Lecture Notes CME108: Introduction to Scientific Computing

Cindy Orozco

Summer 2016

Abstract

These notes are designed for the course CME 108: Introduction to Scientific Computing (MATH 114) during Summer 2016. In addition of lectures notes from previous editions of the course (given by Pr. Eric Dunham), this material follows these references:

- U. M. Ascher, C. Greif, *A First Course on Numerical Methods*, SIAM, Philadelphia, PA, 2011.
- B. Bradie, *A Friendly Introduction to Numerical Analysis*, Pearson Prentice Hall, Upper Saddle River, NJ, 2006

If you want some engineering applications, I recommend:

- S. Chapra, R. Canale *Numerical Methods for Engineers*, Mc-Graw-Hill Science, 6th edition, 2009.
- A. Antoniou, W. Lu., *Practical optimization: Algorithms and engineering applications*, Springer, New York, NY, 2007.

And for a more specialized content:

- L. N. Trefethen, D. Bau, *Numerical Linear Algebra*, SIAM, Philadelphia, PA, 1997.
- J. Nocedal, S. J. Wright, *Numerical Optimization*, Springer, New York, NY, 2006.
- S. J. Wright, *Coordinate Descent Algorithms*, ArXiv e-prints, arXiv:1502.04759, Feb 2015, Provided by the SAO/NASA Astrophysics Data System.
- R. LeVeque, *Finite Difference Methods for Ordinary and Differential Equations: steady-state and time-dependent problems*, SIAM, Philadelphia, PA, 2007.

If you find any typo or mistake, just send me an email at orozcocc@stanford.edu.

Enjoy the material!

Contents

Chapter 1

Basic Concepts of Scientific Computing

Sometimes there is the misconception that the most reliable alternative to solve a problem is to use math and computers. The principal issue is that if we do not keep track of the sources of error, things can go terrible wrong. For example, in 1996, the European Space Agency failed launching Ariane 5 rocket. The reason: 16-bits integer overflow.

In order to keep track of what can go wrong solving a problem, first we need to understand all the decisions made along the way, and more important all the implications and limitations we are subject to. To illustrate the thinking process, let us introduce the following example:

> *It is a sunny Sunday morning in your Grandparents home. You are thinking that is nice to see all the family together but you really need to go back to your office to finish that report, when, out of the sudden, you hear a gigantic crash: your nephew just broke the Grandpas pendulum clock. Since you took carpentry classes last summer, unanimously the family decides that you are in charge of fixing the clock. What would you do?*

Well, if the carpentry classes were true, you would definitely know how much wood you would need for the case, but probably you would not have any idea about the appropriate chord length that achieves the precision of the pendulum. Therefore let us try to model the clock as a simple pendulum. Given the drag coefficient b and a natural frequency $\omega = \sqrt{L/g}$, where L is the arm length and g is the gravity , the angle of oscillation θ made with the vertical in time t can be defined by the second order differential equation:

$$\textbf{Mathematical Model:} \quad \ddot{\theta} + b\dot{\theta} + \omega^2 \sin\theta = 0 \tag{1.1}$$

where we use $\dot{\theta} = \frac{d\theta}{dt}$ and $\ddot{\theta} = \frac{d^2\theta}{dt^2}$.

Notice that this is a nonlinear equation in θ, because of the term $\omega^2 \sin(\theta)$. If you try a couple of minutes, you will notice it is really difficult to solve it analytically. Therefore it is preferable to discretize it. This means that we will choose a finite number of t where we will know the solution of θ. Using these values, we approximate the derivatives using sums and divisions:

$$\textbf{Discretization: } \frac{\theta_{i+1} + \theta_{i-1} - 2\theta_i}{\Delta t^2} + b\frac{\theta_{i+1} - \theta_{i-1}}{2\Delta t} + \omega^2 \sin\theta_i = 0 \tag{1.2}$$

where $\theta_i = \theta(t_i)$ for $i = 0, 1, ..., N$. After we specify some initial and boundary conditions, we can rewrite (1.2) as a non linear system of equations:

$$K\Theta + \omega^2 \sin\Theta = 0 \tag{1.3}$$

where $K \in \mathbb{R}^{n\times n}$ is a matrix of coefficients, and $\Theta = (\theta_1, ..., \theta_n)$ is the vector of unknowns. Notice that other way to see the problem is to minimize the difference between $K\Theta$ and $\omega^2 \sin\Theta$. Therefore (1.3) is equivalent to find θ such that :

$$\min_{\Theta\in\mathbb{R}^n} ||K\Theta + \omega^2 \sin\Theta||_2^2 \tag{1.4}$$

To solve this minimization problem we use an iterative algorithm along feasible points, each time decreasing the value of the optimal function. At each iteration we find a feasible solving a linear system of the shape $Ax = b$. To solve this system we factorize A into lower and upper triangular matrices and then we backward and forward substitution to find the result.

Notice that solving this simple problem we used a lot different techniques: finite differences, numerical optimization and numerical linear algebra. Also this illustrates the way problems are solved using Scientific Computing.

In Figure 1.1 we can see the process starting form a problem in the real world to implementation. Related with each step, there is a different type of error (notice the red text). There are three levels of relevance of this errors:

- *Error relative to each field:* The tolerance of our method needs to include measurement errors and the consequences of assuming certain physical model.

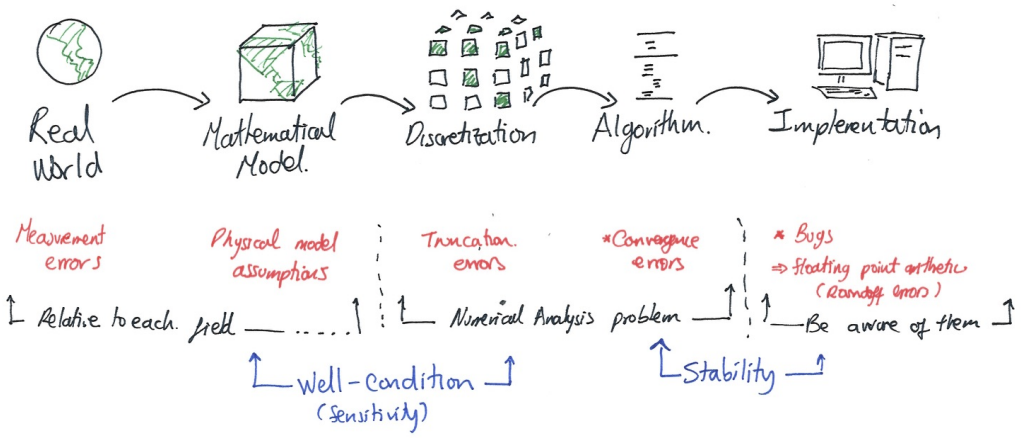

Figure 1.1: Different steps in Scientific Computing

- *Numerical analysis errors:* When we decide to solve a problem using certain method, usually we make two approximations: we solve for a finite number of points (discretization - truncation errors) and we iterate a finite number of times (rate and order of convergence).
- *Implementation errors:* First we will always find bugs in the code. And second, we cannot represent exactly numbers in the computer. This type of errors are called roundoff errors.

Since we can not avoid any of this three type of errors, we will start by discussing how to identify and quantify them. From now on we will use

Definition 1. *Let $\tilde{p}$ an approximation for the value p then*

$$\textbf{\textit{Absolute error:}}\ |\tilde{p}-p| \qquad \textbf{\textit{Relative error:}}\ \frac{|\tilde{p}-p|}{p} \tag{1.5}$$

1.1 Floating point arithmetic and Roundoff errors

How do computers represent numbers?
Which challenges do we have? Can we represent any number?

Computers represent numbers using binary system. Each bit can take values $\{0,1\}$, similar to a light bulb that goes `on/off`. A set of 8 bits creates a byte, and usually every language or architecture measure the amount of memory in bytes. We cannot represent any number because we have a finite amount of memory. Additionally, in the case that memory was not a problem, we have finite time to make

computations. More used memory means more communication time and more computation time.

Therefore it is important to be aware of the type of number representation a programming language uses. For example in MATLAB you can check the convention for integers and floating point numbers.

The floating point representation is based that given a basis $\beta \in \mathbb{N}$, any $x \in \mathbb{R}$ can be represented uniquely as:

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \frac{d_3}{\beta^3} + ... \right) \times \beta^e, \text{ where } 1 \leq d_0 < \beta, 0 \leq d_i < \beta \tag{1.6}$$

Therefore any number x can be represented by:

- A **sign** $\pm$
- An **exponent** e
- A sequence of naturals $d_0, d_1, d_2...$

In a computer we can only store a limited number of d_i. The maximum number of d_i's a computer can store is called **precision** t.

Definition 2. *The given $x \in \mathbb{R}$, **floating number representation** $fl(x)$ is the truncated sum of* (1.6)*, i.e.*

$$\mathbf{fl}(x) = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + ... + \frac{d_t}{\beta^{t-1}} \right) \times \beta^e = \pm \left(\frac{m}{\beta^t} \right) \times \beta^e \tag{1.7}$$

*where $\beta^{t-1} \leq m \leq 2^t - 1$ is called the **mantissa***

A better measure of the tolerance is the **rounding unit**:

$$\epsilon_{machine} = \frac{1}{2} \beta^{1-t} \tag{1.8}$$

which represents the maximum error representing a number between 1 and its closest right neighbor in this representation.

Notice that if $\beta = 2$ and $x \neq 0$, then it must be the case that $d_0 = 1$. Therefore we do not need to store d_0 and we get and additional position. For example, double precision in MATLAB uses 8 bytes, of which 52 bits represent the mantissa. Therefore $t = 53$ and $\epsilon_{machine} = 2^{-53} \approx 1.1e-16$.

Why is it important to know $\epsilon_{machine}$? The floating number representation satisfy the following two properties:

- **Relative error**: If x can be represented in the system then

$$|x - fl(x)| \leq |x|\epsilon_{\text{machine}} \tag{1.9}$$

- **Definition of operations:** For any arithmetic operation $*$, its floating point analogue $\circledast$ is:

$$fl(x) \circledast fl(y) = fl(fl(x) * fl(y)) \tag{1.10}$$

Therefore for any x, y that can be represented in the system:

$$|x * y - fl(x) \circledast fl(y)| \leq \underbrace{|fl(x) * fl(y) - fl(x) \circledast fl(y)|}_{Introduced\ error} + \underbrace{|x * y - fl(x) * fl(y)|}_{Propagated\ error} \tag{1.11}$$

The **introduced error** is related with the representation of the number. If all the floating numbers we can represent are in the interval $[L, U]$, we can have **underflow** if $x < L$ (and the computer will assume $x = 0$) or **overflow** if $x > U$ (and the computer will assume $x = \infty$).

The **propagated error** is related with how the operations change the initial error, i.e. if the error gets magnified or if we get **catastrophic cancellation**. For example if $x >> y$ then $fl(x) \oplus fl(y) = fl(x)$, and if $x \approx y$ then $fl(x) \ominus fl(y) = 0$.

Try the following exercises:

- In MATLAB let $x = 1/7$. What is the result of $x+x+x+x+x+x+x == 1$? Is it what you expected?
- Operations are not longer associative. Suppose the representation system

$$x = \pm \left(d_0 + \frac{d_1}{10} + \frac{d_2}{100} \right) 10^e \text{ where } 1 \leq d_0 \leq 9, 0 \leq d_i \leq 9$$

  let $x = -1.01$, $y = 1.00$ and $z = 1.00e - 3$. What is the value of $(x + y) + z$? What is the value of $x + (y + z)$?.

1.2 Errors related with Algorithms: Robustness and Efficiency

Definition 3. *An **algorithm** is a sequence of steps to perform a task*

People care about algorithms even before computers. And to compare between algorithms, we analyze how close are the approximated solution and the real one, and how expensive is to get to the solution. Designing an algorithm is like designing a building for an earthquake:

You always need a trade-off between admissible loads (Robustness) and cost (efficiency). Old buildings are designed to admit large loads during a seismic event without any visible deformation. Nevertheless their cost is 5 times more in comparison with a new building. This does not mean new buildings will collapse in any earthquake, but they have more probability to have visible damage

The same happens in choosing an algorithm to solve a problem. Since we do not know the real solution in advance (i.e. the real earthquake), we need to be sure how much does it take an algorithm to solve a problem if it is of certain type and the inputs are in certain threshold.

1.2.1 Robustness and stability

Suppose that given a real problem, we can get the real solution as a function f in terms of the x input data, i.e. *real solution* $= f(x)$. Similarly, if we neglect errors in the input, we can define the result of the algorithm as another function in terms of x, i.e. *approx solution* $= \tilde{f}(x)$. If we knew the real solution we could measure how close both solutions are.

Definition 4. *A method is* ***accurate*** *if* $|f(x) - \tilde{f}(x)|$ *is sufficiently small.*

Since in reality we do not know the real solution, then we would like to know under which conditions the method gives a "nearly correct" solution, i.e. we want to analyze the **Robustness** of the method. Instead of talking about accuracy, we talk about stability.

Definition 5. *A method is* ***stable*** *if given two inputs* x *and* $\tilde{x}$ *that are sufficiently close (i.e.* $|x - \tilde{x}|/|x| = \mathcal{O}(\epsilon)$*) then* $|\tilde{f}(x) - f(\tilde{x})|/|f(\tilde{x})|$ *is sufficiently small. Moreover, we can define that a method is* ***backward stable*** *if we can say that the result of the algorithm using the input* $\tilde{f}(x)$ *is equal to the real solution of the problem using a perturbed input* $f(\tilde{x})$*.*

For example, solving y such that $Ay = b$, we can identify the input A, and the real solution $y = f(A) = A^{-1}b$. If my algorithm is $\tilde{f}$, then we can call the result $\tilde{y} = \tilde{f}(A)$. Backward stability says that $\tilde{y} = f(\tilde{A}) = \tilde{A}^{-1}b$.

Notice that stability does not only depend on the method, but also depends on the mathematical problem. If the problem does not behave "nicely", then it does not matter how stable is the method, we can not ensure to get a desirable solution. Think when we are solving $Ay = b$ and A is a singular matrix. Depending on b we can have infinite solutions or zero solutions. How can the algorithm select the solution we are looking for?.

To avoid these issues, it is better to work with well condition problems: "problems that behave nicely".

Definition 6. *A problem is **well conditioned** if:*

- *The solution exists*
- *The solution is unique*
- *The solution depends continuously on the data. Small changes on the initial condition represent small changes on the data.*

1.2.2 Efficiency and Convergence

Usually algorithms in numerical analysis involve many iterations. Therefore to measure **efficiency** we need to count the cost per iteration, and the number of iterations we take. To count the cost per iteration we count **flops** = number of elementary operations performed (`+,-,*,/`) .

To measure the number of iteration an algorithm takes we talk about **convergence**.

Definition 7. *We say a sequence p_n converges to p, i.e. $p_n \to p$ if*

$$\lim_{n\to\infty} p_n = p \tag{1.12}$$

. We say an algorithm converges if the sequence of partial results converges.

Once we know the algorithm converges to the correct result, then we want to know the speed of convergence. For this we have two measures:

Definition 8. *We say a method p_n has **rate of convergence** $\beta(n)$ if there exists a constant $k < \infty$ such that*

$$|p_n - p| \leq K|\beta(n)| \text{ which is equivalent to } |p_n - p| = \mathcal{O}(\beta_n) \tag{1.13}$$

Definition 9. *We say a method p_n has **order of convergence** α if there exists a constant $0 < \lambda < \infty$ such that*

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|^\alpha} = \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^\alpha} = \lambda \tag{1.14}$$

For $\alpha = 1$ we talk about linear convergence and for $\alpha = 2$ we talk about quadratic convergence.

Notice that $\mathcal{O}(\beta_n)$ comes from asymptotic notation. Some useful definitions are

Definition 10. *Let f, g functions in $\mathbb{R}$ (or $\mathbb{N}$), and a defined limit $L = \infty$ or 0 or ....*
We informally say:

- ***Bounded above asymptotically*** $f(x) = \mathcal{O}(g(x)) \iff$

$$\exists M > 0 : |f(x)| \leq M|g(x)| \text{ for } x \to L$$

- ***Dominated above asymptotically*** $lf(x) = o(g(x)) \iff$

$$\forall M > 0 : |f(x)| \leq M|g(x)| \text{ for } x \to L$$

- ***Bounded below asymptotically*** $f(x) = \Omega(g(x)) \iff$

$$\exists M > 0 : |f(x)| \geq M|g(x)| \text{ for } x \to L$$

- ***Dominated below asymptotically*** $f(x) = \omega(g(x)) \iff$

$$\forall M > 0 : |f(x)| \geq M|g(x)| \text{ for } x \to L$$

- ***Bounded above and below asymptotically*** $latex f(x) = \Theta(g(x)) \iff$

$$\exists M_1, M_2 > 0 : M_1|g(x)| \leq |f(x)| \leq M_2|g(x)| \text{ for } x \to L$$

In here, $\exists$ means exists and $\forall$ means for all

1.3 Errors related with Discretization: Calculus tools

If someone asks: "Which problems can be solved using Numerical Analysis?" the answer is not straightforward. If we only count the problems for which there exists a known algorithm that gives the exact answer, then only a few problems in Linear Algebra and Optimization would be covered. But of course, these are not the only interesting problems. Therefore, what is the trick? ... Discretization.

Instead of working with any function, we know how to work with polynomials. And instead of solving for infinite number of points, it is enough in practice to solve for certain points. Of course, these two techniques introduce errors, but once we know how their magnitude depends on the approximation, we can refine it until we get acceptable results.

The principal tool is

Theorem 1. ***Taylor's Theorem*** *If $f \in C^{n+1}[a,b]$ (This means that f and all its derivatives up to $n+1$-th derivative are continuous in the interval $[a,b]$). then for any x and x_0 in $[a,b]$, there exists $\xi(x)$ between x and x_0 such that:*

$$f(x) = \underbrace{\sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k}_{n\text{-th Taylor Polynomial}} + \underbrace{\frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)^{n+1}}_{R_n\text{: Remainder}} \tag{1.15}$$

We can also express the remainder as:

$$R_n = (x-x_0)^{n+1} \int_0^1 \frac{(1-s)^n}{n!} f^{(n+1)}(x_0 + s(x-x_0))ds \tag{1.16}$$

Example 1. *For example, what is the rate of convergence of $(\sin x)/x \to 1$ when $x \to 0$?*

Solution. The Taylor's series of $\sin x$ around 0 is

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \sin(\xi(x))\frac{x^6}{6!}$$

Therefore

$$\lim_{x\to 0} \frac{\sin x}{x} = \lim_{x\to 0} 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \sin(\xi(x))\frac{x^5}{6!} = 1$$

And moreover when $x \to 0$

$$\left|\frac{\sin x}{x} - 1\right| = \left|-\frac{x^2}{3!} + \frac{x^4}{5!} - \sin(\xi(x))\frac{x^5}{6!}\right| = \mathcal{O}(x^2)$$

Why $\mathcal{O}(x^2)$? because of the three terms, x^2 is the slowest going to zero. □

Example 2. *Consider the iterative scheme to find $x = \sqrt{a}$.*

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Which is its order of convergence?

Solution. Notice that

$$\begin{aligned} x_{n+1} - \sqrt{a} &= \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) - \sqrt{a} \\ &= \frac{x_n^2 - 2\sqrt{a}x_n + a}{2x_n} \\ &= \frac{(x_n - \sqrt{a})^2}{2x_n} \end{aligned}$$

Assuming that $x_n \to \sqrt{a}$ when $n \to \infty$ then

$$\lim_{n\to\infty} \frac{|x_{n+1} - \sqrt{a}|}{|x_n - \sqrt{a}|^2} = \lim_{n\to\infty} \frac{1}{2x_n} = \frac{1}{2\sqrt{a}}$$

Therefore the order of convergence is *quadratic*. Notice that

$$\lim_{n\to\infty} \frac{|x_{n+1} - \sqrt{a}|}{|x_n - \sqrt{a}|^\alpha} = \begin{cases} 0 & \alpha = 1 \\ \frac{1}{2\sqrt{a}} & \alpha = 2 \\ \infty & \alpha > 2 \end{cases}$$

□

In addition of Taylor's theorem we have other useful theorems

Theorem 2. ***Intermediate Value*** *Suppose $f \in C[a,b]$ and let s be between $f(a)$ and $f(b)$ therefore there exists $c \in [a,b]$ such that $f(c) = s$.*

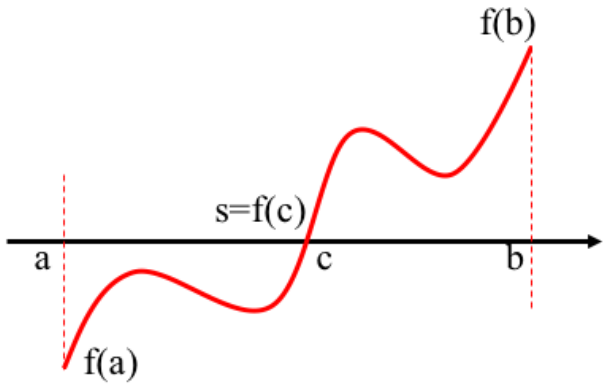

Figure 1.2: Intermediate value theorem

Theorem 3. ***Mean Value*** *Suppose $f \in C[a,b]$ and $f \in C^1(a,b)$. There exists $c \in (a,b)$ such that*

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

.

Theorem 4. ***Rolle's*** *Apply the mean value theorem for the case $f(a) = f(b)$. Therefore there exists $c \in (a,b)$ such that*

$$f'(c) = 0$$

.

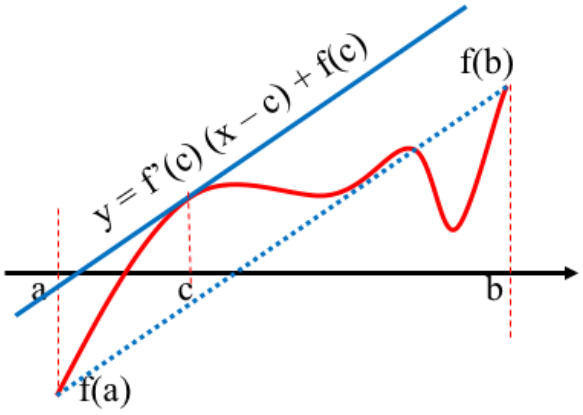

Figure 1.3: Mean value theorem

Theorem 5. ***Fundamental Theorem of Calculus*** *Suppose $f \in C[a,b]$ and F is the indefinite integral of f, i.e. $F'(x) = f(x)$ then*

$$\int_a^b f(x)dx = F(b) - F(a) \tag{1.17}$$

Theorem 6. ***Integration by parts*** *(It is not really a theorem but it is really useful). Assuming $f, g \in C[a,b]$ and $f, g \in C^1(a,b)$ then*

$$\int_a^b f'(x)g(x)dx = [f(b)g(b) - f(a)g(a)] - \int_a^b f(x)g'(x)dx \tag{1.18}$$

In almost every proof in Numerical Analysis we use one or many of this theorems. Always the key step is related with integrating by parts, or approximating a function with a Taylor's series. Therefore, you would really want to remember these theorems, not only for the course, but for sure for your career.

Chapter 2

Root-finding algorithms

Suppose that we want to find x such that $f(x) = 0$. There are two ways of solving it:

- **Closed Methods:** The root is enclosed in an interval. At each iteration we reduce the length of the interval. For example: Bisection
- **Open Methods:** We approximate the function with a model. At each iteration we calculate the new iteration based on the previous one, i.e. $x_n = g(x_{n-1})$. For example: Newton's method, fixed point iteration.

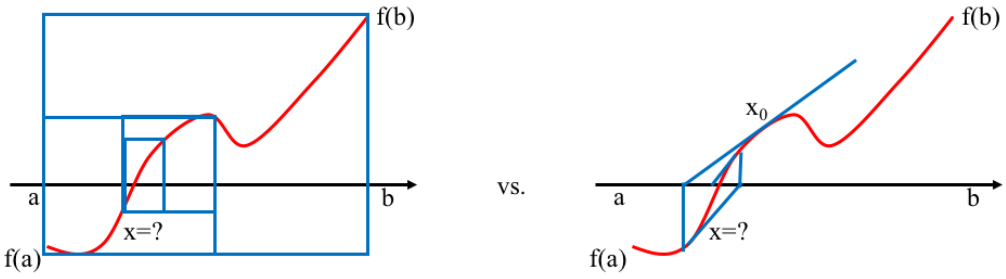

Figure 2.1: Closed methods vs. Open methods

Closed methods are robust, because we ensure that we are always reducing the level of uncertainty, therefore they always converge. In contrast, open methods are not that robust but they are efficient. Since they approximate the function, they preserve more information of the model. If they converge, they converge faster than closed methods. The problem is that we cannot bound the error at each iteration, and that explains why they can diverge.

2.1 Closed Methods

2.1.1 Bisection

Notice that in general, our function can have more than one root. Therefore the first step, regardless the method we are using, is to select an appropriate interval $[a, b]$ where the desired root is. We can do this plotting the function.

Also if we select this interval such that $f(a)$ and $f(b)$ have different sign, using the Intermediate Value theorem there exists $c \in [a, b]$ s.t. $f(c) = 0$. Therefore the simplest method is to reduce the interval $[a, b]$ such that we always preserve that $f(a)$ and $f(b)$ have different signs. If we are only using the sign of f, the healthiest option is to guess that $x_n = (a_n + b_n)/2$ the midpoint. Depending on the sign of $f(c)$ we update the interval, an we ensure that the error is at most the length of the interval.

