock, typically in the gigahertz range[1].

The machine code is assembly language, and consists of simple instructions (take this value at this memory location, add it with this other value, store at this memory location, if this value is zero jump to this piece of code...) which are encoded in instructions. For instance, in the x86 instruction set used in Intel processors, the number 0x4F means "decrement the register EDI"[2]. Assembly language is architecture-dependent: your laptop most likely uses the x86 architecture, and your smartphone the ARM architecture. It is also unpleasant to write, and so we use programming languages that get compiled into machine code.

Information is stored in several places which vary in permanence and access speed. Very close to the processor are registers, which are very small (eg in a typical x86 processor, 16 registers of 64 bits each). The processor operates directly on them, and reading/writing from it takes one CPU cycle. Next is the RAM, which is erased when the computer is turned off and much bigger

[1] The reality is much more complicated because 1) not all operations take the same number of cycles to execute 2) even on plain serial machine code processors try to do things in parallel to go faster. This, vectorization and multithreading mean that frequency is not a meaningful metric of performance.

[2] The prefix "0x" means hexadecimal notation, from 0 to $F = 15$. Sor for instance, $0x4F = 4 \times 16 + 15$.

(several gigabytes) [3]. Accessing it is slower, on the order of hundreds of cycles[4]. Then is permanent storage on hard drives (terabyte range) which is much slower to access, on the order of million cycles. Finally, remote access to files over network is even slower.

A modern computer consists of a CPU with several cores (execution units that can work in parallel), RAM, a hard drive, a video card (graphical processing unit, GPU), and various peripherals for connectivity and user interaction. A recent trend is to use specialized hardware like GPU or FPGA (programmable electronics). Such hardware can be more powerful for specialized applications[5], but more cumbersome to program (needing special, often proprietary software), harder to use efficiently, and subject to fashions. Supercomputers are most often simply "scaled up" versions of regular computers, with very fast network between them[6].

3 Software

3.1 Software environment

You are most likely going to spend a large amount of time in your life writing text (code, reports...), so it makes sense to invest (a reasonable amount of) time in the tools that allow you to do this efficiently. Learn to type quickly. The mouse is inefficient, avoid using it if you can: learn and use keyboard shortcuts (not just copy/paste). For code, use either a good multipurpose editor (emacs, vim, vscode...) or a language-specific IDE (integrated development environment). Customize it. If you have a PC with Windows, consider moving to Linux. Do not be afraid of the command line.

3.2 Programming languages

All reasonable ways of writing programs (as well as several unreasonable ones, such as PowerPoint or Minecraft) are "Turing-complete", meaning that they can all compute everything that is computable. However, some are better suited to some tasks than others: you would not (usually) program an operating system using Python, and you would not write a website in C. The following summary is a vast oversimplification, skips over many distinctions and reflects the personal biases of the author.

Probably the most important distinction between languages is the level of abstraction. Roughly, by increasing order of abstraction:

- Assembly language, which operates on registers and uses jumps for control flow.
- C, which abstracts hardware-specific details and defines the notion of functions, structures, loops, etc, and compiles to assembly code directly.
- C++, (mostly) an extension of C which defines higher-level concepts such as classes, virtual methods, etc, but is still compiled and leaves memory management to the programmer.
- "Managed" languages such as Java or C#, which are similar to C++ but take care of memory allocations, deallocations (*garbage collection*) and bound checking by themselves, and introduce yet more abstractions; these languages are compiled into an (architecture-independent) intermediary representation that is then executed by a runtime.
- "Dynamic" languages such as Python that do not need a precompilation step, and are usually less strict about types.
- Finally and somewhat separate from the others, special-purposes languages, tools and frameworks that are tailored towards specific applications (bash for scripts, Javascript for the web, Mathematica for symbolic mathematics, Matlab/Scilab for numerical mathematics, R for statistics, TensorFlow for machine learning...).

[3] There is actually an intermediate zone between registers and RAM: the different levels of caches, of kilobyte to megabyte size, which are used by the processor to avoid accessing the RAM too much. This has important performance implications: all else being equal, it is better to access memory in predictable, contiguous patterns, in order to allow the caches to do their jobs.

[4] See `http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/` for the cost of each operation.

[5] For instance, neural networks are trained using GPUs, and bitcoins are mined with FPGAs

[6] At the time of writing, the most powerful supercomputer in the world, Fugaku in Japan, consists of about 150,000 CPUs, each with 48 cores. This enables it to attain performance in the order of the exaflop (10^{18} floating-point operations per second).

Another consideration is performance. It is tempting to say that the higher one goes in abstraction, the slower the language becomes. This used to be true, with a clear distinction between compiled (fast, static, low-level) and interpreted (slow, dynamic, high-level) languages: each feature that made the language safer and easier to use (automatic memory management, bounds checking, dynamic types) also made it slower. However, advances in programming language and compiler technology have blurred the lines somewhat. Furthermore, the performance-critical parts of libraries in higher-level languages are usually written in lower-level languages for performance, and a naively hand-coded C program will probably be slower than a Python code that uses optimized libraries to achieve the same task. Performance usually depends more on the algorithm and adherence to the performance good practices for a given language than on the language itself (e.g. in Python, do not use loops but use numpy vectorized operations).

Which language to use depends on the context. When you finish your studies, you will probably have to use whatever language other people around you use. If you have the choice of language, the main rule is: use the highest-level language appropriate for the task at hand. High-level languages enable more productive workflows, maximize code reuse and minimize bugs.

For a good education in programming, learn one language to the point of being reasonably comfortable writing code in it; for most of you, that will probably be Python. Then, learn a low-level language like C++ (or, preferably, C, which is purer and clearer, although less powerful), to see what is going on underneath. Learn about object orientation, and why it is not the only programming paradigm possible. Look at Javascript. Broaden your mind with weird languages you are never going to use like assembly, LISP or OCaml. Most importantly, of course, practice. Do not worry too much about languages: once you have a good grasp on programming itself (in whichever language), you will be able to pick up any new language you come across.

3.3 Environment for this course

This course will use the **Julia** programming language. Julia is a relatively new language that is easy to read and write, and is especially well-suited for scientific computing. Unlike Python, it has first-class support for linear algebra and (when properly handled) is as fast as low-level languages[7]. A good tutorial in notebook form can be found at `https://github.com/JuliaAcademy/JuliaTutorials/tree/main/introductory-tutorials/intro-to-julia/` (video version at `https://www.youtube.com/watch?v=8h8rQyEpiZA`). Regarding workflow, there are three options:

- The low-tech version is to use julia at the command line with a text editor. Use the `include` command to include text files, and the Revise package to faciliate development of multi-file projects.
- IDE fans should use vscode (see installation instructions at `https://www.julia-vscode.org/`).
- Jupyter notebooks can be used with the IJulia extension.

Exercises:

1. Compare the time it takes to sum a vector of double-precision floating-point numbers in python either with a hand-written loop or with the numpy `sum` function. Do the same in Julia.

3.4 Software versioning

Once upon a time there used to be several tools for this. Now there is just one, so learn the basics of how to use Git. Git itself has many features that can seem confusing: at the beginning, do not worry about branches and focus on add, commit, push, pull. Use stashes and resets when something goes wrong. Use the command line directly or graphical interfaces. Version everything: code, latex reports... Make backups of everything that is not versioned. When collaborating with others, use github, gitlab, overleaf.

[7] See `https://julialang.org/blog/2012/02/why-we-created-julia/` for the motivations behind Julia.

3.5 Code organization

Good code organization has the following goals: help understand and extend the code, minimize bugs, facilitate collaboration, maximize code reuse.

