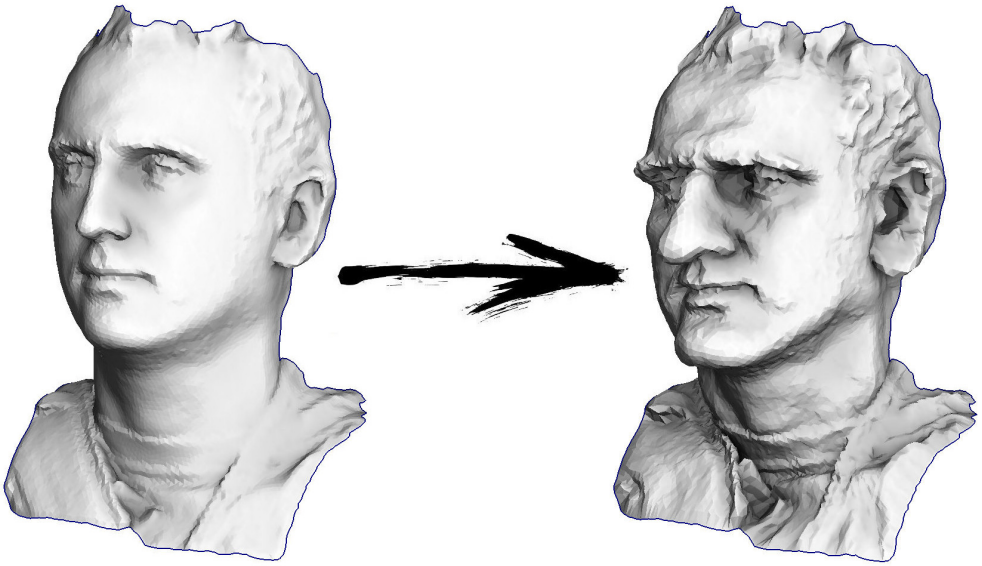
Least squares for programmers

with illustrations

Dmitry Sokolov and Nicolas Ray



Preface

This course is intended for students/engineers/researchers who know how to program in the traditional way: by breaking down complex tasks into elementary operations that manipulate combinatorial structures (trees, graphs, meshes...). Here we present a different paradigm, in which we describe what a good result looks like, and let numerical optimization algorithms find it for us.

This course explains least squares optimization, nowadays a simple and well-mastered technology. We show how this simple method can solve a large number of problems that would be difficult to approach in any other way. This course provides a simple, understandable yet powerful tool that most coders can use, in the contrast with other algorithms sharing this paradigm (numerical simulation and deep learning) which are more complex to master.

Chapter 1

Do you believe in the probability theory?

This short section is not mandatory for the understanding of the main course; the idea behind is to warm up before attacking code and formulae. I'll start with the least squares methods through the maximum likelihood estimation; this requires some (at least superficial) knowledge of probability theory. So, right from the beginning, I would like to digress a little.

I was once asked if I believed in evolutionary theory. Take a short break, think about how you would answer. Being puzzled by the question, I have answered that I find it plausible. Scientific theory has little to do with faith. In short, a theory only builds a model of the world, and there is no need to believe in it. Moreover, the Popperian criterion[] requires a scientific theory be able to be falsifiable. A solid theory must possess, first of all, the power of prediction. For example, if you genetically modify crops in such a way that they produce pesticides themselves, it is only logical that pesticide-resistant insects would appear. However, it is much less obvious that this process can be slowed down by growing regular plants side by side with genetically modified plants. Based on evolutionary theory, the corresponding modelling has made this prediction[], and it seems to have been validated[].

Wait, what is the connection? As I mentioned earlier, the idea is to approach the least squares through the principle of maximum likelihood. Let us illustrate by example. Suppose we are interested in penguins body height, but we are only able to measure a few of these magestic birds. It is reasonable to introduce the body height distribution model into the task; most often it is supposed to be normal. A normal distribution is characterized by two parameters: the average value and the standard deviation. For each fixed value of parameters, we can calculate the probability that the measurements we made would be generated. Then, by varying the parameters, we will find those that maximize the probability.

Thus, to work with maximum likelihood we need to operate in the notions of probability theory. We will informally define the concept of probability and plausibility, but I would like to focus on another aspect first. I find it surprisingly rare to see people paying attention to the word *theory* in "probability theory".

What are the origins, values and scope of probabilistic estimates? For example, Bruno de Finetti said that the probability is nothing but a subjective analysis of the probability that something will happen, and that this probability does not exist out of mind. It's a person's willingness to bet on something to happen. This opinion is directly opposed to the view of people adhering to the classical/frequentist interpretation of probabilty. They assume that the same event can be repeated many times, and the "probability" of a particular result is associated with the frequency of a particular outcome during repeated well-defined random experiment trials. In addition to subjectivists and frequentists, there are also objectivists who argue that probabilities are real aspects of the universe, and not a mere measurement of the observer's degree of confidence.

In any case, all three scientific schools in practice use the same apparatus based on Kolmogorov's axioms. Let us provide an indirect argument, from a subjectivistic point of view, in favor of the probability theory based on Kolmogorov's axioms. We will list the axioms later, first assume that we have a bookmaker who takes bets on the next World Cup. Let us have two events: $a =$ Uruguay will be the champion, $b =$ Germany wins the cup. The bookmaker estimates the chances of the Uruguayan team to win at 40%, and the chances of the German team at 30%. Clearly, both Germany and Uruguay cannot win at the same time, so the chance of $a \wedge b$ is zero. At the same time, the bookmaker thinks that the probability that either Uruguay or

Germany (and not Argentina or Australia) will win is 80%. Let's write it down in the following form:

$$P(a) = .4 \qquad P(a \wedge b) = 0 \qquad P(b) = .3 \qquad P(a \vee b) = .8$$

If the bookmaker asserts that his degree of confidence in the event a is equal to 0.4, i.e., $P(a) = 0.4$, then the player can choose whether he will bet on or against the statement a, placing amounts that are compatible with the degree of confidence of the bookmaker. It means that the player can make a bet on the event a, placing \$4 against \$6 of the bookmaker's money. Or the player can bet \$6 on the event $\neg a$ against \$4 of bookmaker's money.

If the bookmaker's confidence level does not accurately reflect the state of the world, we can expect that in the long run he will lose money to players whose beliefs are more accurate. However, it is very curious that in this particular example, the player has a winning strategy: he can make the bookmaker lose money for *any* outcome. Let us illustrate it:

Player's bets		Result for the bookmaker			
Bet event	Bet amount	$a \wedge b$	$a \wedge \neg b$	$\neg a \wedge b$	$\neg a \wedge \neg b$
a	4-6	-6	-6	4	4
b	3-7	-7	3	-7	3
$\neg(a \vee b)$	2-8	2	2	2	-8
		-11	-1	-1	-1

The player makes three bets, and independently of the outcome, he always wins. Please note that in this case we do not even take into account whether Uruguay or Germany were favorits or outsiders, the loss of the bookmaker is guaranteed! This unfortunate (for the bookmaker) situation happened because he did not respect the third axiom of Kolmogorov, let us list all three of them:

- $0 \leq P(a) \leq 1$: all probabilities range from 0 to 1.
- $P(true) = 1$, $P(false) = 0$: true statements have probability of 1 and false probability of 0.
- $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$: this one is also very intuitive. All cases where the statement a is true, together with those where b is true, cover all those cases where the statement $a \vee b$ is true; however the intersection $a \wedge b$ is counted twice in the sum, therefore it is necessary to subtract $P(a \wedge b)$.

Let us define the word "event" as "a subset of the unit square". Define the word "probability of event" as "area of the corresponding subset". Roughly speaking, we have a large dartboard, and we close our eyes and shoot at it. The chances that the dart hits a given region of the dartboard are directly proportional to the area of the region. A true event in this case is the entire square, and false events are those of zero measure, for example, any given point. Figure 1.1 illustrates the axioms.

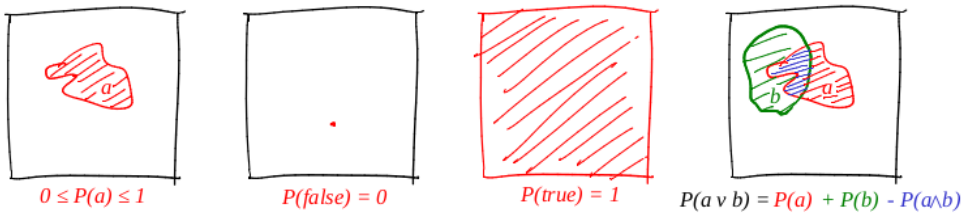


Figure 1.1: A graphical illustration for the Kolmogorov's axioms

In 1931, de Finetti proved a very strong proposition:

If a bookmaker is guided by beliefs which break the axioms of the theory of probability, then there exists such a combination of bets of the player which guarantees the loss for the bookmaker (a prize for the player) at each bet.

Probability axioms can be considered as the limiting set of probabilistic beliefs that some agent can adhere to. Note that if a bookmaker respects Kolmogorov's axioms, it does not imply that he will win (leaving aside the fees), however, if he does not respect the axioms, he is guaranteed to lose. Other arguments have been

put forward in favour of the probability theory; but it is the practical success of probability-based reasoning systems that has proved to be very attractive.

To conclude the digression, it seems reasonable to base our reasoning on the probability theory. Now let us proceed to maximum likelihood estimation, thus motivating the least squares.

Chapter 2

Maximum likelihood through examples

2.1 First example: coin toss

Let us consider a simple example of coin flipping, also known as the Bernoulli's scheme. We conduct n experiments, two events can happen in each one ("success" or "failure"): one happens with probability p, the other one with probability $1 - p$. Our goal is to find the probability of getting exactly k successes in these n experiments. This probability is given by the Bernoulli's formula:

$$P(k; n, p) = C_n^k p^k (1-p)^{n-k}$$

Let us take an ordinary coin ($p = 1/2$), flip it ten times ($n = 10$), and count how many times we get the tails:

$$P(k) = C_{10}^k \frac{1}{2^k} \left(1 - \frac{1}{2}\right)^{10-k} = \frac{C_{10}^k}{2^{10}}$$

Figure 2.1, left shows what a probability density graph looks like.

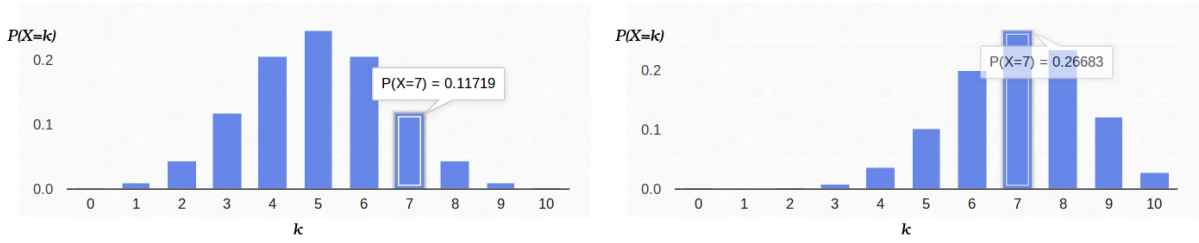


Figure 2.1: **Left:** probability density graph for the Bernoulli's scheme with $p = 1/2$. **Right:** probability density graph for the Bernoulli's scheme with $p = 7/10$.