Set $x_l = a$, $x_r = b$;
while $|x_r - x_l| > tol$ **do**
  $x = \frac{x_r + x_l}{2}$
  if $f(x) = 0$ **then**
    return x
  end
  if $f(x)f(x_l) < 0$ **then**
    $x_r = x$
  else
    $x_l = x$
  end
end
return x

Algorithm 1: Bisection Pseudo-code

Notice that at each iteration the maximum error is reduced by a half, therefore if $\{x_n\}$ is the set of iterates and x is the real root, we know that $x,\ x_n \in [a_n, b_n]$ then:

$$|x_n - x| \leq \frac{1}{2}|b_n - a_n| = \frac{1}{2^2}|b_{n-1} - a_{n-1}| = ... = \frac{1}{2^{n+1}}|b - a|$$

Therefore the rate of convergence is $\mathcal{O}(2^n)$. Additionally we can estimate an upper bound for the maximum number of iterations we would take given a tolerance $|x_n - x| < tol$. We have that

$$n \leq \log_2\left(\frac{b-a}{tol}\right) - 1 \tag{2.1}$$

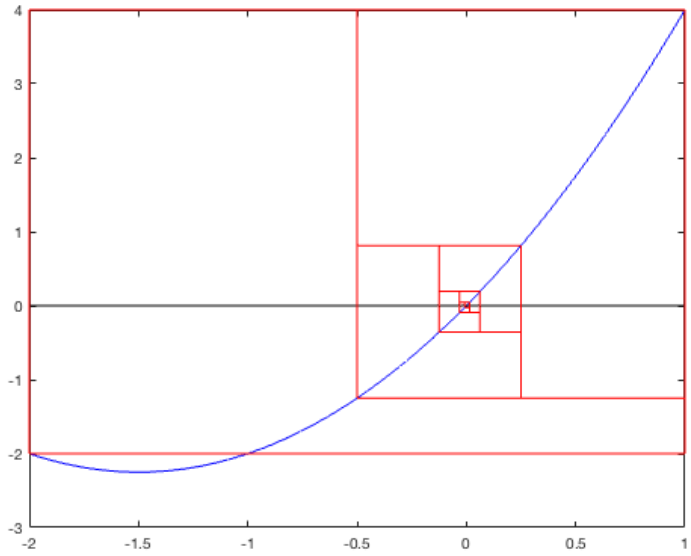

Figure 2.2: Bisection for $x^2 + 3x$

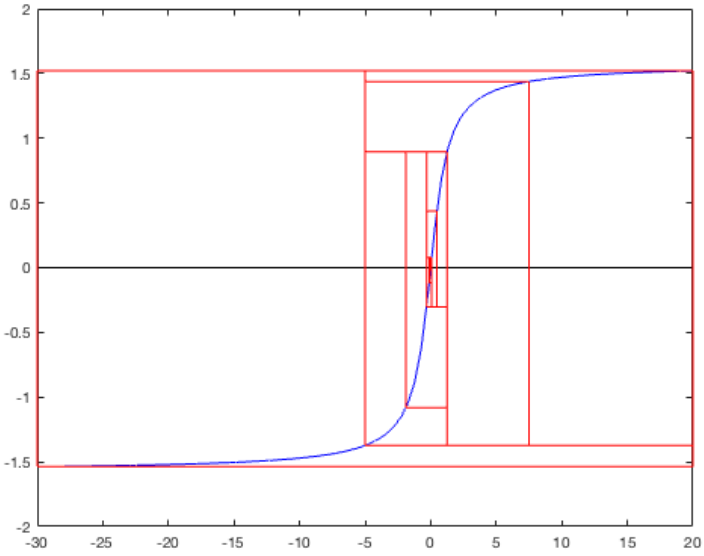

Figure 2.3: Bisection for $atan(x)$

2.1.2 False Position

Notice that in bisection, we only use the sign of the function. What would happen if we approximate the function with a line joining both extreme points?. This method is called **False Position**. Therefore we update the interval, guessing x_n is the root of the straight line between $f(a)$ and $f(b)$

Set $x_l = a$, $x_r = b$;
while $|x_r - x_l| > tol$ **do**
  $x = x_r - f(x_r)\frac{x_r - x_l}{f(x_r) - f(x_l)}$
  if $f(x) = 0$ **then**
    return x
  end
  if $f(x)f(x_l) < 0$ **then**
    $x_r = x$
  else
    $x_l = x$
  end
end
return x

Algorithm 2: False Position Pseudo-code

Although that this looks like a better alternative, we cannot ensure we are reducing the interval as much as $1/2$. Therefore depending on the function, its convergence can be faster or slower than Bisection. For example consider a function that is almost flat and negative, and suddenly goes straight up near one of the end points. In such a case, the interval reduction using false position is less than using bisection.

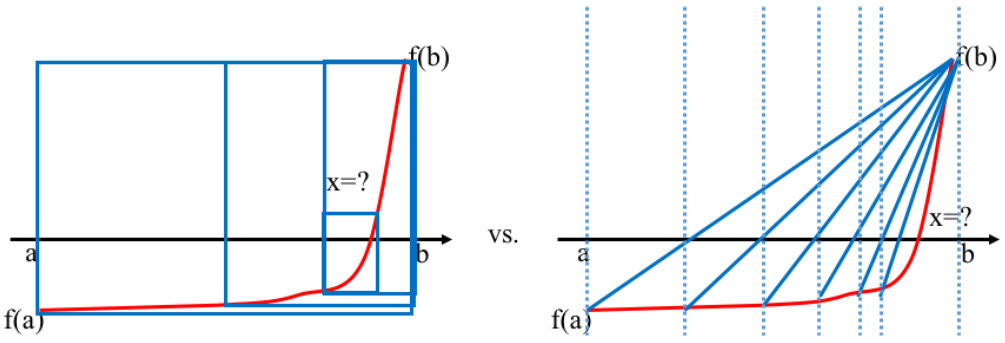

Figure 2.4: When Bisection is not always worse than False Position

2.2 Open methods

2.2.1 Fixed Point iteration

Open methods are based in expressing the next guess in terms of the previous ones. In general we have something of the form

$$x_{n+1} = g(x_n) \tag{2.2}$$

Notice that if we reach the solution of $f(x) = 0$, we would like to stay there, i.e.

$$x = g(x) \tag{2.3}$$

This means that the solution x is a fixed point of g.

Set p_0;
$p_1 = g(p_0)$; $n = 1$
while $|p_n - p_{n-1}| > tol$ *and* $n < max\_iterations$ **do**
  $p_{n+1} = g(p_n)$;
  $n = n + 1$;
end
return p_n

Algorithm 3: Fixed Point Iteration Pseudo-code

How can we choose g if our problem is in terms of f?

There are many different options for choosing g in order that x is a fixed point of it. For example we can take:

$$g_1 = x + f(x) \qquad g_2 = x - 10f(x)^2 \qquad g_3 = x - \frac{f(x)}{f'(x)} \qquad ...$$

The problem will be to know if starting from certain x_0, the sequence $\{x_n = g(x_{n-1})\} \to x$ as $n \to \infty$, i.e. if it converges and at which speed.

Example 3. *Check the implementation in class solving $f(x) = x^3 + x^2 - 3x - 3 = 0$ using:*

$$g_1(x) = \frac{x^3 + x^2 - 3}{3} \text{ Isolating } 3x \text{ and dividing by 3}$$
$$g_2(x) = -1 + \frac{3x + 3}{x^2} \text{ Isolating } x^3 \text{ and dividing by } x^2$$
$$g_3(x) = \sqrt{\frac{3 + 3x - x^2}{x}} \text{Isolating } x^3 \text{ and dividing by } x \text{ and taking sqrt}$$
$$g_4(x) = x - \frac{x^3 + x^2 - 3x - 3}{3x^2 + 2x - 3} \text{ Newton's method}$$

All of the schemes converge to the expected root? How is the order of convergence?

It is not easy to decide if any scheme and initial condition converges or not. Nevertheless, we have a theorem that helps us in a large number of cases.

Theorem 7. ***Fixed Point Theorem*** *If $g \in C[a,b]$ and $g \in C^1(a,b)$ such that:*

1. *$g : [a,b] \to [a,b]$ "The range goes into the domain"*
2. *$\exists k < 1 : |g'(x)| \leq k < 1$ for all $x \in (a,b)$ "The function is not steep"*

Then g has a unique fixed point $p \in [a,b]$ and the sequence $\{p_n = g(p_{n-1})\}$ starting from any $p_0 \in [a,b]$ converges to p such that $|p_n - p| = \mathcal{O}(k^n)$

Notice that if we have a function g that satisfies the two conditions of this theorem, we can say it converges because of the theorem. But of course there are functions that does not satisfy this theorem and they also converge.

How can we explain the **rate of convergence**? Notice that

$$\begin{aligned}
|p_n - p| &= |g(p_{n-1}) - g(p)| \text{ using the fixed point defintion} \\
&= |g'(\xi)||p_{n-1} - p| \text{ for } \xi \text{ between } p_{n-1} \text{ and } p, \text{ using mean value theorem} \\
&\leq k|p_{n-1} - p| \text{ by hypothesis} \\
&\leq k^n|p_0 - p| \text{ applying it } n \text{ times}
\end{aligned}$$

Therefore what is the **order of convergence**? If $p_n \to p$, since $\xi(p_n)$ is between p_n and p then $\xi(p_n) \to p$, then

$$\lim_{n\to\infty} \frac{|p_n - p|}{|p_{n-1} - p|} = \lim_{n\to\infty} |g'(\xi(p_n))| = |g'(p)| \tag{2.4}$$

If $|g'(p)| \neq 0$, then we have linear order of convergence.

If $|g'(p)| = 0$ we have a better order of convergence. Consider the Taylor series of $g(p_n)$ around p

$$g(p_n) = g(p) + g'(p)(p - p_n) + \frac{g''(p)}{2!}(p - p_n)^2 + ... + \frac{g^{(k)}(p)}{k!}(p - p_n)^k + \frac{g^{(k+1)}(\xi)}{(k+1)!}(p - p_n)^{k+1}$$

using the definition of fixed point, and the sequence $\{p_n\}$:

$$p_{n+1} = p + g'(p)(p - p_n) + \frac{g''(p)}{2!}(p - p_n)^2 + ... + \frac{g^{(k)}(p)}{k!}(p - p_n)^k + \frac{g^{(k+1)}(\xi)}{(k+1)!}(p - p_n)^{k+1}$$

Therefore if all the first k derivatives of p are zero, i.e. $g'(p) = g''(p) = ... = g^(k)(p) = 0$ and $g^(k+1)(p) \neq 0$ then

$$|p_{n+1} - p| = \frac{|g^{(k+1)}(\xi)|}{(k+1)!}|p - p_n|^{k+1} \Longrightarrow \lim_{n\to\infty} \frac{|p_n - p|}{|p_{n-1} - p|^{k+1}} = \frac{|g^{(k+1)}(p)|}{(k+1)!}$$

the order of convergence is $k + 1$.

2.2.2 Newton's method

Notice that if $f'(x_n) \neq 0$, we can approximate our function with the tangent line at the point $(x_n, f(x_n))$. And we can approximate the root as the zero of the function

$$y = f'(x_n)(x - x_n) + f(x_n)$$

Therefore we can define Newton's method as a fixed point iteration:

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}} \tag{2.5}$$

Notice that in this case

$$g(x) = x - \frac{f(x)}{f'(x)} \text{ therefore } g'(x) = \frac{f(x)f''(x)}{(f'(x))^2}$$

Let p be the fixed point and assuming $f'(p) \neq 0$, we have two results:

- Since $g'(p) = \frac{f(p)f''(p)}{(f'(p))^2} = 0$ and $g'(p)$ is continuous, we know there exists δ such that $|g'(x)| \leq k < 1$ if $x \in [p - \delta, p + \delta]$
- Since $g'(p) = 0$, therefore we have at least quadratic order of convergence

Notice that the first result tells us that if we are close enough to the root, we can apply the fixed point theorem, and we can ensure convergence. But close enough is hard to define.

Example 4. *Consider the example in class $f(x) = atan(x)$ that converges when $x_0 = -1$ but diverges when $x_0 = -2$. Check the implementation.*

2.2.3 Secant method

If f is really complicated, or is not even explicit, it is almost impossible to compute $f'(x_n)$. Therefore in the secant method we approximate

$$f'(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \tag{2.6}$$

Therefore the iteration is given by

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \tag{2.7}$$

Notice that this iteration looks similar to the update in False Position method. The difference is that in False position we take the line that crosses both extremes of the interval $[a, b]$, for which we know they have different signs. In secant method we take the line between two guesses, which we do not know a priori have different signs or not. Therefore the root is not necessarily between them.

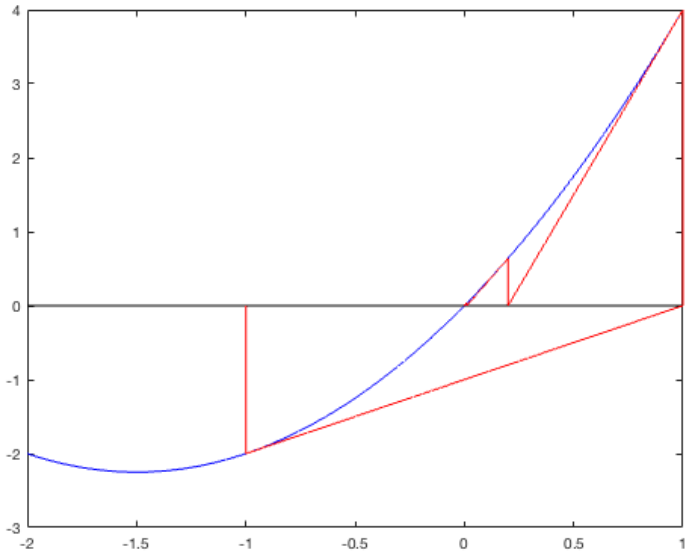

Figure 2.5: Newton for $x^2 + 3x$

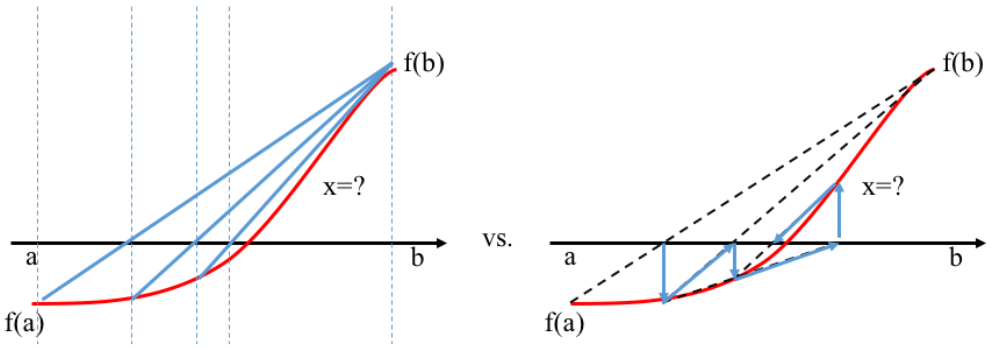

Figure 2.6: False Position v.s. Secant

2.3 Minimizing a function in 1D

Up to now, we have talked about solving nonlinear functions in 1D. This means that given a problem where we need to find $x \in \mathbb{R}$ such that it satisfies an equation, we can solve it if we express it as $f(x) = 0$. We have talked about two strategies: enclosing the root (using bisection) or approximating the model (using a fixed point iteration, such as Newton). We have seen that the first type of method is robust, whereas the second type is faster if it converges to the solution. And depending on whether or not we know more information of the function (sign or value or derivative), our approximation has more properties or not.

Now let us talk about optimizing a function in 1D. In general, this means that we need to find $\hat{x}$ such that minimizes $\phi(x)$.

$$\hat{x} \text{ s.t. } \underset{x \in \mathbb{R}}{\text{minimize}}\, \phi(x) \tag{2.8}$$

What about maximizing a function $\psi(x)$? Just take $\phi(x) = -\psi(x)$.

How can I identify a minimum point?

Recall that when we were finding the root of a function, we could identify if we were approaching the solution, or at least we could measure the error based on evaluating our guess x_n in $f(x)$. Since we were looking for x such that $f(x) = 0$, then we could say that a point x_{n+1} was better than x_n if $|f(x_{n+1})| < |f(x_n)|$, and moreover we can stop if $|f(x_n)|$ is sufficiently small.

But when we are minimizing a function, certainly we can measure if $f(x_{n+1}) < f(x_n)$ but we cannot define a stop criteria only based in the function value, since probably we do not know if there exists another point with even smaller evaluation. Therefore we need to evaluate the derivative at the point.

If we want to minimize a $\phi \in C^2[a, b]$:

Definition 11. *We say that $x^* \in (a, b)$ is a **critical point** if*

$$\phi'(x^*) = 0 \tag{2.9}$$

Using the Taylor series we can approximate $\phi(x)$ using a quadratic model

$$\phi(x^* + \Delta x) = \phi(x^*) + \phi'(x^*)\Delta x + \frac{\phi''(x^*)}{2}\Delta x^2 + \mathcal{O}(\Delta x^3) \tag{2.10}$$

Therefore for sufficiently small Δx, we can see that the approximation has a minimum or a maximum depending on $\phi''(x^*)$:

Definition 12. *Given a critical point x^**

- *If $\phi''(x^*) > 0$ then x^* is a **local minimizer***
- *If $\phi''(x^*) < 0$ then x^* is a **local maximizer***
- *If $\phi''(x^*) = 0$ then we need more information to decide if it is a minimizer, maximizer or a saddle point.*

Therefore if we call $f(x) = \phi'(x)$. To minimize $\phi(x)$ not only we need to find $\hat{x}$ such that $f(\hat{x}) = 0$ but also $f'(\hat{x}) > 0$. Notice also that finding a global minimizer $\tilde{x}$ is much more difficult because we also require that $\phi(x) > \phi(\tilde{x})\ \forall x \in [a, b]$. Therefore unless we know our function is of certain type (i.e. convex) , we only look for local minimizers.

How can we find a local minimizer?

One way is to solve $f(x) = 0$ for $f(x) = \phi'(x)$ using either closed or open methods and once we find x^* such that $f(x^*) \approx 0$, then we verify if $f'(x^*) = \phi''(x^*) > 0$.

If we have the information of ϕ, ϕ', ϕ'', we can apply Newton's method, and to ensure convergence we can use a hybrid method that also reduces the interval of uncertainty. If we do not know $f'(x) = \phi''(x)$, we can approximate it using secant method. We can also perform bisection in $f(x)$ and approximate $f'(x)$ to check the final result.

But what can we do if we do not even know $f(x) = \phi'(x)$?

Notice that in root-finding we always needed 2 pieces of information to reconstruct the model, either $f(a), f(b)$ or $f(x_n), f'(x_n)$, because we see it as a straight line. Straight lines do not have local minimizers. Instead we use a parabola and therefore we need three pieces of information: either $\phi(x_n), \phi'(x_n), \phi''(x_n)$, or $\phi(x_n), \phi'(a_n), \phi'(b_n)$ or $\phi(a), \phi(b), \phi(c)$

2.3.1 Golden Section

When we have three points $a < c < b$, we can ensure there is a (local) minimum in $[a, b]$ if we have the pattern "high, low, high". If we only start with three points, and we iterate to reduce the interval of uncertainty (similar to bisection), how can we do it efficiently? It is not longer as easy as evaluating the middle part as in bisection.

Notice that we need a forth point in order to choose a subinterval. If we do not distribute our points efficiently we will end up evaluating at each iteration 2 internal points. The special distribution to locate our points such that they are optimal ends up to be the golden ratio $\varphi = 1.618$. How can we see it?

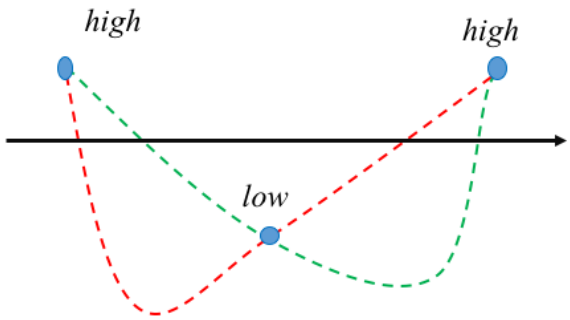

Figure 2.7: "high, low, high" pattern, but how can we reduce the interval to choose either the red or the green model.

Suppose we have the interval $[a_n, b_n]$ and we have the internal points c_n, d_n. First we would like to be symmetric, i.e.

$$c_n - a_n = d_n - b_n$$

. Therefore we can say $c_n = a_n + \alpha(b_n - a_n)$ and $d_n = a_n + (1-\alpha)(b_n - a_n)$

Second, you would like to reuse the point that is enclosed by the subinterval we choose. Without loss of generality, let us assume we choose the subinterval $[a_n, d_n]$ and c_n is the internal point. Therefore we would like

$$\begin{aligned}
\frac{c_n - a_n}{d_n - a_n} &= \frac{d_n - a_n}{b_n - a_n} \\
\frac{\alpha(b_n - a_n)}{(1-\alpha)(b_n - a_n)} &= \frac{(1-\alpha)(b_n - a_n)}{b_n - a_n} \\
\alpha &= (1-\alpha)^2 \\
0 &= \alpha^2 - 3\alpha + 1 \\
\boxed{\alpha = 1 + \frac{1-\sqrt{5}}{2} = 1 - \frac{1}{\varphi}} & \\
\boxed{1 - \alpha = \frac{-1+\sqrt{5}}{2} = \frac{1}{\varphi}} &
\end{aligned}$$

2.3.2 Parabolic Interpolation

Similar as the bisection case, using the golden section we are only using the function value to decide if a point remains or not, but we are not using it to estimate

Set a, b
$\alpha = 1 - 1/\varphi$
$c = a + \alpha * (b - a)$; $\phi_c = \phi(c)$
$d = a + (1 - \alpha)(b - a)$; $\phi_d = \phi(d)$
while $|b - a| > tol$ **do**
  if $\phi_c < \phi_d$ **then**
    $b = d$
    $d = c$; $\phi_d = \phi_c$
    $c = a + \alpha(b - a)$; $\phi_c = \phi(c)$
  else
    $a = c$
    $c = d$; $\phi_c = \phi_d$
    $d = a + (1 - \alpha)(b - a)$; $\phi_d = \phi(d)$
  end
end
return c

Algorithm 4: Golden Section Pseudo-code

the next iteration. When we were finding roots of the function, we introduce the method of false position to estimate the next iterate approximating the function with a straight line. Now, we can use a similar approach, an use three points to estimate a quadratic model for the point.

Between 3 different points there exists a unique parabola passing through them. If the general equation for the parabola is:

$$\psi(x) = \alpha x^2 + \beta x + \gamma$$

We want to find $\alpha,\ \beta,\ \gamma$ that defines the parabola between $(x_{n-2}, f(x_{n-2}))$, $(x_{n-1}, f(x_{n-1}))$, $(x_n, f(x_n))$. Therefore it should satisfy:

$$\begin{aligned} f(x_{n-2}) &= \alpha x_{n-2}^2 + \beta x_{n-2} + \gamma \\ f(x_{n-1}) &= \alpha x_{n-1}^2 + \beta x_{n-1} + \gamma \\ f(x_n) &= \alpha x_n^2 + \beta x_n + \gamma \end{aligned}$$

That if we write it in matrix form, we have

$$\begin{bmatrix} x_{n-2}^2 & x_{n-2} & 1 \\ x_{n-1}^2 & x_{n-1} & 1 \\ x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} f(x_{n-2}) \\ f(x_{n-1}) \\ f(x_n) \end{bmatrix} \tag{2.11}$$

Solving this linear system, we can find the coefficients of the quadratic model, and moreover, the minimum of it

$$\psi(x)' = 2\alpha x + \beta$$

If $\alpha > 0$, we take $x_{n+1} = -\beta/(2\alpha)$, which is the minimum of the quadratic model.

Chapter 3

Linear systems of equations

3.1 Linear Algebra Review

3.1.1 Singular Matrices

The goal for this review is to emphasize some key concepts that will determined the success of algorithms solving linear systems of equations.

Suppose that we have a linear system of equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + ... + a_{1m}x_m &= b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2m}x_m &= b_2 \\ &... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nm}x_m &= b_n \end{aligned}$$

Each of the equations represents a hyperplane in $\mathbb{R}^m$ and the solution is the intersection of all of them. Notice that depending on the equations, there can be a unique solution (only 1 point), infinite solutions (an affine subspace of dimension bigger than 1) or no solutions. We can also express the system as

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1m} \\ a_{21} & a_{22} & ... & a_{2m} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{bmatrix} = \mathbf{b} \tag{3.1}$$

And using the geometric intuition, now we know that not every system $Ax = b$ where $A \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$ has a unique solution in $\mathbb{R}^m$.

We can also see the system as a linear combination of column vectors of A:

$$\sum_{j=1}^{m} \mathbf{a_j} x_j = \begin{bmatrix} a_{11} \\ a_{21} \\ ... \\ a_{n1} \end{bmatrix} x_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ ... \\ a_{n2} \end{bmatrix} x_2 + ... + \begin{bmatrix} a_{1m} \\ a_{2m} \\ ... \\ a_{nm} \end{bmatrix} x_m = \mathbf{b} \tag{3.2}$$

And another way to understand $Ax = b$ is seeing it as a linear transformation $T : \mathbb{R}^m \to \mathbb{R}^n$ that transforms vectors from $\mathbb{R}^m$ to $\mathbb{R}^n$. There are two important sets for any transformation:

Definition 13. *Let $A \in \mathbb{R}^{n \times m}$ a linear transformation then:*

- *The **range** of A, $range(A) \subset \mathbb{R}^n$, is the image of the transformation, i.e.*

$$range(A) = \{y \in \mathbb{R}^n | \exists x : Ax = y\}$$

  or equivalent $y \in range(A) \iff$ it is a linear combination of the column vectors of b

- *The **nullspace** of A, $null(A) \subset \mathbb{R}^m$, is the kernel of the transformation, i.e.*

$$null(A) = \{x \in \mathbb{R}^m | Ax = 0\}$$

Therefore given a system $Ax = b$:

- If $b \notin range(A)$, i.e. b is not a linear combination of columns of A, then the system does not have a solution.
- If $dim(null(A)) > 0$ then if x is a solution (i.e $Ax = b$) then for all $\tilde{x} \in null(A)$, $x + \tilde{x}$ is also a solution of the system.

Definition 14. *So if we talk about a square matrix $A \in \ltimes \times \ltimes$, we say that the system $Ax = b$ has unique solution if A is **invertible**, which is equivalent to:*

- $dim(null(A)) = 0$
- $range(A) = \mathbb{R}^n$
- *The columns of A are linearly independent. Similarly with the rows.*
- $\det(A) \neq 0$
- *There exists A^{-1} such that $AA^{-1} = A^{-1}A = I$*

*A matrix that is not invertible is called **singular***

Why are we concerned about A being singular or not?

We know that if A is invertible, we will have a unique solution. Otherwise if A singular we will have either infinite solution or none.
Now the problem is that there exists matrices that are invertible, but are nearly to be singular. For example consider

$$A = \begin{bmatrix} 1+\epsilon & 1 \\ 3 & 3 \end{bmatrix}$$

with $\epsilon > 0$ the matrix is invertible with

$$A^{-1} = \frac{1}{3\epsilon} \begin{bmatrix} 3 & -1 \\ -3 & 1+\epsilon \end{bmatrix}$$

But when $\epsilon = 0$, A is singular and therefore we cannot longer compute A^{-1}. Small perturbations, i.e. changes in ϵ (for example due to roundoff errors) produce huge changes in the solution. This type of problems are **ill conditioned** problems (recall the discussion in Chapter 1 about well conditioned problems and stability).

3.1.2 Famous matrices

- Diagonal
- Triangular (It has eigenvalues in the diagonal)
- Symmetric $A^t = A$
- Positive definite $\forall x \neq 0$, $x^T A x > 0$
- Semi-Positive definite $\forall x \neq 0$, $x^T A x \geq 0$
- Orthogonal $Q^T Q = I$. Its column vectors form an orthonormal set:

  $$q_i^T q_j = \begin{cases} 0 & i \neq j \text{ (orthogonal vectors)} \\ 1 & i = j \text{ (2-norm is equal to 1)} \end{cases}$$

  If Q is square then we also have $QQ^T = I$, otherwise this is not true.
- Sparse matrix: most of its elements are zero. Instead of having $\mathcal{O}(n^2)$ entries, usually it has $\mathcal{O}(n)$.

3.1.3 Nice Factorizations

Working with any A can be exhausting and messy. Therefore we prefer to factorize A as a product of matrices, each one with nice properties:

Name	Shape	Explanation	Properties
Eigenvalue Decomp.	$X\Lambda X^{-1}$	X invertible: eigenvectors Λ diagonal: eigenvalues	Only when multiplicity algebraic = geometric It is used for: $A^k = X\Lambda^k X^{-1}$
Jordan Form	XJX^{-1}	X invertible J Block Diagonal	Any matrix
Unitary Diagonal.	$Q\Lambda Q^*$	Q unitary: eigenvectors Λ diagonal: eigenvalues	Only Normal matrices $AA^* = A^*A$
Schur Decomp.	QTQ^*	Q unitary T (upper) triangular	Any Matrix Eigenvalue revealing
SVD	$U\Sigma V$	U, V unitary Σ diagonal, $\sigma_i \geq \sigma_{i+1} \geq 0$ $U \in \mathbb{R}^{m\times m}$ (Full) $U \in \mathbb{R}^{m\times n}$ (Thin)	Any Matrix Reveals the behavior of transformation
QR	QR	Q orthogonal R Upper triangular	$Ax = b \Longrightarrow Rx = Q^T b$ (using backward substitution)
LU	LU	L lower triangular, $l_{ii} = 1$ U Upper triangular	$det(U) = det(A)$ To solve $Ax = b$: $Ly = b,\ Ux = y$
Cholesky	$R^T R$	R upper triangular	Stable it A is positive definite

Table 3.1: Useful Factorizations for a matrix A

3.1.4 Eigenvalues

In problems where we want to find stable solutions for physical systems, usually we need to evaluate A^k. If A is dense, each matrix-matrix product cost $\mathcal{O}(n^3)$ operations. Therefore computing $A...A$ k times is too expensive ($\mathcal{O}(kn^3)$), and in addition we deal with roundoff errors.