- Use good and consistent style (indentation, spacing, etc.). Use an appropriate editor to help you with this.
- Code (variable names, comments) is in English.
- Reuse code to the maximum reasonable extent. Avoid copy-pasting like the plague. A lot of bugs result from copy-pasting and then forgetting to change a variable name in one place. Copy-pasting also means that if you change something in one place you have to change it in the other.
- Hidden state (e.g. global variables), nonlocal effects and mutation makes it harder to find bugs and change the code. Try to make your functions not have side effects.
- Avoid the global scope (scripts), and use functions.
- Split large functions.
- In modern languages, pass functions to other functions. Use closures to pass parameters around. For instance,

  ```
  f(x) = my_function(x, parameters) # this defines a closure
  solve_equation(f, x0)
  ```

- Comment to describe structure, purpose, conventions or non-trivial mechanisms. Do not paraphrase code that is easy to understand.
- Name things consistently. Use distinctive index names (eg if solving a PDE, do not use `i` sometimes for a space index, sometimes for a time index).
- The best line of code is the one you don't write. Less code, less bugs.
- Try to be consistent about data structures and representation of objects. If you need to perform conversions, write clearly-named functions that do this conversion
- Test as early as you can. Keep the tests around.
- When you refactor your code, work by small steps, and test after each change.
- Always begin with the simplest version of the code and grow from there instead of beginning with the most general and having to simplify it anyway when you encounter bugs (which you will).
- Be careful about predicting the future evolution of your code. It is better to have a simple code that you can later modify than to plan carefully for a generalization that you then decide is not a good idea anyway.

3.6 Debugging

It is an unavoidable fact of life that a lot of programming time is spent debugging. Bugs should not be thought of as mistakes or personal character faults, but as the unavoidable byproduct of programming. Good programming is not about programming without bugs on the first try, it's about debugging and testing code in order to be confident it works. Code that is not tested must be assumed not to work by default.

A couple of tips:

- Always have the possibility of bugs on your mind. When something does not work as expected, make sure there are no bugs before looking for other explanations.
- Use defensive programming. For instance, if you compute a quantity that you know should be positive, assert that it is the case, so you get an error if it is not.

- When there are two ways of computing the same quantity, check that both match.
- Decouple your code as much as possible. For instance, if you write a nonlinear solver that solves $F(x) = 0$, make it a function that accepts another function `solve(F, x0)`, rather than specializing to the function you want to solve. This will allow you to test the nonlinear solver independently.
- Always first test your code on a simple problem for which you know the result. On this problem, adjust your method, numerical parameters, etc. Only then can you move on to interesting problems for which you do not know the result.
- Use reasonable values for the parameters of your model (ones that do not result in very fine structures, for instance) and for the numerical parameters. Although in mathematics we use ε going to zero and N going to infinity, in numerics as in physics, there is no such thing as a small or large number: small or large with respect to what?
- Simplify your problem to the maximum. Find the simplest problem that still has the issue you are encountering.
- Do not attempt to debug two bugs at once. If you have multiple issues, find the one that appears simplest to fix, fit it and come back to the more complicated ones.
- Do not attempt to debug the math and the code at the same time.
- Do not code at random until you get the desired result. If something you write appears to work but you are not sure why, do not accept it and move on, because there is a 90% chance it will cause trouble later. Then, not only will you have to understand it anyway, but you will have to do so while debugging another unrelated issue.
- Do not trust yourself, and do not try to interpret code in your head. Bugs are not found by looking at code very hard, they are found by making sure that everything that should be true is true, and therefore by elimination finding the thing that should be true but is false.
- Use debuggers rather than print statements.
- Explore at the command line (REPL)

Exercises:

1. You want to send people on the moon. You have coded a simulator, very carefully checked the physics, the formulas and used the same input (initial conditions, fuel planning, etc.) as a past successful mission. However the simulator insists on sending the astronauts to the center of the sun: there is a bug somewhere in the code. What do you do to find it?

3.7 Performance

The first rule of performance optimization is: don't do it. Optimizing code makes it less readable and maintainable, and more prone to bugs. Remember Knuth's quote: "premature optimization is the root of all evil". The performance critical parts of a code are a small part of the overall code: isolate it, and optimize only this.

The performance optimization flowchart follows. Between each of these steps, remember to make sure the code is correct, and measure the impact of each change.

1. Commit your code before optimizing it.
2. Isolate a reproducible example that is representative of your actual use case.
3. Do you really need to improve performance? Remember that human time is more valuable than computer time.
4. Use a profiler to know what in your code actually takes time to run. Profilers you how much time is spent in every function and line of code, and therefore where the bottlenecks (places where the program spends the most time) are. Humans are extremely bad at guessing where time is spent: do not trust your intuition and measure. Do not spend time fixing performance of code that is not a bottleneck. Profile after any modification to your code.

5. What is the complexity of the algorithms you use? Can you use a more efficient algorithm? If applicable, can you use a more efficient discretization scheme? A better representation of your data? Try changing it.

6. Are you using a high-level programming language? If yes, make sure you're using it as it should be used (don't write loops in Python/Matlab/R, write type-stable code in Julia, etc.). Read performance advice for your language.

7. Look at how people solving problems similar to yours do it. Are there libraries available to do parts of what you're doing? If so, can you replace some of your code with calls to these libraries?

8. Try common-sense optimizations: precompute expensive computations, fuse operations that can be fused, etc. At each stage, measure and profile to check whether the modifications that you have made have an impact. If they don't, don't make them.

9. If using Python/Matlab/R and if performance really is an issue, rewrite the performance-critical portions in a lower-level language.

10. If at this point the performance is still unsatisfactory, you are entering dangerous territory. Learn about multithreading, caches, vectorization. Experiment, measure and profile.

There are two main factors that govern the performance of a well-optimized code: the raw processing power, and the memory access. Codes that have high *arithmetic intensity* (FLOPs per memory access) are limited by processing power and are said to be compute-bound; those limited by memory access are memory-bound. Achieving optimal performance (being compute-bound) is very hard because it requires problems that have many FLOPs per data (typically, matrix-matrix multiplication, but not matrix-vector multiplication) and optimized codes that make good use of caches. Most algorithms will only achieve a small fraction of peak performance, no matter how well they are optimized.

Exercises:

1. If you have written code for a past assignment, run a profiler on it.

2. Benchmark a performant implementation of the `sum` function for different array sizes, and plot the time per element as a function of size. Compare with the cache sizes of your processor (on Linux, use the `lscpu` command). Compute the performance in FLOPs and compare with the peak performance of your computer as achieved by matrix multiplication (in Julia, do `BLAS.set_num_threads(1); peakflops()`)

3.8 Libraries

In programming, laziness is a virtue. If you want to use a standard algorithm, there is likely already an existing good implementation of it somewhere: use it. Using good-quality libraries results in code that is faster, more automatic and more robust than what you could do yourself. Even some very simple algorithms turn out to be very hard to implement in an optimal way: for instance, hand-coded matrix multiplication using the simple triple loop algorithm, even coded in the best way possible, is much slower than optimized implementations, which use a number of tricks to optimize cache usage.

Trust libraries that are widely used or that have been around for decades; do not blindly trust code posted on a blog post somewhere on the internet. Read the documentation of the libraries you use. Prefer open source libraries, both on philosophical and practical grounds. If you find a bug, a typo or a confusing explanation in an open source library, contribute back to it!

Exercises:

1. Benchmark a hand-coded square matrix multiplication algorithm in a performant programming language against a library version. Discuss.

4 Floating point arithmetic

Science often deals with continuous numbers, but computers understand only sequences of bits: reals have to be discretized in some way. The format ubiquitously used is floating-point numbers (by opposition to fixed-point numbers, which are only used in specific hardware), which are codified as IEEE754. The most important thing to know is that floating-point numbers use a *relative* representation. To store a number on a computer, one writes it in the form $x \times b^n$, where x is the mantissa, b is the basis (on computers, $b = 2$) and n is an integer (the exponent). Both x and n are discretized using a finite number of bits[8].

There are two formats in common use and widely supported in hardware: 32-bits simple precision (`float` in C) and 64-bits double precision (`double` in C). The double-precision format uses 11 bits for the exponent and 53 bits for the mantissa. The exponent can go roughly from -2^{10} to 2^{10}: this is enough to represent numbers from about 10^{-300} to 10^{300}, and the exponent is not a serious limitation in practice. More interesting is the mantissa limitations, which means that there is no difference between 1 and $1 + \varepsilon$, where ε is roughly $2^{-53} \approx 10^{-16}$. In single precision, the number of bits for exponent and mantissa are 8 and 24; in particular, $\varepsilon \approx 10^{-8}$.