Thus, if we have fixed the probability of "success" (1/2) and also fixed the number of experiments (10), then the possible number of "successes" can be any integer between 0 and 10, but these outcomes are not equiprobable. It is clear that five "successes" are much more likely to happen than none. For example, the probability encountering seven tails is about 12%.

Now let us look at the same problem from a different angle. Suppose we have a real coin, but we do not know its distribution of a priori probability of "success"/"failure". However, we can toss it ten times and count the number of "successes". For example, we have counted seven tails. Would it help us to evaluate p?

We can try to fix $n = 10$ and $k = 7$ in the Bernoulli's formula, leaving p as a free parameter:

$$\mathcal{L}(p) = C_{10}^7 p^7 (1-p)^3$$

Then the Bernoulli's formula can be interpreted as the plausibility of the parameter being evaluated (in this case p). I have even changed the function notation, now it is denoted as $\mathcal{L}$ (likelihood). That is being said, the likelihood is the probability to generate the observation data (7 tails out of 10 experiments) for the given value of the parameter(s). For example, the likelihood of a balanced coin ($p = 1/2$) with seven tails

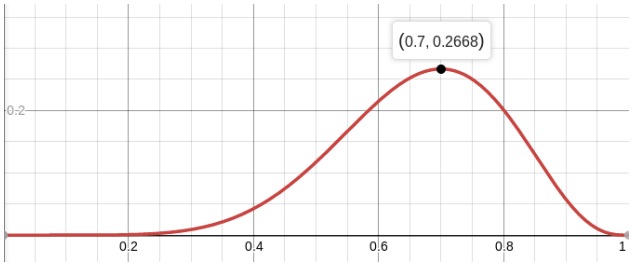


Figure 2.2: The plot of the likelihood function $\mathcal{L}(p)$ for the observation data with 7 tails out of 10 experiments.

out of ten tosses is approximately 12%. Figure 2.2 plots the likelihood function for the observation data with 7 tails out of 10 experiments.

So, we are looking for the parameter value that maximizes the likelihood of producing the observations we have. In our particular case, we have a function of one variable, and we are looking for its maximum. In order to make things easier, I will not search for the maximum of $\mathcal{L}$, but for the maximum of $\log \mathcal{L}$. The logarithm is a strictly monotonous function, so maximizing both is equivalent. The logarithm has a nice property of breaking down products into sums that are much more convenient to differentiate. So, we are looking for the maximum of this function:

$$\log \mathcal{L}(p) = \log C_{10}^7 + 7 \log p + 3 \log(1-p)$$

That's why we equate it's derivative to zero:

$$\frac{d \log \mathcal{L}}{dp} = 0$$

The derivative of $\log x = \frac{1}{x}$, therefore:

$$\frac{d \log \mathcal{L}}{dp} = \frac{7}{p} - \frac{3}{1-p} = 0$$

That is, the maximum likelihood (about 27%) is reached at the point $p = 7/10$. Just in case, let us check the second derivative:

$$\frac{d^2 \log \mathcal{L}}{dp^2} = -\frac{7}{p^2} - \frac{3}{(1-p)^2}$$

In the point $p = 7/10$ it is negative, therefore this point is indeed a maximum of the function $\mathcal{L}$:

$$\frac{d^2 \log \mathcal{L}}{dp^2}(0.7) \approx -48 < 0$$

Figure 2.1 shows the probability density graph for the Bernoulli's scheme with $p = 7/10$.

2.2 Second example: analog-to-digital converter (ADC)

Let us imagine that we have a constant physical quantity that we want to measure; for example, it can be a length to measure with a ruler or a voltage with a voltmeter. In the real world, any measurement gives *an approximation* of this value, but not the value itself. The methods I am describing here were developed by Gauß at the end of the 18th century, when he measured the orbits of celestial bodies [1]. [?]

For example, if we measure the battery voltage N times, we get N different measurements. Which of them should we take? All of them! So, let us say that we have N measurements U_j:

$$\{U_j\}_{j=1}^N$$

[1] Note that Legendre has published an equivalent method in 1805, whereas Gauß' first publication is dated by 1809. Gauß has always claimed that he had been using the method since 1795, and this is a very famous priority dispute [?] in the history of statistics. There are, however, numerous evidence to support the thesis that Gauß possessed the method before Legendre, but he was late in his communication.

Let us suppose that each measurement U_j is equal to the real value plus the Gaussian noise. The noise is characterized by two parameters — the center of the Gaussian bell and its "width". In this case, the probability density can be expressed as follows:

$$p(U_j) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(U_j - U)^2}{2\sigma^2}\right)$$

That is, having N measurements U_j, our goal is to find the parameters U and σ that maximize the likelihood. The likelihood (I have already applied the logarithm) can be written as follows:

$$\begin{aligned}
\log \mathcal{L}(U, \sigma) &= \log\left(\prod_{j=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(U_j - U)^2}{2\sigma^2}\right)\right) = \\
&= \sum_{j=1}^{N} \log\left(\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(U_j - U)^2}{2\sigma^2}\right)\right) = \\
&= \sum_{j=1}^{N} \left(\log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{(U_j - U)^2}{2\sigma^2}\right) = \\
&= -N\left(\log\sqrt{2\pi} + \log\sigma\right) - \frac{1}{2\sigma^2}\sum_{j=1}^{N}(U_j - U)^2
\end{aligned}$$

And then everything is strictly as it used to be, we equate the partial derivatives to zero:

$$\frac{\partial \log \mathcal{L}}{\partial U} = \frac{1}{\sigma^2}\sum_{j=1}^{N}(U_j - U) = 0$$

The most plausible estimation of the unknown value U is the simple average of all measurements:

$$U = \frac{\sum_{j=1}^{N} U_j}{N}$$

And the most plausible estimation of σ turns out to be the standard deviation:

$$\frac{\partial \log \mathcal{L}}{\partial \sigma} = -\frac{N}{\sigma} + \frac{1}{\sigma^3}\sum_{j=1}^{N}(U_j - U)^2 = 0$$

$$\sigma = \sqrt{\frac{\sum_{j=1}^{N}(U_j - U)^2}{N}}$$

Such a convoluted way to obtain a simple average of all measurements... In my humble opinion, the result is worth the effort. By the way, averaging multiple measurements of a constant value in order to increase the accuracy of measurements is quite a standard practice. For example, ADC averaging. Note that the hypothesis of Gaussian noise is not necessary in this case, it is enough to have an unbiased noise.

2.3 Third exampe, still 1D

Let us re-consider the previous example with a small modification. Let us say that we want to measure the resistance of a resistor. We have a bench top power supply with current regulation. That is, we control the current flowing through the resistance and we can measure the voltage required for this current. So, our "ohmmeter" evaluates the resistance through N meausrements U_j for each reference current I_j:

$$\{I_j, U_j\}_{j=1}^{N}$$

If we draw these points on a chart (Figure 2.3), the Ohm's law tells us that we are looking for the slope of the blue line that approximates the measurements.

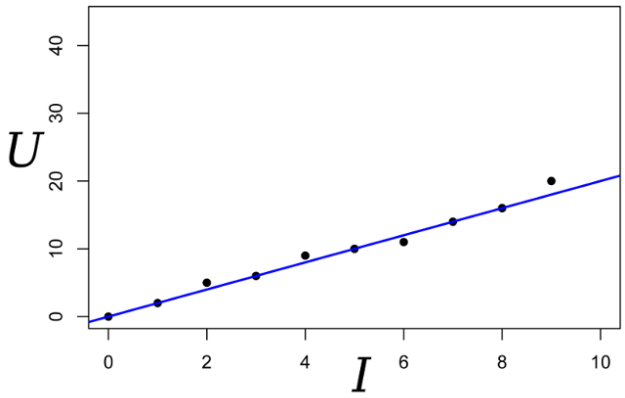


Figure 2.3: Having N meausrements U_j for each reference current I_j, we are looking for the slope of the blue line that approximates the measurements through the Ohm's law.

Let us write the expression of the (logarithm of) likelihood of the parameters:

$$\log \mathcal{L}(R, \sigma) = \log \left(\prod_{j=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(U_j - RI_j)^2}{2\sigma^2} \right) \right) =$$

$$= -N \left(\log \sqrt{2\pi} + \log \sigma \right) - \frac{1}{2\sigma^2} \sum_{j=1}^{N} (U_j - RI_j)^2$$

As usual, we equate the partial derivatives to zero:

$$\frac{\partial \log \mathcal{L}}{\partial R} = -\frac{1}{2\sigma^2} \sum_{j=1}^{N} -2I_j(U_j - RI_j) =$$

$$= \frac{1}{\sigma^2} \left(\sum_{j=1}^{N} I_j U_j - R \sum_{j=1}^{N} I_j^2 \right) = 0$$

Then the most plausible resistance R can be found with the following formula:

$$R = \frac{\sum\limits_{j=1}^{N} I_j U_j}{\sum\limits_{j=1}^{N} I_j^2}$$

This result is somewhat less obvious than the simple average of all measurements in the previous example. Note that if we take one hundred measurements with $\approx 1A$ reference current and one measurement with $\approx 1kA$ reference current, then the first hundred measurements would barely affect the result. Let's remember this fact, we will need it later.

2.4 Fourth example: back to the least squares

You have probably already noticed that in the last two examples, maximizing the logarithm of the likelihood is equivalent to minimizing the sum of squared estimation errors. Let us consider one more example. Say we want to calibrate a spring scale with a help of reference weights. Suppose we have N reference weights of mass x_j; we weigh them with the scale and measure the length of the spring. So, we have N spring lengths y_j:

$$\{x_j, y_j\}_{j=1}^{N}$$

Hooke's law tells us that spring stretches linearly on the force applied; this force includes the reference weight and the weight of the spring itself. Let us denote the spring stiffness as a, and the spring length

streched under under its own weight as b. Then we can express the plausibility of our measurements (still under the Gaussian measurement noise hypothesis) in this way:

$$\log \mathcal{L}(a,b,\sigma) = \log\left(\prod_{j=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_j - ax_j - b)^2}{2\sigma^2}\right)\right) =$$

$$= -N\left(\log\sqrt{2\pi} + \log\sigma\right) - \frac{1}{2\sigma^2}\sum_{j=1}^{N}(y_j - ax_j - b)^2$$

Maximizing the likelihood of $\mathcal{L}$ is equivalent to minimizing the sum of the squared estimation error, i.e., we are looking for the minimum of the function S defined as follows:

$$S(a,b) = \sum_{j=1}^{N}(y_j - ax_j - b)^2$$

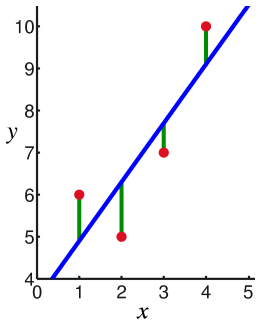


Figure 2.4: To calibrate the spring scale, we can solve the linear regression problem.