Nevertheless, for certain type of matrices, *diagonalizable matrices*, we can express $A = X\Lambda X^{-1}$ where Λ is diagonal. In such a case $A^k = X\Lambda^k X^{-1}$ will cost $\mathcal{O}(n^3)$ for any k.

In general, if for $\lambda \in \mathbb{R}$ there exists $x \neq 0$ such that

$$Ax = \lambda x$$

we say that λ is an eigenvalue of A, and x is a corresponding eigenvector.

Notice that to find eigenvalues of A is equivalent to find the roots of the characteristic polynomial $\rho_A(\lambda)$.

$$Ax = \lambda x \implies \rho_A(\lambda) = det(A - \lambda I) = 0$$

And we define to types of multiplicity for an eigenvalue:

- **Algebraic**: Multiplicity of root in $\rho_A(\lambda)$
- **Geometric**: $dim(E_\lambda) = \{x \in \mathbb{R}^n | Ax = \lambda x\}$

And we know that Algebraic multiplicity $\geq$ Geometric multiplicity. If we have equality for all the eigenvalues, then A is diagonalizable.

Do you think we find the eigenvalues finding the roots of $\rho_A(\lambda)$?.

Not really. Usually the matrices we are interested in belong to $\mathbb{R}^{n\times n}$, where n is large. Therefore the characteristic polynomial is order n too. And generally finding the roots of high order polynomials is really unstable. Therefore to find eigenvalues and eigenvectors we use iterative methods to solve them (For example, power method). This methods are out of scope of this course.

3.2 Direct Methods: Gauss Elimination

Given a system $Ax = b$ we can create the augmented version

$$[A|b] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & ... & a_{1m} & b_1 \\ a_{21} & a_{22} & ... & a_{2m} & b_2 \\ ... & ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nm} & b_n \end{array}\right] \tag{3.3}$$

Depending on A this system can be really easy or really hard to solve. For example of A is diagonal, we only need to divide b_j by a_{jj} to find x_j.

Another relatively easy type of matrices are the lower and upper triangular matrices. For example with an upper triangular matrix in $\mathbb{R}^{n\times n}$, every variable x_k only depends on the following $x_{k+1}, ..., x_{n-1}, x_n$. Therefore we can move *backward*, starting from x_n, then x_{n-1}, and so on. This method is called *Backward Substitution*.

In general, the goal of **Gauss Elimination** is first reduce any matrix A into an upper triangular form and then solve the system using backward substitution. To do this we use the following three rules:

- $r_i \leftrightarrow r_j$ Interchange rows i and j
- $r_i \leftarrow \alpha r_i$ Multiply a row by a scalar
- $r_i \leftarrow r_i + \beta r_j$ Add a multiple to the row j to the row i.

Let us consider the system

$$[A|b] = \left[\begin{array}{ccc|c} 2 & 1 & 0 & 2 \\ -1 & 1 & -1 & 1 \\ -3 & 4 & -4 & 2 \end{array}\right]$$

And apply Gaussian elimination only using the third rule:

$$[A|b] = \left[\begin{array}{ccc|c} 2 & 1 & 0 & 2 \\ -1 & 1 & -1 & 1 \\ -3 & 4 & -4 & 2 \end{array}\right] \sim \begin{array}{l} r_2 \leftarrow r_2 + 1/2 r_1 \\ r_3 \leftarrow r_3 + 3/2 r_1 \end{array} \left[\begin{array}{ccc|c} 2 & 1 & 0 & 2 \\ 0 & 3/2 & -1 & 2 \\ 0 & 11/2 & -4 & 5 \end{array}\right]$$

$$\sim \begin{array}{l} r_3 \leftarrow r_3 - 11/3 r_2 \end{array} \left[\begin{array}{ccc|c} 2 & 1 & 0 & 2 \\ 0 & 3/2 & -1 & 2 \\ 0 & 0 & -1/3 & -7/3 \end{array}\right]$$

And then doing backward substitution

$$\begin{aligned} x_3 &= \frac{-7/3}{-1/3} = 7 \\ x_2 &= \frac{1}{3/2}\left(2 - (-1)x_3\right) = \frac{2}{3}(2+7) = 6 \\ x_1 &= \frac{1}{2}\left(2 - (1)x_2 - (0)x_3\right) = \frac{1}{2}\left(2 - 6\right) = -2 \end{aligned}$$

Notice that we can express the set of transformations as multiplying A by the matrices

$$L_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 3/2 & 0 & 1 \end{bmatrix} \qquad L_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -11/3 & 1 \end{bmatrix}$$

Therefore

$$L_2 L_1 A = U = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 3/2 & -1 \\ 0 & 0 & -1/3 \end{bmatrix}$$

Moreover notice that

$$L_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ -3/2 & 0 & 1 \end{bmatrix} \qquad L_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 11/3 & 1 \end{bmatrix}$$

Then

$$A = \underbrace{L_1^{-1} L_2^{-1}}_{L} U = LU$$

Where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ -3/2 & 11/3 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 3/2 & -1 \\ 0 & 0 & -1/3 \end{bmatrix}$$

Therefore in general, using Gauss elimination, we can get a LU decomposition of A. If we are solving only for one b, this decomposition is not relevant. But imagine the case where we have the system $Ax_i = b_i$, where we need the solution of the system x_i for different right hand side vectors b_i. Therefore with a LU factorization we can transform the problem as follows:

$$Ax = LUx = b \Longrightarrow \begin{cases} \text{First solve} & Ly = b \\ \text{then solve} & Ux = y \end{cases} \tag{3.4}$$

Notice that for the first step you perform *Forward substitution* and for the second one *Backward substitution.*

3.2.1 Number of flops

Why do we want to use Gauss elimination (or LU), instead of just computing the inverse?

```
for k = 1 : n − 1 do
    for i = k + 1 : n do
        l_ik = a_ik / a_kk
        for j = k + 1 : n do
            a_ij = a_ij − l_ik a_kj
        end
        b_i = b_i − l_ik b_k
    end
end
```

Algorithm 5: Gauss Elimination Pseudo-code

```
for k = n : −1 : 1 do
    sum = 0
    for j = k + 1 : n do
        sum = sum + a_kj x_j
    end
    x_k = (b_k − sum) / a_kk
end
```

Algorithm 6: Backward substitution Pseudo-code

Notice that we can find the inverse of a matrix applying Gauss elimination to the system

$$\left[\begin{array}{c|c} A & I \end{array}\right] \sim ... \sim \left[\begin{array}{c|c} I & A^{-1} \end{array}\right]$$

Imagine that we do not have roundoff errors, therefore why don't we just compute the inverse of a matrix instead of computing the LU factorization. The answer is because of the number of operations (flops). Let us compute the number of flops for different methods. We count the sums and the products, and each of the sums are the iterations inside the loops. Gauss Jordan refers to the method that instead

of reducing the matrix to a triangular form, leaves it as a diagonal.

$$
\begin{aligned}
flops(Gauss\ Elimination) &= \sum_{k=1}^{n-1}\left(\sum_{i=k+1}^{n}\left(3+\sum_{j=k+1}^{n}2\right)\right)\\
&= \sum_{k=1}^{n-1}\left(\sum_{i=k+1}^{n}(3+2(n-k))\right)\\
&= \sum_{k=1}^{n-1}(n-k)(3+2(n-k))\\
&= 2\sum_{k=1}^{n-1}k^2+\frac{3}{2}k\\
&\boxed{= \frac{2}{3}n^3+\mathcal{O}(n^2)}
\end{aligned}
$$

$$
\begin{aligned}
flops(Backward\ substitution) &= \sum_{k=1}^{n}\left(2+\sum_{j=k+1}^{n}2\right)\\
&= 2\sum_{k=1}^{n}(1+n-k)\\
&= 2\sum_{k=1}^{n}k\\
&\boxed{= n^2+\mathcal{O}(n)}
\end{aligned}
$$

$$
\begin{aligned}
flops(Gauss\ Jordan) &= \sum_{k=1}^{n} \left(\sum_{\substack{i=1:n \\ i \neq k}} \left(3 + \sum_{j=k+1}^{n} 2 \right) \right) \\
&= \sum_{k=1}^{n} \left(\sum_{\substack{i=1:n \\ i \neq k}} (3 + 2(n-k)) \right) \\
&= \sum_{k=1}^{n} (n-1)(3 + 2(n-k)) \\
&= 2(n-1) \sum_{k=1}^{n} k + \frac{3}{2} \\
&\boxed{= n^3 + \mathcal{O}(n^2)}
\end{aligned}
$$

$$
\begin{aligned}
flops(Inverse) &= \sum_{k=1}^{n-1} \left(\sum_{i=k+1}^{n} \left(1 + 2n + \sum_{j=k+1}^{n} 2 \right) \right) \\
&= \sum_{k=1}^{n-1} \left(\sum_{i=k+1}^{n} (1 + 2n + 2(n-k)) \right) \\
&= \sum_{k=1}^{n-1} (n-k)(1 + 2n + 2(n-k)) \\
&= 2 \sum_{k=1}^{n-1} k^2 + nk + k \\
&\boxed{= \frac{5}{3} n^3 + \mathcal{O}(n^2)}
\end{aligned}
$$

As you can see, finding the Inverse or finding LU factorization are both $\mathcal{O}(n^3)$. The difference is in the coefficient in front. Finding the inverse is almost three times as expensive as finding the LU.

Also notice that backward substitution (and therefore Forward substitution) are $\mathbb{O}(n^2)$. Therefore once we have the LU, solving (3.4) is really cheap. Think about what is the number of flops of solving a diagonal or quasi-diagonal system (i.e. tridi-

agonal)!

3.2.2 Pivoting

Does this method work for any invertible matrix?

For example take

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

In this case, with standard Gauss elimination, we will divide by 0. This will break the method!. Therefore we need to use the rule of switching rows in order to get a value different than zero (**a pivot**). Notice that in a computer, we will not only find the problem when the entry is exactly zero but when is relatively small with respect to the rest of the entries of the column. Therefore we would like to apply **pivoting** in order to have a more stable algorithm.

Clearly, this pivoting changes the factorization of A as:

$$PA = LU$$

where L is lower triangular, U is upper triangular and P is a permutation matrix. To make the algorithm more stable we can even permute rows and columns, including an additional permutation matrix Q

$$PAQ = LU$$

In the case of solving the system using this permutation matrices, we need to modify x and b appropriately.

3.2.3 Cholesky Factorization

What happens if A is symmetric? Or even symmetric positive definite?

Notice that from

$$A = LU$$

if all the diagonal entries of U are different from zero, we can form the decomposition

$$A = LD\tilde{U}$$

where $\tilde{U}$ is upper triangular, with diagonal entries equal to one, and we can get it dividing each row U_i by the diagonal entry u_{ii}. Notice that if A is symmetric, we can have a symmetric decomposition

$$A = LDL^T$$

And moreover, if all the entries of D are > 0 we can take the square root and

$$A = LD^{1/2}D^{1/2}L^T$$

If we call $G = LD^{1/2}$ then we have that

$$A = GG^T$$

where G is lower triangular matrix with positive entries in the diagonal. This is called **Cholesky factorization**. If the matrix is far from being singular and moreover it is symmetric positive definite (i.e. all the eigenvalues are positive and far from zero), then the Cholesky factorization for A exists and the method is stable.

3.3 Conditioning

3.3.1 Vector and Matrix Norms

Definition 15. *A **norm** is a function $||\cdot|| : V \to \mathbb{R}$ such that:*

- $||x|| \geq 0$; $||x|| = 0 \iff x = 0$
- $||\alpha x|| = |\alpha|\ ||x||$
- $||x + y|| \leq ||x|| + ||y||\ \forall x, y \in \mathbb{R}^n$

Some typical vector norms defined in $\mathbb{R}^n$ are

$$||x||_2 = \sqrt{x^t x} = \left(\sum_{j=1}^{n} |x_j|^2\right)^{1/2}$$

$$||x||_1 = \sum_{j=1}^{n} |x_j|$$

$$||x||_p = \left(\sum_{j=1}^{n} |x_j|^p\right)^{1/p}$$

$$||x||_\infty = \max_{1 \leq j \leq n} |x_i|$$

How can we define a matrix norm?

First we can see a Matrix as object in $\in \mathbb{R}^{\times \times >}$. This means that we can unwrapped the rows of A and we can have a large vector belonging to $\mathbb{R}^{\times \times >}$. If we compute the 2-norm os this vector, it is called the **Frobenius norm**

$$||A||_F = \sqrt{\sum_{i,j} a_{ij}^2} \tag{3.5}$$

And as a nice fact, we can even define an Inner Product between matrices based on this norm:

$$\langle A, B \rangle = tr(A^T B)$$

On the other hand we can see a Matrix as operator : $\mathbb{R}^n \to \mathbb{R}^m$. This means we can analyze how the matrix transforms the vectors from one space to the other. In this way we can define the **Induced norm**:

$$||A||_p = \sup_{x \neq 0} \frac{||Ax||_p}{||x||_p} \text{ for } 1 \leq p \leq \infty \tag{3.6}$$

Some of the typical induced norms are:

$$||A||_1 = \max_j \sum_i |a_{ij}| \qquad ||A||_\infty = \max_i \sum_j |a_{ij}| \qquad ||A||_2 = \max |\sigma_i|$$

where σ_i corresponds to the singular value, that are the same as the diagonal terms in Σ in the SVD, singular value decomposition. Also

$$||A||_2 = \sqrt{\rho\left(AA^T\right)} \text{ where } \rho(M) = \max_{\lambda_i \in \Lambda(M)} |\lambda_i| \text{ is the spectral radious}$$

All the induced norms satisfy the **sub-multiplicative property**:

$$||AB|| \leq ||A||\ ||B||$$

3.3.2 Condition number

How can we measure the expected error in solving a system?

As we talked in section 1, if we have perfect arithmetic (i.e. we do not count roundoff errors), the sources of error depend on the problem itself (if it is a well conditioned problem) and the method we use (if it is stable or not). If we know before hand that our problem tends to be ill-conditioned, then we will use a method that is really robust.

In the case of linear systems, let us understand where the error comes from. First let us define the residual (that we can measure in reality, without knowing the real solution) as:

$$r = b - A\tilde{x} = Ax - A\tilde{x} = A(x - \tilde{x})$$

then the absolute error is:

$$||x - \tilde{x}|| = ||A^{-1}r|| \leq ||A^{-1}||\ ||r||$$

If we prefer to consider the relative error, and using that $||b|| \leq ||A||\ ||x||$, we have

$$\frac{||x - \tilde{x}||}{||x||} \leq ||A||\ ||A^{-1}|| \frac{||r||}{||b||} \tag{3.7}$$

Here we see two sources from error:

- The method and how the residual can change from method to method $||r||$
- And the problem itself regardless of the method we use.

In this setting, choosing a specific norm, we define the **condition number of a matrix** as

$$\kappa(A) = ||A||\ ||A^{-1}|| \tag{3.8}$$

Notice that we have the following properties:

- Using the sub-multiplicative property $1 = ||I|| = ||AA^{-1}|| \leq ||A||\ ||A^{-1}||$
- Using that for orthogonal matrices $Q^T = Q^{-1}$ and $||Q|| = ||Q^T||$ then $1 = ||Q||||Q^{-1}||$
- And using the SVD decomposition

$$k_2 = \frac{\sigma_{\max}}{\sigma_{\min}}$$

Notice that from (3.7), if a matrix has a large condition number, the bound in the relative error of $Ax = b$ is bigger, therefore we would need a robust method. If you want to explore more about condition number I recommend to read Ascher section 5.8 or Bradie section 3.4, or for deeper understanding a Numerical Linear Algebra book such as the one from Trefethen and Bau.

3.4 Linear Least Squares

Let us assume we have a set of stock prices along certain period of time, and we want to adjust a model to it. Each measure has certain type of error, and of course, if we pass by all the points, probably we will be overfitting the model. Therefore we prefer to adjust a model and minimize the error between the curve and the data. If the model is

$$f(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + ... + \alpha_{m-1} t^{m-1}$$

Then the error between the model and the data (t_k, p_k) is given by

$$\begin{bmatrix} r_1 \\ r_2 \\ ... \\ r_n \end{bmatrix} = \begin{bmatrix} p_1 - f(t_1) \\ p_2 - f(t_2) \\ ... \\ p_n - f(t_n) \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ ... \\ p_n \end{bmatrix} - \begin{bmatrix} 1 & t_1 & ... & t_1^{m-1} \\ 1 & t_2 & ... & t_1^{m-1} \\ ... & ... & ... & ... \\ 1 & t_n & ... & t_n^{m-1} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ ... \\ \alpha_{m-1} \end{bmatrix} \tag{3.9}$$

In matrix form we have

$$r = b - Ax$$

In here we are assuming that A has full column rank, this means that all the variables are independent one from the other. If A were square, we could just solve the system such that $r = 0$. But given that A is not square, we should minimize $||r||_2$.

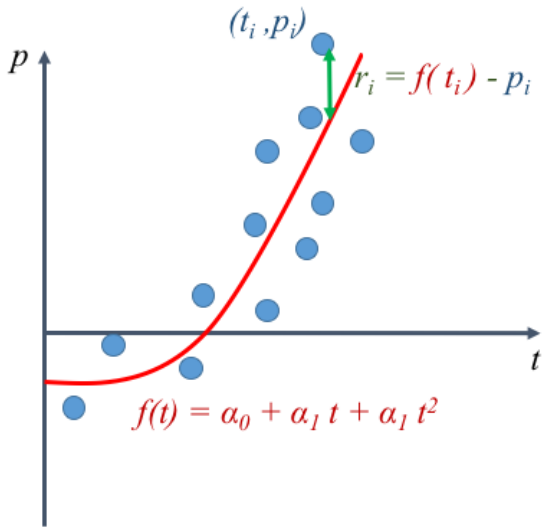

Figure 3.1: **Data Fitting problem**: Let us suppose we have the data series $\{(t_i, p_i)\}$ and we want to see how the model $f(t) = \alpha_0 + \alpha_1 t + \alpha_2^2 t^2$ approximates the price.

Why $||\ ||_2$ *and not other norm?*

The other 2 popular norms are $||\ ||_1$ and $||\ ||_\infty$. The first one helps to get rid of outliers (it makes the coefficients of the model also to be sparse) whereas the second one keeps track of the worst case error (a min-max problem). The problem with this 2 norms is that they are not differentiable, therefore we can not apply the same techniques as for $||\ ||_2$. In such a case we use Linear Programing, a constrained optimization problem with both linear objective function and constraints.

Therefore in linear least squares, we are looking for x such that minimizes

$$\min_{x \in \mathbb{R}^m} \frac{1}{2}||b - Ax||_2^2$$

Let us call

$$\begin{aligned}
\psi(x) &= \frac{1}{2}||b - Ax||_2^2 \\
&= \frac{1}{2}(b - Ax)^T(b - Ax) \\
&= \frac{1}{2}\left(b^T b - 2b^T Ax + x^T A^T Ax\right) \\
&= \frac{1}{2}x^T A^T Ax - (A^T b)^T x + \frac{1}{2}b^T b
\end{aligned}$$

In this case to have a minimum, as in 1D we need a critical point, (i.e. $\nabla\psi = 0$) and a second derivative requirement ($\nabla^2\psi$ is positive definite).

If it is not easy to see how are the derivatives from ψ let us see it by components:

$$\psi(x) = \frac{1}{2}\sum_{i=1}^{n}\left(b_i - \left(\sum_{k=1}^{m} a_{ik}x_k\right)\right)^2$$

Therefore taking the partial derivative with respect of x_j we have

$$\begin{aligned}
\frac{\partial\psi}{\partial x_j} &= \frac{1}{2}\frac{\partial}{\partial x_j}\sum_{i=1}^{n}\left(b_i - \left(\sum_{k=1}^{m} a_{ik}x_k\right)\right)^2 \\
&= \sum_{i=1}^{n}\left(b_i - \left(\sum_{k=1}^{m} a_{ik}x_k\right)\right)\frac{\partial}{\partial x_j}\left(b_i - \left(\sum_{k=1}^{m} a_{ik}x_k\right)\right) \\
&= \sum_{i=1}^{n}\left(b_i - \left(\sum_{k=1}^{m} a_{ik}x_k\right)\right)\left(\frac{\partial b_i}{\partial x_j} - \left(\sum_{k=1}^{m} a_{ik}\frac{\partial x_k}{\partial x_j}\right)\right) \\
&= \sum_{i=1}^{n}\left(b_i - \left(\sum_{k=1}^{m} a_{ik}x_k\right)\right)\left(-\sum_{k=1}^{m} a_{ik}\delta_{kj}\right) \\
&= \sum_{i=1}^{n}\left(b_i - \left(\sum_{k=1}^{m} a_{ik}x_k\right)\right)(-a_{ij}) \\
&= -\sum_{i=1}^{n} a_{ij}\left(b_i - \left(\sum_{k=1}^{m} a_{ik}x_k\right)\right)
\end{aligned}$$

If we express the gradient in matrix form we have that

$$\nabla\psi(x) = -A^T(b - Ax) = A^TAx - A^Tb \tag{3.10}$$

And the components of the Hessian are $\frac{\partial^2 g}{\partial x_j \partial x_l}$. Therefore:

$$\begin{aligned}
\frac{\partial^2\psi}{\partial x_j\partial x_l} &= \frac{\partial}{\partial x_l}\sum_{i=1}^{n} a_{ij}\left(\left(\sum_{k=1}^{m} a_{ik}x_k\right) - b_i\right) \\
&= \sum_{i=1}^{n} a_{ij}\left(\sum_{k=1}^{m} a_{ik}\frac{\partial x_k}{\partial x_l}\right) \\
&= \sum_{i=1}^{n} a_{ij}a_{il}
\end{aligned}$$

Then the Hessian in matrix form is given by

$$\nabla^2\psi(x) = A^TA \tag{3.11}$$

Since A^TA is positive definite, then now we can solve the least squares problem as finding x such that

$$A^TAx = A^Tb \tag{3.12}$$

And they are called the **normal equations**, because they satisfy that

$$A^T(Ax - b) = A^Tr = 0$$

this means that the residual is normal to the range of A.

How can we solve them?

Notice that (3.12), if A^TA is invertible (i.e A is full column rank) is equivalent to

$$x = \left(A^TA\right)^{-1}A^Tb$$

And therefore we can call $A^\dagger = \left(A^TA\right)^{-1}A^T$ the pseudo inverse of A, and as in linear systems we can find that

$$\frac{||x - \tilde{x}||}{||x||} \leq ||A||\ ||A^\dagger|| \frac{||r||}{||b||} \tag{3.13}$$

Therefore we can also define a condition number for rectangular matrices $A \in \mathbb{R}^{n\times m}$ and in 2-norm is equal to

$$\kappa(A) = \frac{\sigma_1}{\sigma_m}$$

Therefore if we solve (3.12) using Gauss elimination, or in this case Cholesky Factorization (since A^TA is symmetric positive definite), then the error will not depend on $\kappa(A)$ but on $\kappa(A^TA)$. And from the SVD we can see that

$$\kappa(A^TA) = \frac{\sigma_1^2}{\sigma_m^2} = (\kappa(A))^2$$

Therefore if the condition number of A is large, when we solve using normal equations we duplicate, therefore it is more ill conditioned.

In order to avoid that, we would like to use another methods, such us

- Use **QR factorization**. At the begin of the chapter, we talked about factorizing $A = QR$ where Q is orthogonal and R is upper triangular. We can get this factorization as applying Gram-Schmidt to the column vectors of A, to transform it into a orthonormal basis of the range. Although Gram-Schmidt is not really stable by itself, there are other methods, such as Householder reflection and Givens rotations, to achieve this factorization. It is useful because:

$$||Ax - b||_2^2 = ||QRx - b||_2^2 = ||Rx - Q^Tb||_2^2$$

Therefore the only way to reduce the residual is to solve exactly for the first m equations of R. And the condition number of R is comparable with the one of A. As an additional comment when we have $A \in \mathbb{R}^{n \times m}$ $n! = m$ full column rank, and $b \in \mathbb{R}^n$, if we write in MATLAB `A\b`, it returns the least squares solution using QR factorization. Check the last part of `http://www.mathworks.com/help/matlab/ref/mldivide.html` .

- We can only use QR factorization if A is non singular. In the case A is singular or almost singular we can use **SVD factorization**. Therefore we get

$$||Ax - b||_2^2 = ||U\Sigma V^T x - b||_2^2 = ||\Sigma V^T x - U^T b||_2^2$$

  And we can neglect the equations where σ_i is almost zero.

As we can always expect, as the method gets more robust, the cost of computing it increases. Therefore depending on the problem and the conditioning of A, we would like to use a different method.

Chapter 4

Non Linear systems of equations

4.1 Newton's Method

After giving a general introduction of Scientific Computing, in chapter 2 we started with finding the solutions of nonlinear equations of the shape $f(x) = 0$. Some of the methods were based in reducing the uncertainty interval (closed methods), whereas others used a fixed point iteration to define the guess (Open methods: $x_{n+1} = g(x_n)$). In general closed methods were robust but open methods had faster convergence. Therefore in HW 1, we explored hybrid methods, mixing both approaches to ensure convergence.

After in chapter 3, we deal with linear systems of equations of the shape $Ax = b$, and they are linear because all the operations between unknowns are additions or multiplications by a scalar. Now we will deal with a more general problem. Let us consider

$$\text{Find } \mathbf{x} = \begin{bmatrix} x_1 \\ ... \\ x_m \end{bmatrix} \in \mathbb{R}^m \text{ s.t. } \mathbf{F}(\mathbf{x}) = 0 \text{ where } \mathbf{F} = \begin{bmatrix} f_1(x_1, ..., x_m) \\ f_2(x_1, ..., x_m) \\ ... \\ f_n(x_1, ..., x_m) \end{bmatrix} \tag{4.1}$$

The first remark we should do is that in general we can not generalize closed methods in this multidimensional setting, because we do not have a graphic picture of the function neither basic conditions on the neighborhood of a root that should been satisfied. Therefore we only have open methods. Here we will talk about Newton's method.