The limitation in the mantissa means that numbers can be represented up to a *relative* precision of roughly ε. In other words, if $\mathrm{fl}(x)$ is the floating-point representation of the real x, we have, for all x in a very large range,

$$\mathrm{fl}(x) = x + O(\varepsilon|x|).$$

Elementary arithmetic between floating-point numbers effectively behaves as if the correct result was computed, then rounded to the nearest representable number: performing $x+y$ on a computer results in $\mathrm{fl}(\mathrm{fl}(x) + \mathrm{fl}(y))$. This means that floating-point arithmetic is commutative, but not associative: on a computer, $(a + b) + c$ computes $\mathrm{fl}(\mathrm{fl}(\mathrm{fl}(a) + \mathrm{fl}(b)) + \mathrm{fl}(c))$ and rounds it to the nearest number, then adds c and rounds it to the nearest number, which is not the same as $a+(b+c)$. A rule of thumb is that unless you are manipulating numbers of very different magnitude, the result you get will have a relative accuracy of $O(\varepsilon)$. On the other hand, if you subtract numbers that are very close, you will get a poor relative accuracy on the result.

Exercises:

1. Compute $\sqrt{2}^2 - 2$ in various floating point formats.
2. Compare $(1 - 1) + 10^{-30}$ and $1 + ((-1) + 10^{-30})$
3. Consider the finite difference approximation $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$. What h gives the best result for $f'(x)$? Do this analysis both theoretically and numerically. How can you improve on this?
4. Explain why the following gives a virtually exact result for $\sin'(x)$:

   ```
   eps = 1e-100
   imag(sin(x+eps*im))/eps
   ```

   Hint: for analytic functions $f(z) = u(z) + iv(z)$ of a complex variable $z = x + iy$, the Cauchy-Riemann equations $\partial_x u = \partial_y v, \partial_y u = -\partial_x v$ hold.
5. Consider the function $\mathrm{logsumexp}(x, y) = \log(e^x + e^y)$, which acts as a smooth approximation of the max function. Plot it and the max function. What possible issue can arise in floating point arithmetic? For what kind of inputs? How can you fix it?
6. Without solving explicitly the polynomial, find the approximate location of the roots of $\varepsilon x^2 + x + 1 = 0$ in the asymptotic limit $\varepsilon \to 0$. What happens when using numerically the explicit formulas for the roots of a degree 2 polynomial for ε small? How can you fix it?

[8] There is also a special encoding for non-standard numbers. For instance, $1/0$ is Inf (positive zero is distinct from negative zero, so $1/(-0.0)$ is -Inf). Undefined operations such as $0/0$ or Inf−Inf result in NaN: "Not A Number". Computations that "explode" usually end up producing NaNs. The IEEE754 format also has a number of more exotic features, including signaling NaNs and subnormal numbers, that are usually best left alone.

5 Conditioning and stability

5.1 Matrix algebra, singular and eigenvalues

We measure the space $\mathbb{R}^N$ with the Euclidean metric $\|x\|^2 = \sum_{n=1}^N |x_n|^2$. All the notions we develop depend on the metric used[9]. We will deal exclusively with the reals; the extension to complex numbers are usually just a matter of replacing transposes by adjoints (transconjugates).

Let A be a $N_1 \times N_2$ matrix, and $r \leq \min(N_1, N_2)$ be its rank. The (compact) singular value decomposition (SVD) of A is

$$A = USV^T.$$

where U (resp. V) is the $N_1 \times r$ (resp. $N_2 \times r$) matrix with orthonormal columns of the left (resp. right) singular vectors, and S is the diagonal $r \times r$ matrix of singular values. The SVD exists and is essentially unique (up to reordering and subspace transformations) for any matrix A. Here is the very short proof, in the case where $r = N_2$ for simplicity. The matrix $A^T A$ is symmetric positive definite, hence can be diagonalized as $A^T A = VS^2V^T$ with $V^T V = 1$. Let $U = AVS^{-1}$: we have $A = USV^T$ and $U^T U = S^{-1}V^T A^T A = VS^{-1} = 1$.

In contrast to the eigenvalue decomposition, which is an algebraic statement about the vectors that are left invariant with respect to the transformation A, the SVD is a statement about the growth in the norm of various vectors. Note in particular that the singular values are not intrinsic to A but depend on the metric (hidden in the transpose, which implicitly assumes the canonical inner product). There is no specific relationship between the eigenvalue and singular decomposition, unless A is symmetric (in which case the singular values are the absolute value of the eigenvalues). The right singular vectors v_n are such that $\|Av_n\| = s_n$, and so give the directions in which A is large or small. They often contain valuable information and are a useful diagnostic tool[10].

The operator norm of a matrix is

$$\|A\|_{\text{op}} = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

This norm is often the most useful theoretically. It is equal to the largest of the singular values of A. Easier to compute in practice is the Frobenius (or Hilbert-Schmidt in the context of Hilbert spaces) norm $\|A\|_{\text{fro}} = \sqrt{\sum_{i,j} |A_{ij}|^2}$, which is equal to the Euclidean norm of the singular values of A. Both these norms are not to be confused with the spectral radius $\rho(A) = \lim_{n\to\infty} \|A^n\|^{1/n}$ (which is equal to the largest eigenvalue of A in modulus). The difference between the spectral radius and the matrix A can be dramatic, for instance on the matrix

$$\begin{pmatrix} 0.1 & 1000 \\ 0 & 0.1 \end{pmatrix}.$$

Exercises:

1. Show that

$$\rho(A) \leq \|A\|_{\text{op}} \leq \|A\|_{\text{fro}}.$$

2. Find out how to compute the SVD in your language, and check the formulas in this section.

3. Plot the dynamics of $x_{n+1} = Ax_n$ with A the matrix above. Why must one be careful when using the eigenvalues as stability criterion of a dynamical system? What do the singular values tell you about the dynamics? Why is the situation simpler when A is symmetric?

[9] In particular, while the p-norms are not particularly useful since they do not come from inner products, it will be very fruitful later to consider the change of inner product $\langle x, y\rangle_P = \langle x, Py\rangle$ where P is a symmetric positive definite matrix.

[10] For instance, singular vectors associated with small singular values v are the directions in which $Av \approx 0$ which are likely to be problematic when trying to invert or solve least-squares problems with A. In statistics, the singular vectors of a matrix representing repeated observations of several random variables capture the linear correlations between these variables, and keeping only the ones associated with the largest singular values is a powerful dimensionality reduction technique (the principal component analysis).

5.2 Conditioning and stability

Consider a (possibly nonlinear) mapping $x \to f$ from data to a quantity of interest, where $f \in \mathbb{R}^{N_f}$ and $x \in \mathbb{R}^{N_x}$. If around a given input x we make a small mistake δx, the error in the output f is (to first order) $\delta f = f'(x)\delta x$. The absolute conditioning (or condition number) of the mapping f bounds this error:

$$\kappa_{\text{abs}} = \|f'(x)\|_{\text{op}}, \text{ so that } \|\delta f\| \leq \kappa_{\text{abs}} \|\delta x\|$$

The main problem with this quantity is that it has a dimension. Is a conditioning of 1000 good or bad? It depends on the range of x and f. For this reason, a much more useful number is the (dimensionless) relative conditioning

$$\kappa_{\text{rel}} = \frac{\|x\|}{\|f\|} \|f'(x)\|_{\text{op}}, \text{ so that } \frac{\|\delta f\|}{\|f\|} \leq \kappa_{\text{rel}} \frac{\|\delta x\|}{\|x\|},$$

usually denoted simply by κ. Note that this relies on x and f having a nonzero magnitude, which fixes the scales to measure δx and δf: as with all relatives measures, it is less relevant if either x or f are zero. The conditioning of a mapping f gives intrinsic information on its robustness to noise in the input x, regardless of the origin of the noise.

The conditioning of an operation provides a useful way of thinking about floating-point arithmetic (as well as other sources of error, for instance, the early stopping in an iterative solver). Since all quantities are only ever known to a relative precision of ε, the output of a given mathematical function f cannot be known to a better relative precision than $\varepsilon\kappa$ (because two inputs that are ε close to each other produce outputs that are $\varepsilon\kappa$ close). Note that this is *independent* on the actual implementation of f, and is a best-case scenario. A "good" algorithm is one that achieves an error on the order of the best-case scenario $\varepsilon\kappa$: we call these algorithms *numerically backward stable*.