Figure 2.4 illustrates the formula: we are looking for such a straight line that minimizes the sum of squared lengths of green segments. And then the derivation is quite straightforward:

$$\frac{\partial S}{\partial a} = \sum_{j=1}^{N} 2x_j(ax_j + b - y_j) = 0$$

$$\frac{\partial S}{\partial b} = \sum_{j=1}^{N} 2(ax_j + b - y_j) = 0$$

We obtain a system of two linear equations with two unknowns:

$$\begin{cases} a\sum\limits_{j=1}^{N} x_j^2 + b\sum\limits_{j=1}^{N} x_j &= \sum\limits_{j=1}^{N} x_j y_j \\ a\sum\limits_{j=1}^{N} x_j + bN &= \sum\limits_{j=1}^{N} y_j \end{cases}$$

Use your favorite method to obtain the following solution:

$$a = \frac{N\sum\limits_{j=1}^{N} x_j y_j - \sum\limits_{j=1}^{N} x_j \sum\limits_{j=1}^{N} y_j}{N\sum\limits_{j=1}^{N} x_j^2 - \left(\sum\limits_{j=1}^{N} x_j\right)^2}$$

$$b = \frac{1}{N}\left(\sum_{j=1}^{N} y_j - a\sum_{j=1}^{N} x_j\right)$$

Conclusion

The least squares method is a particular case of maximizing likelihood in cases where the probability density is Gaussian. If the density is not Gaussian, the least squares approach can produce an estimate different from the MLE (maximum likelihood estimation). By the way, Gauß conjectured that the type of noise is of no importance, and the only thing that matters is the independence of trials.

As you have already noticed, the more we parameters we have, the more cumbersome the analytical solutions are. Fortunately, we are not living in XVIII century anymore, we have computers! Next we will try to build a geometric intuition on least squares, and see how can least squares problems be efficiently implemented.

Chapter 3

Introduction to systems of linear equations

3.1 Smooth an array

The time has come to write some code. Let us examine the following program:

```
# initialize the data array
x = [0, .8, 1, .6, .1, .4, .2, .1, .6, .3, 1, .7, .4, 0, .6, 1]

# smooth the data
for iter in range(512):
    x = [ x[0] ] + [ (x[i-1]+x[i+1])/2. for i in range(1, len(x)-1) ] + [ x[-1] ]
```

We start from a 16 elements array, and we iterate over and over a simple procedure: we replace each element with a barycenter of its neighbours; the first and the last element are fixed. What should we get at the end? Does it converge or would it oscillate infinitely? Intuitively, each peak in the signal is cut out, and therefore the array will be smoothed over time. Top left image of the Figure 3.1 shows the initialization of the array `x`, other images show the evolution of the data.

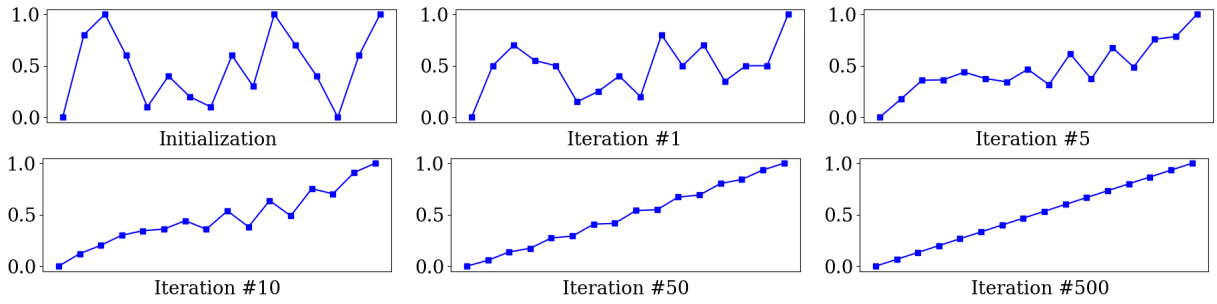


Figure 3.1: Smoothing an array: first 500 iterations of the program from § 3.1.

Is there a way to predict the result without guessing or executing the program? The answer is yes; but first let us recall how to solve systems of linear equations.

3.2 The Jacobi and Gauß-Seidel iterative methods

Let us suppose that we have an ordinary system of linear equations:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

It can be rewritten by leaving x_i only on the left side of the equations:

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n) \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n) \\ &\vdots \\ x_n &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1}) \end{aligned}$$

Suppose that we have an arbitrary vector $\vec{x}^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\right)$, approximating the solution (an initial guess, for example, a zero vector). Then, if we plug it into the right side of the equations, we can compute an updated approximated solution $\vec{x}^{(1)} = \left(x_1^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)}\right)$. In other words, $\vec{x}^{(1)}$ is derived from $\vec{x}^{(0)}$ as follows:

$$\begin{aligned} x_1^{(1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - \cdots - a_{1n}x_n^{(0)}) \\ x_2^{(1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - \cdots - a_{2n}x_n^{(0)}) \\ &\vdots \\ x_n^{(1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(0)} - a_{n2}x_2^{(0)} - \cdots - a_{n,n-1}x_{n-1}^{(0)}) \end{aligned}$$

Repeating the process k times, the solution can be approximated by the vector $\vec{x}^{(k)} = \left(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}\right)$. Let us write down the recursive formula just in case:

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k-1)}\right), \quad \text{for } i = 1, 2, \ldots, n$$

Under some assumptions about the system (for example, it is quite obvious that diagonal elements must not be zero), this procedure converges to the true solution. This iteration is known as the Jacobi method. Of course, there are other much more powerful numeric methods, for example, the conjugate gradient method, but the Jacobi method is the simplest one.

What is the connection with the code from §3.1? It turns out that the program solves the following system with the Jacobi method:

$$\begin{cases} x_0 &= 0 \\ x_1 - x_0 &= x_2 - x_1 \\ x_2 - x_1 &= x_3 - x_1 \\ &\vdots \\ x_{13} - x_{12} &= x_{14} - x_{13} \\ x_{14} - x_{13} &= x_{15} - x_{14} \\ x_{15} &= 1 \end{cases} \tag{3.1}$$

Do not take my word for it, grab a pencil and verify it! So, if we consider the array as a sampled function, the linear system prescribes a constant derivative and fixes the extremities, therefore the result can only be a straight line.

There is a very interesting modification of the Jacobi method, named after Johann Carl Friedrich Gauß and Philipp Ludwig von Seidel. This modification is even easier to implement than the Jacobi method, and it often requires fewer iterations to produce the same degree of accuracy. With the Jacobi method, the values of obtained in the k-th approximation remain unchanged until the entire k-th approximation has been calculated. With the Gauß-Seidel method, on the other hand, we use the new values of each as soon as they are known. The recursive formula can be written as follows:

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)}\right), \quad \text{for } i = 1, 2, \ldots, n$$

It allows to perform all the computations in place. The following program solves the same equation (3.1) by the Gauß-Seidel method:

```
x = [0, .8, 1, .6, .1, .4, .2, .1, .6, .3, 1, .7, .4, 0, .6, 1]

for iter in range(512):
    for i in range(1, len(x)-1):
        x[i] = ( x[i-1] + x[i+1] )/2.
```

Figure 3.2 shows the behaviour of this program.

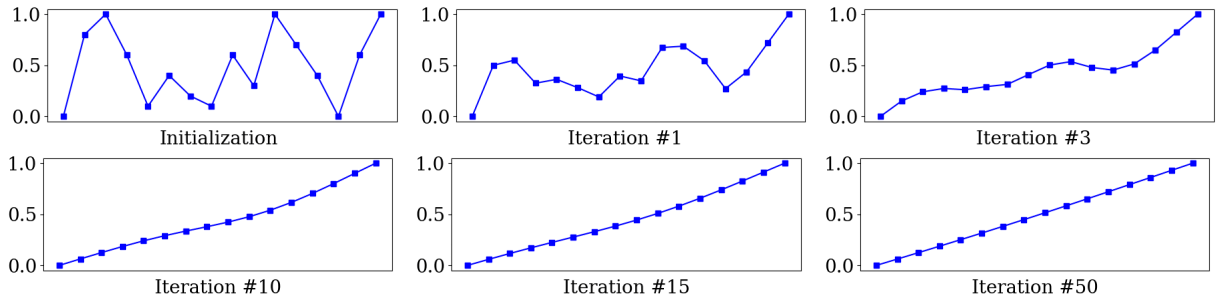


Figure 3.2: Linear function via Gauß-Seidel iteration (§3.2).

3.3 Smoothing a 3D surface

There is no much fun in studying small 1D arrays, let us play with a triangulated surface! The following listing is a direct equivalent of the previous one:

```
#include "model.h"

int main(void) {
    Model m("../input.obj"); // parse the input mesh

    // smooth the surface through Gauss-Seidel iterations
    for (int it=0; it<1000; it++) {
        for (int v=0; v<m.nverts(); v++) { // for all vertices
            if (m.is_boundary_vert(v)) continue; // fix the boundary
            m.point(v) = m.one_ring_barycenter(v);
        }
    }

    std::cout << m; // drop the result
    return 0;
}
```

All boundary vertices are fixed, and all the interiour vertices are iteratively placed in the barycenter of their immediate neighbors. Can you predict the result?

First of all, let us see that the system (3.1) can be rewritten as follows:

$$\begin{cases} x_0 & & & & & & & & & & & & = 0 \\ -x_0 & +2x_1 & -x_2 & & & & & & & & & & = 0 \\ & -x_1 & +2x_2 & -x_3 & & & & & & & & & = 0 \\ & & -x_2 & +2x_3 & -x_4 & & & & & & & & = 0 \\ & & & & & \ddots & & & & & & & \vdots \\ & & & & & & -x_{11} & +2x_{12} & -x_{13} & & & & = 0 \\ & & & & & & & -x_{12} & +2x_{13} & -x_{14} & & & = 0 \\ & & & & & & & & -x_{13} & +2x_{14} & -x_{15} & & = 0 \\ & & & & & & & & & & x_{15} & & = 1 \end{cases} \tag{3.2}$$

You can see the pattern for the second-order finite difference. Computing a constant derivative function is equivalent to computing a function with zero second derivative, it is only logic that the result is a straight line.

It is all the same for our 3D surface, the above code solves the Laplace's equation $\Delta f = 0$ with Dirichlet boundary conditions. Again, do not trust me, grab a pencil and write down the corresponding matrix for a mesh with one interior vertex. So, the result must be the minimal surface respecting the boundary conditions. In other words, make a loop from a rigid wire, soak it in a liqud soap, the soap film on this loop is the solution. Figure 3.3 provides an illutration.

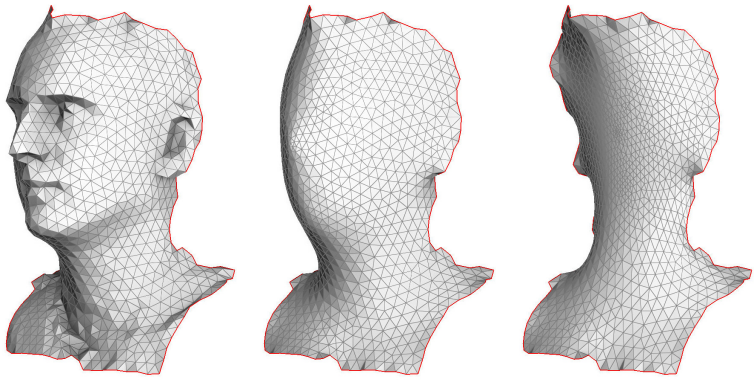


Figure 3.3: Smoothing a 3D surface (§3.3). **Left:** the input surface; **middle:** 10 iterations; **right:** 1000 iterations.