As in 1D case, we approximate our function using Taylor series:

Definition 16. *Let* $\mathbf{x} = [x_1, ..., x_m]^T \in \mathbb{R}^m$ *and* $\mathbf{F} = [f_1, ..., f_n]^T$ *with bounded derivatives up to order at least 2, then for any direction* $\mathbf{p} = [p_1, ..., p_m]^T \in \mathbb{R}^m$ *the*

Taylor expansion of $\mathbf{F}$ *is*

$$\mathbf{F}(\mathbf{x}+\mathbf{p}) = \mathbf{F}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\boldsymbol{p} + \mathcal{O}(||\mathbf{p}||^2) \tag{4.2}$$

where $\mathbf{J} \in \mathbb{R}^{n\times m}$ *is the Jacobian matrix*

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_m} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \dots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \tag{4.3}$$

Therefore for each function f_i *we have that*

$$f_i(x_1+p_1, ..., x_m+p_m) = f_1(x_1, ..., x_m) + \sum_{i=1}^{m} \frac{\partial f_i}{\partial x_m} p_m + \mathcal{O}(||\mathbf{p}||^2) \tag{4.4}$$

Let us start for the case when $n = m$. Then starting at $\mathbf{x}^{(0)}$ we want to find $\mathbf{x}^*$ such that $\mathbf{F}(\mathbf{x}^*) = 0$. We can start approximating $\mathbf{F}$ with the Taylor series to find $\mathbf{x}^{(1)}$. Let $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{p}^{(0)}$ then

$$\mathbf{F}(\mathbf{x}^{(1)}) = \mathbf{F}(\mathbf{x}^{(0)} + \mathbf{p}^{(0)}) = \mathbf{F}(\mathbf{x}^{(0)}) + \mathbf{J}(\mathbf{x}^{(0)})\mathbf{p}^{(0)} + \mathcal{O}(||\mathbf{p}||^2)$$

But since we would like $\mathbf{F}(\mathbf{x}^{(1)}) = 0$ then we can choose $\mathbf{p}^0$ such that

$$\underbrace{\mathbf{J}(\mathbf{x}^{(0)})}_{\text{matrix } \in \mathbb{R}^{n\times n}} \quad \mathbf{p}^{(0)} = -\underbrace{\mathbf{F}(\mathbf{x}^{(0)})}_{\text{vector } \in \mathbb{R}^n} \tag{4.5}$$

Notice that we get a linear system of equations that we can solve using any of the methods we discussed in Chapter 3.

Similar to the 1D case, we have quadratic convergence if we are close enough to the solution $\mathbf{x}^*$ i.e. there exists a constant M such that if $||\mathbf{x}^{(k)} - \mathbf{x}^*||$ is sufficiently small then

$$||\mathbf{x}^{(k+1)} - \mathbf{x}^*|| \leq M||\mathbf{x}^{(k)} - \mathbf{x}^*||^2 \tag{4.6}$$

As stopping criteria you can choose the absolute error between iterations, the relative error, the norm of the function value, ...

Example 5. *(See Ascher Section 9.1. Example 9.1) Consider the intersection between the circle centered at the origin with radius of 1* $x_1^2 + x_2^2 = 1$ *and the parabola* $x_2 = (x_1 - 1)^2$*. We want to find the points such that the 2 curves intersect.*

Solution. Notice that we can express the problem as:

$$\text{Find } \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2 \text{ s.t. } \mathbf{F}(\mathbf{x}) = 0 \text{ where } \mathbf{F} = \begin{bmatrix} x_1^2 + x_2^2 - 1 \\ (x_1-1)^2 - x_2 \end{bmatrix} \tag{4.7}$$

Set $\mathbf{x}^{(0)}$; $k = 0$;
while *abs error*$(\mathbf{x}^{(k)}) > tol$ *and* $k < max\_iterations$ **do**
  Solve for $\mathbf{p}^{(k)}$ in $\mathbf{J}\left(\mathbf{x}^{(k)}\right)\mathbf{p}^{(k)} = -\mathbf{F}\left(\mathbf{x}^{(k)}\right)$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{p}^{(k)}$;
  $k = k + 1$;
end
return $\mathbf{x}^{(k)}$

Algorithm 7: Multidimensional Newton Pseudo-code

Therefore the Jacobian at $\mathbf{x}$ is

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} 2x_1 & 2x_2 \\ 2(x_1 - 1) & 1 \end{bmatrix}$$

And therefore starting from $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)})$ we have that

$$\mathbf{x}_{(k+1)} = \mathbf{x}_{(k)} + \mathbf{p}^{(k)} \text{ where } \begin{bmatrix} 2x_1 & 2x_2 \\ 2(x_1 - 1) & 1 \end{bmatrix} \mathbf{p}^{(k)} = -\begin{bmatrix} x_1^2 + x_2^2 - 1 \\ (x_1 - 1)^2 - x_2 \end{bmatrix}$$

□

What are the down sides of Newton's method?

As in the 1D case, we need to be near the root to have convergence. And now we do not have any other method to combine with to ensure convergence. Nevertheless we can use $\mathbf{p}^{(k)}$ as a direction instead of just a solution and find the best α such that

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\mathbf{p}^{(k)}$$

is a good guess. These are called **Line-search methods** and they are a keystone in Numerical Optimization.

On the other hand, notice that at each step we need to compute the Jacobian matrix at $\mathbf{x}^{(k)}$, i.e. $\mathbf{J}\left(\mathbf{x}^{(k)}\right)$, and depending on $\mathbf{F}$ this can be a very expensive process. Therefore, we would like to approximate J with a matrix $B^{(k)}$ that we update at each iteration in order to be near the value of $\mathbf{J}\left(\mathbf{x}^{(k)}\right)$. These are called **Quasi-Newton methods**. One of them is the **Broyden's method**:

$$B^{(k)} = B^{(k-1)} + \frac{\left(\left(\mathbf{F}\left(\mathbf{x}^{(k)}\right) - \mathbf{F}\left(\mathbf{x}^{(k-1)}\right)\right) - B^{(k-1)}\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right)\right)\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right)}{\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right)^T\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right)} \tag{4.8}$$

4.2 Non-Linear Least Squares

In section 3.4, we talked about minimizing a specific type of problems: linear least squares problems, where we had a matrix $A \in \mathbb{R}^{n \times m}$ with $n > m$ and we want to find

$$\underset{x}{\text{minimize}} \frac{1}{2}||Ax - b||_2^2$$

In most of the cases Ax represents a model linear in the parameters to approximate b, i.e. if $x = (\alpha_1, ..., \alpha_m)$ then the model is of the shape

$$f(t) = \alpha_1 f_1(t) + \alpha_2 f_2(t) + ... + \alpha_m f_m(t)$$

Now let us consider the case where the model is not linear in the parameters. For example the logistic model for population growth:

$$N(t) = \frac{N_0 K e^{rt}}{K + N_0(e^{rt} - 1)}$$

That comes from the differential equation

$$\frac{dN}{dt} = rN\left(1 - \frac{N}{K}\right); \qquad N(0) = N_0$$

In the logistic model, the parameters are: N_0 the initial population, r the growth rate, K the carrying capacity. And it is clear that $N(t)$ is not linear in the parameters. Let us define $x = (x_1, x_2, x_3) = (N_0, r, K)$, and let us suppose that we have $1 \leq i \leq n$ measurements of bacteria population (t_i, N_i). Therefore we have the system of equations

$$\begin{aligned}
\frac{x_1 x_3 e^{x_2 t_1}}{x_3 + x_1(e^{x_2 t_1} - 1)} &= N_1 \\
\frac{x_1 x_3 e^{x_2 t_2}}{x_3 + x_1(e^{x_2 t_2} - 1)} &= N_2 \\
... &= ... \\
\underbrace{\frac{x_1 x_3 e^{x_2 t_n}}{x_3 + x_1(e^{x_2 t_n} - 1)}}_{\mathbf{g(x)}} &= \underbrace{N_n}_{\mathbf{b}}
\end{aligned}$$

If we express it as a vector form, we want to find $\mathbf{x} = (x_1, x_2, x_3)$ such that $\mathbf{g(x)} = \mathbf{b}$. Since we only have 3 unknowns and n equations, in the linear case this is not possible to find. Therefore we prefer to formulate the problem as

$$\underset{x \in \mathbb{R}^m}{\text{minimize}} \frac{1}{2}||\mathbf{g(x)} - \mathbf{b}||_2^2$$

Following the same steps we did in the linear case, let us call

$$\begin{aligned}\psi(x) &= \frac{1}{2}||\mathbf{g}(\mathbf{x}) - \mathbf{b}||_2^2 \\ &= \frac{1}{2}(\mathbf{g}(\mathbf{x}) - \mathbf{b})^T(\mathbf{g}(\mathbf{x}) - \mathbf{b}) \\ &= \frac{1}{2}\mathbf{g}(\mathbf{x})^T\mathbf{g}(\mathbf{x}) - \mathbf{g}(\mathbf{x})^T\mathbf{b} + \mathbf{b}^T\mathbf{b}\end{aligned}$$

And in component-wise

$$\psi(x) = \frac{1}{2}\sum_{i=1}^{n}(g_i(x) - b_i)^2$$

In this case to have a minimum, as in 1D we need a critical point, (i.e. $\nabla\psi = 0$) and a second derivative requirement ($\nabla^2\psi$ is positive definite). Therefore computing the gradient component-wise:

$$\begin{aligned}\frac{\partial\psi}{\partial x_j} &= \frac{\partial}{\partial x_j}\frac{1}{2}\sum_{i=1}^{n}(g_i(x) - b_i)^2 \\ &= \sum_{i=1}^{n}(g_i(x) - b_i)\underbrace{\frac{\partial g_i}{\partial x_j}}_{\text{Jacobian } A(x) \text{ of } g(x)}\end{aligned}$$

In matrix form we get the general form of the Normal equations

$$0 = \nabla\psi(x) = A^T(x)(g(x) - b) \tag{4.9}$$

where $A(x)$ is the Jacobian of $g(x)$. Notice that the Hessian of ψ component-wise is given by:

$$\begin{aligned}\frac{\partial^2\psi}{\partial x_l\partial x_j} &= \frac{\partial}{\partial x_l}\sum_{i=1}^{n}(g_i(x) - b_i)\frac{\partial g_i}{\partial x_j} \\ &= \sum_{i=1}^{n}\underbrace{\frac{\partial g_i}{\partial x_j}\frac{\partial g_i}{\partial x_l}}_{A^T(x)A(x)} + \underbrace{(g_i(x) - b_i)\frac{\partial^2 g_i}{\partial x_l\partial x_j}}_{L(x)}\end{aligned}$$

Therefore

$$\nabla^2\psi(x) = A^T(x)A(x) + L(x) \succ 0 \tag{4.10}$$

Notice that to solve (4.9) we need to solve a non-linear system of equations. If we call $\phi(x) = A^T(x)(g(x) - b)$, then we want to find x such that $\phi(x) = 0$. Therefore we can solve it using **Newton**, starting from an initial guess, and finding a new p such that $\phi(x + p) \approx 0$. We use the Taylor series of $\phi(x)$ and we find

$$J_\phi(x_0)p = -\phi(x_0)$$

But recall that $\phi(x) = \nabla\psi(x)$ then we can find p from

$$\nabla^2\psi(x_0) = -\nabla\psi(x_0)$$
$$\left(A^T(x_0)A(x_0) + L(x_0)\right)p = -A^T(x_0)(g(x_0) - b)$$

Notice that this equation is a bit complicated because it involves the function g with its first (in $A(x)$) and second derivatives (in $L(x)$).

What if we drop the second derivatives? (i.e. $L(x)$)

When we neglect the second derivatives, the method is called **Gauss-Newton**, and computes p from

$$A^T(x_0)(A(x_0)p) = -A^T(x_0)(g(x_0) - b)$$

which correspond to the normal equations of the problem

$$\underset{p\in\mathbb{R}^m}{\text{minimize}}\ \frac{1}{2}||A(x_0)p + (g(x_0) - b)||_2^2 = \underset{p\in\mathbb{R}^m}{\text{minimize}}\ \frac{1}{2}||\underbrace{(g(x_0) + A(x_0)p)}_{\text{Linear approx. of } g(x)} - b||_2^2 \quad (4.11)$$

Notice that Gauss-Newton have the following properties:

- The approximation for the Hessian (the second derivative) is always positive definite (even when $A^TA + L$ is not)
- It does not compute the second derivative of g, therefore it is cheaper per iteration
- Since there is always a difference between Newton and Gauss-Newton (i.e L), then Gauss-Newton does not have quadratic convergence.

4.3 Unconstrained Optimization

Up to now, we have solved linear and non-linear systems going from $\mathbb{R}^m$ to $\mathbb{R}^n$. If $m = n$ we used Newton's method to find the solution. If $m > n$ we used a least squares approach. Least squares is a particular type of minimization problem. But it is not the only one. In general given a function $\phi : \mathbb{R}^n \to \mathbb{R}$, we are interested

Set $\mathbf{x}^{(0)}$; $k = 0$;
while *abs error*$(\mathbf{x}^{(k)}) > tol$ *and* $k < max\_iterations$ **do**
    Solve the linear least squares problem

$$\underset{p \in \mathbb{R}^m}{\text{minimize}} \frac{1}{2}||A(\mathbf{x}^{(k)})p + (g(\mathbf{x}^{(k)}) - b)||_2^2$$

    Set the minimizer $p_k = p$
    Update $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{p}^{(k)}$;
    $k = k + 1$;
end
return $\mathbf{x}^{(k)}$

Algorithm 8: Gauss-Newton for Non-Linear Least Squares Pseudo-code

in finding x^* such that there exists $\delta > 0$ that for any y with $||y - x|| < \delta$ then $\phi(x^*) \leq \phi(y)$

The simplest way to find a minimum is by simple inspection of the function. We can guess x^* choosing randomly a set of points $\{x^{(1)}, x^{(2)}, ...x^{(k)}\}$ and taking x^* as the minimum evaluation, i.e. $x^* = \{x^(i)|\phi(x^{(i)}) = \min_j \phi(x^{(j)})\}$. But as you can imagine this is a very inefficient method.

Another way is to minimize ϕ in each direction. Start with a guess $x^{(1)} = (x_1^{(1)}, x_2^{(1)}, ..., x_n^{(1)})$ and fix all the components except the first one, i.e. consider

$$f(\alpha) = \phi(\alpha, x_2^{(1)}, ..., x_n^{(1)})$$

This is a univariate function in α and we can find its minimum using any of the methods described in chapter 2. Once we find α^* we can update our guess as $x^{(2)} = (\alpha^*, x_2^{(1)}, ..., x_n^{(1)})$ and now we can optimize in the second component x_2 and so on. Once we reach x_n, we start again in x_1 and we repeat the process until we cannot minimize any more. This method is called **Coordinate descent**.

Do you think this is the best way of minimizing a function?

Consider the two functions in Figure 4.1. They are exactly the same function and the same initial guess. The only difference is that we rotated our coordinate system. For the fist one we get the minimum in one iteration whereas in the second one we get in two. But probably if we selected in a better way our search direction, we could find the minimum in both cases in the same number of iterations.

To choose this direction, let us consider the Taylor series for a function $\phi : \mathbb{R}^n \to \mathbb{R}$

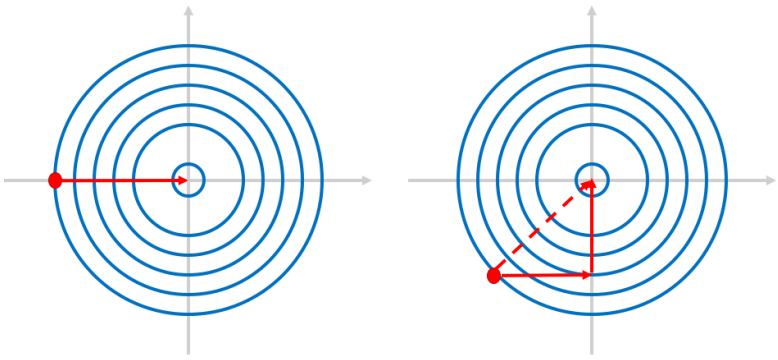

Figure 4.1: Minimize $\phi(x, y) = x^2 + y^2$ choosing one direction at a time for different initial conditions. How can we just do everything in one step?

Definition 17. *Let* $\mathbf{x} = [x_1, ..., x_n]^T \in \mathbb{R}^n$ *and* $\phi : \mathbb{R}^n \to \mathbb{R}$ *with bounded derivatives up to order at least 2, then for any direction* $\mathbf{p} = [p_1, ..., p_n]^T \in \mathbb{R}^n$ *the Taylor expansion of* ϕ *is*

$$\phi(\mathbf{x} + \mathbf{p}) = \phi(\mathbf{x}) + \nabla\phi(\mathbf{x})^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\nabla^2\phi(\mathbf{x})\mathbf{p} + \mathcal{O}(||\mathbf{p}||^3) \tag{4.12}$$

where $\nabla\phi(\mathbf{x})^T$ *is the gradient of* ϕ

$$\nabla\phi(\mathbf{x})^T = \left[\frac{\partial\phi}{\partial x_1}, ..., \frac{\partial\phi}{\partial x_n}\right] \tag{4.13}$$

and $\nabla^2\phi(\mathbf{x})$ *is the Hessian of* ϕ

$$\nabla^2\phi(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2\phi}{\partial x_1^2} & \frac{\partial^2\phi}{\partial x_1\partial x_2} & \cdots & \frac{\partial^2\phi}{\partial x_1\partial x_n} \\ \frac{\partial^2\phi}{\partial x_1\partial x_2} & \frac{\partial^2\phi}{\partial x_2^2} & \cdots & \frac{\partial^2\phi}{\partial x_2\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2\phi}{\partial x_1\partial x_n} & \frac{\partial^2\phi}{\partial x_2\partial x_n} & \cdots & \frac{\partial^2\phi}{\partial x_n^2} \end{bmatrix} \tag{4.14}$$

4.3.1 Steepest descent

If we just consider the first order Taylor expansion, we notice that for sufficiently small p, $\phi(x + p) < \phi(x)$ if

$$\nabla\phi^T p < 0 \tag{4.15}$$

If a direction p satisfies (4.15) then it is called a **descent direction** at x. Let θ is the angle between the two vectors then $\nabla\phi^T p = ||\nabla\phi||\ ||p|| \cos\theta$. Therefore among

all directions p such that $||p|| = 1$ the biggest descent is

$$p = -\nabla\phi(x) \tag{4.16}$$

normalized. When we choose the direction p that satisfies (4.16) then the method is called **steepest descent** or **gradient descent** method.

Now if we try in our functions of Figure 4.1, we take only one iteration if we use steepest descent direction. But now let us consider Figure 4.2. We have the same function and we are choosing two different initial conditions, both at the same level curve of ϕ. We optimize choosing the steepest descent direction. In the first case we get the minimum in only one iteration, whereas in the second case we start to zig-zag towards the minimum. Why is this happening?

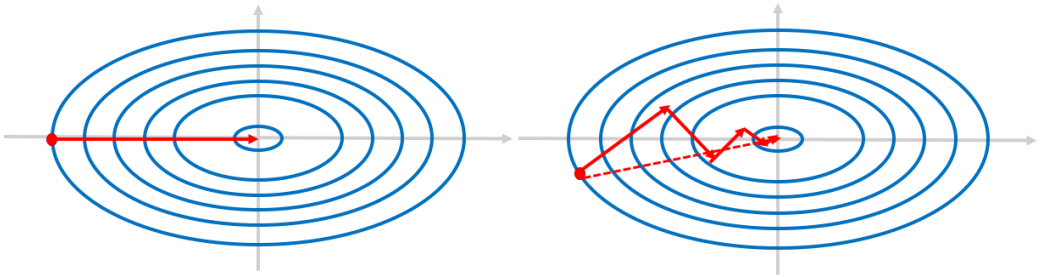

Figure 4.2: Minimize $\phi(x, y) = x^2 + 4y^2$ choosing one direction at a time for different initial conditions. How can we just do everything in one step?

4.3.2 Newton's Method

Notice that using only a first order approximation of ϕ is replacing our function by a plane. And a plane is not bounded. Therefore it does not seem a clever choice to approximate a bounded function with an unbounded function to find the minimum. Therefore let us go one step further and take the second order Taylor series approximation. Let

$$\phi(\mathbf{x} + \mathbf{p}) \approx \psi(\mathbf{p}) = \phi(\mathbf{x}) + \nabla\phi(\mathbf{x})^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\nabla^2\phi(\mathbf{x})\mathbf{p} \tag{4.17}$$

Let us recall which are the necessary and sufficient conditions to be a local minimum.

Definition 18. *We say that $x^* \in \mathbb{R}^\ltimes$ is a **critical point** of ϕ if*

$$\nabla\phi(x) = 0 \tag{4.18}$$

Definition 19. *Given a critical point x^*, then*

- *If it is a local minimizer then $\nabla^2\phi(x^*) \succeq 0$ (necessary)*
- *If $\nabla^2\phi(x^*) \succ 0$ then it is a local minimizer (sufficient)*

If x is sufficiently near x^* then ψ is a quadratic function in p, such that its Hessian is positive definite. Since $\nabla\psi(p) = \nabla\phi(x) + \nabla^2\phi(x)p$, then ϕ has a unique minimizer p^* such that

$$\nabla^2\phi(x)p^* = -\nabla\phi(x) \tag{4.19}$$

Therefore we just need to find p^* solving a linear system of equations (4.19). With this direction, we only take one iteration in both function in Figure (4.2).

Why is it called Newton's method?

Notice that if we consider the Taylor series of the gradient of ϕ, let $G = \nabla\phi$ using (4.2) we have that for $x, p \in \mathbb{R}^n$ if ϕ is smooth enough then:

$$G(x+p) = G(x) + J_G(X)p + \mathcal{O}(p) \tag{4.20}$$

where J_G is the Jacobian of $G = \nabla\phi = \left[\frac{\partial\phi}{\partial x_1}, ..., \frac{\partial\phi}{\partial x_n}\right]^T$, i.e

$$J_G = \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial g_n}{\partial x_1} & \cdots & \frac{\partial g_n}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1}\left(\frac{\partial\phi}{\partial x_1}\right) & \cdots & \frac{\partial}{\partial x_n}\left(\frac{\partial\phi}{\partial x_1}\right) \\ \cdots & \cdots & \cdots \\ \frac{\partial}{\partial x_1}\left(\frac{\partial\phi}{\partial x_n}\right) & \cdots & \frac{\partial g_n}{\partial x_n}\left(\frac{\partial\phi}{\partial x_n}\right) \end{bmatrix} = \begin{bmatrix} \frac{\partial^2\phi}{\partial x_1^2} & \cdots & \frac{\partial^2\phi}{\partial x_n\partial x_1} \\ \cdots & \cdots & \cdots \\ \frac{\partial^2\phi}{\partial x_1\partial x_n} & \cdots & \frac{\partial^2\phi}{\partial x_n^2} \end{bmatrix}$$

Therefore $J_G = \nabla^2\phi$. If we apply Newton's method of section 4.1 to solve $G(x) = 0$, then in each iteration with a guess $x^{(k)}$ we find $p^{(k)}$ such that:

$$J_G(x^{(k)})p^{(k)} = -G(x^{(k)})$$
$$\nabla^2\phi(x^{(k)})p^{(k)} = -\nabla\phi(x^{(k)})$$

which is indeed the same equation we got from approximating our function with a quadratic model from Taylor's series.

Nevertheless, similar to the problem in 1D (section 2.3) it is not enough to solve (4.19) in order to find a minimum. We also need that $\nabla\phi(x^{(k)})$ is positive definite. If we do not enforce this condition, the direction that we get might not be a descent direction, or even worse, we could not even solve (4.19) if $\nabla\phi(x^{(k)})$ is singular or ill-conditioned.

Notice that if we assume that $\nabla\phi(x^{(k)})$ is positive definite, we can solve (4.19) using a Cholesky Factorization. Therefore one of the methods to solve (4.19) when we do not know if $\nabla\phi(x^{(k)})$ is to modify its diagonal until a Cholesky factorization exists. This method is called **Modified Cholesky** and it is one of the most used methods.

4.3.3 Line Search methods

Up to now we have talked about three different methods to minimize $\phi : \mathbb{R}^n \to \mathbb{R}$: Coordinate descent, Steepest descent and Newton's method. In all of them we can split each iteration in 2 steps:

1. Find or select a search direction $p^{(k)}$, that it is a descent direction (4.15):
   - Coordinate decent:
   $$p^{(k)} = -\left(\frac{\partial\phi}{\partial x_{i_k}}(x^{(k)})\right) e_{i_k}$$
   - Steepest decent:
   $$p^{(k)} = -\nabla\phi(x^{(k)})$$
   - Newton's method:
   $$\nabla^2\phi(x^{(k)})p^{(k)} = -\nabla\phi(x^{(k)})$$
2. Minimize your function in the direction $p^{(k)}$. Consider $f(\alpha) = \phi(x^{(k)} + \alpha p^{(k)})$. This is the restriction of your function to the direction $p^{(k)}$, and $f : \mathbb{R} \to \mathbb{R}$. Therefore we can apply any of the techniques of chapter 2 to find $\alpha^{(k)}$ such that minimizes $f(\alpha)$. Then we update the guess as
   $$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$$

With this method we construct a sequence of points $\{x^{(1)}, x^{(2)}, x^{(3)}, ...\}$ such that $\phi(x^{(1)}) \geq \phi(x^{(2)}) \geq \phi(x^{(3)}) \geq ...$ and it converges to a local minimum of ϕ.

Any minimization method based on this two steps is called a **Line Search** method. And as you can expect, there is always a trade off between finding a descent direction and optimizing the function in that direction.

For example, steepest descent has linear rate of convergence if we solve exactly the 1D minimization problem at each step, whereas Newton's method has quadratic rate of convergence if we always take $\alpha^{(k)} = 1$ and we are sufficiently close to the real minimum. Nevertheless, steepest descent only requires to know the first derivative of the function and Newton's method requires first and second derivative for every direction.

Example 6. *Consider the function* $\phi(x_1, x_2) = (x_1 + x_2)^2 + 4(x_1 - x_2)^2$. *How is the first iteration with each of the methods if* $x^{(0)} = (-1, -2)$*?*

Solution. Let us compute the gradient and the Hessian matrix of ϕ:

$$\nabla\phi(x_1, x_2) = [2(x_1 + x_2) + 8(x_1 - x_2), 2(x_1 + x_2) - 8(x_1 - x_2)]^T$$
$$\nabla^2\phi(x_1.x_2) = \begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix}$$

- **Coordinate descent** We have that:

$$p^{(0)} = -\left(\frac{\partial\phi}{\partial x_1}(-1, -2)\right) e_1 = -(2, 0)$$

Therefore we need to optimize

$$f(\alpha) = \phi((-1, -2) - \alpha(2, 0)) = ((-1 - 2\alpha) - 2)^2 + 4((-1 - 2\alpha) + 2)^2$$

Taking first derivative

$$\begin{aligned} 0 = \frac{\partial f}{\partial\alpha} &= 2((-1 - 2\alpha) - 2)(-2) + 8((-1 - 2\alpha) + 2)(-2) \\ 0 &= -3 - 2\alpha + 4(1 - 2\alpha) \\ \alpha &= \frac{1}{10} \end{aligned}$$

Therefore the new guess is $x^{(1)} = x^{(0)} - (1/10)(2, 0) = (-6/5, -2)$

- **Steepest descent** We have that:

$$p^{(0)} = -\nabla\phi(-1, -2) = (-2, 14)$$

Therefore we need to optimize

$$\begin{aligned} f(\alpha) &= \phi((-1, -2) + \alpha(-2, 14)) \\ &= ((-1 - 2\alpha) + (-2 + 14\alpha))^2 + 4((-1 - 2\alpha) - (-2 + 14\alpha))^2 \end{aligned}$$

Taking first derivative

$$\begin{aligned} 0 &= \frac{\partial f}{\partial\alpha} \\ 0 &= 2((-1 - 2\alpha) + (-2 + 14\alpha))(12) + 8((-1 - 2\alpha) - (-2 + 14\alpha))(-16) \\ 0 &= (-3 + 12\alpha)(3) + 4(1 - 16\alpha)(-4) \\ \alpha &= \frac{25}{292} = 0.0856 \end{aligned}$$

Therefore the new guess is $x^{(1)} = x^{(0)} + \frac{25}{292}(-2, 14) = (-0.8287, -0.8014)$

- **Newton's method** We have that:

$$\begin{bmatrix} 10 & -6 \\ -6 & 10 \end{bmatrix} p^{(0)} = -\nabla\phi(-1,-2) = \begin{bmatrix} -2 \\ 14 \end{bmatrix}$$

Therefore $p^{(0)} = [1,2]^T$ and we need to optimize

$$\begin{aligned} f(\alpha) &= \phi((-1,-2) + \alpha(1,2)) \\ &= ((-1+\alpha) + (-2+2\alpha))^2 + 4((-1+\alpha) - (-2+2\alpha))^2 \end{aligned}$$

Taking first derivative

$$\begin{aligned} 0 &= \frac{\partial f}{\partial \alpha} \\ 0 &= 2((-1+\alpha) + (-2+2\alpha))(3) + 8((-1+\alpha) - (-2+2\alpha))(-1) \\ 0 &= -9(1-\alpha) - 4(1-\alpha) \\ \alpha &= 1 \end{aligned}$$

Therefore the new guess is $x^{(1)} = x^{(0)} + (1,2) = (0,0)$

□

Some comments about finding the search direction

Since computing the second derivatives of ϕ can be really expensive or complicated, one of the approaches is to approximate $\nabla^2\phi(x^{(k)})p^{(k)}$ at each iteration. There are two ways to approximate it:

- Approximate term by term with finite differences. For example, if we know the gradient then we can approximate:

$$\frac{\partial^2\phi}{\partial x_k \partial x_l}(x^{(k)}) \approx \frac{\frac{\partial\phi}{\partial x_l}(x^{(k)} + h\ e_k) - \frac{\partial\phi}{\partial x_l}(x^{(k)})}{h}$$

- Approximate $\nabla^2\phi$ as an operator: Find a matrix with a similar behavior. For example, if we approximate $\nabla^2\phi = I$ the identity, then we get the steepest descent direction. But of course doing this we loose all the curvature information of ϕ.