We emphasize that the concepts of conditioning and of backward stability are orthogonal to each other: one is the property of a problem, the second of an algorithm. There are stable algorithms to solve ill-conditioned problems, and unstable algorithms to solve well-conditioned problems. For an (approximate) algorithm to produce a reasonable answer for the solution of a problem, three conditions must be satisfied: the problem itself must be well-conditioned, the algorithm used must be stable, and the sources of error must be small. In practice, most standard algorithms are backward stable, leading to the mantra of numerical analysis, applicable in various forms in many contexts: "well-conditioning + consistency = convergence".

Exercises:

1. What is the conditioning of the mapping $f(x) = \alpha x$ for α a given scalar? What is that of $f(x) = x + \alpha$?

2. Compute the eigenvalues of the matrix

$$M = \begin{pmatrix} 1 & 1 \\ \varepsilon & 1 \end{pmatrix}$$

   for ε small, and deduce that the mapping $M \to \lambda$ is ill-conditioned. By contrast, one can show that, for symmetric matrices, $\delta\lambda \leq \|\delta M\|$.

3. Let f be the mapping from (a, b, c) to the smallest (in magnitude) root of the quadratic equation $ax^2 + b + c = 0$. Compute f to first order near $(a, b, c) = (0, 1, 1)$, and show that f is well-conditioned at that point. Is the numerical method $\widetilde{f}(a, b, c) = \frac{b - \sqrt{b^2 - 4ac}}{2a}$ stable in floating-point arithmetic?

5.3 Conditioning of matrices

If A is an invertible matrix, the relative conditioning of the mapping $x \to Ax$ is easily computed to be

$$\kappa(A) = \|A\|_{\text{op}} \|A^{-1}\|_{\text{op}}, \tag{1}$$

It is a remarkable property that $\kappa(A) = \kappa(A^{-1})$: $\kappa(A)$ is a dimensionless measure of how well-posed the problem of solving $Ax = b$ is.

There can be several sources of ill-conditioning:

- Ill-posed problem, with nearly-non-invertible matrices. In general, ill-conditioning is often a sign that the problem is close to an ill-posed one (which has no solution, or several).
- Bad choice of units: if some of the unknowns (or equations) are expressed in units of wildly differing typical magnitude (e.g. if you solve a coupled mechanics + fluid problem, and for some reason measure the displacements of the solids in kilometers and the velocity of the fluid in nm/s), the resulting matrix will be ill-conditioned. This is usually easy to avoid by selecting appropriate units.
- Phenomena at different scales, as in the example of the discrete Laplace operator: low and high spatial frequencies correspond to small and large eigenvalues respectively. One solution to this problem is to change the way we measure vectors to give more weight to high spatial frequencies than to low. Mathematically, this leads to Sobolev spaces, the Lax-Milgram theory, etc. Numerically, this leads to the notion of preconditioning, to be discussed later[11].

It is often interesting to explore the singular vectors (*modes*) associated with the large and small singular values of a matrix, which can for a given problem shed light on the source of ill-conditioning by giving directions in which A is large or small.

Exercises:

1. Show (1). Show that $\kappa(A)$ is equal to the ratio of the largest to smallest singular values of A (which extends the above definition to the case of non-square matrices). In which special case is this equal to the ratio of largest to smallest eigenvalues of A? Show that $\kappa(A) \geq 1$.
2. Find out how to compute condition numbers in your language, and check the formulas in this section.
3. Compute the conditioning of the matrix $A = \begin{pmatrix} 1 & 0 \\ 0 & \alpha \end{pmatrix}$.
4. Write down the matrix of the finite-differences discretization of the Laplacian on $[0, 1]$ with Dirichlet boundary conditions. Plot its eigenvalues and some of its eigenvectors. What is the conditioning of this matrix as a function of N?

5.4 Least-squares and regularization

Consider the following generalized least squares problem: let $(\phi_p)_{p=1,\dots,P}$ be smooth $\mathbb{R} \to \mathbb{R}$ functions, and $(x_n, y_n)_{n=1,\dots,N}$ be data points. Define the least-squares objective function

$$E = \sum_{n=1}^{N} \left(y_n - \sum_{p=1}^{P} \alpha_p \phi_p(x_n) \right)^2$$

For instance if $\phi_1(x) = 1$, $\phi_2(x) = x$, this is the classical least squares fitting of a set of points by a straight line.

Exercises:

1. Reformulate the problem in the vector form: find $\alpha \in \mathbb{R}^P$ such that $\|A\alpha - b\|$ is minimal.
2. Write down the optimality conditions.
3. Use this for polynomial regression ($\phi_p(x) = x^{p-1}$).
4. Test on noisy data. What happens when P gets large?
5. What are the conditioning of A and $A^T A$? What modes are this ill-conditioning associated with? Why?
6. Redo this with the objective function

$$E = \sum_{n=1}^{N} \left(y_n - \sum_{p=1}^{P} \alpha_p \phi_p(x_n) \right)^2 + \eta \sum_{p=1}^{P} \alpha_p^2.$$

Interpret. How should η depend on the level of noise?

[11] Still, the existence of various scales (for instance in space, time or energy) for a given physical problem is always challenging both theoretically and numerically, especially in the nonlinear case (see e.g. turbulence).

6 Linear equations

Consider the problem of solving $Ax = b$ in $\mathbb{R}^N$, where A is a given invertible matrix. This is an ubiquitous problem: either by itself, or as a building block of more complex problems.

6.1 Direct methods

Direct methods work on the entries of A directly. If A has a specific form, linear systems can be solved efficiently:

- Diagonal matrices, in $O(N)$.
- Upper or lower triangular matrices, which can be solved by back-substitution (for a lower triangular A, use the first line to solve for x_1, substitute in the second line, etc.) in $O(N^2)$
- Orthogonal matrices ($A^{-1} = A^T$), in $O(N^2)$
- Tridiagonal matrices, in $O(N)$ (the Thomas algorithm)
- Circulant matrices (cyclic convolutions), in $O(N \log N)$ via the fast Fourier transform
- Low-rank perturbations of easy-to-invert matrices: for instance, $(I+uu^T)^{-1} = I - \frac{uu^T}{1+\|u\|^2}$ (this generalizes to the Sherman-Morrison-Woodbury formula)

Otherwise, one has to use a general-purpose method, the most basic version of which is Gaussian elimination[12]. Subtract an appropriate multiple of the first equation to all others equations in order to remove the dependency of all but the first equation on x_1. Do the same to remove the dependence on all but the first two equations on x_1 and x_2, and carry on this process iteratively until the last equation depends on x_N only. Solve for x_N, substitute back into the second last equation to solve for x_{N-1}, etc[13]. Algebraically, the first part of this process turns A into an upper triangular matrix by "triangular" row manipulations (ie the n-th step does not touch the previous rows), and can be represented by the matrix factorization

$$A = LU$$

where L is lower-triangular and U upper-triangular. Then $Ax = b$ can be solved by back-substitution. The complexity of the first step (matrix factorization) is $O(N^3)$, and that of the second step (back-substitution) is $O(N^2)$. The interest of writing Gaussian elimination as a matrix factorization is that once the factorization $A = LU$ is completed, the system $Ax = b$ can be solved for several right-hand sides quickly [14].

Gaussian elimination can also be performed blockwise. Say that you have a variable $X = (x, y)$ with $x \in \mathbb{R}^{N_x}, y \in \mathbb{R}^{N_y}, X \in \mathbb{R}^{N_x+N_y}$, and the block system of equations

$$Ax + By = a$$
$$Cx + Dy = b$$

Then we can solve for y in the second equation: $y = D^{-1}(b - Cx)$ and replace in the first to obtain

$$(A - BD^{-1}C)x = a - BD^{-1}b$$

Therefore, if D is easy to invert, we obtain a reduced system for x with matrix $A - BD^{-1}C$. This matrix is called the Schur complement, and forms the basis of some very powerful techniques in many fields of applied mathematics[15].

Exercises:

[12] You may be familiar with Cramer's rule, which expresses the entries of the inverse of a matrix using determinants. Determinants however imply a sum over permutations, which scale exponentially with the matrix size and are therefore impractical. Generally speaking, determinants are not extremely useful in numerical linear algebra.

[13] This description ignores the fact that diagonal elements (pivots) might become zero, or close to zero; in this case one needs to permute rows and/or columns to ensure a non-zero pivot.