3.4 Prescribe the right hand side

Let us return to the equation (3.2), but this time the right hand side would not be zero:

```
x = [0, .8, 1, .6, .1, .4, .2, .1, .6, .3, 1, .7, .4, 0, .6, 1]

for iter in range(512):
    for i in range(1, len(x)-1):
        x[i] = ( x[i-1] + x[i+1] - (i+15)/15**3 )/2.
```

What is the result of this program? Well, it is easy to answer. We are looking for a function whose second derivative is a linear function, therefore the solution must be a cubic polynomial. Figure 3.4 shows the evolution of the array, the ground truth polynomial is shown in green.

Congratulations, you have just solved a Poisson's equation!

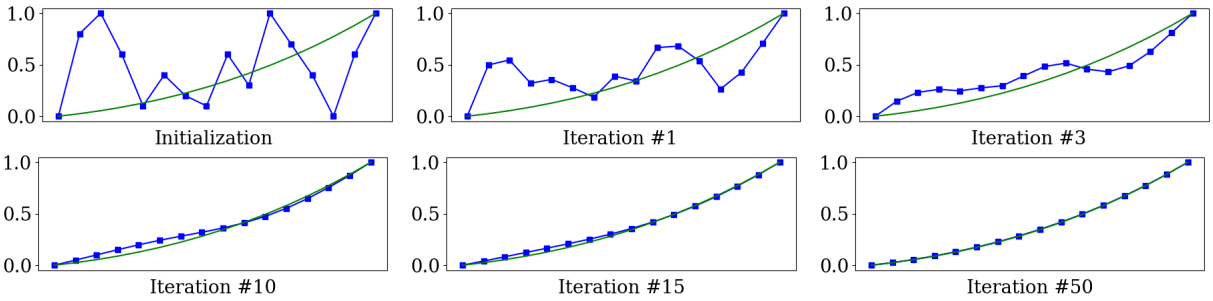


Figure 3.4: Reconstructing a cubic function (§3.4); the ground truth is shown in green.

Chapter 4

Finite elements example: the Galerkin weighted residual method

In the previous section we saw that mere 3 lines of code can be sufficient to solve a linear system, and this linear system corresponds to a differential equation. While it is extremely cool, what are the practical consequences for a programmer? How do we build these systems? Where do we use them? Well, there is a huge community of people who start with a continuous problem, then the problem is carefully discretized and eventually reduced to a linear system. In this chapter I show a tiny bit of the bottomless pit called finite element methods. Namely, I show a mathematician's approach to the program we saw in §3.4.

Note that this chapter is marked as optional. It is here to make connections to adjacent domains; feel free to skip it, the core text about least squares continues in the next chapter.

4.1 Approximation of vectors

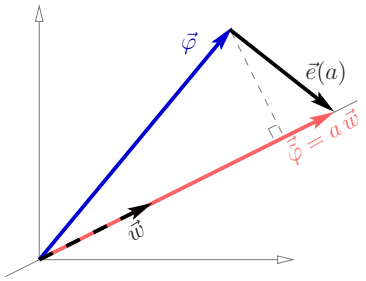


Figure 4.1: .

4.2 Approximation of functions

In this section we will consider a simple boundary value problem and its relation to systems of linear equations. The boundary value problem is the problem of finding a solution to a given differential equation satisfying boundary conditions at the the boundary of a region. Let us start with a ground truth function defined as

$$\varphi(x) := \frac{1}{6}x^3 + \frac{1}{2}x^2 + \frac{1}{3}x.$$

This function is unknown, and we list it here to compare our solution to the ground truth. Let us define a function $f(x) := x + 1$. It is simply the second derivative of $\varphi(x)$, but remember that $\varphi(x)$ is not known! So, the problem is to find a function $\varphi(x)$ with prescribed second derivative and constrained at the limits of

a region. One possible instance of the boundary value problem can be written as follows:

$$\begin{cases} \dfrac{d^2}{dx^2}\varphi(x) = f(x), & 0 < x < 1 \\ \varphi(0) = 0, \quad \varphi(1) = 1 \end{cases}$$

Here $f(x)$ is known, $\varphi(x)$ is unknown, but we have specified its values at the boundary.

In general, finite elements method searches for an *approximated* solution of the problem. Let us split the interval in n parts, these subsegments are the *elements*. For example, for $n = 3$ we can define four equispaced nodes x_i:

$$x_0 := 0, \quad x_1 := \frac{1}{3}, \quad x_2 := \frac{2}{3}, \quad x_3 := 1.$$

Note that equal distance is chosen here for the simplicity of presentation, and is not a requirement for the method. Then a set of functions is to be defined over the elements; the approximated solution $\tilde{\varphi}$ is defined as a weighted sum of these functions.

For example, let us choose a set of piecewise linear functions (refer to Figure 4.2 for an illustration):

$$w_i(x) := \begin{cases} \dfrac{x - x_{i-1}}{x_i - x_{i-1}}, & x_{i-1} \le x \le x_i \\ \dfrac{x_{i+1} - x}{x_{i+1} - x_i}, & x_i \le x \le x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

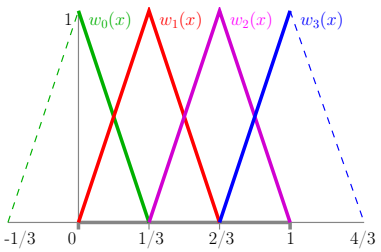


Figure 4.2: Hat function basis for FEM approximation.

This choice is somewhat arbitrary, there are numerous possible bases. Then the approximate solution is a linear combination of the weighting functions:

$$\tilde{\varphi}(x) := 0 \cdot w_0(x) + b \cdot w_1(x) + c \cdot w_2(x) + 1 \cdot w_3(x),$$

where the coefficients of $w_0(x)$ and $w_3(x)$ are the direct consequence of the boundary condition $\varphi(0) = 0$, $\varphi(1) = 1$; note that b and c give the values of the approximation for the points $x = \frac{1}{3}$ and $x = \frac{2}{3}$, respectively.

With a differential equation we do not know how to measure the true error, so we must instead require the residual $\frac{d^2\tilde{\varphi}}{dx^2} - f$ to be orthogonal to $\operatorname{span}\{w_1, \dots w_{n-1}\}$.

TODO : *explanation, figure, residual orthogonal to the solution space*

The residual being orthogonal to the solution space can be written as follows:

$$\int_0^1 w_i \cdot \left(\frac{d^2\tilde{\varphi}}{dx^2} - f \right) dx = 0, \quad i = 1 \dots n-1$$

Note that $w_i(x)$ is equal to zero outside the interval $[x_{i-1}, x_{i+1}]$, this allows us to rewrite the system:

$$\int_{x_{i-1}}^{x_{i+1}} w_i \cdot \frac{d^2\tilde{\varphi}}{dx^2}\, dx - \int_{x_{i-1}}^{x_{i+1}} w_i \cdot f\, dx = 0, \quad i = 1 \dots n-1$$

Most often both integrals are evaluated numerically, or symbolic calculations are peformed over the left integral only. The left integral is a dot product between the basis functions and the differential operator

defined on a combination of basis functions; it depends on the choice of the basis and can be precomputed. As our problem is very simple, we will find the solution analytically.

Attention! The next step will be done on a slippery ground. Our weighting functions are not differentiable everywhere, and caution must be taken. Anyhow, let us integrate by parts the first integral:

$$\int_{x_{i-1}}^{x_{i+1}} w_i \cdot \frac{d^2\tilde{\varphi}}{dx^2}\,dx = \int_{x_{i-1}}^{x_i} w_i \cdot \frac{d^2\tilde{\varphi}}{dx^2}\,dx + \int_{x_i}^{x_{i+1}} w_i \cdot \frac{d^2\tilde{\varphi}}{dx^2}\,dx =$$

$$= \underbrace{\lim_{\varepsilon\to 0} w_i \cdot \frac{d\tilde{\varphi}}{dx}\Big|_{x_{i-1}+\varepsilon}^{x_i-\varepsilon} + \lim_{\varepsilon\to 0} w_i \cdot \frac{d\tilde{\varphi}}{dx}\Big|_{x_i+\varepsilon}^{x_{i+1}-\varepsilon}}_{=0} - \int_{x_{i-1}}^{x_{i+1}} \frac{dw_i}{dx} \cdot \frac{d\tilde{\varphi}}{dx}\,dx =$$

$$= -\int_{x_{i-1}}^{x_{i+1}} \frac{dw_i}{dx} \cdot \frac{d\tilde{\varphi}}{dx}\,dx$$

It allows us to rewrite the system of equations:

$$\int_{x_{i-1}}^{x_{i+1}} \frac{dw_i}{dx} \cdot \frac{d\tilde{\varphi}}{dx}\,dx + \int_{x_{i-1}}^{x_{i+1}} w_i \cdot f\,dx = 0, \quad i = 1 \dots n-1 \tag{4.1}$$

$$\frac{d\tilde{\varphi}}{dx}(x) := \begin{cases} 3b, & 0 < x < \frac{1}{3} \\ -3b + 3c, & \frac{1}{3} < x < \frac{2}{3} \\ -3c + 3, & \frac{2}{3} < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{dw_1}{dx}(x) := \begin{cases} 3, & 0 < x < \frac{1}{3} \\ -3, & \frac{1}{3} < x < \frac{2}{3} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{dw_2}{dx}(x) := \begin{cases} 3, & \frac{1}{3} < x < \frac{2}{3} \\ -3, & \frac{2}{3} < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

Then the system (4.1) can be rewritten as follows:

$$\begin{cases} \int_0^{1/3} 3 \cdot 3b\,dx + \int_{1/3}^{2/3} -3 \cdot (-3b + 3c)\,dx + \int_0^{1/3} 3x(x+1)\,dx + \int_{1/3}^{2/3} (2-3x)(x+1)\,dx = 0 \\ \int_{1/3}^{2/3} 3 \cdot (-3b + 3c)\,dx + \int_{2/3}^{1} -3 \cdot (-3c + 3)\,dx + \int_{1/3}^{2/3} (3x-1)(x+1)\,dx + \int_{2/3}^{1} (3-3x)(x+1)\,dx = 0 \end{cases}$$

$$\begin{cases} 2b - c = -\frac{4}{27} \\ -b + 2c = \frac{22}{27} \end{cases}$$

$b = \frac{14}{81}$, $c = \frac{40}{81}$

TODO : *Mention Ritz and guide towards least squares*

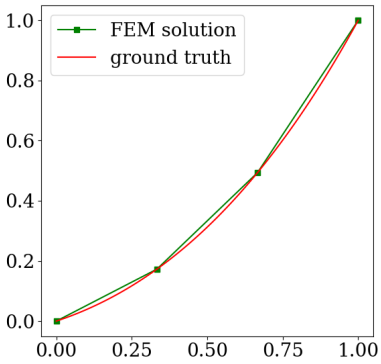


Figure 4.3: The ground truth $\varphi(x) = \frac{1}{6}x^3 + \frac{1}{2}x^2 + \frac{1}{3}x$ and its approximation found by the FEM.