  Another way to approximate $\nabla^2\phi$ is to consider again the Taylor series of $\nabla\phi(x^{(k)}$ around $x^{(k+1)}$, then we get that

$$\nabla\phi(x^{(k)}) \approx \nabla\phi(x^{(k+1)}) - \nabla^2\phi(x^{(k+1)})(x^{(k+1)} - x^{(k)}) \tag{4.21}$$

Therefore if we want B_{k+1} being a nice approximation of $\nabla^2\phi(x^{(k+1)})$ then it should satisfy (4.21) i.e.

$$B_{k+1}s_k = y_k \tag{4.22}$$

where $s_k = x^{(k+1)} - x^{(k)}$ and $y_k = \nabla\phi(x^{(k+1)}) - \nabla\phi(x^{(k)})$. The equation (4.22) is called the **secant condition**, and the methods that use this condition to approximate the Hessian are called **Quasi-Newton** methods.

In all of them we find the search direction solving

$$B_k p^{(k)} = -\nabla\phi(x^{(k)}) \tag{4.23}$$

And some of the updates are:

- **BFGS method** It is the most popular of the Quasi-Newton methods because of its fast convergence in practice. And it is the heart of some optimization packages:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k} \tag{4.24}$$

- **DFP method** It applies the same update as the BFGS to the inverse of the Hessian instead of the Hessian itself. This means that if we call $B_k^{-1} = H_k$ then DFP update is

$$H_{k+1} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{s_k^T y_k} \tag{4.25}$$

- **SR1 method** It is a symmetric rank-1 of the matrix. It is easier to compute but it can be non-positive definite

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T y_k} \tag{4.26}$$

Some comments about minimizing $\phi(x + \alpha p)$

How much effort should we invest in finding the minimum of $f(\alpha) = \phi(x^{(k)} + \alpha p^{(k)})$ if we are far from x^*? Do we have an initial guess for α? Notice that if we are considerably far from the minimum, it is not worthy to put a lot of time in finding the minimum in certain direction if that does not bring me sufficiently closer to x^*. This means that most of the times, the methods used in finding the minimum are not the most sophisticated. For example we can use **backtracking**, i.e. start with $\alpha_{\max}$ and iterate using $\alpha_i = (1/2)^i \alpha_{max}$ until we find a reasonable minimum. Other

methods use **bisection** or **parabolic interpolation** with loose tolerances.

Also notice that for steepest descent and coordinate decent we do not have a clear idea of the size of α because it comes from a linear model where the optimal α is ∞. In contrast, Newton's method comes from a quadratic model, where the optimal α is 1. Therefore we have an initial guess for Newton's method, and also Quasi-Newton methods, but not for steepest descent or coordinate descent. In more general cases, when the problem also has constraints (i.e. another set of conditions that the solution should satisfy), then $\alpha_{\max}$ is designed in order to satisfy all the constraints.

Another important concept in choosing the α is that it should be large enough to ensure convergence to the real minimum (and not to another point) and small enough to avoid divergence or zig-zagging. For this, we use a set of conditions that a new guess should satisfy. Some of them are:

- **Armijo Condition** To avoid divergence: For $c \in (0, 1)$

$$\phi(x^{(k)} + \alpha p^{(k)}) \leq \phi(x^{(k)}) + c\ \alpha\ \nabla\phi(x^{(k)})^T p^{(k)}$$

- **Goldstein Condition** To avoid small α and also divergence: For $c \in (0, 1/2)$

$$\phi(x^{(k)})+(1-c)\ \alpha\ \nabla\phi(x^{(k)})^T p^{(k)} \leq \phi(x^{(k)}+\alpha p^{(k)}) \leq \phi(x^{(k)})+c\ \alpha\ \nabla\phi(x^{(k)})^T p^{(k)}$$

For Newton's method, we can take $c = 1e-4$, but for steepest descent it depends on the problem. Sometimes it should be around $c = 0.1$.

Just an additional comment: Coordinate decent is a really basic algorithm, and that is the reason that it is not a main topic in many optimization books. Nevertheless, its performance is considerably good, especially nowadays in applications such as machine learning and statistics. If you want to learn more about this topic, check Wright's paper in Arxiv. You find its reference at the beginning of the notes.

Chapter 5

Interpolation

5.1 Polynomial Interpolation

Interpolation is a special way to approximate functions. This can be used to find a function that passes through all given data (data fitting) or to approximate a complicated function with a simple one. Once we have this model, we can use it to predict new points or to approximate derivatives and integrals.

What are we looking for in interpolation?

Example 7. *Find the line between 2 data points. This is called* ***linear interpolation***

Solution. We know that the slope of the straight line should satisfy

$$\frac{f(x) - f(x_0)}{x - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \tag{5.1}$$

If we have the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$ then we have that

$$\frac{f(x) - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0}$$

$$f(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) \tag{5.2}$$

□

In general, when we talked about interpolation, we are looking for a function of the form

$$f(x) = c_0\phi_0(x) + c_1\phi_1(x) + \cdots + c_n\phi_n(x) \tag{5.3}$$

such that for our given set of data $(x_0, y_0), ..., (x_n, y_n)$ satisfies that

$$y_i = f(x_i) = c_0\phi_0(x_i) + c_1\phi_1(x_i) + \cdots + c_n\phi_n(x_i)$$

If we fixed the **basis functions** ϕ_i, in order to know the **coefficients** c_i we need to solve a linear system of n equations and n unknowns:

$$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \dots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \dots & \phi_n(x_1) \\ \dots & \dots & \dots & \dots \\ \phi_0(x_n) & \phi_1(x_n) & \dots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \dots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_n \end{bmatrix} \tag{5.4}$$

When we talk about polynomial interpolation, we choose the basis functions to be polynomials. But since the polynomials are a Vector space, we can take different set of basis functions to represent the same polynomial interpolation. For example:

- **Monomial Interpolation** if we choose $\phi_i(x) = x^i$
- **Lagrange Interpolation** if we choose $\phi_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$
- **Newton Interpolation** if we choose $\phi_i(x) = \prod_{j<i}(x - x_j)$

Of course, with each choose of basis, the coefficients of the polynomial will change, but the result will be the same.

5.1.1 Monomial Interpolation

The simplest interpolation to imagine is the Monomial interpolation. In this case we are looking for the coefficients $\{c_0, \dots, c_n\}$

$$f(x) = c_0 + c_1 x + c_2 x^2 + \dots + c_n x^n \tag{5.5}$$

to pass exactly through $(x_0, y_0), ..., (x_n, y_n)$. Therefore we should solve the linear system

$$\underbrace{\begin{bmatrix} 1 & x_0 & \dots & x_0^n \\ 1 & x_1 & \dots & x_1^n \\ \dots & \dots & \dots & \dots \\ 1 & x_n & \dots & x_n^n \end{bmatrix}}_{Vandermonde\ matrix} \begin{bmatrix} c_0 \\ c_1 \\ \dots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_n \end{bmatrix} \tag{5.6}$$

We know that the determinant of the Vandermonde matrix X is

$$det(X) = \prod_{i=0}^{n-1} \prod_{j=i+1}^{n} (x_j - x_i)$$

And therefore if all the x entries are different, then the determinant is different from zero and X is invertible. Then there is a unique solution for the coefficients. This means there is a unique polynomial of order n that passes exactly through $n + 1$ points with different abscissas. Therefore it does not matter which basis we are working with, we will get the same polynomial. The only difference is that the representation in some basis is easier to understand than in others.

Which are the drawbacks of the monomial representation

- X is often ill-conditioned, and solving the system with Gauss elimination takes $2/3\ n^3$
- The coefficients do not give a lot of insight of the behavior of the interpolant. For example notice that it is easier to imagine

$$f(x) = (x-3)^3$$

than

$$f(x) = x^3 - 9x^2 + 27x - 27$$

5.1.2 Lagrange interpolation

What if we want the coefficients to be exactly the points? i.e. $c_i = y_i$

Recall that we are looking for an interpolation that given $(x_0, y_0), ..., (x_n, y_n)$, satisfies

$$y_i = f(x_i) = c_0\phi_0(x_i) + c_1\phi_1(x_i) + \cdots + c_n\phi_n(x_i)$$

For example if we consider the basis functions $\phi_i(x) = L_i(x)$ such that

$$L_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{5.7}$$

Then we have that

$$\begin{aligned} y_i &= c_0L_0(x_i) + c_1L_1(x_i) + \cdots + c_{i-1}L_{i-1}(x_{i-1}) + c_iL_i(x_i) + c_{i+1}L_{i+1}(x_{i+1}) + \cdots + c_nL_n(x_i) \\ &= c_0 * 0 + c_1 * 0 + \cdots + c_{i-1} * 0 + c_i * 1 + c_{i+1} * 0 + \cdots + c_n * 0 \\ &= c_i \end{aligned}$$

But now, how do we construct $L_i(x)$?

Let us think the case when $n = 2$, then we have $(x_0, y_0), (x_1, y_1), (x_2, y_2)$. To construct $L_0(x)$ we need:

- $L_0(x_1) = 0$ and $L_0(x_2) = 0$
- $L_0(x_0) = 1$

Then notice that we can choose

$$L_0(x) = a(x - x_1)(x - x_2)$$

That satisfies the first condition. For the second condition, we need to find the value of a such that $L_0(x_0) = 1$. Therefore

$$1 = L_0(x_0) = a(x_0 - x_1)(x_0 - x_2) \Longrightarrow a = \frac{1}{(x_0 - x_1)(x_0 - x_2)}$$

And similarly we can solve for $L_1(x)$ and $L_2(x)$ to get:

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} \quad L_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} \quad L_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Now for a general n we can define the **Lagrange polynomials** $L_i(x)$ as

$$L_i(x) = \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)} \tag{5.8}$$

And therefore the interporlant is given by

$$f(x) = \sum_{i=0}^{n} y_i L_i(x) = \sum_{i=0}^{n} y_i \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)} \tag{5.9}$$

There is another expression for (5.9) that makes it easier to compute using the **barycentric weights** w_i. Define

$$\rho_i = \prod_{j \neq i} (x_i - x_j) \qquad\qquad w_i = \frac{1}{\rho_i} \tag{5.10}$$

And also define the function

$$\psi(x) = (x - x_0)(x - x_1) \dots (x - x_n) = \prod_{i=0}^{n} (x - x_i)$$

Then for any set of points $\{(x_i, y_i)\}_{i=0}^{n}$

$$f(x) = \sum_{i=0}^{n} y_i w_i \frac{\psi(x)}{x - x_i}$$

In particular, if we interpolate the constant function one, then all $y_i = 1$ and

$$1 = \sum_{i=0}^{n} (1) w_i \frac{\psi(x)}{x - x_i} = \psi(x) \sum_{i=0}^{n} \frac{w_i}{x - x_i} \Longrightarrow \psi(x) = \frac{1}{\sum_{i=0}^{n} \frac{w_i}{x - x_i}}$$

And therefore we get the **barycentric interpolation** formula. For $x \neq x_i$:

$$f(x) = \sum_{i=0}^{n} y_i L_i(x) = \frac{\sum_{i=0}^{n} y_i \frac{w_i}{x - x_i}}{\sum_{i=0}^{n} \frac{w_i}{x - x_i}} \tag{5.11}$$

5.1.3 Newton Interpolation

What if we want to an adaptive algorithm where we can add point one by one?

Sometimes, all the interpolation points are not available when we start to interpolate them. Imagine an experiment where we collect few points, we see how is the result, and we measure more points to add to the interpolation. When we compute the coefficients for the new polynomial, if we use either Lagrangian or monomial basis, we need to recompute all the values, we cannot reuse the information we had before.

Therefore let us consider the following basis functions:

$$\phi_i(x) = \prod_{j=0}^{i-1}(x - x_j) \tag{5.12}$$

This means

$$\phi_0 = 1, \quad \phi_1 = (x - x_0), \quad \phi_2 = (x - x_0)(x - x_1), \quad \phi_3 = (x - x_0)(x - x_1)(x - x_2), \quad \ldots$$

Therefore we will look for the coefficients c_i such that

$$y_i = p_n(x_i) = c_0\phi_0(x_i) + c_1\phi_1(x_i) + \cdots + c_n\phi_n(x_i)$$

Therefore notice that if p_n interpolates $\{(x_0, y_0), \ldots, (x_n, y_n)\}$, then p_{n+1} that interpolates all the previous n points plus an extra one (x_{n+1}, y_{n+1}) is given by:

$$p_{n+1}(x) = p_n(x) + c_{n+1}(x - x_0)(x - x_1)...(x - x_n)$$

How can we compute the coefficients? ***Divided Differences***

Since the definition of the polynomial is recursive, we can expect that the definition of the coefficients is recursive. Notice that

1. $f(x_0) = c_0$
2. $f(x_1) = c_0 + c_1(x_1 - x_0)$. Therefore:
$$c_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$
3. $f(x_2) = c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1)$. Therefore:
$$f(x_2) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1)$$
$$f(x_2) - f(x_1) = f(x_0) - f(x_1) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1)$$
$$f(x_2) - f(x_1) = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1) + c_2(x_2 - x_0)(x_2 - x_1)$$
$$\frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2 - x_0} = c_2$$

4. ...

So in general, we can define the ith coefficient as the ith *divided difference* where:

$$\begin{aligned} c_0 = f[x_0] &= f(x_0) \\ c_1 = f[x_0, x_1] &= \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f[x_1] - f[x_0]}{x_1 - x_0} \\ c_2 = f[x_0, x_1, x_2] &= \frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2 - x_0} = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} \\ \ldots &= \ldots \\ c_i = f[x_0, x_1, \ldots, x_i] &= \frac{f[x_1, \ldots, x_i] - f[x_0, \ldots, x_{i-1}]}{x_i - x_0} \end{aligned} \tag{5.13}$$

Therefore we can construct a table $i, x_i, f[x_i], f[x_i, x_{i+1}], \ldots$ and so on, and each time we add a new value to interpolate, we just add a new extra row to the table.

How can we estimate the error of the interpolation?

Notice that the divided differences look like approximations of higher derivatives, indeed we have the following theorem

Theorem 8. *Let f a function with k bounded derivatives in the interval $[a, b]$ and let $z_0, ..., z_k$ $k+1$ distinct points, then there is $\xi \in [a, b]$ such that*

$$f[x_0, ..., x_k] = \frac{f^{(k)}(\xi)}{k!} \tag{5.14}$$

So now let us consider the points $\{(x_0, f(x_0)), ..., (x_n, f(x_n))\}$ and $p_n(x)$ the interpolant. If we want to compute the error at x:

$$e_n(x) = f(x) - p_n(x) \tag{5.15}$$

We can construct an interpolant for the points $\{(x_0, f(x_0)), ..., (x_n, f(x_n))\} \cup \{(x, f(x)))\}$ given by

$$p_{n+1}(y) = p_n(y) + f[x_0, ..., x_n, x] \prod_{i=0}^{n} (y - x_i)$$

Therefore when we evaluate $y = x$ we get that

$$e_n(x) = f(x) - p_n(x) = f[x_0, ..., x_n, x] \prod_{i=0}^{n} (x - x_i) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \tag{5.16}$$

Therefore we can bound the maximum error as

$$e_n(x) \leq \max |f(x) - p_n(x)| = \max_{a \leq xi \leq b} \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} \right| \max_{a \leq s \leq b} \left| \prod_{i=0}^{n} (s - x_i) \right| \tag{5.17}$$

Name	Basis	Construction cost	Properties
Monomial	$\{x^i\}_{i=0}^n$	$2/3n^3$	Intuitive, Prove existence
Lagrange	$\left\{\prod_{j\neq i} \frac{x-x_j}{x_i-x_j}\right\}_{i=0}^n$	n^2	Stable, $c_i = y_i$
Newton	$\left\{\prod_{j=0}^{i-1} x - x_j\right\}_{i=0}^n$	$3/2n^2$	Adaptive, compute the error

Table 5.1: Properties of different basis polynomials

5.2 Piecewise Polynomial Interpolation

5.2.1 Runge's Phenomenon

It looks that everything works well with Polynomial Interpolation, any set of points has a unique polynomial, is that enough?

Let us consider the function $f(x) = \frac{1}{1+x^2}$ for $x \in [-5, 5]$, and suppose that we sample the data $(x_0, f(x_0)), (x_1, f(x_1)), ..., (x_n, f(x_n))$ where all the x-coordinates are equally spaced, i.e. $x_i = -5 + i * 10/n$. Let us see how is the behavior of the interpolant in Figure 5.1.

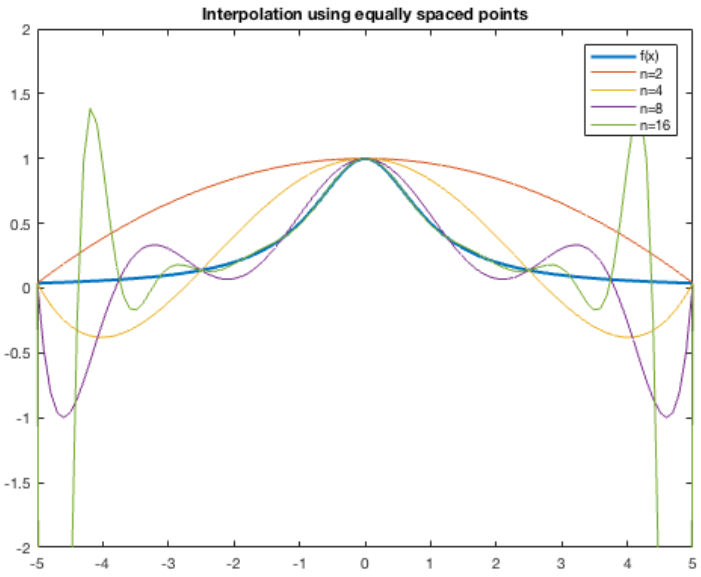

Figure 5.1: Equally spaced interpolation of Runge phenomenon

As you can see, as we increase n, the interpolant has more and more oscillations at the tails of the interval, and the error is concentrated there. Adding more points only makes the situation worse.

Now let us use a different set of x-coordinates called the **Chebyshev points**. This coordinates correspond to the roots of the Chebyshev polynomials in the interval $[-1, 1]$ and they are given by:

$$x_k = \cos\left(\frac{2k-1}{2n}\pi\right) \tag{5.18}$$

In order to scale them to the interval $[-5, 5]$ we multiply them by 5, and in this case we also include the beginning and the last point of the interval.

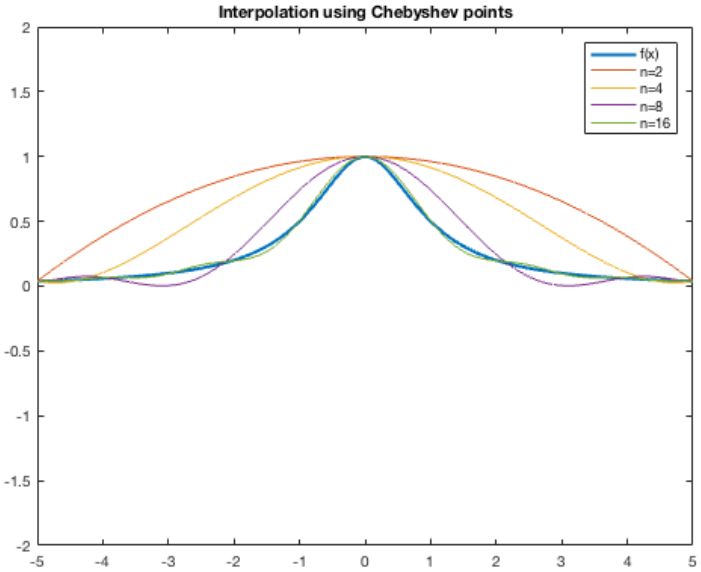

Figure 5.2: Interpolation of Runge phenomenon using Chebyshev points

Now when we plot the interpolant of this new set of $\{(x_k, f(x_k))\}$, we do not have as many oscillations as before and the error keeps bounded.

Why does this happen?

This is called the **Runge's phenomenon** and illustrates why we usually do not consider polynomial interpolations of high degree unless the set of x coordinates is really special. If we recall the equation (5.17), the error depends on the maximum of the k-th derivative of the function, and also, on the distribution of the points, i.e.

$$\max_{a \leq s \leq b} \left| \prod_{i=0}^{n} (s - x_i) \right| \tag{5.19}$$

In particular, the Chebyshev points properly scaled minimizes (5.19), and that explains the different behavior between Figures 5.1 and 5.2.

Therefore, in general, if we can choose the x-coordinates to interpolate, we can special set of points, for example Chebyshev points, and we can have high order polynomial interpolation. But in general, we do not have the freedom to choose the abscissas. Therefore we need another way to approximate our functions without having huge oscillations.

5.2.2 Piecewise linear interpolation

If I have a hundred pairs of data, how can I interpolate them?

In the previous section, we saw that interpolating a small set of data ends up in a small order polynomial, and everything was well behaved. Therefore the idea now is to split the data into small chunks, and interpolate each piece separately.

The simplest interpolation we can propose is **piecewise constant interpolation** where given the points $\{(x_i, f(x_i))\}_{i=0}^n$ we can define our interpolant $\phi(x)$ as

$$\phi(x) = f(x_i) \quad \text{if} \quad \frac{x_i + x_{i-1}}{2} \leq x \leq \frac{x_i + x_{i+1}}{2} \tag{5.20}$$

The problem with this interpolation is that is not even continuous, because it has jumps at $(x_i + x_{i+1})/2$ for all $i = 0, 1, ..., n-1$.

The following most simple interpolation is **piecewise linear interpolation**, that consist in joining each 2 adjacent points with a straight line. Given the points $\{(x_i, f(x_i))\}_{i=0}^n$ we can define our interpolant $\phi(x)$ as

$$\phi(x) = f(x_i)\frac{x - x_{i-1}}{x_i - x_{i-1}} + f(x_{i-1})\left(1 - \frac{x - x_{i-1}}{x_i - x_{i-1}}\right) \quad \text{if} \quad x_{i-1} \leq x \leq x_i \tag{5.21}$$

This representation is really ease to compute, and looks like a pretty accurate approximation. And sometimes that is all what we need. But other times we are interested in a smoother approximation. That means, we should increase the order of the polynomial. And here is where a lot of options are open: there is not a single path to take.

In one hand, notice that in the constant interpolation we took 1 point, in the linear interpolation we took 2 points, and therefore we can increase the order of the polynomial taking more points to define 1 curve (bigger chunks). For example if we want to construct **piecewise quadratic interpolation**, we can take $x_{2i-1}, x_{2i}, x_{2i+1}$, and construct a parabola between them. The only problem with

this approach is what happens in the break points. In the case of the parabola, it is C^2 in (x_{2i-1}, x_{2_i+1}) but the derivative is not continuous in x_{2i-1} and x_{2i+1}.

The second approach is to work with the same chunks (only two points defining each polynomial) and not only impose continuity in each x_i but also continuity of the derivatives at each x_i.

5.2.3 Cubic Spline Interpolation

Suppose we want to do a **piece-wise cubic interpolation** between 2 points. Therefore we want to fit the polynomial

$$\phi_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad \text{for} \quad x_i \leq x \leq x_{i+1}$$

Then we need 4 equations. If we know $(x_i.f(x_i), f'(x_i))$ and $(x_{i+1}.f(x_{i+1}), f'(x_{i+1}))$, we can solve this system of equations and find the cubic interpolation. This process is called **piece-wise cubic Hermite interpolation**, because it is using the information of the derivative at each point. And if we modify one point, it will only change 2 curves. So the interpolation is local. Notice that the function is C^2 in (x_i, x_{i+1}) and the second derivatives are not continuous in x_i.

Let us see which are the equations we get:

$$\begin{aligned} f(x_i) &= a_i \\ f(x_{i+1}) &= a_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 + d_i(x_{i+1} - x_i)^3 \\ f'(x_i) &= b_i \\ f(x_{i+1}) &= b_i + c_i 2(x_{i+1} - x_i) + 3d_i(x_{i+1} - x_i)^2 \end{aligned}$$

Therefore to find the coefficients a_i, b_i, c_i, d_i we solve:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & (x_{i+1} - x_i) & (x_{i+1} - x_i)^2 & (x_{i+1} - x_i)^3 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2(x_{i+1} - x_i) & 3(x_{i+1} - x_i)^2 \end{bmatrix} \begin{bmatrix} a_i \\ b_i \\ c_i \\ d_i \end{bmatrix} = \begin{bmatrix} f(x_i) \\ f(x_{i+1}) \\ f'(x_i) \\ f'(x_{i+1}) \end{bmatrix} \tag{5.22}$$

But what if we do not have the derivatives?

If we do not have the derivatives, we can use a finite difference approximation of them, or we can just impose that the derivative at x_i of two adjacent curves is the same, without fixing its value. Therefore we will have the following set of equations:

$$\begin{aligned} \phi_i(x_i) &= f(x_i) \\ \phi_i(x_{i+1}) &= f(x_{i+1}) \\ \phi_i'(x_i) &= \phi_{i+1}'(x_i) \\ \phi_i''(x_i) &= \phi_{i+1}''(x_i) \end{aligned}$$

Therefore if we have $n+1$ points, we have n splines, each one with 4 unknowns, and up to now we have $2n + 2(n-1)$ equations. Therefore we need to impose 2 additional conditions that can be:

- **Free boundary** or **Natural spline**:

$$\phi''(x_0) = \phi''(x_n) = 0$$

- **Champed boundary** or **Complete spline**:

$$\phi'(x_0) = f'(x_0), \quad \phi'(x_n) = f'(x_n)$$

- **Not-a-knot**

$$\phi'''(x_0) = \phi'''(x_1), \ \phi'''(x_{n-1}) = \phi'''(x_n)$$

As you can see, in this case all the $4n$ equations are coupled, therefore the interpolation although it looks like local, it is not in reality. To solve for the coefficients, we need to construct a linear system $Ax = b$, where A is a tridiagonal matrix. Once we find the coefficients, we get a spline that is C^2 in all the interval (x_0, x_n).