[14] Alternative factorizations, all $O(N^3)$ are the QR factorization (Gram-Schmidt process) with Q orthogonal and R triangular (useful for solving least-squares problems), and the Cholesky factorization $A = LL^T$ with L triangular of a positive-definite matrix (faster than LU for these matrices). It would appear from this list that the complexity of solving a linear system is $O(N^3)$. This is not the case and sub-cubic algorithms exist, but are not competitive in practice.

[15] Some applications of Schur complements are collected at `http://antoine.levitt.fr/schur.pdf`.

1. What is the largest dense linear system you can solve on your computer without crashing it? (make sure to save your work before attempting this!)

2. Check numerically that LU factorizations scale roughly as N^3.

3. A LU factorization uses about $\frac{2}{3}N^3$ floating point operations. Compare the runtime of a linear solver against the expected cost in a simple model where each floating point operation takes up one clock cycle. How can you explain the discrepancy?

6.2 Iterative methods: the Richardson iteration

Very often matrices of practical interest, such as those arising from finite differences, elements or volume discretizations, are *sparse* (contain $O(1)$ non-zero elements per row). This structure can be used in two ways: first, by specialized direct solvers that try to exploit this property, and second by iterative methods, which use the fact that matrix-vector products can be sped up (to $O(N)$ rather than $O(N^2)$) by skipping the zero entries. We focus here on this second class[16].

The very simplest iterative method is the iteration

$$x_{n+1} = x_n - \alpha(Ax_n - b) \tag{2}$$

where α is a scalar parameter (sometimes called the modified Richardson iteration). The idea behind this method is that, if we allow ourselves only to compute matrix-vector products, there is not much more we can do but combine the *residual* $Ax_n - b$ with the current iterate x_n in some way. We now study this method in the case of symmetric positive-definite matrices A for simplicity.

Clearly, this iteration has the solution x_* of $Ax_* = b$ as a fixed point. This is a first-order inhomogeneous recurrence. To study the convergence, the first step is to reduce is to a homogeneous problem by setting $e_n = x_n - x_*$[17]. Then we have

$$e_{n+1} = (1 - \alpha A)e_n$$

Now project this in an orthonormal (since A is symmetric) eigenbasis of A: $Av_i = \lambda_i v_i$. Then,

$$\langle v_i, e_n \rangle = (1 - \alpha\lambda_i)^n \langle v_i, e_0 \rangle$$

For a generic x_0, the $\langle v_i, e_0 \rangle$ are all non-zero. Then, $e_n \to 0$ if and only if $|1 - \alpha\lambda_i| < 1$ for all i. Since we have assumed that A is positive-definite, this is equivalent to $\alpha < \frac{2}{\lambda_N}$, where $0 < \lambda_1 \leq \cdots \leq \lambda_N$ are the eigenvalues of A sorted by magnitude. This is the first important result: the iteration always converges to x_* when α is small enough (note that this does not hold if A is not positive definite).

Now what is its speed of convergence? Squaring and summing the above equality, we get

$$\|e_n\| \leq (\max_i |1 - \alpha\lambda_i|)^n \|e_0\|.$$

The quantity $r(\alpha) = \max_i |1 - \alpha\lambda_i|$ is called the convergence rate: the smaller it is, the faster convergence will occur. It is easy to see that the α that minimizes r is given by

$$\alpha_{\text{opt}} = \frac{2}{\lambda_1 + \lambda_N},$$

and that in this case

$$r(\alpha_{\text{opt}}) = 1 - \frac{2\lambda_1}{\lambda_1 + \lambda_N} = 1 - \frac{2}{1 + \kappa(A)} \tag{3}$$

where we have used the condition number $\kappa(A) = \|A\|_{\text{op}}\|A^{-1}\|_{\text{op}} = \lambda_N/\lambda_1$. The larger this condition number is, the closer $r(\alpha_{\text{opt}})$ is to 1, indicating slow convergence. This is a general lesson: the more ill-conditioned a problem is, the harder it is to solve numerically.

Exercises:

[16] Note that iterative methods apply also to matrices that are not sparse but for which matrix-vector products can be performed efficiently, such as matrices representing convolutions.

[17] Equivalently, we could try to apply the Banach fixed-point theorem by showing that the map $x \to x - \alpha(Ax - b)$ is contractive.

1. Plot the iterates of this method in the case where $A = \begin{pmatrix} 1 & 0 \\ 0 & 0.1 \end{pmatrix}$

2. Use this method to solve the equation $-u'' = 1$, $u(0) = u(1) = 0$ discretized using finite differences. How should α be selected as a function of N? After a number of steps, on what modes will the error $x_n - x_*$ be concentrated on?

6.3 Krylov methods

In practice, one should never use the simple iteration (2), for several reasons. First, it requires a manual selection of α. Second, it does not work for all invertible matrices (for instance, $A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$). Third, it is not the most efficient way of combining the residual with the current iterate. A better way is to combine it with the whole subspace of previous iterates; this is known as a Krylov method. There are several avatars of this, but the simplest conceptually is the GMRES method, which is given by

$$x_n = \text{argmin}_{x \in K_n} \|Ax - b\|$$

where $K_n = \text{Span}(\{b, Ab, \dots, A^{n-1}b\}$ [18]. To implement this idea, one builds an orthonormal basis of K_n by $n-1$ applications of A [19], and then solves the resulting linear least-squares as in Section 5.4. Note that the Richardson method, as do all the methods that combine linearly residuals with current iterates, has the property that $x_n \in K_n$: therefore the GMRES method is the optimal method in this class. There are numerous implementations details in the GMRES method, as well as several variants (the most notable being the conjugate gradient method, applicable only to symmetric positive matrices, which achieves similar results to the GMRES method but requires considerably less storage and scalar products).

The Krylov methods do not need any manual selection of parameters, always converge for any invertible matrix, and converge faster than the Richardson iteration. They also have the property that they converge exactly in N steps since by then the Krylov space will span the whole of $\mathbb{R}^N$, although this is not very useful when N is large. Their convergence rate is still affected by ill-conditioning, albeit to a lesser extent[20].

Exercises:

1. Find out how to call a library that implements Krylov methods, and use it to solve the linear systems in the previous section.

6.4 Preconditioning

We have seen that ill-conditioned problems are harder to solve iteratively. Geometrically, this is because the residual and the error are not aligned. For a given x, if $r = Ax - b$ and $e = x - x_*$, then we have the error-residual relation

$$Ae = r.$$

The less well-conditioned (different from a multiple of the identity matrix) A is, the more r will be distorted from the ideal direction e, and therefore the harder it will be to converge.

A solution is preconditioning. Assume that we know a matrix P which is "close" to A but simpler to invert. Then, instead of solving $Ax = b$, we can solve $P^{-1}Ax = P^{-1}b$: this will work nicely if 1) $P^{-1}A$ is more well-conditioned than A and 2) P^{-1} is sufficiently cheap to apply. Note that we have here made the choice to apply P^{-1} to the equations (left preconditioning); we could have applied it on the unknowns (right preconditioning) as $AP^{-1}(Px) = b$, or even on both as $P_1^{-1}AP_2^{-1}(P_2x) = P_1^{-1}b$.

Preconditioning is problem-dependent; sources can include

[18] This is an instance of a Galerkin approach (technically, Petrov-Galerkin), which uses a small basis to approximate the solution of a larger problem

[19] Note that this should *not* be done by computing A^ib for $i = 0, \dots, n-1$ and then orthogonalizing these vectors, as this is very unstable numerically: A^nb tends to align with the eigenvector of A associated to its largest eigenvalue in modulus, so that all the vectors in the basis will be very similar, resulting in a loss of accuracy after orthogonalization. Rather, this is done in practice using the Arnoldi or Lanczos process, which orthogonalizes *then* applies A.

[20] A careful analysis based on approximation by Chebyshev polynomials shows that the convergence rate degrades at a rate similar to (3), but involving instead the square root of the condition number.

- Approximations of the underlying physics that admit a simpler solution (for instance, replacing an inhomogeneous medium by an homogeneous one)
- The pre-computed factorization of a related problem (for instance, with a close set parameters)
- Algebraic preconditioners such as the incomplete LU (ILU) factorization
- The solution of the problem at a coarser discretization (this can be used recursively to yield the very efficient multigrid method)

Exercises:

1. Consider the integral equation

$$\alpha(x)u(x) + \varepsilon \int_0^1 \beta(x,y)u(y)dy = f(x)$$

on $[0,1]$, where α is a positive continuous function, β and f are continuous functions and ε is a scalar. Show that it admits a unique solution when ε is small enough. What is the expansion of u in powers of ε? Show that the series converges for ε small enough. How can you interpret the power expansion in the framework of this chapter? Is it an efficient numerical method to solve the problem numerically?