Chapter 5

Minimization of quadratic functions and linear systems

Recall that the main goal is to study least squares, therefore our main tool will be the minimization of quadratic functions; however, before we start using this power tool, we need to find where its on/off button is located. First of all, we need to recall what a matrix is; then we will revisit the definition of a positive numbers, and only then we will attack minimization of quadratic functions.

5.1 Matrices and numbers

In this sections, matrices will be omnipresent, so let's remember what it is. Do not peek further down the text, pause for a few seconds, and try to formulate what the matrix is.

5.1.1 Different interpretations of matrices

The answer is very simple. A matrix is just a locker that stores stuff. Each piece of stuff lies in its own cell, cells are grouped in rows and columns. In our particular case, we store real numbers; for a programmer the easiest way to imagine a matrix A is something like:

```
float A[m][n];
```

Why would we need a storage like this? What does it describe? Maybe I will upset you, but the matrix by itself does not describe anything, it stores stuff. For example, you can store coefficients of a function in it. Let us put aside matrices for a second imagine that we have a number a. What does it mean? Who knows what it means... For example, it can be a coefficient inside a function that takes one number as an input and gives another number as an output:

$$f(x) : \mathbb{R} \to \mathbb{R}$$

One possible instance of such a function a mathematicion could write down as:

$$f(x) = ax$$

In the programmers' world it would look something like this:

```
1 float f(float x) {
2     return a*x;
3 }
```

On the other hand, why this function and not another one? Let's take another one!

$$f(x) = ax^2$$

A programmer would write it like this:

```
float f(float x) {
    return x*a*x;
}
```

One of these functions is linear and the other is quadratic. Which one is correct? Neither one. The number a does not define it, it just stores a value! Build the function you need.

The same thing happens to matrices, they give storage space when simple numbers (scalars) do not suffice, a matrix is a sort of an hyper-number. The addition and multiplication operations are defined over matrices just as over numbers.

Let us suppose that we have a 2×2 matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

The matrix does not mean anything by itself, for example, it can be interpreted as a linear function:

$$f(x) : \mathbb{R}^2 \to \mathbb{R}^2, \quad f(x) = Ax$$

Here goes the programmer's view on the function:

```
vector<float> f(vector<float> x) {
    return vector<float>{a11*x[0] + a12*x[1],  a21*x[0] + a22*x[1]};
}
```

This function maps a two-dimensional vector to a two-dimensional vector. Graphically, it is convenient to imagine it as an image transformation: we give an input image, and the output is the stretched and/or rotated (maybe even mirrored!) version. The top row of Figure 5.1 provides few different examples of this interpretation of matrices.

On the other hand, nothing prevents to interpret the matrix A as a function that maps a vector to a scalar:

$$f(x) : \mathbb{R}^2 \to \mathbb{R}, \quad f(x) = x^\top Ax = \sum_i \sum_j a_{ij} x_i x_j$$

Note that the square is not very well defined for the vectors, so I cannot write x^2 as I wrote in the case of ordinary numbers. For those who are not at ease with matrix multiplications, I highly recommend to revisit it right now and check that the expression $x^\top Ax$ indeed produces a scalar value. To this end, we can explicitly put brackets $x^\top Ax = (x^\top A)x$. Recall that in this particular example x is a two-dimensional vector (stored in a 2×1 matrix). Let us write all the matrix dimensions explicitly:

$$\underbrace{\left(\underbrace{x^\top}_{1\times 2} \times \underbrace{A}_{2\times 2}\right)}_{1\times 2} \times \underbrace{x}_{2\times 1}$$
$$\underbrace{\qquad\qquad\qquad}_{1\times 1}$$

Returning to the cozy world of programmers, we can write the same quadratic function as follows:

```
float f(vector<float> x) {
    return x[0]*a11*x[0] + x[0]*a12*x[1] + x[1]*a21*x[0] + x[1]*a22*x[1];
}
```

5.1.2 What is a positive number?

Allow me to ask a very stupid question: what is a positive number? We have a great tool called the predicate "greater than" $>$. Do not be in a hurry to answer that the number a is positive if and only if $a > 0$, it would be too easy. Let us define the positivity as follows:

Definition 1. *The real number a is positive if and only if for all non-zero real $x \in \mathbb{R}$, $x \neq 0$ the condition $ax^2 > 0$ is satisfied.*

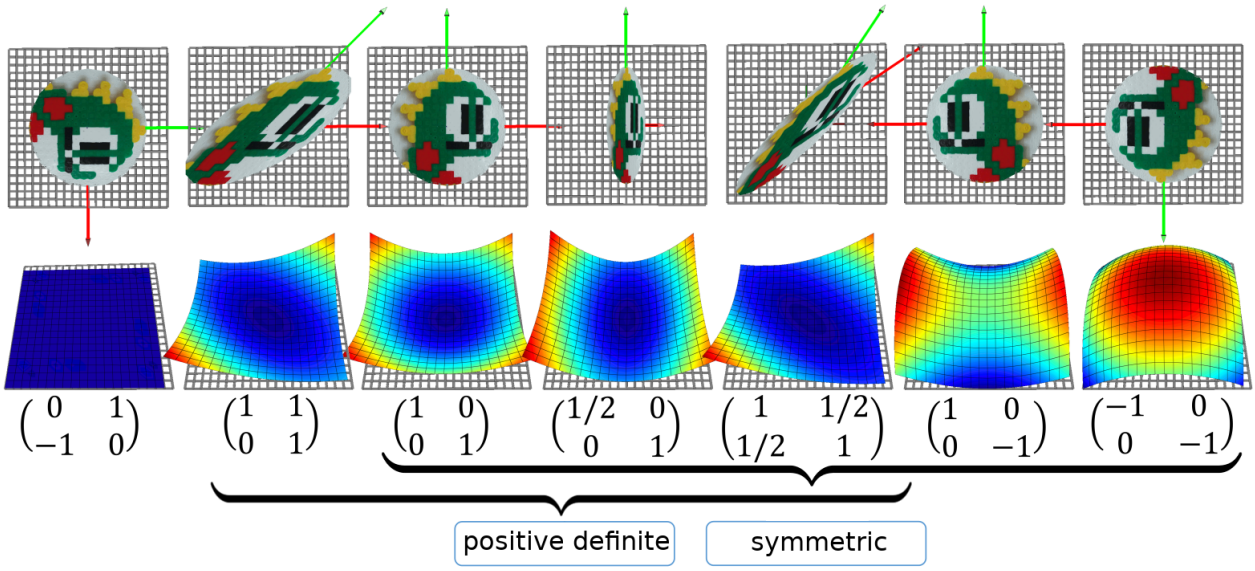


Figure 5.1: Seven examples of 2×2 matrices, some of them are positive definite and/or symmetric. **Top row:** the matrices are interpreted as linear functions $f(x) : \mathbb{R}^2 \to \mathbb{R}^2$. **Middle row:** the matrices are interpreted as quadratic functions $f(x) : \mathbb{R}^2 \to \mathbb{R}$.

Looks pretty awkward, but it applies perfectly to the matrices:

Definition 2. *The square matrix A is called positive definite if for any non-zero x the condition $x^\top Ax > 0$ is met, i.e. the corresponding quadratic form is strictly positive everywhere except at the origin.*

What do we need the positivity for? As we have already mentioned, our main tool will be the minimization of quadratic functions. It would be nice to be sure that the minimum exists! For example, the function $f(x) = -x^2$ clearly has no minimum, because the number -1 is not positive, both branches of the parabola $f(x)$ look down. Positive definite matrices guarantee that the corresponding quadratic forms form a paraboloid with a (unique) minimum. Refer to the Figure 5.1 for an illustration.

Thus, we will work with a generalization of positive numbers, namely, positive definite matrices. Moreover, in our particular case, the matrices will be symmetric! Note that quite often, when people talk about positive definiteness, they also imply symmetry. This can be partly explained by the following observation (optional for the understanding of the rest of the text):

A digression on quadratic forms and matrix symmetry

Let us consider a quadratic form $x^\top Mx$ for an arbitrary matrix M. Next we add and subtract a half ot its transpose:

$$M = \underbrace{\frac{1}{2}(M + M^\top)}_{:=M_s} + \underbrace{\frac{1}{2}(M - M^\top)}_{:=M_a} = M_s + M_a$$

The matrix M_s is symmetric: $M_s^\top = M_s$; the matrix M_a is antisymmetric: $M_a^\top = -M_a$. A remarkable fact is that for any antisymmetric matrix the corresponding quadratic form is equal to zero everywhere. This follows from the following observation:

$$q = x^\top M_a x = (x^\top M_a^\top x)^\top = -(x^\top M_a x)^\top = -q$$

It means that the quadratic form $x^\top M_a x$ equals q and $-q$ at the same time, and the only way to have this condition is to have $q \equiv 0$. From this fact it follows that for an arbitrary matrix M the corresponding quadratic form $x^\top Mx$ can be expressed through the symmetric matrix M_s as well:

$$x^\top Mx = x^\top (M_s + M_a)x = x^\top M_s x + x^\top M_a x = x^\top M_s x.$$

5.2 Minimizing a quadratic function

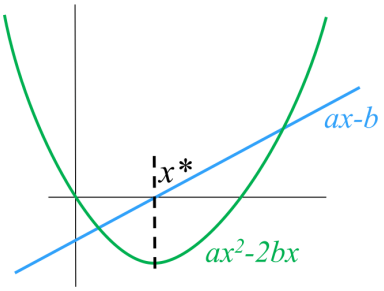


Figure 5.2: In 1d, the solution x^* of the equation $ax - b = 0$ solves the minimization problem $\arg\min_x(ax^2 - 2bx)$ as well.

Let us return to the one-dimensional world for a while; I want to find the minimum of the function $f(x) = ax^2 - 2bx$. The number a is positive, therefore the minimum exists; to find it, we equate with the corresponding derivative with zero: $\frac{d}{dx}f(x) = 0$. It is easy to differentiate a one-dimensional quadratic function: $\frac{d}{dx}f(x) = 2ax - 2b = 0$; so our problem boils down to the equation $ax - b = 0$. With some effort we can find the solution $x^* = b/a$. Figure 5.2 illustrates the equivalence of two problems: the solution x^* of the equation $ax - b = 0$ coincides with the minimizer $\arg\min_x(ax^2 - 2bx)$.

My point is that our main goal is to minimize quadratic functions (we are talking about least squares here!). The only thing that the humanity knows to do well is to solve linear equations, and it is great that one is equivalent to the other! The last thing is to check whether this equivalence holds for the case of $n > 1$ variables. To do so, we will first prove three theorems.

5.2.1 Three theorems, or how to differentiate matrix expressions

The first theorem states that 1×1 matrices are invariant w.r.t the transposition:

Theorem 1. $x \in \mathbb{R} \Rightarrow x^\top = x$

The proof is left as an exercise.

The second theorem allows us to differentiate linear functions. In the case of a real function of one variable we know that $\frac{d}{dx}(bx) = b$, but what happens in the case of a real function of n variables?