5.3 Some additional comments

When we introduced interpolation, we talked about basis functions ϕ_i, and we said that we were looking for a function

$$f(x) = c_0\phi_0(x) + c_1\phi_1(x) + \cdots + c_n\phi_n(x)$$

but when we started to talk about piecewise interpolation, it looks like we forgot about them. Basis functions are important because they help us to describe any function, in the cases where we do not have any idea of its shape. For example, when we are solving differential equations. Some of them have a simple form, and some others not that simple.

In the case of piecewise-linear functions, the basis functions are called **hat functions**, and they are really famous in *Finite Elements method* to solve differential equations. As the Lagrange polynomials they satisfy $\phi_j(x_i) = \delta_{i,j}$, but the difference is that they have compact support (i.e. they are different from zero in a small interval). They are defined by

$$\phi_j(x_i) = \begin{cases} \frac{x - x_{j-1}}{x_j - x_{j-1}} & x_{j-1} \leq x \leq x_j \\ \frac{x - x_{j+1}}{x_j - x_{j+1}} & x_j \leq x \leq x_{j+1} \\ 0 & \text{otherwise} \end{cases} \tag{5.23}$$

The splines also have related basis functions called **B-splines**, and they are the basis of Computer-aided Design (CAD), and sometimes they are also used for solving differential equations.

What if my curve is too complicated or I want to interpolate points in more dimensions?

In those we use parametric curves. Therefore if we have the data $\{(x_i, y_i)\}$, we include a new variable τ_i that **parametrize** the curve. Then we solve two separate interpolation problems with the data $\{(\tau_i, x_i)\}$ and $\{(\tau_i, y_i)\}$, and then the resulting curve will be $(x(\tau), y(\tau))$.

But if I have a 100 of points, which is the best method to choose?

There is not a simple answer in this case. As we saw in chapter 4, we can formulate any model, and we can fit the data solving a least squares problem. In such a case, we can choose the order of the polynomial that we want, and fit as many points we have, and there is not a correspondence between points and polynomial degree. The problem (or sometimes the advantage) of this approach is that the model is not ensure to pass exactly through all the points, but, the residual error is reduced. Therefore if we have some experiment data, that we know it has measurement errors, the best option is to go for least squares, and it is up to you which model to select.

But then, why does interpolation exist? Well, not every model wants to fit experimental data. Sometimes you are sure your function has to satisfy certain points. And moreover, you are interested in the integral or the derivative of your function. Think in the case of a really complicated function, that you want to estimate its integral in certain interval. Then you would like to take as many points as you can in order to approximate that interval. Therefore you can use a global interpolation using "special" points, such as Chebyshev, or you can use piecewise interpolation, such as constant, linear or cubic.

Chapter 6

Numerical Integration

The idea behind numerical integration is to remember that if f is nice enough, the integral of f in an interval is the limit of Riemann sums. Therefore if we want to approximate an integral, it sounds reasonable to propose the following:

$$I = \int_a^b f(x)dx \approx \sum_{i=0}^{n} \omega_i f(x_i) \tag{6.1}$$

Where x_i are called the abscissas, and ω_i the weights. This form is called a **quadrature rule**.

Does this formula look familiar?

When we were interpolating points $\{(x_i, f(x_i)\}_{i=0}^n$, we chose certain basis functions such that

$$f(x) \approx p_n(x) = \sum_{i=0}^{n} f(x_i)\phi_i(x)$$

For example in the case of polynomial interpolation $\phi_i(x) = L_i(x)$ the Lagrange polynomials, and in the case of piecewise linear interpolation $\phi_i(x)$ were the hat functions.

Therefore using the interpolant, we have have that

$$I = \int_a^b f(x)dx \approx \int_a^b p_n(x)dx = \int_a^b \sum_{i=0}^{n} f(x_i)\phi_i(x)dx = \sum_{i=0}^{n} f(x_i) \underbrace{\int_a^b \phi_i(x)dx}_{\omega_i}$$

In conclusion, when we approximate an integral using a quadrature rule, we interpolate the function using a set basis functions for which we know the integral, and then we just transform the integral into a sum of terms.

6.1 Newton-Cotes Formulas

To introduce the different Newton-Cotes Formulas, let us follow the same steps from Polynomial Interpolation. First we have a function, and let us choose a set of $n+1$ points in the interval $[a, b]$. Once we have the points $\{(x_i, f(x_i))\}_{i=0}^{n}$, then we can find a unique polynomial of order n that passes through all the points.

For example, if $n = 0$ and we choose $x_0 = \frac{b+a}{2}$, then

$$p_0(x) = f(x_0) = f\left(\frac{b+a}{2}\right)$$

And the corresponding quadrature formula is the **Midpoint rule**

$$I \approx \int_a^b p_0(x)dx = f\left(\frac{b+a}{2}\right)(b-a)$$

Now, if $n = 1$ and we choose $x_0 = a$, and $x_1 = b$ then

$$p_1(x) = f(x_0)\frac{(x-x_1)}{x_0-x_1} + f(x_1)\frac{(x-x_0)}{x_1-x_0} = \frac{f(b)(x-a) - f(a)(x-b)}{b-a}$$

And the corresponding quadrature formula is the **Trapezoidal Rule**

$$I \approx \int_a^b p_1(x)dx = \frac{b-a}{2}\left(f(a) + f(b)\right)$$

And if $n = 2$ and we choose $x_0 = a$, $x_1 = \frac{a+b}{2}$, $x_2 = b$ then

$$\begin{aligned} p_2(x) &= f(x_0)\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f(x_1)\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f(x_2)\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \\ &= \frac{2}{(b-a)^2}f(a)(x-\frac{b+a}{2})(x-b) + f(\frac{b+a}{2})(x-a)(x-b) + f(b)(x-a)(x-\frac{b+a}{2}) \end{aligned}$$

And the corresponding quadrature formula is **Simpson's Rule**

$$I \approx \int_a^b p_2(x)dx = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{b+a}{2}\right) + f(b)\right)$$

Using equally spaced points, with $n = 3$ we get the three-eights rule and with $n = 4$ we get Boole's rule.

On the other hand, Midpoint rule is an example of an **open quadrature formula** because it does not include the endpoints a, b, whereas trapezoidal rule and Simpson's rule are called **closed quadrature formulas**, because they include the endpoints.

But following the same reasoning as in Interpolation, you can imagine that using equidistant points with a large n is not the most stable algorithm. Therefore we have two choices: split the interval into small chunks (which is equivalent to do a piecewise interpolation) or choose an optimal set of points for the quadrature rule. The first approach is called **Composite Newton-Cotes formulas** and the second one is called **Gauss quadrature**

6.1.1 Composite Newton-Cotes formula

Are we interested in preserving the derivatives at every point?

In the case of integration, we can split the interval into different segments, and compute the integral separately in each of them. Therefore, we can use an interpolation scheme that can have discontinuous derivatives or even discontinuities in the function at finite points.

$$I = \sum_{i=1}^{n} \int_{a_i}^{b_i} f(x)dx \approx \underbrace{\sum_{i=1}^{n} \int_{a_i}^{b_i} p^{(i)}(x)dx}_{\text{Integrate each segment}} = \underbrace{\int_{a}^{b} \phi(x)dx}_{\text{Piecewise Interpolant}} \tag{6.2}$$

If we use piecewise linear interpolation using $n+1$ points $x_0, x_1, ..., x_n$, it is equivalent to split the interval $[a, b]$ in n segments and apply trapezoidal rule to each one. Therefore the **composite trapezoidal rule** is

$$I \approx \sum_{i=1}^{n} \int_{a_i}^{b_i} p_1(x)dx = \sum_{i=1}^{n} \frac{x_i - x_{i-1}}{2} \left(f(x_{i-1}) + f(x_i)\right) \tag{6.3}$$

If all of the points are equally spaced, i.e. $x_i = a + ih$ where $h = (b-a)/n$ then

$$I \approx \sum_{i=1}^{n} \int_{a_i}^{b_i} p_1(x)dx = \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right) \tag{6.4}$$

Similarly with piecewise quadratic interpolation, we get the **composite Simpson's rule** (in here, we assume that the x_{2i} es the midpoint between x_{2i-i} and x_{2i+1}. This is not the most general case)

$$I \approx \sum_{i=1}^{n/2} \int_{a_i}^{b_i} p_2(x)dx = \sum_{i=1}^{n/2} \frac{x_{2i} - x_{2(i-1)}}{6} \left(f(x_{2(i-1)}) + 4f\left(x_{2i-1}\right) + f(x_{2i})\right)$$

Using equally spaced points,i.e. $x_i = a + ih$ where $h = (b-a)/n$ we get

$$I \approx \sum_{i=1}^{n/2} \int_{a_i}^{b_i} p_2(x)dx = \frac{h}{3}\left(f(a) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=2}^{n/2-1} f(x_{2i}) + f(b)\right) \tag{6.5}$$

6.1.2 Error of Newton-Cotes formulas

Since we are using a polynomial to interpolate $f(x)$ and then integrate, we can express the error based on the interpolation error. This means that if we call:

$$I = \int_a^b f(x)dx \qquad\qquad I_n = \int_a^b \phi_n(x)dx \tag{6.6}$$

Where $\phi_n(x)$ is an interpolant of $f(x)$ using $n+1$ points, we know that

$$I - I_n = \int_a^b f(x) - \phi_n(x)dx \tag{6.7}$$

For example if $\phi_n(x)$ is the polynomial that interpolates the points $\{(x_i, f(x_i)\}$ then we know from (5.16) that

$$e = I - I_n = \int_a^b f(x) - \phi_n(x)dx = \int_a^b f[x_0, x_1, ..., x_n, x] \prod_{i=0}^{n} (x - x_i)dx \tag{6.8}$$

And therefore for the previous Newton-Cotes formulas defined we have

- **Midpoint rule**
$$e = \frac{f''(\xi_1)}{24}(b-a)^3$$
- **Trapezoidal rule**
$$e = -\frac{f''(\xi_2)}{12}(b-a)^3$$
- **Simpson's rule**
$$e = -\frac{f^{(4)}(\xi_3)}{90}\left(\frac{b-a}{2}\right)^5$$
- **Composite Trapezoidal rule** (equally spaced coordinates)
$$e = \sum_{i=1}^{n} \int_{a_i}^{b_i} I - I_n dx = -\frac{f''(\eta_1)}{12}(b-a)h^2$$
- **Composite Simpson's rule** (equally spaced coordinates)
$$e = \sum_{i=1}^{n/2} \int_{a_i}^{b_i} I - I_n dx = -\frac{f^{(4)}(\eta_2)}{180}(b-a)\left(\frac{h}{2}\right)^4$$

So for example, how can we get the error of the *Midpoint rule*? Using Taylor series around $(b+a)/2$:

$$\begin{aligned} e &= \int_a^b f(x) - f\left(\frac{b+a}{2}\right) dx \\ &= \int_a^b f'\left(\frac{b+a}{2}\right)\left(x - \frac{b+a}{2}\right) + \frac{1}{2}f''(\xi)\left(x - \frac{b+a}{2}\right)^2 dx \\ &= \frac{f''(\xi_1)}{24}(b-a)^3 \end{aligned}$$

Definition 20. *We call* ***precision*** *of a quadrature formula the largest integer p such that the quadrature error is zero ($e = 0$)*

Example 8. *Which is the precision of the previous quadrature formulas?*

Solution. Since all the errors are expressed using derivatives, it is easy to see that:

- *Midpoint rule:* Uses 1 point and has precision $p = 1$
- *Trapezoidal rule:* Uses 2 points and has precision $p = 1$
- *Simpson's rule:* Uses 3 points and has precision $p = 3$
- *In general a Newton-Cotes rule* that uses $n+1$ points has precision $p = n$ if n is odd and $p = n+1$ if n is even (Because of symmetry around intermediate points)

□

6.2 Gaussian Quadrature

Going back to (6.1), we want to find weights ω_i and abscissas x_i such that:

$$I = \int_a^b f(x)dx \approx \sum_{i=0}^{n} \omega_i f(x_i)$$

With the Newton-Cotes approach, we saw that using $n+1$ points we can get at most $p = n+1$ precision, this means we can interpolate exactly polynomials of degree at most $p = n+1$. But if we see, in (6.1) we have $2(n+1)$ choices, therefore something tells us that we can optimize the choice of weights and abscissas to get a better precision with the same number of points.

Let us suppose we want to find a quadrature rule that exactly integrates all the polynomials of degree at most 3 in the interval $[-1, 1]$ using the minimum number

of points. In particular it should integrate $\{1, x, x^2, x^3\}$. Let us suppose we use 2 abscissas, they should satisfy:

$$\int_{-1}^{1} 1dx = \omega_1(1) + \omega_2(1)$$
$$\int_{-1}^{1} xdx = \omega_1(x_1) + \omega_2(x_2)$$
$$\int_{-1}^{1} x^2dx = \omega_1(x_1)^2 + \omega_2(x_2)^2$$
$$\int_{-1}^{1} x^3dx = \omega_1(x_1)^3 + \omega_2(x_2)^3$$

Now we have a non-linear system of equations to solve for $\{\omega_1, \omega_2, x_1, x_2\}$ Using symmetry we know that $x_1 = -x_2$ and $w_1 = w_2$ then we get

$$w_1 = 1 \qquad x_1 = -\frac{1}{\sqrt{3}} \qquad w_2 = 1 \qquad x_2 = \frac{1}{\sqrt{3}}$$

Therefore using only 2 points we can get precision $p = 3$ (in contrast to the Trapezoidal rule, where we just get $p = 1$). In general, we can extend the same procedure for any number of points. And we can show that using $n+1$ "optimal" points we get a precision of $p = 2n+1$. This quadrature approach is called **Gaussian Quadrature**.

The theory behind Gaussian quadrature is a family of orthogonal polynomials called *Legendre Polynomials*. The idea is that when we use Legendre polynomials to interpolate a function, we minimize the error in its integral. If ϕ_k is the Legendre polynomial of order k, then for any polynomial $q(x)$ of order $j < k$ we have that

$$\int_{-1}^{1} q(x)\phi_k(x)dx = 0 \tag{6.9}$$

Therefore if we want to integrate a polynomial $f(x)$ of order $p = 2k-1$, there exists $q(x), r(x)$ of degree less than k such that

$$f(x) = q(x)\phi_k(x) + r(x)$$

Therefore

$$\int_{-1}^{1} f(x)dx = \int_{-1}^{1} q(x)\phi_k(x) + r(x)dx = \int_{-1}^{1} r(x)dx \tag{6.10}$$

and if we evaluate f in x_i the roots of ϕ_k, then

$$f(x_i) = q(x_i)\phi_k(x_i) + r(x_i) = r(x_i) \tag{6.11}$$

Then if the quadrature rule integrates exactly $r(x)$, then it also integrates exactly $f(x)$.

If you want to see a complete explanation check Bradie, Section 6.6.

6.3 Some additional comments

What does it mean to intepolate (integrate) using "..." polynomials? For example using Legendre polynomials?

It means that if we want to use $n+1$ points, use the roots of the "..." polynomial of order $n+1$ as abscissas.

The Legendre polynomials are optimal to integrate polynomials, but what if my function is not a polynomial?

The Gaussian quadrature can be generalize to integrals of the shape

$$I = \int_a^b w(x)f(x)dx \approx \sum_{i=0}^{n} \omega_i f(x_i) \tag{6.12}$$

where $w(x)$ is a **weight function** that is positive and integrable. For each weight function, we can find an orthogonal family of polynomials which roots are the optimal choice for the abscissas. Some examples are

- Laguerre polynomials: $w(x) = e^{-x}$ on the interval $[0, \infty)$
- Hermite polynomials: $w(x) = e^{-x^2}$ on the interval $(-\infty, \infty)$
- Chebyshev polynomials: $w(x) = \frac{1}{\sqrt{1-x^2}}$ on the interval $[-1, 1]$

How can I integrate in multiple dimensions?

For example consider a simple domain $\Omega \in \mathbb{R}^m$, then the integral of the function over the domain can be decomposed into iterated integrals for each variable

$$I = \int_\Omega f(x)dx = \int_{a_{(1)}}^{b_{(1)}} \int_{a_{(2)}}^{b_{(2)}} \dots \int_{a_{(n)}}^{b_{(m)}} f(x_1, x_2, ..., x_m)dx_1dx_2...dx_m \tag{6.13}$$

An then we can apply a quadrature rule in each integral. For example if we use a quadrature rule of $n+1$ points in each direction, we will end up with a grid of $(n+1)^m$ points to evaluate the whole integral.

Chapter 7

Numerical Differentiation

When we started the course, one of the principal goals in developing all the methods was to solve numerically differential equations. Therefore, first we learned how to solve systems of equations (linear and non-linear) and we applied them to solve unconstrained optimization problems. Some of those optimization problems involved data fitting using linear and non-linear least squares. At the heart of all this methods was to approximate the function with a Taylor series expansion around certain point x_0

Later, we switched a bit from subject, and we look for methods to fit polynomials that exactly satisfy some data, i.e. that interpolates it. So given certain points (x_i, y_i), we proposed a model

$$\phi(x) = \sum_{j=1}^{n} c_j \phi_j(x)$$

i.e. a linear combination of basis functions, and we looked for coefficients c_j such that $y_i = \phi(x_i)$. First we proposed global polynomials, and using different basis we proved existence and error bounds. Then to avoid oscillations, we proposed local polynomials (piecewise functions), and we enforced continuity at the break points.

Lastly, we talked about integration as a direct application of integration. In this case we approximate the function with an interpolant, and instead of integrating the function, we integrate the interpolant. With this idea behind, we developed quadrature rules such that

$$I \approx \sum_{i=1}^{n} f(x_i)\omega_i$$

.

But if the idea is to solve differential equations, what about *numerical differentiation*?. In this chapter we will show two methods to find numerical differentiation

formulas. The first one is intuitive using Taylor series and the second one is more general using Lagrange Interpolation.

7.1 Numerical Differentiation using Taylor series

Along this quarter, we have been using numerical differentiation without introduce it formally. Recall Theorem 1, we have that

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(\xi) \text{ where } \xi \in [x_0, x_0+h] \tag{7.1}$$

Therefore we have

$$\begin{aligned} f'(x_0) &= \frac{f(x_0+h)-f(x_0)}{h} - \frac{h}{2}f''(\xi) \text{ where } \xi \in [x_0, x_0+h] \\ &= \frac{f(x_0+h)-f(x_0)}{h} + \mathcal{O}(h) \end{aligned} \tag{7.2}$$

Notice that when $h \to 0$, $\frac{f(x_0+h)-f(x_0)}{h} \to f'(x_0)$.

The formula (7.2) is called **forward difference approximation** and it is first order accurate becaouse the error is $\mathcal{O}(h)$. Similarly, using the Taylor series of $f(x_0-h)$ we get the **backward difference** formula

$$\begin{aligned} f'(x_0) &= \frac{f(x_0)-f(x_0-h)}{h} + \frac{h}{2}f''(\xi) \text{ where } \xi \in [x_0-h, x_0] \\ &= \frac{f(x_0)-f(x_0-h)}{h} + \mathcal{O}(h) \end{aligned} \tag{7.3}$$

Now using Taylor series up to order 2 around x_0 we have that there exist ξ_1 and ξ_2 near x_0 such that

$$\begin{aligned} f(x_0+h) &= f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{3!}f'''(\xi_1) \\ f(x_0-h) &= f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) - \frac{h^3}{3!}f'''(\xi_2) \end{aligned}$$

Therefore subtracting both equations and using intermediate value theorem, we get that for $\xi \in [x_0-h, x_0+h]$

$$\begin{aligned} f(x_0+h) - f(x_0-h) &= 2hf'(x_0) + \frac{h^3}{3}f'''(\xi) \\ f'(x_0) &= \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{h^2}{6}f'''(\xi) \end{aligned} \tag{7.4}$$

Equation (7.4) is called **centered difference** formula and it is second order accurate.

Now to approximate the second derivative, we can use again the Taylor series of $x_0 + h$ and $x_0 - h$ of order 3:

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{3!}f'''(x_0) + \frac{h^4}{4!}f^{(4)}(\xi_1)$$
$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) - \frac{h^3}{3!}f'''(x_0) + \frac{h^4}{4!}f^{(4)}(\xi_2)$$

Adding both equations we get the **centered formula for the second derivative**:

$$f(x_0 + h) + f(x_0 - h) = 2f(x_0) + h^2 f''(x_0) + 2\frac{h^4}{4!}f^{(4)}(\xi)$$
$$f''(x_0) = \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} - \frac{h^2}{12}f^{(4)}(\xi) \qquad (7.5)$$

And equation (7.5) is second order accurate.

Using more points tends to gives us a higher accuracy. So how can we generalize this methods to involve more points?

For example if we want to use $x_0 - 2h$, $x_0 - h$, x_0, $x_0 + h$ and $x_0 + 2h$ to generate a forth order approximation formula for the first derivative, we can solve the system

$$\begin{bmatrix} f(x_0) & f(x_0) & f(x_0) & f(x_0) & f(x_0) \\ 0 & (-2h)f'(x_0) & (-h)f'(x_0) & (h)f'(x_0) & (2h)f'(x_0) \\ 0 & \frac{(-2h)^2}{2!}f''(x_0) & \frac{(-h)^2}{2!}f''(x_0) & \frac{(h)^2}{2!}f''(x_0) & \frac{(2h)^2}{2!}f''(x_0) \\ 0 & \frac{(-2h)^3}{3!}f'''(x_0) & \frac{(-h)^3}{3!}f'''(x_0) & \frac{(h)^3}{3!}f'''(x_0) & \frac{(2h)^3}{3!}f'''(x_0) \\ 0 & \frac{(-2h)^4}{4!}f^{(4)}(x_0) & \frac{(-h)^4}{4!}f^{(4)}(x_0) & \frac{(h)^4}{4!}f^{(4)}(x_0) & \frac{(2h)^4}{4!}f^{(4)}(x_0) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_{-2h} \\ \alpha_{-h} \\ \alpha_h \\ \alpha_{2h} \end{bmatrix} = \begin{bmatrix} 0 \\ f'(x_0) \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

and find the coefficients for which we should multiply each Taylor expansion.

We divide each row by $f^{(i)}(x_0)h^i/i!$ and we get

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & -2 & -1 & 1 & 2 \\ 0 & (-2)^2 & (-1)^2 & (1)^2 & (2)^2 \\ 0 & (-2)^3 & (-1)^3 & (1)^3 & (2)^3 \\ 0 & (-2)^4 & (-1)^4 & (1)^4 & (2)^4 \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_{-2h} \\ \alpha_{-h} \\ \alpha_h \\ \alpha_{2h} \end{bmatrix} = \begin{bmatrix} 0 \\ 1/h \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Nevertheless, with this approach we do not have enough insight what does it mean each of the coefficients, and how does the underlined approximation behave.

7.2 Numerical Differentiation using Interpolation

Now let us go back to the case of approximating the first derivative at x_0 using 5 points: $x_0 - 2h$, $x_0 - h$, x_0, $x_0 + h$. In this case let us consider the Lagrange polynomial that interpolates the 5 points:

$$\phi(x) = \sum_{i=-2}^{2} f(x_0 + ih) L_i(x) \text{ where } L_i(x) = \prod_{\substack{j=-2 \\ j \neq i}}^{2} \frac{(x - (x_0 + jh))}{((x_0 + ih) - (x_0 + jh))}$$

Therefore

$$\phi'(x) = \sum_{i=-2}^{2} f(x_0 + ih) L_i'(x)$$

$$\text{where } L_i'(x) = \sum_{\substack{k=-2 \\ k \neq i}}^{2} \frac{1}{((x_0 + ih) - (x_0 + kh))} \prod_{\substack{j=-2 \\ j \neq i,k}}^{2} \frac{(x - (x_0 + jh))}{((x_0 + ih) - (x_0 + jh))}$$

In particular we have that

$$L_i'(x_i) = \frac{1}{h} \sum_{\substack{k=-2 \\ k \neq i}}^{2} \frac{1}{(i - k)} \tag{7.6}$$

$$L_i'(x_l) = \frac{1}{h} \frac{1}{(i - l)} \prod_{\substack{j=-2 \\ j \neq i,l}}^{2} \frac{(l - j)}{(i - j)} \tag{7.7}$$

Therefore

$$L_0'(x_0) = -\frac{1}{h} \sum_{\substack{k=-2 \\ k \neq 0}}^{2} \frac{1}{k} = \frac{1}{-2} + \frac{1}{-1} + \frac{1}{1} + \frac{1}{2} = 0$$

$$L_i'(x_0) = \frac{1}{ih} \prod_{\substack{j=-2 \\ j \neq i,0}}^{2} \frac{(-j)}{(i - j)}$$

Then

$$L'_{-2}(x_0) = -\frac{1}{2h}\left(\frac{(1)}{(-2+1)}\frac{(-1)}{(-2-1)}\frac{(-2)}{(-2-2)}\right) = \frac{1}{12h}$$
$$L'_{-1}(x_0) = -\frac{1}{1h}\left(\frac{(2)}{(-1+2)}\frac{(-1)}{(-1-1)}\frac{(-2)}{(-1-2)}\right) = -\frac{8}{12h}$$
$$L'_{1}(x_0) = \frac{1}{1h}\left(\frac{(2)}{(1+2)}\frac{(1)}{(1+1)}\frac{(-2)}{(1-2)}\right) = \frac{8}{12h}$$
$$L'_{2}(x_0) = \frac{1}{2h}\left(\frac{(2)}{(2+2)}\frac{(1)}{(2+1)}\frac{(-1)}{(2-1)}\right) = -\frac{1}{12h}$$

Therefore using the error formula for interpolation we have that

$$f'(x_0) = \phi'(x_0) + \frac{\partial}{\partial x}\left(f[x_0-2h, x_0-h, x_0, x_0+h, x_0+2h, x]\prod_{i=-2}^{2}(x-(x_0+ih))\right)_{x=x_0}$$
$$= \sum_{i=-2}^{2} f(x_0+ih)L'_i(x_0) + f[x_0-2h, x_0-h, x_0, x_0+h, x_0+2h, x_0] \prod_{\substack{i=-2 \\ i\neq 0}}^{2} (ih)$$
$$= \frac{1}{12h}\left(f(x_0-2h) - 8f(x_0-h) + 8f(x_0+h) - f(x_0+2h)\right) + \frac{h^4}{30}f^{(5)}(\xi)$$

Which is exactly the same result as solving the linear system given by the Taylor expansions around x_0. The difference is that now we understand that the derivative comes from fitting a polynomial of order 4 in the points x_0-2h, x_0-h, x_0, x_0+h.

The same behavior we had in interpolation, now we have it in differentiation. Then unless we are working with special abscissas (i.e. Chebyshev points, we call this spectral methods), we do not want to use all the points in the grid to approximate the derivative, otherwise we can have oscillations because of the Runge phenomenon.

Therefore we prefer to do is use local approximations of the derivatives (for example (7.2),(7.3),(7.4),(7.5)) . This is equivalent to use piecewise polynomial interpolation.

On the other hand, notice that regardless of the method we use to approximate the derivatives, we can write

$$f'(x_i) \approx \sum_{j=i_1}^{i_k} a_{ij} f(x_j) = (Af)_i \tag{7.8}$$

This means that we can find a matrix A such that transforms the vector $f = [f(x_0), ..., f(x_n)]$ into $f' = [f'(x_0), ..., f'(x_n)]$. This matrix is called differentiation

matrix. Therefore once we construct the matrix, we can find the derivative for all the points in just "one operation".