6.5 Lax-Milgram revisited

The Lax-Milgram theorem is usually presented in terms of bilinear functionals, and works in a single function space X, using the Riesz isomorphism to identify X^* with X. Depending on the structure of X, this might not be a simple operation: typically when $X = H^1$, the linear application $(v \to \int fv) \in X^*$ is *not* represented by f. To be more explicit and better highlight the connection with this section, we refrain from making this identification, but rather introduce a "sandwich" space H (typically, L^2) for which the dual is simpler to identify.

Theorem (Lax-Milgram, operator version)**.** *Let H be a separable Hilbert space, $X \subset H$ a dense sub-Hilbert space, and $X^* \supset H^*$ the dual of X (linear functionals continuous in the X norm). Let A be continuous from X to X^* and coercive on X: for all $x, y \in X$,*

$$\langle x, Ay\rangle_{X\times X^*} \leq C\|x\|_X\|y\|_X$$
$$\langle x, Ax\rangle_{X\times X^*} \geq \alpha\|x\|_X^2.$$

Then, for all $b \in X^*$, the equation $Ax = b$ has a unique solution in X.

Note that A is not bounded from H to H^*. The typical application is on elliptic equations: $H = L^2$ and $X = H^1$, $X^* = H^{-1}$, and A is an operator with two derivatives (bounded from H^1 to H^{-1}).

Proof. Let $M : X \to H$ be the isomorphism between X and H (recall that all separable Hilbert spaces are isomorphic to ℓ^2 through the existence of a Hilbert basis). This naturally induces an isomorphism M^* from H^* to X^*. Then $Ax = b$ can be reformulated as

$$\underbrace{M^{-*}AM^{-1}}_{H\to H^*}\underbrace{Mx}_{\in H} = \underbrace{M^{-*}b}_{\in H^*}$$

Up to Riesz identification of H^* with H, we can consider the operator $M^{-*}AM^{-1} : H \to H^*$ as an operator from H to H. It is easily checked that this operator is bounded and coercive on H. Consider the iteration

$$y_{n+1} = y_n - \alpha(M^{-*}AM^{-1}y_n - M^{-*}b) \tag{4}$$

The spectral radius $\rho(L) = \lim_{n\to\infty} \|L^n\|^{1/n}$ of the iteration operator $L = 1 - \alpha M^{-*}AM^{-1}$ is below 1 for α small enough, and therefore the iteration converges to a fixed point[21]. □

[21] To show this properly needs a few elementary results on the spectral radius we have skipped, and can be found e.g. in 2A spectral theory lectures.

This (fully constructive) proof links the Lax-Milgram theorem to the solution of an infinite-dimensional equation by a preconditioned iterative method. It is a good example of the proof-algorithm equivalence: a constructive existence proof gives a converging algorithm, and vice-versa.

Exercises:

1. Identify (4) explicitly when $X = H_0^1([0,1])$, $H = L^2([0,1])$, and $Au = -u''$.

7 Eigenvalue problems

Consider the problem $Ax = \lambda x$. How to compute the eigenvalues and eigenvectors numerically[22]? The elementary method of computing the roots of the characteristic polynomial turns out to be both inefficient and numerically unstable. As for linear systems, there are two classes of methods: direct methods operating on dense matrices, and iterative methods for sparse matrices.

The very simplest iterative method is the power method

$$x_{n+1} = \frac{Ax_n}{\|Ax_n\|}$$

which converges to an eigenvector associated with the largest eigenvalue of A in magnitude. Using the same analysis as before we can show that its convergence rate is equal to the ratio of the second-largest to the largest eigenvalue. Therefore, this method is efficient if the largest eigenvalue is well-separated from the others.

As for linear systems, this simple iteration is suboptimal in practice, and Krylov methods like the Arnoldi or Lanczos method should be used.

Exercises:

1. Assuming that A is symmetric, show that x_{n+1} converges, and give its convergence rate.

2. Implement the power method and test it on a random matrix. Find a package that implements an appropriate Krylov method and compare.

3. A simplified version of the Pagerank algorithm that used to form the backbone of Google's search engine works as follows. Consider a set of webpages $1, \dots, N$, and let $l(i,j)$ be 1 if page i points to page j, and 0 otherwise. Consider a user that clicks on links at random. Letting p_i^n be the probability that the surfer is on webpage i at step n, write a recurrence relation for p_i^n. What happens after a very long time? The Pagerank algorithm assigns to each webpage i the eventual value of p_i.

8 Nonlinear equations

Consider a nonlinear system

$$F(x) = 0.$$

where $F : \mathbb{R}^N \to \mathbb{R}^N$ is smooth. How do we find zeros of this equation? The very simplest method, generalizing the Richardson iteration, is to subtract a multiple of the current residual:

$$x_{n+1} = x_n - \alpha F(x_n)$$

This is known as the Picard iteration, and is not usually convergent unless in specific cases. This is because the differential at x of this nonlinear mapping is $dx \to dx - \alpha F'(x)dx$, which is not contractive unless F' has all positive eigenvalues. To solve this, we need to modify the direction $F(x_n)$ given by the residual to align it better with the error.

The optimal way of doing this is the Newton iteration

$$x_{n+1} = x_n - F'(x_n)^{-1}F(x_n).$$

[22]Note that, unlike linear systems, finding eigenvalues is necessarily an iterative process that does not converge in a finite number of steps. This is because the problem of finding the roots of a polynomial can be recast as an eigenvalue problem using the companion matrix, and Galois theory shows that roots of a polynomial of degree ≥ 5 can not be expressed using elementary functions only.

The differential of this mapping at x is

$$dx \to dx - F'(x)^{-1}F'(x)dx - (J(x)dx)F(x) = -(J(x)dx)F(x)$$

where $J(x) : \mathbb{R}^N \to \mathbb{R}^{N\times N}$ is the differential at x of the mapping $x \to F'(x)^{-1}$. This is a complicated object, but fortunately it multiplies $F(x)$, so that it vanishes at a solution x_* of $F(x_*) = 0$. This underlines the very rapid convergence of the Newton algorithm[23] when close to a solution.

Another way of deriving the Newton method is as "linearize and solve". At current iterate x_n, build a linear model of F: $F(x) \approx F(x_n) + F'(x_n)(x - x_n)$, set $F(x) = 0$ and solve for x.

The practical problems of the Newton method are that one needs to 1) start close enough to a solution 2) compute the jacobian $F'(x)$ 3) invert it. Globalization strategies (linesearches, trust-region) try to extend the region of convergence. Quasi-Newton algorithms (such as Broyden) try to avoid computing or inverting the Jacobian.

Using the Newton method and the Banach fixed-point theorem, we can prove the implicit function theorem:

Theorem. *Let $f : X \times Y \to X$ of class C^2, with X and Y Banach. Suppose that $f(0,0) = 0, \partial_x f(0,0)$ invertible. Then for all y in a neighborhood of 0 in Y, there is $x(y)$ such that $f(x(y), y) = 0$. This $x(y)$ is the only solution in a neighborhood of 0 in X.*

This very powerful theorem says that if an equation $f(x) = 0$ has a non-degenerate solution x_* (in the sense that $f'(x_*)$ is invertible), then any small perturbation of that equation will have a unique solution close to x_*. Whenever $f'(x_*)$ is not invertible, existence or uniqueness can fail: this is called a *bifurcation*. For instance, $f(x, y) = x^2 - y$ has 2 solutions when $y > 0$, but zero solutions when $y < 0$.

Exercises:

1. Consider solving for $f(z) = 0$ where $f(z) = z^3 - 1$ is from $\mathbb{C}$ to $\mathbb{C}$. Mapping $\mathbb{C}$ to $\mathbb{R}^2$, write down the Newton iteration, and show that it is equivalent to the complex iteration

   $$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}.$$

   We call *basin of attraction* of a root the set of starting points z_0 such that the Newton method eventually converge to that root. Compute numerically and visualize the basin of attraction of the three roots.