Theorem 2. $\nabla b^\top x = \nabla x^\top b = b$

No surprises here, the same result in a matrix notation. The proof is straightforward, it suffices to write down the definition of the gradient:

$$\nabla(b^\top x) = \begin{bmatrix} \frac{\partial(b^\top x)}{\partial x_1} \\ \vdots \\ \frac{\partial(b^\top x)}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial(b_1x_1+\cdots+b_nx_n)}{\partial x_1} \\ \vdots \\ \frac{\partial(b_1x_1+\cdots+b_nx_n)}{\partial x_n} \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = b$$

Now applying the first theorem: $b^\top x = x^\top b$, and this concludes the proof.

Now let us switch to quadratic forms. We know that in the case of a real function of one variable we have $\frac{d}{dx}(ax^2) = 2ax$, but what happens with the quadratic forms?

Theorem 3. $\nabla(x^\top Ax) = (A + A^\top)x$

Note that if A is symmetric, then $\nabla(x^\top Ax) = 2Ax$. The proof is straightforward, let us express the quadratic form as a double sum:

$$x^\top Ax = \sum_i \sum_j a_{ij}x_ix_j$$

Now let us differentiate this double sum w.r.t the variable x_i:

$$\begin{aligned}
\frac{\partial(x^\top Ax)}{\partial x_i} &= \frac{\partial}{\partial x_i}\left(\sum_{k_1}\sum_{k_2} a_{k_1k_2}x_{k_1}x_{k_2}\right) = \\
&= \frac{\partial}{\partial x_i}\left(\underbrace{\sum_{k_1\neq i}\sum_{k_2\neq i} a_{ik_2}x_{k_1}x_{k_2}}_{k_1\neq i,k_2\neq i} + \underbrace{\sum_{k_2\neq i} a_{ik_2}x_ix_{k_2}}_{k_1=i,k_2\neq i} + \underbrace{\sum_{k_1\neq i} a_{k_1i}x_{k_1}x_i}_{k_1\neq i,k_2=i} + \underbrace{a_{ii}x_i^2}_{k_1=i,k_2=i}\right) = \\
&= \sum_{k_2\neq i} a_{ik_2}x_{k_2} + \sum_{k_1\neq i} a_{k_1i}x_{k_1} + 2a_{ii}x_i = \\
&= \sum_{k_2} a_{ik_2}x_{k_2} + \sum_{k_1} a_{k_1i}x_{k_1} = \\
&= \sum_j (a_{ij}+a_{ji})x_j
\end{aligned}$$

I split the double sum into four cases, shown by the curly brackets. Each of these four cases is trivial to differentiate. Now let us collect the partial derivatives into a gradient vector:

$$\nabla(x^\top Ax) = \begin{bmatrix} \frac{\partial(x^\top Ax)}{\partial x_1} \\ \vdots \\ \frac{\partial(x^\top Ax)}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \sum_j (a_{1j}+a_{j1})x_j \\ \vdots \\ \sum_j (a_{nj}+a_{jn})x_j \end{bmatrix} = (A+A^\top)x$$

5.2.2 Minimum of a quadratic function and the linear system

Recall that for a positive real number a solving the equation $ax = b$ is equivalent to the quadratic function $\arg\min_x(ax^2 - 2bx)$ minimization.

We want to show the corresponding connection in the case of a symmetric positive definite matrix A. So, we want to find the minimum quadratic function

$$\arg\min_{x\in\mathbb{R}^n}(x^\top Ax - 2b^\top x).$$

As before, let us equate the derivative to zero:

$$\nabla(x^\top Ax - 2b^\top x) = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}.$$

The Hamilton operator is linear, so we can rewrite our equation as follows:

$$\nabla(x^\top Ax) - 2\nabla(b^\top x) = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Now let us apply the second and third differentiation theorems:

$$(A+A^\top)x - 2b = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Recall that A is symmetric, let us divide the equation by 2, and we obtain the final linear system:

$$Ax = b.$$

Hurray, passing from one variable to many, we have not lost a thing, and can effectively minimize quadratic functions!

5.3 Back to the least squares

Finally we can move on to the main content of this course. Imagine that we have two points on a plane (x_1, y_1) and (x_2, y_2), and we want to find an equation for the line passing through these two points. The equation of the line can be written as $y = \alpha x + \beta$, that is, our task is to find the coefficients α and β. This is a secondary school exercise, let us write down the system of equations:

$$\begin{cases} \alpha x_1 + \beta = y_1 \\ \alpha x_2 + \beta = y_2 \end{cases}$$

We have two equations with two unknowns (α and β), the rest is known. In general, there exists a unique solution. For convenience, let's rewrite the same system in a matrix form:

$$\underbrace{\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix}}_{:=A} \underbrace{\begin{bmatrix} \alpha \\ \beta \end{bmatrix}}_{:=x} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}}_{:=b}$$

We obtain the equation $Ax = b$, which is trivially soved as $x^* = A^{-1}b$.

And now let us imagine that we have **three** points through which we need to draw a straight line:

$$\begin{cases} \alpha x_1 + \beta = y_1 \\ \alpha x_2 + \beta = y_2 \\ \alpha x_3 + \beta = y_3 \end{cases}$$

In a matrix form, this system can be expressed as follows:

$$\underbrace{\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \end{bmatrix}}_{:=A\,(3\times 2)} \underbrace{\begin{bmatrix} \alpha \\ \beta \end{bmatrix}}_{:=x\,(2\times 1)} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}}_{:=b\,(3\times 1)}$$

The matrix A is rectangular, and thus it is not invertible! Sounds legit, we have only two variables and three equations, and in general this system has no solution. This is a perfectly normal situation in the real world where the data comes from noisy measurements, and we need to find the parameters α and β that fit the measurements the best. We have already encountered this situation in §2.4, where we calibrated a spring scale. However, back then the solution we have obtained was purely algebraic and very cumbersome. Let's try a more intuitive way.

Our system can be written down as follows:

$$\alpha \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{:=\vec{i}} + \beta \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{:=\vec{j}} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Or, in more shortly,

$$\alpha\vec{i} + \beta\vec{j} = \vec{b}.$$

The vectors $\vec{i}$, $\vec{j}$ and $\vec{b}$ are known, we are looking for (unknown) scalars α and β. Obviously, the linear combination $\alpha\vec{i} + \beta\vec{j}$ generates a plane, and if the vector $\vec{b}$ does not lie in this plane, there is no exact solution; however, we are looking for an *approximate* solution.

Let's denote the solution error as $\vec{e}(\alpha, \beta) := \alpha\vec{i} + \beta\vec{j} - \vec{b}$. Our goal is to find the values of parameters α and β that minimize the norm of the vector $\vec{e}(\alpha, \beta)$. In other words, the problem can be formulated as $\arg\min_{\alpha,\beta} \|\vec{e}(\alpha, \beta)\|$. Refer to Fig. 5.3 for an illustration.

But wait, where are the least squares? Just a second. The square root function $\sqrt{\cdot}$ is monotonous, so $\arg\min_{\alpha,\beta} \|\vec{e}(\alpha, \beta)\| = \arg\min_{\alpha,\beta} \|\vec{e}(\alpha, \beta)\|^2$!

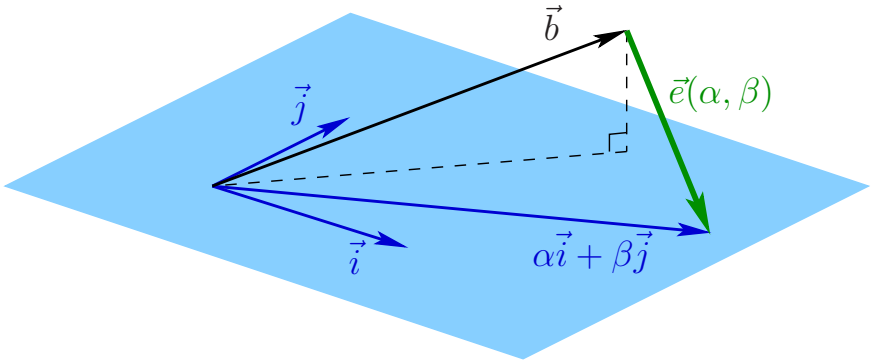


Figure 5.3: Given the vectors $\vec{i}$, $\vec{j}$ and $\vec{b}$, we want to minimize the norm of the error vector $\vec{e}$. Obviously the norm is minimized when the vector is orthogonal to the plane generated by $\vec{i}$ and $\vec{j}$.

It is quite obvious that the norm of the error vector is minimized when it is orthogonal to the plane generated by the vectors $\vec{i}$ and $\vec{j}$. We can express it by equating corresponding dot products to zero:

$$\begin{cases} \vec{i}^{\top}\vec{e}(\alpha, \beta) = 0 \\ \vec{j}^{\top}\vec{e}(\alpha, \beta) = 0 \end{cases}$$

We can write down the same system in a matrix form:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ 1 & 1 & 1 \end{bmatrix} \left(\alpha \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \beta \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

or

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ 1 & 1 & 1 \end{bmatrix} \left(\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

But we won't stop there, because the expression can be further shortened:

$$A^{\top}(Ax - b) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

And the very last transformation:

$$A^{\top}Ax = A^{\top}b.$$

Let us do a sanity check. Matrix A is 3×2, thus $A^{\top}A$ is 2×2. Matrix b is 3×1, so the matrix $A^{\top}b$ is 2×1. Therefore, in a general case the matrix $A^{\top}A$ can be invertible! Moreover, $A^{\top}A$ has a couple more nice properties.

Theorem 4. *$A^{\top}A$ is symmetric.*

It is very easy to show:

$$(A^{\top}A)^{\top} = A^{\top}(A^{\top})^{\top} = A^{\top}A.$$

Theorem 5. *$A^{\top}A$ positive semidefinite: $\forall x \in \mathbb{R}^n \quad x^{\top}A^{\top}Ax \geq 0$.*

It follows from the fact that $x^{\top}A^{\top}Ax = (Ax)^{\top}Ax > 0$. Moreover, $A^{\top}A$ is positive definite in the case where A has linearly independent columns (rank A is equal to the number of the variables in the system).

So, for a system with two unknowns, we have proven that to minimize the quadratic function $\arg\min_{\alpha,\beta} \|\vec{e}(\alpha, \beta)\|^2$ is equivalent to solve the linear system $A^{\top}Ax = A^{\top}b$. Of course, all this reasoning applies to any other

number of variables, but let us write it all down again a compact algebraic way. Let us start with a least squares problem:

$$\begin{aligned}
\arg\min \|Ax-b\|^2 &= \arg\min (Ax-b)^\top (Ax-b) = \\
&= \arg\min \left(x^\top A^\top - b^\top\right)(Ax-b) = \\
&= \arg\min \left(x^\top A^\top Ax - b^\top Ax - x^\top A^\top b + \underbrace{b^\top b}_{\text{const}} \right) = \\
&= \arg\min \left(x^\top A^\top Ax - 2b^\top Ax\right) = \\
&= \arg\min \left(x^\top \underbrace{\left(A^\top A\right)}_{:=A'} x - 2 \underbrace{\left(A^\top b\right)}_{:=b'}{}^\top x \right)
\end{aligned}$$

Having started with a least squares problem, we have come to the quadratic function minimization problem we know already. Thus, the least squares problem $\arg\min \|Ax-b\|^2$ is equivalent to minimizing the quadratic function $\arg\min \left(x^\top A' x - 2b'^\top x\right)$ with (in general) a symmetric positive definite matrix A', which in turn is equivalent to solving a system of linear equations $A'x = b'$. Phew. The theory is over.