7.3 Richardson Extrapolation

Up to now, all the error estimates that we have are based in the theoretical derivation of the methods (i.e using Taylor series or Interpolation). Nevertheless, we would like to have a practical error estimate. Let us suppose we have 2 discretizations. In the first one the points are equally spaced at a distance h. In the second one, they are equally spaced at a distance $h/2$. Now let us use the same differentiation method in both to find $f'(x_0)$. For example centered differences. Let D be the real derivative, D_h the approximation from the first grid and $D_{h/2}$ the approximation from the second one. We know from Taylor series that

$$D = D_h + \frac{1}{6}f'''(x_0)h^2 + O(h^3) \tag{7.9}$$

$$D = D_{h/2} + \frac{1}{6}f'''(x_0)(h/2)^2 + O(h^3) \tag{7.10}$$

If we are approximating the derivative, it means that we also do not have any idea fo the value of $f'''(x_0)$. If we only had one discretization, we could not find the error, but now with 2 approximations we can subtract them and get:

$$0 = (D_h - D_{h/2}) + \frac{1}{6}f'''(x_0)\frac{3}{4}h^2 + O(h^3)$$

Therefore we have an approximation for the error of $\mathcal{O}(h^2)$.

$$\frac{1}{6}f'''(x_0)h^2 = -\frac{4}{3}(D_h - D_{h/2}) + O(h^3)$$

And this implies that we have a better approximation for the derivative (of $\mathcal{O}(h^3)$)

$$\begin{aligned} D &= D_h - \frac{4}{3}(D_h - D_{h/2}) + O(h^3) \\ &= \frac{4D_{h/2} - D_h}{3} + O(h^3) \\ &= \frac{1}{3}\left(4\frac{f(x_0 + h/2) - f(x_0 - h/2)}{h} - \frac{f(x_0 + h) - f(x_0 - h)}{2h}\right) + O(h^3) \\ &= \frac{8f(x_0 + h/2) - 8f(x_0 - h/2) - f(x_0 + h) + f(x_0 - h)}{12(h/2)} + O(h^3) \end{aligned} \tag{7.11}$$

Therefore this gives you the same differentiation rule using 5 points derived using Taylor series or interpolation. The meaning here is a bit different. The idea is to have an adaptive method, where we reduce h to half in order to have a better approximation, and we estimate the error, using a previous guess.

Chapter 8

Initial Value Ordinary Differential Equations

In this part of the course we will start to work with the simplest model that includes a derivative. We will look for a function $y(t)$ such that

$$\frac{dy}{dt} = f(t, y), \ for \ a \leq t \leq b \tag{8.1}$$

We call t the **independent variable** and y the **dependent variable**. What makes it difficult to solve is that f depends on the unknown variable.

Also notice that (8.1) has infinite number of solutions. For example if $f(t, y) = 0$, then $y(t) = constant$ and it can take any value. Therefore we need to specify additional conditions in order to have a unique solution. In this case we will specify the value of the function at $t = a$

$$y(a) = \alpha \tag{8.2}$$

And together (8.1) plus (8.2) are called **Initial Value problem**. Also to have a unique solution, we need $f(t, y)$ to be Lipschitz continuous (See Bradie Section 7.1, LeVeque Section 5.2)

8.1 Euler's method

Instead of finding a numerical approximation of y for all the $t \in [a, b]$, we will choose a discrete set of points:

$$a = t_0 < t_1 < t_2 < \cdots < t_{N-1} < t_N = b \tag{8.3}$$

And for simplicity let us suppose they are equally spaced, i.e. $t_i = a + ih$ where $h = (b - a)/N$.

For each of this t's we have:

$$\begin{aligned} y(t_0) &= \alpha \\ \frac{dy}{dt}(t_i) &= f(t_i, y(t_i)) \end{aligned} \tag{8.4}$$

How can we get rid of the derivative?

Let us call y_i the numerical approximation of $y(t_i)$. We can use any of the approximations form numerical differentiation to approximate the derivative. For example if we use **Forward differences** we transform (8.4) into

$$\begin{aligned} y_0 &= \alpha \\ \frac{y_{i+1} - y_i}{h} &= f(t_i, y_i) \implies y_{i+1} = y_i + hf(t_i, y_i) \end{aligned} \tag{8.5}$$

This is the so called **Euler's method**. Notice that the advantage of (8.5) is that y_{i+1} is *explicitly* defined in terms of y_i. This type of method is called an **explicit method**.

Notice that we could also use **Backward differences** to approximate the derivative and get:

$$\begin{aligned} y_0 &= \alpha \\ \frac{y_i - y_{i-1}}{h} &= f(t_i, y_i) \implies y_{i+1} = y_i + hf(t_{i+1}, y_{i+1}) \end{aligned} \tag{8.6}$$

This is the so called **Backward Euler's method**. The shape between Euler and backward Euler is the same, but the substantial difference is that that y_{i+1} depends *implicitly* on itself (notice the $f(t_{i+1}, y_{i+1})$ term). This is called an **implicit method**. The difficulty of implicit methods is that in general we need to solve a nonlinear system of equations. Nevertheless, later we will se that they have advantages.

8.2 Properties of the methods

How can we decide which method to use?

Let us remember the discussion about scientific computing we had at the beginning of the quarter. When we talked about algorithms, there were 2 important properties an algorithm should have: Robustness and Efficiency. The first one was related with the stability of the method (if I perturb the input, how much does the output change?). The second one was related with convergence of the method (how many steps should I take in order to get a small error?).

This two concepts are again present in the solution of differential equations. More properly we will look for a one step method:

$$y_{i+1} = y_i - h_i\phi(f, t_i, y_i, y_{i+1}, h_i)$$

- **Consistent** Define the *local truncation error* as

$$\tau_i = \frac{y(t_{i+1}) - y(t_i)}{h_i} - \phi(f, t_i, y(t_i), y(t_{i+1}), h_i) \tag{8.7}$$

This means, τ_i is the error that we get when we evaluate the approximation using the exact solution of y. For example, for Euler's method we have

$$\tau_i = \frac{y(t_{i+1}) - y(t_i)}{h} - f(t_i, y(t_i)) \tag{8.8}$$

Using Taylor series around y_{t_i} we have:

$$\begin{aligned}\tau_i &= \frac{y(t_i) + y'(t_i)h + y''(t_i)h^2/2 + \mathcal{O}(h^3) - y(t_i)}{h} - f(t_i, y(t_i)) \\ &= y'(t_i) + y''(t_i)h/2 + \mathcal{O}(h^2) - f(t_i, y(t_i)) \text{ using the ODE} \\ &= \frac{h}{2}y''(t_i) + \mathcal{O}(h^2)\end{aligned}$$

Therefore it is a first order method because $t_i = \mathcal{O}(h)$. A method is *consistent* if $\tau_i \to 0$ as $h_i \to 0$

- **Convergent** Define the *global discretization error* as $y(t_i) - y_i$, it is the cumulative error introduced in all the steps. We say that a method is convergent if

$$\lim_{h\to 0} \max_{1\le i\le N} |y(t_i) - y_i| = 0 \tag{8.9}$$

- **Stable**: How can we relate local truncation error and global discretization error?
Let us suppose that we want to solve the ODE

$$\begin{aligned}\frac{dy}{dt} &= \lambda y + g(t),\ 0 \le t \le T \\ y(0) &= y_0\end{aligned}$$

Integrating in both sides (or using Duhamel's principle), the solution is given by

$$y(t) = y_0 e^{\lambda t} + \int_{i=0}^{t} e^{\lambda(t-s)} g(s) ds$$

On the other hand using Forward Euler we have that

$$y_{i+1} = y_i + \lambda h y_i + hg(t_i) = (1 + \lambda h)y_i + hg(t_i)$$

If we evaluate Forward Euler with the exact solution, and using the local truncation error, we get:

$$\begin{aligned} y_{i+1} &= (1 + \lambda h)y_i + hg(t_i) \\ y(t_{i+1}) &= (1 + \lambda h)y(t_i) + hg(t_i) + h\tau_i \\ \boxed{e_{i+1} &= (1 + \lambda h)e_i - h\tau_i} \end{aligned} \tag{8.10}$$

Using the recurrence relation, we have

$$e_n = -h \sum_{i=1}^{n} (1 + \lambda h)^{n-i} \tau_{i-1}$$

Since $|1+\lambda h| < e^{|\lambda| h}$ (this is called Zero-Stability) we can bound $|(1+\lambda h)^{n-i}| \leq e^{T|\lambda|}$ and since we have n terms then $nh = T$. Therefore if $\tau = [\tau_0, \tau_1, ...\tau_{n-1}]$, then

$$|e_n| \leq = T e^{T|\lambda|} ||\tau||_\infty$$

Since Euler is consistent then $||\tau||_\infty \to 0$ and therefore $|e_n| \to 0$, i.e. the method is convergent.

But notice that this is not enough. If we go back to (8.10), at each time step, the previous error is amplified by a factor of $(1 + \lambda h)$. If $\lambda < 0$, the solution is characterized by an exponential decay. Therefore, if the method is **absolutely stable**, we expect that the errors will also decay. This requires that:

$$|1 + \lambda h| < 1 \iff -1 < 1 - |\lambda| h < 1 \iff h < \frac{2}{|\lambda|} \tag{8.11}$$

Therefore we need to choose h sufficiently small and $\lambda < 0$ to have a nice behavior in Euler.

For the case of backward differences, we have that

$$y_{i+1} = y_i + \lambda h y_{i+1} + hg(t_{i+1}) \Longrightarrow y_{i+1} = \frac{1}{(1 - \lambda h)}(y_i + hg(t_{i+1}))$$

If we evaluate Backward Euler with the exact solution, and using the local truncation error, we get:

$$\begin{aligned} y_{i+1} &= \frac{1}{(1 - \lambda h)}(y_i + hg(t_{i+1})) \\ y(t_{i+1}) &= \frac{1}{(1 - \lambda h)}(y(t_i) + hg(t_{i+1})) + h\tau_i \\ \boxed{e_{i+1} &= \frac{1}{(1 - \lambda h)} e_i - h\tau_i} \end{aligned}$$

Then

$$e_n = -h \sum_{i=1}^{n} \left(\frac{1}{(1-\lambda h)} \right)^{n-i} \tau_{i-1}$$

And as before we can show convergence. To have absolute stability we need

$$\left| \frac{1}{(1-\lambda h)} \right| < 1 \Longleftrightarrow 1 < |1 - \lambda h|$$

If $\lambda < 0$ or $h > 2/\lambda$ we have absolute stability. In particular if $\lambda < 0$ we do not have any restriction in h.

As we can see, the choice of time step h determines 2 different properties:

- *Truncation error* Since $\tau_i = \mathcal{O}(h^p)$ for certain p, then we need to choose a really small h in order to avoid truncation error.
- *Stability* In order to have absolute stability, we have a restriction on h given λ. In the linear case, it is easy to see what is λ. In the non-linear case, we can always think in linearizing $f(t, y)$ using Taylor series:

$$f(t,y) = \underbrace{f(t_0,t_0) + \left.\frac{\partial f}{\partial t}\right|_{(t_0,y_0)} (t-t_0)}_{\approx g(t)} + \underbrace{\left.\frac{\partial f}{\partial y}\right|_{(t_0,y_0)} (y-y_0)}_{\approx \lambda y} + \mathcal{O}(|t-t_0|^2 + |y-y_0|^2)$$

Therefore λ is usually related with $\frac{\partial f}{\partial y}$ around the points of the solution. For example we can take

$$\lambda = \max_{t,y \in \Omega} \left| \left. \frac{\partial f}{\partial y} \right|_{(t,y)} \right|$$

For a redefined set Ω where we know the solution lives.

8.3 Higher order methods

Up to now, forward and backward Euler are only first order accurate. It means that we need to take really small h in order to have small truncation error. If we want to find higher order methods, let us see from where we can derive Euler method first:

1. **Finite Differences**: We are looking for an approximation of $\frac{dy}{dx}$. Therefore we can have:
   - *Forward Differences* : Euler's method

$$\frac{dy}{dx}(t_i, y_i) = \frac{y_{i+1} - y_i}{h} + \mathcal{O}(h)$$

- *Backward Differences* : Backward Euler's method

$$\frac{dy}{dx}(t_i, y_i) = \frac{y_i - y_{i-1}}{h} + \mathcal{O}(h)$$

- *Centered Differences* : **Multi-step method**

$$\frac{dy}{dx}(t_i, y_i) = \frac{y_{i+1} - y_{i-1}}{2h} + \mathcal{O}(h^2)$$

Therefore replacing in the ODE we get

$$\begin{aligned} \frac{y_{i+1} - y_{i-1}}{2h} &= f(t_i, y_i) \\ y_{i+1} &= y_{i-1} + 2hf(t_i, y_i) \end{aligned} \tag{8.12}$$

And it is a multi-step method because y_{i+1} depends on 2 time-steps: y_i and y_{i-1}. This type of methods are more complicated because we need to specify more initial conditions and the analysis of global error and stability is more sensitive. Nevertheless they have a higher order of accuracy.

2. **Taylor's Series** Recall the Theorem1, and let us use it for $y(t_{i+1})$ around $y(t_i)$ then we have

$$y(t_{i+1}) = y(t_i) + y'(t_i)h + y''(t_i)\frac{h^2}{2!} + y'''(t_i)\frac{h^3}{3!} + ... \tag{8.13}$$

And using the ODE we have that

$$\begin{aligned} y'(t_i) &= f(t_i, y_i) \\ y''(t_i) &= \frac{d}{dt}\left(f(t_i, y_i)\right) \\ &= \frac{\partial}{\partial t} f(t_i, y_i) + \frac{\partial}{\partial y} f(t_i, y_i)\frac{dy}{dt} \\ &= \frac{\partial}{\partial t} f(t_i, y_i) + f(t_i, y_i)\frac{\partial}{\partial y} f(t_i, y_i) \\ y'''(t_i) &= \dots \end{aligned}$$

Therefore second order scheme, we can ignore all the terms of third order or higher and we can just replace the equivalences given by the ODE:

$$y(t_{i+1}) = y(t_i) + f(t_i, y_i)h + \left(\frac{\partial}{\partial t} f(t_i, y_i) + f(t_i, y_i)\frac{\partial}{\partial y} f(t_i, y_i)\right)\frac{h^2}{2!} \tag{8.14}$$

This is called a **Taylor method**, and it is explicit, high order accurate one-step method. The only drawback of this family of methods is that we need to compute the derivatives of f, and as the order increases, handling them starts to be complicated. It is a nice theoretical method, but not useful in practice.

3. **Numerical Integration** If we go back to (8.1), integrating in both sides we have that

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t))dt \tag{8.15}$$

Therefore we can use different quadrature rules to approximate the integral:

- If we use the value on the left $\Longrightarrow$ Euler's method:

$$\int_{t_i}^{t_{i+1}} f(t, y(t))dt = h(f(t_i, y_i) + \mathcal{O}(h))$$

- If we use the value on the right $\Longrightarrow$ Backward Euler's method:

$$\int_{t_i}^{t_{i+1}} f(t, y(t))dt = h(f(t_{i+1}, y_{i+1}) + \mathcal{O}(h))$$

- If we use the value on the middle $\Longrightarrow$ Midpoint method:

$$\int_{t_i}^{t_{i+1}} f(t, y(t))dt = h(f(t_{i+1/2}, y_{i+1/2}) + \mathcal{O}(h^2))$$

- If we use both endpoints $\Longrightarrow$ Trapezoidal method:

$$\int_{t_i}^{t_{i+1}} f(t, y(t))dt = h\left(\frac{f(t_i, y_i) + f(t_{i+1}, y_{i+1})}{2} + \mathcal{O}(h^2)\right)$$

- If we use both endpoints and middle $\Longrightarrow$ Simpson's method:

$$\int_{t_i}^{t_{i+1}} f(t, y(t))dt = h\left(\frac{f(t_i, y_i) + 4f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})}{6} + \mathcal{O}(h^4)\right)$$

Of all of this approximations, only Euler is explicit, and moreover some of the methods involve points that are not in the original discretization (for example $(t_{i+1/2}, y_{i+1/2})$).

> *If we want to transform all the methods in explicit one-step methods: What should we do?*

In this case, we can approximate the intermediate points using Euler's method. This family of methods is called **Runge-Kutta methods.** For example:

- For the Midpoint method, we can approximate

$$y_{i+1/2} \approx y_i + \frac{h}{2} f(t_i, y_i)$$

Therefore we can define the **Modified Euler method** as

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2}) = y_i + hf\left(t_i + \frac{h}{2}, y_i + \frac{h}{2} f(t_i, y_i)\right) \tag{8.16}$$

- For the Trapezoidal method, we can approximate

$$y_{i+1} \approx y_i + hf(t_i, y_i)$$

Therefore we can define the **Heun's method** as

$$\begin{aligned} y_{i+1} &= y_i + \frac{h}{2}\left(f(t_i, y_i) + f(t_{i+1}, y_{i+1})\right) \\ &= y_i + \frac{h}{2}\left(f(t_i, y_i) + f(t_i + h, y_i + hf(t_i, y_i))\right) \end{aligned} \tag{8.17}$$

- For Simpson's method, we can approximate

$$\begin{aligned} y^{(1)}_{i+1/2} &= y_i + \frac{h}{2} f(t_i, y_i) \text{ using Forward Euler} \\ y^{(2)}_{i+1/2} &= y_i + \frac{h}{2} f(t_{i+1/2}, y^{(1)}_{i+1/2}) \text{ Using Backward differences} \\ y^{(1)}_{i+1} &= y_i + hf(t_{i+1/2}, y^{(2)}_{i+1/2}) \text{ Using Modified Euler} \end{aligned}$$

Therefore when we use Simpson's formula we have the famous **4-th order Runge-Kutta**

$$\begin{aligned} y_{i+1} = y_i + \frac{h}{6}\Big(&f(t_i, y_i) + 2f(t_i + 1/2h, y^{(1)}_{i+1/2}) \\ &+2f(t_i + 1/2h, y^{(2)}_{i+1/2}) + f(t_i + h, y^{(1)}_{i+1})\Big) \end{aligned} \tag{8.18}$$

A general Runge-Kutta method of k stages is of the form

$$y_{i+1} = y_i + \sum_{j=1}^{k} w_j k_j \tag{8.19}$$

$$\text{where } k_j = hf\left(t_i + c_j h, y_i + \sum_{l=1}^{j-1} a_{j,l} k_l\right) \tag{8.20}$$

with $c_1 = 0$

How does the ODE's methods are implemented in MATLAB?

Depending on the requirements, MATLAB offers multiple ODE solvers `http://www.mathworks.com/help/matlab/math/choose-an-ode-solver.html`. Most of them are generalizations of Runge-Kutta methods (implicit or explicit) and multi-step methods.

The principal characteristic among them is that the points in time are not equally spaced (i.e. h_i is not constant). Moreover, at each iteration, they adjust the position

of the new $t_{i+1} = t_i + h$ depending on the local error tolerance they want to have. To have an estimation for the local error, they solve the value of y_{i+1} using 2 methods of order q and $q+1$, and then they compute the difference between the results. If $\hat{y}_{i+1}$ is the estimation for order q and y_{i+1} the estimation for order $q+1$, then we accept the value of y_{i+1} if

$$|\hat{y}_{i+1} - y_{i+1}| \leq h\ tol$$

. If not we compute a new $\bar{h}$

$$\bar{h} = h\left(\mu \frac{h\ tol}{|\hat{y}_{i+1} - y_{i+1}|}\right)^{1/q}$$

where μ is a factor between 0 and 1 (we can take $\mu = 0.9$).

For example, `ode45` is based in 2 Runge-Kutta methods of order 4 and 5. For each of the ODE solvers that MATLAB offers, you can check the documentation and at the end of it, you can find a description of the algorithm that is used behind the scenes. Most of them have fancy names, but all of them are based in the basic ideas described here.

8.4 Systems of equations, Stiffness

Most of the problems do not deal with an isolated ODE equation. In general, we have multiple variables that depend on time and the relations between them. For example think in a biological system analyzing a predator - prey model

$$\text{Prey Population}: \frac{dN_1}{dt} = \underbrace{\alpha N_1}_{\text{growth}} - \underbrace{\beta N_1 N_2}_{\text{loss for predation}}$$

$$\text{Predator Population}: \frac{dN_1}{dt} = \underbrace{\delta N_1 N_2}_{\text{growth from predation}} - \underbrace{\gamma N_2}_{\text{death rate}}$$

Other type of models come from chemical reactions. If I have u_1 concentration of chemical compound A, and u_2 concentration of chemical compound B, and A is transform into B at rate K_1 and B is transform into A at rate K_2 :

$$A \underset{K_2}{\overset{K_1}{\rightleftharpoons}} B$$

Then we get a set of ODEs for u_1 and u_2

$$\frac{du_1}{dt} = -K_1 u_1 + K_2 u_2$$
$$\frac{du_2}{dt} = K_1 u_1 - K_2 u_2$$

And in general, even if we are working with only one differential equation, most of the times it involves higher order derivatives. For example, the pendulum equation we had at the beginning of the course:

$$\frac{d^2\theta}{dt^2} + b\frac{d\theta}{dt} + \omega^2 \sin\theta = 0$$

We can call $\phi = \frac{d\theta}{dt}$ and we have the following system:

$$\begin{aligned} \frac{d\theta}{dt} &= \phi \\ \frac{d\phi}{dt} &= -b\phi - \omega^2 \sin\theta \end{aligned}$$

In general, when we talk about an Initial Value problem for a ODE system, we want to solve for $\mathbf{u}(t) = [u_1(t), u_2(t)..., u_n(t)]'$ vector of unknowns such that

$$\begin{cases} \frac{d\mathbf{u}}{dt} = \mathbf{f}(t, u_1(t), u_2(t), ..., u_n) \\ \mathbf{u}(0) = [\alpha_1, \alpha_2, ..., \alpha_n] \text{ known} \end{cases} \tag{8.21}$$

And as we can expect, all the numerical methods that we developed previously (*Euler, Backward Euler, Runge-Kutta*) extend naturally to solve systems of ODEs. The only difference is that we need to reconsider the stability properties of the methods, and how that affects the performance of it.

The simplest non-trivial case is when $\mathbf{f}(t, u_1(t), u_2(t), ..., u_n)$ is linear in $\mathbf{u}$, i.e.

$$\begin{cases} \frac{d\mathbf{u}}{dt} = A\mathbf{u} \\ \mathbf{u}(0) = [\alpha_1, \alpha_2, ..., \alpha_n] \text{ known} \end{cases} \tag{8.22}$$

If we assume that A is diagonal, i.e. $A = diag(\lambda_1, \lambda_2, ..., \lambda_n)$, then we can uncouple the equations:

$$\begin{cases} \frac{du_1}{dt} = \lambda_1 u_1; & u_1(0) = \alpha_1 \\ \frac{du_2}{dt} = \lambda_2 u_2; & u_2(0) = \alpha_2 \\ \dots & \dots \\ \frac{du_n}{dt} = \lambda_n u_n; & u_n(0) = \alpha_n \end{cases} \implies \begin{array}{c} u_1(t) = \alpha_1 e^{\lambda_1 t} \\ u_2(t) = \alpha_2 e^{\lambda_2 t} \\ \dots \\ u_n(t) = \alpha_n e^{\lambda_n t} \end{array} \tag{8.23}$$

Let us assume all $\lambda_j < 0$. Therefore each λ_j is associated with the rate of decay of u_j. To use Euler to solve the system we need that each spacing $h_j < 2/|\lambda_j|$, where

$$\frac{u_j^{(i+1)} - u_j^{(i)}}{h_j} = \lambda_j u_j^{(i)}$$

If we solve the system as a whole, to have stability we need $h \leq \min_j h_j < 2/\max_j |\lambda_j|$. But in the other hand, if I am interesting in a large scale behavior, the variables that will dominate are those with the smallest rate of decay, i.e. with the smallest values of $|\lambda_i|$.

This means that in order to have stability, we need to choose h according to the "modes" or $|\lambda_i|$ that we are less interested on, making the computation really slow. If the gap between the largest λ_i and the smallest one is considerably big, then we say that the problem is **stiff**.

Usually stiff problems are characterized by a separation of time scale, and when we are using *explicit* methods, we required a really small time step to preserve the stability of the method avoiding effects of the small scale. This is a very common situation in reaction-diffusion and transport equations.

In these cases, it is strongly recommended to use *implicit methods* based on Backward Euler and Trapezoidal method, because they are absolutely stable regardless of the size of h and λ. This means that if we solve the system using implicit methods, we can choose h based only in the accuracy we are looking for the method (based on the slowest modes) without being worried of stability issues given by fastest modes.

But in general, we are not solving a diagonal system, how can we know a problem is stiff?

To determine if a problem is stiff is not an easy task. It is similar to the case of well or ill-conditioned matrices. There is not a magic threshold that can define stiffness. Nevertheless it is important to know that if we have a linear ODE system (8.22), such that A is diagonalizable, then we can apply a change of variables to each eigenspace and we can end up with (8.23) where each λ_i are the eigenvalues of A.

In a more general case, if $\mathbf{f}(t, u_1, u_2, ..., u_n)$ is not linear, we can always think in its linear approximation using the Taylor series. In such a case A would be the Jacobian matrix of f at $(t, u_1, u_2, ..., u_n)$.

Chapter 9

Boundary Value Ordinary Differential Equations

In te previous chapter, we started with a first order ODE of the form

$$\begin{cases} \frac{dy}{dt} = f(t,y) & \text{for } \ t_0 \leq t \leq t_f \\ y(t_0) = y_0 \end{cases}$$

Since it is a first order, we only require one extra condition to have a unique solution, $y(t_0) = y_0$. But what about higher order ODE's?. For example in the homework we have

$$\theta'' + b\theta' + \omega^2 \sin(\theta) = 0 \text{ for } \ 0 \leq t \leq T$$

and we can solve it if θ and θ' are specified at the initial point $t = 0$, i.e. if it is an initial value problem. But what about if we have other type of conditions? for example if we specify values $\theta(0) = \theta_0$ and $\theta(T) = \theta_T$?

In the case where the extra conditions are specified in more than one point, generally at both endpoints ($t = t_0 \& t = t_f$) we say we have a boundary value problem. And now we can have different types of conditions:

- **Dirichlet Condition** The value of the function is specified, i.e $\theta(T) = 0$
- **Neumann Condition** The value of the (normal) derivative of the function is specified, i.e $\theta(0)' = 1$
- **Robin Condition** A linear combination of function and derivative is specified, i.e. $\alpha\theta(0) + \beta\theta'(0) = \gamma$
- **Periodic Condition** We identify initial point with end point as the same point, i.e. $\theta(0) = \theta(T)$ and $\theta'(0) = \theta'(T)$

In this chapter we will talk about two different methods to solve a boundary value problem: Shooting problem and finite differences. The first one will use the tools we developed in the previous chapter to redefine the problem as a Initial Value problem, whereas the second approach will be extended to Partial differential equations, and solves all the discretization points at once.

9.1 Shooting Method

Suppose we have the following Boundary Value problem

$$\begin{cases} \theta'' + b\theta' + \omega^2 \sin(\theta) = 0 & \text{for } \ 0 \leq t \leq T \\ \theta(0) = \theta_0 & \theta(T) = \theta_T \end{cases} \tag{9.1}$$

If we would like to use the methods to solve Initial Value Problems, how would you solve this problem?

To use an initial value approach we would like to solve the system for $0 \leq t \leq T$:

$$\begin{cases} \theta' = \phi \\ \phi' = -b\phi - \omega^2 \sin(\theta) \\ \theta(0) = \theta_0 \\ \phi(0) = \phi_0 \end{cases} \tag{9.2}$$

The only problem is that we do not know a priori ϕ_0. Therefore in the shooting method we "try" different values of ϕ_0 until we get $\theta(T) = \theta_T$.