2. Prove the implicit function theorem.

9 Optimization

Assume that we are given a smooth function $E : \mathbb{R}^N \to \mathbb{R}$, and we want to optimize it. The very simplest method is the gradient descent

$$x_{n+1} = x_n - \alpha\nabla E(x_n).$$

It reduces to the Richardson method when E is quadratic: $E = \frac{1}{2}x^T Ax - b^T x$. Therefore, the previous analysis shows that convergence occurs close to a minimum (where the function is approximately quadratic with $A = \nabla^2 E(x)$ symmetric positive definite), for α small enough, with a convergence rate that depends on the conditioning of $\nabla^2 E(x)$.

We can obtain more global results by noting that

$$E(x_{n+1}) = E(x_n) - \alpha\|\nabla E(x_n)\|^2 + O(\alpha^2\|\nabla E(x_n)\|^2).$$

Therefore, for α small enough, the value of the objective function will always decrease. This allows optimization methods to be much more robust than root-finding. Furthermore, α can be chosen adaptively by ensuring a sufficient decrease in the objective function (line search).

[23] In fact, refining this analysis shows that the Newton method converges quadratically, meaning that the error at step $n + 1$ is roughly the square of that at step n. This is an incredibly fast mode of convergence where the number of accurate digits doubles per iteration; much quicker than the "linear" convergence we have seen before as r^n, where the number of accurate digits increases linearly per iteration.

Like the Richardson method, gradient descent suffers from ill-conditioning. Near a minimum, one can adapt Newton's method, giving the iteration

$$x_{n+1} = x_n - (\nabla^2 E(x_n))^{-1} \nabla E(x_n),$$

which however needs the hessian and its inverse. As with the previous methods, one can use quasi-Newton updates (the most successful version of which is the LBFGS method) or preconditioning: $x_{n+1} = x_n - \alpha P \nabla E(x_n)$[24]

We have here only talked about unconstrained local optimization (convergence to a local minimum) of a continuous, differentiable and deterministic function.

- Constrained optimization can be tackled using specific methodologies. The most important concept is that of the Lagrange multipliers (or dual variables). Linear constraints can be handled particularly efficiently (see linear programming).
- Global optimization (finding *the* minimum) of a non-convex objective function is extremely hard, and can only be done heuristically (e.g. genetic algorithms which introduce "mutations" of the variables and evaluate their fitness).
- Optimization of discrete (e.g. over integers, or binary variables) objective functions leads to combinatorial explosion of the state space. This is often tackled by relaxing the integers to reals and/or eliminating portions of the tree (*branch and bound* strategies)
- Continuous but not differentiable functions are often encountered due to absolute values, which are "almost" differentiable (see e.g. subgradient methods).
- Optimization in the presence of noise is a subject in itself. A common case is that of minimizating an objective function ("loss") written as a sum (expectation value) over several data samples. Stochastic gradient methods avoid making a full pass over the data before making a minimization step and look only at one or several ("mini-batching") samples. This method (or an accelerated variant) is the standard method in training neural networks.

Exercises:

1. Plot the Rosenbrock function $(1-x)^2 + 100(y-x^2)^2$. Optimize it by a hand-coded gradient method and a more sophisticated version from a library.

10 Numerical differentiation

Computing derivatives of complicated functions (not simple univariate mathematical expressions) by hand is often tedious and easily leads to mistakes. Alternatives are

- Symbolic differentiation, using specialized software like Mathematica, or libraries like SymPy. This method analyses the expression (parsed as a tree) and differentiates it mechanically.
- Numerical differentiation by finite differences, which are simple to use (using the function only as a black box) but suffer from numerical error.
- Automatic differentiation, using specialized libraries or frameworks like TensorFlow. This comes in two flavors: forward autodiff (which stores the sensitivities of each intermediate variable with respect to input variables, and propagates them *forward* along the computation) and reverse autodiff (which stores the sentivities of the output with respect to intermediate variables, and propagates them *backward* along the computation). [25]

Derivatives should always be checked by comparing two independent methods.

[24] In the context of optimization, preconditioning has a nice interpretation in terms of a change of metric. Recall that the differential $df(x)$ of E at x is defined as the unique linear functional such that $E(x+h) = E(x) + df(x) \cdot h + O(h^2)$. This does *not* define a gradient without an additional structure: an inner product, which allows the identification of linear functionals with vectors through $df(x) \cdot h = \langle \nabla E(x), h \rangle$. Depending on the choice of the inner product, different gradients result. An objective function $E : \mathbb{R}^N \to \mathbb{R}$ has a Euclidean gradient $\nabla E(x)$. But if instead we use the metric $(x, y) = \langle x, P^{-1} y \rangle$ where P is a symmetric positive definite matrix, then the gradient is $\nabla_P E(x) = P \nabla E(x)$. Preconditioned gradient descent can therefore be interpreted as a natural gradient descent, in a different metric.

[25] Backward autodiff (also known as *adjoint method* or *backpropagation*) in particular is an extremely powerful methodology. It implies that "gradients are cheap": the cost of computing a gradient can always be made to be of the same complexity as computing the function *once*, no matter the size of the input. We refer to `http://antoine.levitt.fr/adjoint.pdf` for details.

11 Interpolation

Imagine that we know a function $f : \mathbb{R} \to \mathbb{R}$ at several points. How do we evaluate at other points? Interpolation (when the evaluation points are surrounded by data points) has to be distinguished from extrapolation[26].

The simplest method in one dimension is to use the value at the data point closest to the evaluation point. For more accuracy, one tries to guess the behavior of the function between the data points, for instance as a polynomial, and fits the coefficients of the polynomial to the data point. At first order, this is piecewise-linear approximation. Higher-order approximations are asymptotically more accurate but involve more and more data points for a single evaluation. This makes them sensitive to discontinuities of f, and to ill-conditioning issues. In practice it is often useful to restrict to a moderate order (for instance, cubic splines).

The above method can be extended easily to higher dimensions. It is however completely useless in very high dimensions, because the number of points needed scales exponentially with the dimension. There, more advanced methods such as sparse grids, kriging or neural networks must be used.

Exercises:

1. Fit a polynomial of degree $N-1$ to N equispaced samples of the Runge function $1/(1+25x^2)$ on $[0, 1]$.

12 Integration

Given a function, assume we want to compute its integral. The most basic method is the method of rectangles (Riemann sum):

$$\int_a^b f(x)dx \approx \frac{(b-a)}{N} \sum_{n=1}^{N} f\left(a + \frac{(b-a)n}{N}\right).$$

The accuracy of this method is easily seen by Taylor expansion to be of order $1/N$, which is quite slow[27]. Better methods interpolate the function f at data points and integrate the resulting interpolant (for instance, Gaussian quadrature), making numerical integration largely a subset of interpolation. Integration codes can be adaptive, meaning they try to focus on places where the function is harder to integrate. An alternative in high dimensions is the Monte-Carlo method, see Section 15.

Exercises:

1. Derive the trapezoidal rule, which integrates a piecewise linear interpolation of the function.
2. Show that the Riemann method applied to a smooth periodic function converges more quickly than any inverse polynomial in N.
3. Compare the efficiency of the Riemann sum, the trapezoidal sum and the Monte-Carlo method on a simple integral. Compare with a specialized integration package.

13 Space discretization

When faced with partial differential equations, one must discretize space. The main methods are:

- Finite differences. The unknowns are the functions at discrete points, and derivatives are approximated by finite differences. This is the very simplest method.
- Finite elements. The unknowns are the coefficients of the solution in an expansion on localized basis functions, and the problem is converted to a variational form. This makes it relatively straightforward to handle complex geometries (meshes), and this is often the method of choice in mechanical engineering.

[26] Extrapolation is hard unless there is reason to expect a particularly simple behavior (e.g. power law), in which case this behavior can be fitted to data.

[27] A notable exception is when the integrand is periodic, in which case the Riemann sum turns out to integrate the low-frequency Fourier harmonics exactly, and converges extremely quickly.

- Spectral methods. The unknowns are coefficients in delocalized, higher-order basis functions (for instance, Fourier harmonics). This needs regular geometries but is able to achieve a very high precision with relatively few unknowns.
- Finite volumes. The unknowns are the averages of the function in discrete cells, and fluxes between cells are approximated. This method guarantees the conservation of a number of properties, and is often used in fluid mechanics.
- Meshfree or particle methods. The unknowns are the positions of particles that represent the function in some sense. This makes it easier to solve free surface flows for instance.