Chapter 6

Least squares through examples

6.1 Linear-quadratic regulator

Let us start this chapter with a tiny example from the optimal control theory. Optimal control deals with the problem of finding a control law for a given system such that a certain optimality criterion is achieved. This phrase is too unspecific, let us illustrate it. Imagine that we have a car that advances with some speed, say 0.5m/s. The goal is to accelerate and reach, say, 2.3m/s. We can not control the speed directly, but we can act on the acceleration via the gas pedal. We can model the system with a very simple equation:

$$v_{k+1} = v_k + u_k,$$

where the signals are sampled every 1 second, v_k is the car speed and u_k is the acceleration of the car. Let us say that we have half a minute to reach the given speed, i.e, $v_0 = 0.5\text{m/s}$, $v_n = 2.3\text{m/s}$, $n = 30\text{s}$. So, we need to find $\{u_i\}_{i=0}^{n-1}$ that optimizses some quality criterion $J(\vec{v}, \vec{u})$:

$$\min J(\vec{v}, \vec{u}) \quad s.t.\ v_{i+1} = v_i + u_i \ \ \forall i \in 0..n-1$$

The case where the system dynamics are described by a set of differential equations and the cost is described by a quadratic functional is called a linear quadratic problem. Let us test few different quality criteria. What happens if we ask for the car to reach the final speed as quickly as possible? It can be written as follows:

$$J(\vec{v}, \vec{u}) := \sum_{i=1}^{n} (v_i - v_n)^2 = \sum_{i=1}^{n} \left(\sum_{j=0}^{i-1} u_j - v_n + v_0 \right)^2$$

To minimize this criterion, we can solve the following system in the least squares sense:

$$\begin{cases} u_0 & & & & = v_n - v_0 \\ u_0 & +u_1 & & & = v_n - v_0 \\ & & & \vdots & \\ u_0 & +u_1 & \dots & +u_{n-1} & = v_n - v_0 \end{cases}$$

Listing 1 (refer to the appendices) solves the system. The resulting arrays $\{u_i\}_{i=0}^{n-1}$ and $\{v_i\}_{i=0}^{n}$ are shown in the leftmost image of Figure 6.1. It can be seen that in this case the system reaches the final state in one timestep; clearly it is physicall impossible for a car to produce such an acceleration.

Okay, no problem, let us try to penalize large accelerations:

$$J(\vec{v}, \vec{u}) := \sum_{i=0}^{n-1} u_i^2$$

Minimization of this criterion is equivalent to solving the following system in the least squares sense:

$$\begin{cases} u_0 & = 0 \\ u_1 & = 0 \\ \dots & \\ u_{n-1} & = 0 \end{cases}$$

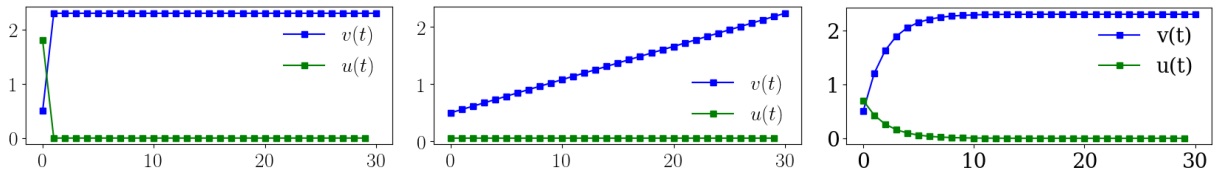


Figure 6.1: 1D optimal control problem. **Left:** lowest settling time goal; **middle:** lowest control signal goal; **right:** a trade-off between the control signal amplitude and the settling time.

Listing 2 solves this system, and the resulting arrays are shown in the middle image of Figure 6.1. This criterion indeed produces low acceleration, however the transient time becomes unacceptable.

Minimization of the transient time and low acceleration are competing goals, but we can find a trade-off by mixing both goals:

$$J(\vec{v},\vec{u}) := \sum_{i=1}^{n}(v_i - v_n)^2 + 4\sum_{i=0}^{n-1} u_i^2 = \sum_{i=1}^{n}\left(\sum_{j=0}^{i-1} u_j - v_n + v_0\right)^2 + 4\sum_{i=0}^{n-1} u_i^2$$

This criterion asks to reach the goal as quickly as possible, while penalizing large accelerations. It can be minimized by solving the following system:

$$\begin{cases} u_0 & & & & = v_n - v_0 \\ u_0 & +u_1 & & & = v_n - v_0 \\ & & & \vdots & \\ u_0 & +u_1 & \dots & +u_{n-1} & = v_n - v_0 \\ 2u_0 & & & & = 0 \\ & 2u_1 & & & = 0 \\ & & & \vdots & \\ & & & 2u_{n-1} & = 0 \end{cases}$$

Note the coefficient 2 in the equations $2u_i = 0$ and recall that we solve the system in the least squares sense. By changing this coefficient, we can attach more importance to one of the competing goals. Listing 3 solves this system, and the resulting arrays are shown in the right image of Figure 6.1.

Note that the signal $u(t)$ is equal to the signal $v(t)$ up to a multiplication by a constant gain:

$$u(t) = -Fv(t),$$

This gain is necessary to build a closed-loop regulator, it can be computed from J. In practice, just like we did in this section, engineers try different combinations of competing goals until they obtain a satisfactory transient time while not exceeding regulation capabilites.

6.2 Caricature

```
x = [100,100,97,93,91,87,84,83,85,87,88,89,90,90,90,88,87,86,84,82,80,
     77,75,72,69,66,62,58,54,47,42,38,34,32,28,24,22,20,17,15,13,12,9,
     7,8,9,8,6,0,0,2,0,0,2,3,2,0,0,1,4,8,11,14,19,24,27,25,23,21,19]
y = [0,25,27,28,30,34,37,41,44,47,51,54,59,64,66,70,74,78,80,83,86,90,93,
     95,96,98,99,99,100,99,99,99,98,98,96,94,93,91,90,87,85,79,75,70,65,
     62,60,58,52,49,46,44,41,37,34,30,27,20,17,15,16,17,17,19,18,14,11,6,4,1]
n = len(x)

cx = [x[i] - (x[(i-1+n)%n]+x[(i+1)%n])/2 for i in range(n)] # precompute the
cy = [y[i] - (y[(i-1+n)%n]+y[(i+1)%n])/2 for i in range(n)] # discrete curvature

for iter in range(1000):
    for i in range(n):
```

```
        x[i] = (x[(i-1+n)%n]+x[(i+1)%n])/2 + cx[i]*1.9
        y[i] = (y[(i-1+n)%n]+y[(i+1)%n])/2 + cy[i]*1.9
```

```
import numpy as np

def amplify(x):
    n = len(x)
    A = np.matrix(np.zeros((2*n,n)))
    b = np.matrix(np.zeros((2*n,1)))
    for i in range(n):
        A[i, i] = 1. # amplify the curvature
        A[i, (i+1)%n] = -1.
        b[i, 0] = (x[i] - x[(i+1)%n])*1.9

        A[n+i, i] =    1*.3 # light data fitting term
        b[n+i, 0] = x[i]*.3
    return (np.linalg.inv(A.T*A)*A.T*b).tolist()

x = [100,100,97,93,91,87,84,83,85,87,88,89,90,90,90,88,87,86,84,82,80,
     77,75,72,69,66,62,58,54,47,42,38,34,32,28,24,22,20,17,15,13,12,9,
     7,8,9,8,6,0,0,2,0,0,2,3,2,0,0,1,4,8,11,14,19,24,27,25,23,21,19]
y = [0,25,27,28,30,34,37,41,44,47,51,54,59,64,66,70,74,78,80,83,86,90,93,
     95,96,98,99,99,100,99,99,99,98,98,96,94,93,91,90,87,85,79,75,70,65,
     62,60,58,52,49,46,44,41,37,34,30,27,20,17,15,16,17,17,19,18,14,11,6,4,1]

x = amplify(x)
y = amplify(y)
```

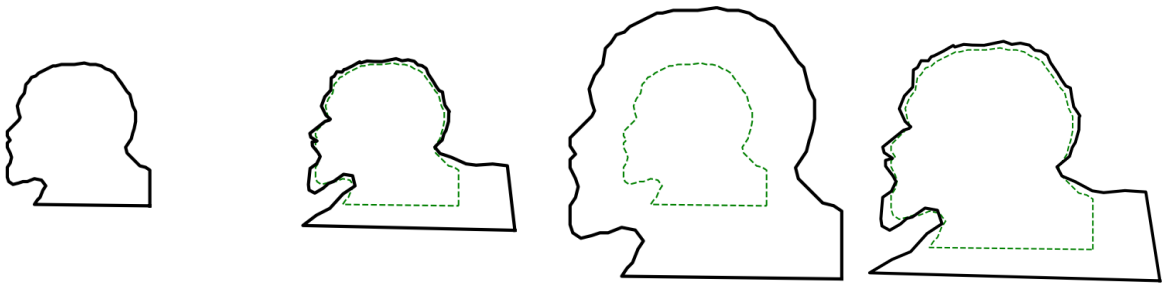


Figure 6.2:

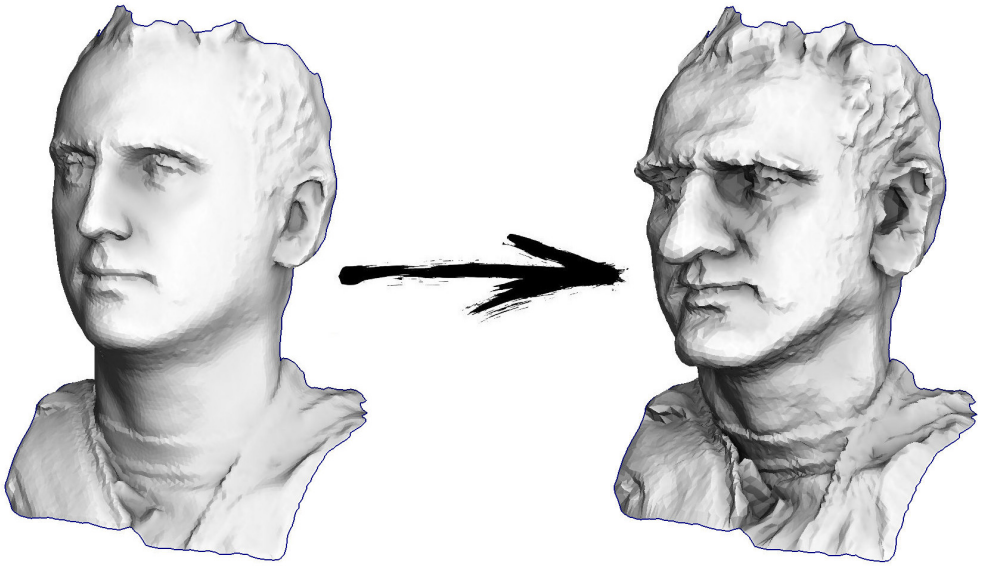


Figure 6.3: .