Does this setting look familiar?

Indeed! We are solving a root finding problem. This means that we want to find ϕ_0 such that θ as a function of its initial values, gives us $\theta(T; \phi_0) = \theta_T$. Therefore we should use any of the methods in Chapter 2 to find a result. At each iteration we will solve an initial value problem for a guess of ϕ_0. If $|\theta(T; \phi_0) - \theta_T| > tol$, then we update ϕ_0 using a root finding algorithm.

Since each iteration is really expensive, we would like to start with an appropriate first guess, and use a fast convergence algorithm, such as Newton. The problem of Newton's method for root finding is that if we call $g(\phi_0) = \theta(T; \phi_0)$, it requires the derivative of g with respect to ϕ_0, which is not easy to compute. Instead we prefer to use Secant method and we update ϕ_0 as

$$\phi_0^{(i+1)} = \phi_0^{(i)} - g\left(\phi_0^{(i)}\right) \frac{\phi_0^{(i)} - \phi_0^{(i-1)}}{g\left(\phi_0^{(i)}\right) - g\left(\phi_0^{(i-1)}\right)} \tag{9.3}$$

How to choose an initial guess for ϕ_0

If the ODE is linear (i.e. $\theta'' = p(t)\theta' + q(t)\theta + r(t)$) then we can start assuming that $\phi_0(0) = (\theta(T) - \theta(0))/T$. If the ODE is nonlinear, it is more difficult to predict how the method behaves, and moreover it could exists more than one solution. Therefore just try with different initial guesses, to see how it performs.

9.1.1 The linear case

Consider the following ODE

$$\begin{cases} y'' = p(t)y' + q(t)y + r(t) \quad t_0 \leq t \leq t_f \\ y(t_0) = y_0 \\ y(t_f) = y_f \end{cases} \tag{9.4}$$

With linear equations, the superposition principle holds. It means that we can decompose the extra conditions (either boundary or initial conditions) into homogeneous and nonhomogenous part, solve them separately and add the two solutions as the final result. Following this approach we can consider the following 2 ODE systems:

$$\begin{cases} u'' = p(t)u' + q(t)u + r(t) \quad t_0 \leq t \leq t_f \\ u(t_0) = y_0 \\ u'(t_0) = 0 \end{cases} \tag{9.5}$$

$$\begin{cases} w'' = p(t)w' + q(t)w \quad t_0 \leq t \leq t_f \\ w(t_0) = 0 \\ w'(t_0) = 1 \end{cases} \tag{9.6}$$

And assume that $y = u + cw$. Therefore in order to satisfy (9.4), we need $y(t_f) = u(t_f) + cw(t_f)$. Solving (9.5) and (9.6) numerically, we can approximate c as:

$$c = \frac{y_f - u(t_f)}{w(t_f)} \tag{9.7}$$

Since the superposition principle does not hold in nonlinear equations, we cannot use this approach to solve them.

9.1.2 Other types of Boundary Conditions

Up to now, we have talked only about Dirichlet Boundary conditions. What about Robin and Neumann? Since Robin is the most general one, we will focus only in it.

For the Linear case, instead of solving (9.5) and (9.6), we can consider the following set of Initial Value Problems:

$$\begin{cases} u'' = p(t)u' + q(t)u \quad t_0 \leq t \leq t_f \\ u(t_0) = 1 \\ u'(t_0) = 0 \end{cases} \tag{9.8}$$

$$\begin{cases} w'' = p(t)w' + q(t)w \quad t_0 \leq t \leq t_f \\ w(t_0) = 0 \\ w'(t_0) = 1 \end{cases} \tag{9.9}$$

$$\begin{cases} z'' = p(t)z' + q(t)z + r(t) \quad t_0 \leq t \leq t_f \\ z(t_0) = 0 \\ z'(t_f) = 0 \end{cases} \tag{9.10}$$

And we assume that $y(t) = \alpha_1 u(t) + \alpha_2 w(t) + \alpha_3 z(t)$, and we solve for the boundary conditions.

In general, for the nonlinear case, we continue solving a root finding problem. If the conditions are Neumann ($y'(t_0)$ is given), then we start with a guess of the function value $y(t_0) = \alpha$, and we iterate until the boundary conditions are satisfied. If the conditions are Robin, we can decide either to guess $y'(t_0)$ or $y(t_0)$, and the other term is given by the Boundary conditions. If the problem is nonlinear, we cannot ensure the stability of the algorithm beforehand. Therefore if it is possible, it is always good to try both from right to left and left to right, and solving for $y(t_0)$ or for $y'(t_0)$

9.2 Finite Differences

Let us recall the Boundary Value problem we had in the Shooting method

$$\begin{cases} \theta'' + b\theta' + \omega^2 \sin(\theta) = 0 \quad \text{for } \ 0 \leq t \leq T \\ \theta(0) = \alpha \qquad\qquad\qquad\quad \theta(T) = \beta \end{cases} \tag{9.11}$$

Before, we solved this problem based on an initial value problem, using a guess for the derivate of the function θ' at zero, and then applying root finding techniques to converges to the exact solution.

But now if we just analyze the problem, we can discretize both θ'' and θ' using finite differences and replace them in (9.11). Notice that we would like to have the

same truncation error in both approximations, otherwise only one of them would be responsible of the error. Therefore let us use centered differences for the first and second derivative, and for every $\theta_i \approx \theta(t_i)$, where $t_i = t_0 + ih = iT/N$, we have

$$\begin{cases} \theta_0 = \alpha \\ \frac{\theta_{i+1} - 2\theta_i + \theta_{i-1}}{h^2} + b\frac{\theta_{i+1} - \theta_{i-1}}{2h} + \omega^2 \sin(\theta_i) = 0 \quad 1 \leq i \leq N-1 \\ \theta_N = \beta \end{cases} \tag{9.12}$$

Therefore we can rearrange this into a system of equations:

$$\begin{bmatrix} 1 & & & & & \\ \frac{1}{h^2} - \frac{b}{2h} & -\frac{2}{h^2} & \frac{1}{h^2} + \frac{b}{2h} & & & \\ & \cdots & \cdots & \cdots & & \\ & & \frac{1}{h^2} - \frac{b}{2h} & -\frac{2}{h^2} & \frac{1}{h^2} + \frac{b}{2h} & \\ & & & & 1 & \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \cdots \\ \theta_{N-1} \\ \theta_N \end{bmatrix} = \begin{bmatrix} \alpha \\ -\omega^2 \sin(\theta_1) \\ \cdots \\ -\omega^2 \sin(\theta_{N-1}) \\ \beta \end{bmatrix} \tag{9.13}$$

Since this is a nonlinear system of equations ($N+1$ equations, $N+1$ unknowns), then we solve it iteratively using Newton's method. Since it is a nonlinear equation, different initial guesses could result in different solutions.

It is important to notice that we get a nonlinear system, because the original ODE is nonlinear. Similarly, we can expect a linear system from a linear second order ODE. So in general if we have

$$\begin{cases} y'' = f(t, y, y') & t_0 \leq t \leq t_f \\ y(t_0) = \alpha \ , \ y(t_f) = \beta \end{cases} \tag{9.14}$$

To find a numerical approximation we solve the system of equations:

$$\begin{cases} \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = f\left(t_i, y_i, \frac{y_{i+1} - y_{i-1}}{2h}\right) \quad t_i = t_0 + ih \ , \ 1 \leq i \leq N-1 \\ y_0 = \alpha \ , \ y_N = \beta \end{cases} \tag{9.15}$$

What is the difference between shooting method and Finite Differences?

In the shooting method, the approach is to solve y_{i+1} only using the information at y_i, or in multistep, some other points before that one. Therefore we get the values of y_i sequentially, and at each step, the total number of points in the grid does not influence the cost per step.

On the other hand, Finite Differences gives us a general stencil for all the points in the grid, and to solve it, we should solve a system of equations involving all the points in the grid. Since the matrix of coefficients is usually banded (for example tridiagonal), There are efficient ways to solve all the points at once. Nevertheless, the cost of solving the system increases with the number of total points in the grid.

9.2.1 Other types of Boundary Conditions

Let us consider now the following problem

$$\begin{cases} \theta'' + b\theta' + \omega^2 \sin(\theta) = 0 \quad \text{for } \ 0 \leq t \leq T \\ \alpha_0\theta(0) + \beta_0\theta'(0) = \gamma_0 \\ \alpha_T\theta(T) + \beta_T\theta'(T) = \gamma_T \end{cases} \tag{9.16}$$

So how to construct a system of equations similar to (9.13)*, but involving the Robin conditions?*

One way is to approximate the Robin conditions using finite differences. We could use forward differences for $\theta'(0)$ and backward for $\theta'(T)$. But we would break the second order approximation, and the error would be concentrated in the endpoints.

Another option is to have a second order approximation for the derivative. Since t_0, and t_N are at the boundary, the only way to approximate the derivatives using the existing points is to take t_0, t_1 & t_2 and on the other hand t_{N-2}, t_{N-1} & t_N. The problem of this approach is that the tridiagonal structure of (9.13) is lost.

If t_0 and t_N were not boundary points, we could approximate the derivative using centered differences, and the tridiagonal pattern would not be lost. Therefore the idea of this approach is to include *fictitious points* t_{-1} and t_{N+1}, and approximate the Robin conditions with centered differences.

But adding 2 additional unknowns that don't exist does not look like the best answer

Well if we combine the Robin conditions with the numerical equation at θ_0 we get:

$$\alpha_0\theta_0 + \beta_0\frac{\theta_1 - \theta_{-1}}{2h} = \gamma_0 \tag{9.17}$$

$$\frac{\theta_1 - 2\theta_0 + \theta_{-1}}{h^2} + b\frac{\theta_1 - \theta_{-1}}{2h} + \omega^2 \sin(\theta_0) = 0 \tag{9.18}$$

From (9.17), if $\beta_0 \neq 0$, we have that:

$$\frac{\theta_1 - \theta_{-1}}{2h} = \frac{1}{\beta_0}(\gamma_0 - \alpha_0\theta_0)$$

$$\theta_{-1} = \theta_1 - \frac{2h}{\beta_0}(\gamma_0 - \alpha_0\theta_0)$$

Replacing in (9.18) we get:

$$\frac{\theta_1 - 2\theta_0}{h^2} + \frac{1}{h^2}\left(\theta_1 - \frac{2h}{\beta_0}\left(\gamma_0 - \alpha_0\theta_0\right)\right) + b\frac{1}{\beta_0}\left(\gamma_0 - \alpha_0\theta_0\right) + \omega^2\sin(\theta_0) = 0$$

$$\frac{2\theta_1 - 2\theta_0}{h^2} + \frac{\gamma_0 - \alpha_0\theta_0}{\beta_0}\left(b - \frac{2}{h}\right) + \omega^2\sin(\theta_0) = 0$$

Chapter 10

Solution of Partial Differential Equations

In Ordinary Differential Equations (ODEs) we had an independent variable t and a dependent variable y, which was only a function of t, and we could find the behavior of y with respect to t solving an equation that involves derivatives of y.

In contrast, in a Partial Differential Equation (PDE), the dependent variable u is a function of multiple independent variables, let us say $t,\ x\ \&\ y$, and we find the behavior of u solving an equation that involves the partial derivatives of u with respect to $t,\ x\ \&\ y$

Let us say $u = u(x, y)$, this means u is a function of x and y. Then first order semilinear PDE is given by:

$$a(x,y)\frac{\partial u}{\partial x} + b(x,y)\frac{\partial u}{\partial y} = c(x,y) \tag{10.1}$$

This equation defines the directional derivative of u with respect to the vector field $V = (a(x, y), b(x, y)$. This means that given certain initial conditions, the solution of the equation moves following characteristic curves defined by the vector field. In other words, if we know the solution in the characteristic curves, we know the solution in the whole domain.

One simple example of a first order PDE is the **advection equation**:

$$\begin{cases} \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \\ u(x,0) = \eta(x) \end{cases} \tag{10.2}$$

And the exact solution is given by

$$u(x,y) = \eta(x - at) \tag{10.3}$$

When we want to solve numerically this equation, we really need to understand how the solution should behave. For example if in (10.2), $a > 0$ it means that the characteristics are straight lines with positive slope in the t v.s. x plane. Therefore if we use a forward discretization in x, we won't capture the behavior of the characteristic curves, and, as a result, the numerical method would be unstable.

For this reason solving PDEs numerically is not only a matter of approximating the derivatives (for example using finite differences), but also being sure that the numerical approximation follows the same physical principles that the original PDE does.

A second order constant coefficient PDEs in 2D is given by:

$$a_1\frac{\partial^2 u}{\partial x^2} + a_2\frac{\partial^2 u}{\partial x \partial y} + a_3\frac{\partial^2 u}{\partial y^2} + a_4\frac{\partial u}{\partial x} + a_5\frac{\partial u}{\partial y} + a_6 u = f \tag{10.4}$$

and we can classify it in 3 types:

- **Elliptic** if $a_2^2 - 4a_1a_3 < 0$. For example **Lagrange Equation**:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{10.5}$$

  or the **Poisson Equation**

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{10.6}$$

  both related with the steady-state solution of the Heat equation, where $f(x)$ is a source term.

- **Hyperbolic** if $a_2^2 - 4a_1a_3 > 0$. For example the **Wave equation**

$$c^2\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial t^2} = 0 \tag{10.7}$$

  Since they can be factorize as 2 first order PDEs, its behavior is similar to the advection equation. This means that the solution at a point is given by the characteristic curve that passes through it, and it is not affected by the solution of the whole domain.

- **Parabolic** if $a_2^2 - 4a_1a_3 = 0$. For example the **Heat equation**

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2} \tag{10.8}$$

And the principal difference with the wave equation is that the speed of propagation is infinity, this means that any change in the function in any point of the domain affects the whole domain. Think in how is the behavior of a heated pan versus the waves generated by throwing a stone into a lake. Therefore this two phenomenons are summarize as:

- **Diffusion** It has an instantaneous smoothing effect of the initial conditions. *For example Elliptic and Parabolic equations*
- **Advection or Transport** It carries the initial conditions along the domain using characteristic curves. *Wave equation: First order PDE and Hyperbolic equations*

In a more general setting, we could have equations that combine parabolic and hyperbolic behaviors, for example the *advection-diffusion-reaction* equation:

$$\frac{\partial u}{\partial t} = \underbrace{k\frac{\partial^2 u}{\partial x^2}}_{diffusion} - \underbrace{a\frac{\partial u}{\partial x}}_{advection} + \underbrace{R(x)}_{reaction} \tag{10.9}$$

For this type of equations, we should develop specific numerical methods that can capture the effects of both systems, dealing with the stability issues of a stiff problem.

10.1 Elliptic Equations

We call Δ the Laplacian operator, and it is defined as

$$\Delta u(x) = \sum_{i=1}^{D} \frac{\partial^2 u}{\partial x_i^2} \tag{10.10}$$

where D is the dimension of the domain, and it can be 1,2 or 3, $x = (x_i)_{i=1}^{D}$. The general heat equation with a source term ψ is given by

$$\frac{\partial u}{\partial t} = k\Delta u + \psi \tag{10.11}$$

The steady-state solution for the heat equation means that u is not changing with respect to time, therefore $\frac{\partial u}{\partial t} = 0$, and at the same time the source is independent of time $\psi(x,t) = f(x)$. With this conditions we derive the Poisson equation as

$$\Delta u = f(x) \tag{10.12}$$

Notice that in 1D, (10.12) is a second order ODE $u'' = f(x)$, and we can solve it using the methods of the previous chapter. In 2D we get:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \tag{10.13}$$

Additional to (10.13), we should specify boundary conditions, and these can be Dirichlet, Newman, Robin or more general conditions. As you can assume, discretizing (10.13) using finite differences is not really complicated. The challenge now is the different type of domains we could have.

First let us consider a rectangular domain $x_0 \leq x \leq x_f$, $y_0 \leq y \leq y_f$, and we divide them in (N_x, N_y) equally spaced points, i.e. $x_i = x_0 + ih_x$ and $x_j = y_0 + jh_y$ such that $h_x = (x_f - x_0)/N_x$ and $h_y = (y_f - y_0)/N_y$.

Then we want to approximate the function at the grid points as $u(x_i, y_j) \approx u_{i,j}$. Therefore using centered finite difference to approximate the second order derivatives in (10.13), we get:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_x^2} = f(x_i, y_i) \tag{10.14}$$

If $h_x = h_y = h$ we get:

$$\frac{1}{h^2}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}\right) = f(x_i, y_i) \tag{10.15}$$

This is the 5-points stencil for the Laplacian operator.
And notice that if we solve for $u_{i,j}$ we get that

$$u_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4} - \frac{h^2}{4} f(x_i, y_i)$$

Therefore, if we do not have any forcing (i.e. $f = 0$) then each point is the average of its 4 neighbors. This proves that the numerical scheme also satisfies the Maximum Principle: let Ω be the set of all grid points, and $\partial\Omega$ the set of all boundary points then:

$$\max_{\Omega} u_{ij} = \max_{\partial\Omega} u_{ij}$$

The maximum principle is one of the most important properties of the Laplace equation, because it explains why solutions are smooth inside the domain.

How can we solve this system?

If we have $N_x - 1$ internal nodes in x and $N_y - 1$ internal nodes in y, depending on the boundary conditions we have at least $(N_x - 1) * (N_y - 1)$ unknowns, and for each internal node we have (10.14), which only involves 5 points per equation, therefore we have a sparse linear system to solve. In 1D the way to organize the equations was straightforward, we just followed the orientation of x. But now in 2D, we do not have a natural order to follow. One of the simplest ways is to enumerate the nodes row by row. That produces a nice pattern in the matrix but it is not the

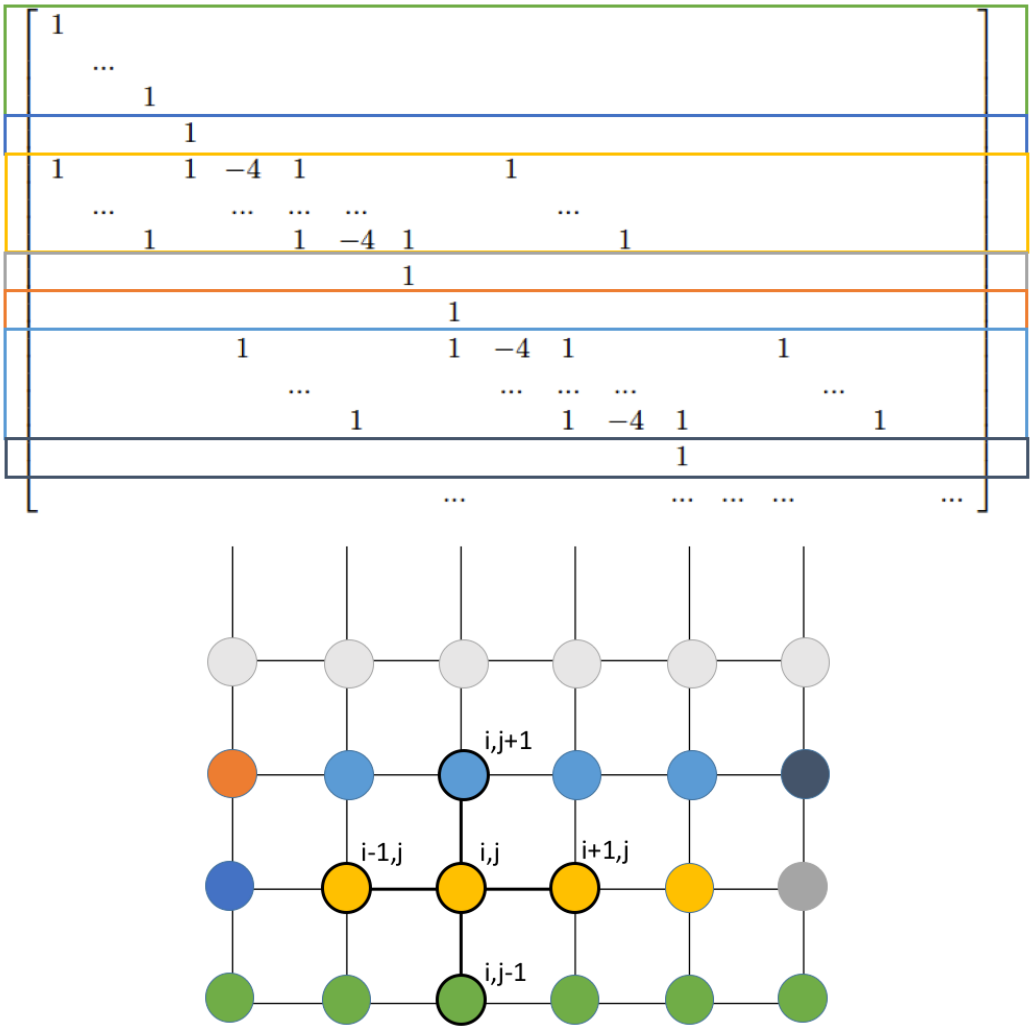

Figure 10.1: Assembly of the matrix to solve the Poisson equation in 2D with equally spaced points and Dirichlet Boundary conditions. Notice that we can remove the boundary terms from the system, just passing their values to the right hand side of the system of equations. Having other type of boundary conditions will only change the rows corresponding to the boundary nodes.

optimal choice in order to solve the system. A lot of people have worked in finding optimal configurations for this matrix.

On the other hand, dealing with different types of boundary conditions is exactly the same as in the Boundary Value problems with ODE's.

What to do in more general domains?

Depending on the domain, sometimes it would be necessary to have different spacing along the points. This is possible to do in finite differences, as far as they include that change in the expansion. For those cases, we could think in the Lagrangian polynomial that is behind the finite difference approximation.

10.2 Parabolic Equations

The canonical Parabolic equation is the heat equation. If we have a domain Ω, the heat equation is defined as:

$$\frac{\partial u}{\partial t} = D\Delta u \ for \ x \in \Omega \ \& \ t > 0 \tag{10.16}$$

And now to have a particular solution we need to specify Initial conditions in time and Boundary conditions in space.

Notice that the right hand side only involves partial derivatives with respect to space, therefore for each t we can discretize it using methods for elliptic equations. Once we do that, we will have a linear system that depends on $u(x,t)$ and its neighbors. We can call this approximation $\tilde{\Delta}_h u$.

And then (10.16) can be approximated as

$$\frac{\partial u}{\partial t} = D\tilde{\Delta}_h u(t) \ t > 0 \tag{10.17}$$

And for each x, we can this Initial Value ODE problem in time, using any of the methods derived in Chapter 8, for example Euler, Backward Euler or any other Runge-Kutta or multistep method. And of course, as we could expect from the stability analysis of ODE methods, there will be a tight relationship between the discretization in space and the discretization in time.

Let us work with the simplest case to see how this procedure works. If Ω is just one line segment, then we will be working in one dimension. The heat equation for

1D with Dirichlet Boundary conditions is

$$\begin{cases} \frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} & x_0 \leq x \leq x_f, t > 0 \\ u(x_0, t) = \alpha(t) \\ u(x_f, t) = \beta(t) \\ u(x, 0) = v(x) \end{cases} \tag{10.18}$$

To discretize in space, let us take $N+1$ equally spaced points $x_0, .., x_N = x_f$, where $x_i = x_0 + ih$ where $h = (x_f - x_0)/N$. Let $u_i(t) \approx u(x_i, t)$, and let us approximate the second derivative using centered differences. Therefore for each $1 \leq i \leq N-1$ we have that

$$\begin{cases} \frac{du_i(t)}{dt} = D\frac{u_{i+1}(t) - 2u_i(t) + u_{i-1}(t)}{h^2} & 1 \leq i \leq N-1 \\ u_0(t) = \alpha(t) \\ u_N(t) = \beta(t) \\ u_i(0) = v(x_i) \end{cases} \tag{10.19}$$

Notice that if we consider the vector $U(t) = [u_1(t), u_2(t), ..., u_{N-1}(t)]$ then (10.19) is an initial value ODE of the form

$$\begin{cases} \frac{dU(t)}{dt} = -\frac{D}{h^2}\left(AU(t) + b(t)\right) \\ U(0) = V \end{cases}$$

where

$$A = \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & ... & ... & ... & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix} \quad b(t) = \begin{bmatrix} -\alpha(t) \\ 0 \\ 0 \\ ... \\ 0 \\ -\beta(t) \end{bmatrix} \quad V = \begin{bmatrix} v(x_1) \\ v(x_2) \\ v(x_3) \\ ... \\ v(x_{N-2}) \\ v(x_{N-1}) \end{bmatrix}$$

And now we can solve (10.19) using Euler, Backward Euler or Trapezoidal method. If we define $t^{(n)} = n * k$ where k is the time-step, then we will approximate our solution as $u(x_i, t^{(n)}) = u_i^{(n)}$. Therefore applying the different time schemes we have

- **Euler⇒FTCS** *Forward in Time, Centered in Space*

$$\frac{u_i^{(n+1)} - u_i^{(n)}}{k} = D\frac{u_{i+1}^{(n)} - 2u_i^{(n)} + u_{i-1}^{(n)}}{h^2} \tag{10.20}$$

- **Backward Euler⇒BTCS** *Backward in Time, Centered in Space*

$$\frac{u_i^{(n+1)} - u_i^{(n)}}{k} = D\frac{u_{i+1}^{(n+1)} - 2u_i^{(n+1)} + u_{i-1}^{(n+1)}}{h^2} \tag{10.21}$$

- **Trapezoidal⇒Crank-Nicolson**

$$\frac{u_i^{(n+1)} - u_i^{(n)}}{k} = \frac{D}{2}\left(\frac{u_{i+1}^{(n)} - 2u_i^{(n)} + u_{i-1}^{(n)}}{h^2} + \frac{u_{i+1}^{(n+1)} - 2u_i^{(n+1)} + u_{i-1}^{(n+1)}}{h^2}\right) \tag{10.22}$$

For the three cases the initial conditions are

$$u_i^{(0)} = v(x_i) \tag{10.23}$$

And the boundary conditions are

$$u_0^{(n)} = \alpha(t^{(n)}) \qquad\qquad u_N^{(n)} = \beta(t^{(n)}) \tag{10.24}$$

What about convergence?

Notice that the truncation error of **FTCS** and **FTCS** is $\mathcal{O}(h^2 + k)$ whereas in **Crank Nicholson** is $\mathcal{O}(h^2 + k^2)$, because the trapezoidal scheme is second order accurate in time.

What about stability?

Since (10.19) is a linear system of first order ODEs, we can analyze stability of each method based on the eigenvalues of A. The nice property about A is that it is a Toeplitz symmetric tridiagonal matrix, and therefore its eigenvalues are well known. If A is $N \times N$ and the value in the main diagonal is a and in the outer diagonal is b then

$$\lambda_s = a + 2b\cos\left(\frac{s\pi}{N+1}\right) \quad s = 1, 2, 3, .., N$$

Therefore in this case, we have that

$$\lambda_s = 2 - 2\cos\left(\frac{s\pi}{N}\right) \quad s = 1, 2, 3, .., N-1 \tag{10.25}$$

We know that Backward Euler and Trapezoidal are unconditionally stable, this means that we can take any k, and the error per iteration will remain bounded. But for Euler method, we had the condition that

$$|1 + k\lambda| < 1$$

In this case, this is translated to

$$\left|1 - 2D\frac{k}{h^2}\left(1 - \cos\left(\frac{s\pi}{N}\right)\right)\right| < 1 \quad s = 1, 2, 3, .., N-1 \tag{10.26}$$

which implies that

$$k \leq \frac{h^2}{2D} \tag{10.27}$$