14 Time integration

Consider the ODE

$$\dot{x} = f(x)$$

with $x(0) = x_0 \in \mathbb{R}^N$. The simplest integration method is the explicit (or forward) Euler scheme: partition time into discrete times t_n, rewrite $\dot{x}(t_n) \approx (x(t_{n+1}) - x(t_n))/(t_{n+1} - t_n)$, and so deduce

$$x^{\mathrm{FE}}(t_{n+1}) \approx x^{\mathrm{FE}}(t_n) + (t_{n+1} - t_n) f(x^{\mathrm{FE}}(t_n)). \tag{5}$$

Under mild conditions on the regularity of f, this converges in the sense that, for all $T > 0$, there is $C > 0$ such that, assuming an equispaced grid of stepsize Δt,

$$|x(t) - x^{\mathrm{FE}}| \leq C\Delta t$$

for all gridpoints $0 \leq t \leq T$. The proof proceeds by showing that 1) the method is consistent, in the sense that the error made at each step is small 2) it is stable, in the sense that the errors made at each step simply sum (rather than accumulate in a nonlinear fashion).

The explicit Euler is particularly simple, but has the following drawbacks

- It is only first-order accurate, making it hard to achieve accurate solutions. Higher-order methods proceed by better approximations of (5), which can be achieved in two ways: either by evaluating f at intermediary points (Runge-Kutta methods) or by trying to extrapolate from previous values of x (multistep methods, which, because of the use of extrapolation, may suffer from stability issues).
- It does not preserve the conserved quantities that might be present in the solution (energy, momentum, mass...) and therefore might be unusable for long trajectories. Symplectic integrators, taylored to the specific equation, can be more efficient there.
- It is not suitable for *stiff* equations, which are equations with multiple timescales that can present rapid variations (as might commonly be found in electronics or biology).

To illustrate this last point, consider the system

$$\dot{x} = -\lambda x \tag{6}$$

for $\lambda > 0$, which converges to zero. The explicit Euler method gives $x_{n+1} = (1 - \lambda\Delta t)x_n$, which eventually explodes for timesteps $\Delta t > 2/\lambda$. For more complicated systems, this means that the timestep Δt must be smaller than the *smallest* timescale in the system or lead to instabilities. This is particularly problematic for discretizations of PDEs, which often have an unbounded array of frequency scales.

It turns out that the implicit (or backward) Euler method cures this. The backward Euler simply takes the opposite approximation $x'(t_{n+1}) \approx (x(t_{n+1}) - x(t_n))/(t_{n+1} - t_n)$, leading to

$$x^{\mathrm{BE}}(t_{n+1}) \approx x^{\mathrm{BE}}(t_n) + (t_{n+1} - t_n) f(x^{\mathrm{BE}}(t_{n+1})).$$

This is now an *implicit* equation for $x^{\mathrm{BE}}(t_{n+1})$ (which is well-posed when $t_{n+1} - t_n$ is small). On the model equation (6), it gives $x_{n+1} = x_n/(1 + \lambda\Delta t)$, which goes to zero for any value of Δt. The backward Euler is therefore more stable, at the cost of solving an equation at each step; whether that is worth it depends very much on the problem.

Exercises:

1. Consider the heat equation, discretized with finite differences. Show the forward Euler method explodes unless Δt satisfies the *CFL* (Courant-Friedrichs-Lewy) condition $\Delta t \leq \frac{2}{\Delta x^2}$. Show that the backward Euler method always converges to zero. Test this numerically.

2. Consider the equation $\dot{x} = Ax$ where A is an anti-hermitian matrix. Show that the discretized wave equation and Schrödinger equation are formally of this form. Show that $\|x\|$ is preserved along the dynamics. Show that $\|x\|$ increases for the forward Euler method, decreases for the backward Euler method, and stays constant with the Crank-Nicolson method

$$x_{n+1} = x_n + \frac{t_{n+1} - t_n}{2} \left(f(x_n) + f(x_{n+1}) \right).$$

15 Random numbers and the Monte-Carlo method

Computers do not know how to generate random numbers and one must instead use pseudo-random numbers: numbers generated deterministically that are still sufficiently similar to "true" random numbers. This is a very important topic in computer security because random numbers are used to design cryptographic codes: a predictable random number generator can be used to attack the codes. Perhaps the simplest random number generators are the linear congruential generators

$$x_{n+1} = (ax_n + c) \bmod m$$

where a, c and m satisfy certain properties, designed so that the x_n are as independent and uniformly distributed as possible. These are very bad and the ones found in standard libraries are much more robust.

Random numbers are used in probabilistic modeling and inference, as well as in the solution of many deterministic problems in computational physics and engineering. Most numerical libraries expose the `rand` function, which generates uniform numbers between 0 and 1, and the `randn`, which generates normally generated numbers[28].

Exercises:

1. What is the probability that a random variable distributed randomly on the square of length 2 centered around the origin will lie in the disk of radius 1? Use this to deduce an algorithm to compute the value of π, and implement it. What can you use to measure the error? What is the order of magnitude of the number of trials you expect to obtain 2 digits after of accuracy after the decimal point?

16 Summary

Problem	Questions to ask
$Ax = b$	A dense? Sparse? Well-conditioned? Preconditioner available?
$Ax = \lambda x$	A dense/sparse? Symmetric/normal? Eigenvalues well-separated?
$F(x) = 0$	F' available? Invertible? Good initial guess?
$\min f(x)$	Problem discrete/constrained/stochastic? Gradient/Hessian available/invertible?
Integration/interpolation	High-dimensional? Smooth?
PDE	Type of the equation? Geometry?
$x' = f(x)$	Preserved quantities? Stiff?

[28] Vectors of normally generated numbers tend to be more useful than vectors of uniformly generated ones, because the multivariate normal distribution is rotationally invariant.

Problem	Simplest method	Recommended method
$Ax = b$, A dense	Gaussian elimination	LU/Cholesky factorization
$Ax = b$, A sparse	Richardson method	(preconditioned) CG/GMRES
$Ax = \lambda x$, A dense	Power method	QR algorithm
$Ax = \lambda x$, A dense	Power method	Lanczos/Arnoldi
$F(x) = 0$	Newton	Globalized (quasi-)Newton
$\min f(x)$	Gradient descent	LBFGS
Interpolation	Nearest neighbor	Splines/Chebyshev
Integration	Rectangles	Adaptive higher-order
PDE	Finite differences	FD/FE/FV/spectral
$x' = f(x)$, non-stiff	Explicit Euler	Adaptive explicit Runge-Kutta/multistep
$x' = f(x)$, non-stiff	Implicit Euler	Adaptive implicit Runge-Kutta/multistep

17 Project

Read `https://www.dna.caltech.edu/courses/cs191/paperscs191/turing.pdf` Sections 1-8. The model he uses in the continuous case is

$$\frac{\partial u}{\partial t} = f(u) + D\frac{\partial^2 u}{\partial x^2}$$

where $u(x,t) \in \mathbb{R}^2$ and D is a diagonal 2×2 matrix, together with an appropriate domain and boundary conditions (we take $[0, 2\pi]$ with periodic boundary conditions for simplicity).

- Linearize f near a spatially homogeneous equilibrium u^*, and reproduce Turing's argument that the equilibrium might be broken by a spatially varying perturbation. Why is it important that the two entries of D are different?
- Apply this to

  $$f(u,v) = \begin{pmatrix} p - uv^2 \\ q - v + uv^2 \end{pmatrix}$$

  with $p = 2.0, q = 0.05$. Find the spatially homogeneous equilibrium u^*, and check that it is stable.
- Setting

  $$D = \begin{pmatrix} 1 & 0 \\ 0 & d \end{pmatrix}$$

  simulate the partial differential equation and observe the behavior of the system for various values of d (you might need rather small values for d to get interesting behavior). Compare explicit Euler, implicit Euler and a black-box ODE solver.

Bonus:

- Compute accurately the largest value of d at which the eventual equilibrium is non-homogeneous.
- Reproduce the stability analysis in the discrete case (in both space and time). Are the finite step sizes in space and time stabilizing or destabilizing the equation?
- Solve the implicit Euler equation as efficiently as you can.
- ...