Figure 6.4: .

(a) **(b)** **(c)** **(d)**

Figure 6.5: Poisson image editing. We want to replace the baseball from the image **(a)** with the football **(b)**. A direct overlay leads to a unsatisfactory result **(c)**, whereas the Poisson image editing produces an image with much less visible seams **(d)**.

Figure 6.6: Least squares conformal mapping. **Left:** input mesh; **middle:** unfolded mesh textured by an artist; **right:** final textured surface. Two pinned vertices are shown in red.

Chapter 7

Under the hood of the conjugate gradient method

The resolution of linear systems is the keystone of a large number of numerical optimization algorithms encountered in many application contexts. For example, such linear systems play a central role in least-squares-type optimization problems, in finite-element-based numerical simulation methods as well as in non-linear function optimization algorithms that solve a series of linear problems.

Given an invertible $n \times n$ matrix A and an n-vector b, we would like to solve the matrix equation $Ax = b$. One way to do so is to invert A and multiply both sides by A^{-1}. While this approach is theoretically valid, there are several problems with it in practice. Computing the inverse of a large [1] matrix is expensive and susceptible to numerical error due to the finite precision of floating-point numbers. Moreover, matrices which occur in real problems tend to be sparse and one would hope to take advantage of such structure to reduce work, but matrix inversion destroys sparsity.

In this chapter we describe the conjugate gradient method, an algorithm for solving linear systems. It is widely used because it is very efficient and relatively simple to implement. It is rather difficult to have a good intuition of its functioning, which we propose to study here. The conjugate gradient method can very well be used as a black box, but it is more intellectually satisfying to understand how it works. Moreover, from a practical point of view, this allows to use it in a more efficient way, taking into account the conditioning of the matrix, by adjusting the number of iterations according to the type of problem, or by programming it in a way to perform all the computations in place (i.e., without explicitly storing large matrices in memory).

Our approach is to take the point of view of someone looking to reinvent the algorithm. We will first specify which problem is solved by the method, then we will present two methods of resolution (gradient descent and conjugation), which can be combined to obtain the conjugate gradient. We will also illustrate the behaviour of these algorithms under the most common conditions, i.e. with sparse matrices.

7.1 Problem statement

The conjugate gradient method solves the following problem:

> Given a symmetric positive definite $n \times n$ matrix A and a vector $b \in \mathbb{R}^n$, find the vector $x \in \mathbb{R}^n$ such that $Ax - b = 0$.

From a geometric point of view, it is quite understandable that to solve a linear system corresponds to calculating the intersection of the hyperplanes corresponding to each equation (see figure 7.1). It is less obvious to understand the conditions on the matrix A, and we have dedicated the entire chapter 5 to this subject. In short,

[1] It is quite common to manipulate $10^6 \times 10^6$ sparse matrices in image and geometry processing. Recall the Poisson image editing: we want to compute an image, each pixel is a variable of the system. For a rather small 1000×1000 pixels grayscale image we have 10^6 variables!

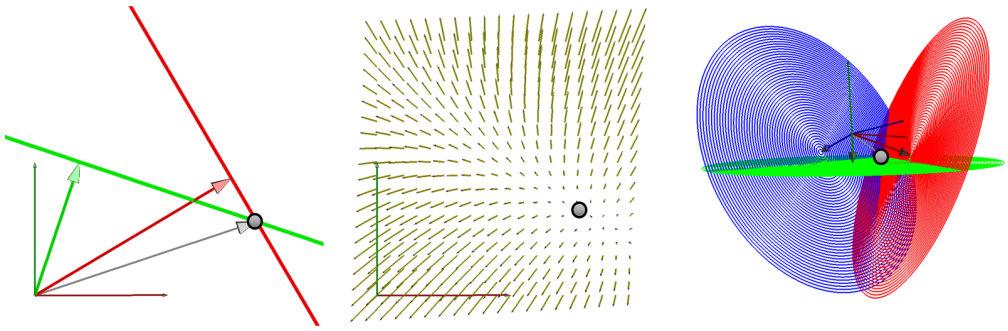


Figure 7.1: Geometric interpretations of the resolution of $Ax - b = 0$. The dot product of x with each row of A is fixed by b. Each of these constraints imposes that x is located on a hyperplane (a straight line in 2D, a plane in 3D). We can define x as the intersection of these hyperplanes: green and red lines in our 2D example (left) and the three 3D planes (right). In the middle, $v = Ax - b$ is shown as a vector field: the solution is the point where the vector field vanishes. This field can be seen as the gradient of a function to be minimized... under certain conditions.

Our problem is equivalent to the minimization of the quadratic form $x^\top Ax - 2b^\top x$. Since A is symmetric positive definite, then the solution exists and is unique.

7.2 Three ways to solve the problem

Since we want to perform an unconstrained minimization of a convex function, it is only natural to use an iterative descent method. In this chapter we study three different methods that share the common structure:

- Make an initial guess $x^{(0)} \leftarrow \vec{0}$;
- Iterate until convergence:
  - compute the next step direction $d^{(k)}$;
  - choose the step size $\alpha^{(k)}$;
  - update the solution vector $x^{(k+1)} \leftarrow x^{(k)} + \alpha^{(k)} d^{(k)}$;

These three algorithms differ only in the calculation of the step direction. The idea is to start with a very simple method and gradually build the way up to the conjugate gradient method. So, we start with the famous **gradient descent (§??)**. We can minimize the quadratic form $x^\top Ax - 2bx$, like any other convex function, by a gradient descent. Having a quadratic form even allows to calculate an optimal descent step.

Then we present a **conjugation method (§??)**. This method works not in the original space, but in the space transformed by some linear operator M, and directly projects the solution on an orthogonal basis. The vectors whose images by M form an orthogonal base are said to be conjugate (with respect to the scalar product defined by A). This is why this method is called conjugation. *Warning: this method has only a pedagogical interest, it should not be used in practice!*

Finally, we will combine two above methods to obtain the **conjugate gradient method (§??)**. The conjugate gradient is a gradient descent in which the directions of descent are modified so that they are conjugated with respect to each other.

Chapter 8

From least squares to neural networks

8.1 Logistic regression

Two possible classes encoded as $y \in \{0, 1\}$ and assume

$$p(y = 1|x, w) = \frac{1}{1 + e^{-w^\top x}},$$

where w is a $(m + 1)$-parameter vector and the last element of x is the constant 1.

Follows that

$$p(y = 0|x, w) = 1 - P(y = 1|x, w) = \frac{1}{1 + e^{w^\top x}}$$

Given n independently distributed data points x_i with corresponding labels y_i, we can write the log-likelihood as

$$\log \mathcal{L}(w) = \log \prod_{i=1}^{n} p_i(w)^{y_i}(1 - p_i(w))^{1-y_i} = \sum_{i=1}^{n} \left(y_i w^\top x_i - \log \left(1 + e^{w^\top x} \right) \right),$$

where $p_i(w) := p(y_i = 1|x_i, w)$.

To maximize $\log \mathcal{L}$, we set its derivatives to 0 and obtain

$$\frac{\partial \log \mathcal{L}}{\partial w}(w) = \sum_{i=1}^{n} x_i(y_i - p_i(w)) = 0,$$

so we have $m + 1$ non-linear equations in w. We can rewrite the system as

$$\frac{\partial \log \mathcal{L}}{\partial w}(w) = X^\top (y - p) = 0, \tag{8.1}$$

where X is the $n \times (m + 1)$ matrix whose rows are x_i, $y := \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix}^\top$ and $p := \begin{bmatrix} p_1(w) & \dots & p_n(w) \end{bmatrix}^\top$.

We can solve the system (8.1) iteratively using Newton-Raphson steps:

$$w^{(k+1)} = w^{(k)} - \left(\frac{\partial^2 \log \mathcal{L}}{\partial w \partial w^\top} \left(w^{(k)} \right) \right)^{-1} \frac{\partial \log \mathcal{L}}{\partial w} \left(w^{(k)} \right) \tag{8.2}$$

Let V be $n \times n$ diagonal matrix with $V_{i,i} = p_i(w)(1 - p_i(w))$, it is straightforward to verify that the Hessian matrix has the following expression:

$$\frac{\partial^2 \log \mathcal{L}}{\partial w \partial w^\top}(w) = -X^\top V X.$$

Then (8.2) becomes:

$$\begin{aligned} w^{(k+1)} &= w^{(k)} + \left(X^\top V^{(k)} X \right)^{-1} X^\top (y - p^{(k)}) \\ &= \left(X^\top V^{(k)} X \right)^{-1} X^\top V^{(k)} \left(X w^{(k)} + {V^{(k)}}^{-1} (y - p^{(k)}) \right) \\ &= \left(X^\top V^{(k)} X \right)^{-1} X^\top V^{(k)} z^{(k)}, \end{aligned} \tag{8.3}$$

where $z^{(k)} := Xw^{(k)} + {V^{(k)}}^{-1}(y - p^{(k)})$ and where $V^{(k)}$ and $p^{(k)}$ are V and p, respectively, evaluated at $w^{(k)}$.

Note that $w^{(k+1)}$ as given by (8.3) also satisfies

$$w^{(k+1)} = \arg\min_{w} \left(z^{(k)} - Xw\right)^{\top} V^{(k)} \left(z^{(k)} - Xw\right),$$

a weighted least-squares problem, and hense iterating (8.3) is often called iteratively reweighted least squares. It is easy to see that the Hessian matrix $X^{\top}VX$ is positive definite (it follows from the fact that $V > 0$), and thus is a convex optimization problem.

Note that convergence fails if the two classes are linearly separable. In that case we can handle this via a regularization.

Appendix A

Program listings

```python
import numpy as np

n = 30
A = np.matrix(np.zeros((n, n)))
b = np.matrix(np.zeros((n, 1)))

for i in range(0,n): # x_i = 2.3
    for j in range(0,i+1):
        A[i, j] = 1
    b[i, 0] = 2.3 - 0.5

u = np.linalg.inv(A.T*A)*A.T*b
v = [.5 + np.sum(u[:i]) for i in range(0,n+1)]
```

Listing 1: LQR example (§6.1). This program minimizes the criterion $J(\vec{v},\vec{u}) := \sum_{i=1}^{n}(v_i - v_n)^2$.

```python
import numpy as np

n = 30
A = np.matrix(np.zeros((n+1, n)))
b = np.matrix(np.zeros((n+1, 1)))

b[0, 0] = 2.3 - 0.5
for i in range(0,n):
    A[0,   i] = 1 # x_n = 2.3
    A[1+i, i] = 1 # u_i = 0

u = np.linalg.inv(A.T*A)*A.T*b
v = [.5 + np.sum(u[:i]) for i in range(0,n+1)]
```

Listing 2: LQR example (§6.1). This program minimizes the criterion $J(\vec{v},\vec{u}) := \sum_{i=0}^{n-1} u_i^2$.

```python
import numpy as np

n = 30
A = np.matrix(np.zeros((2*n, n)))
b = np.matrix(np.zeros((2*n, 1)))

for i in range(0,n): # x_i = 2.3
    for j in range(0,i+1):
        A[i, j] = 1
    b[i, 0] = 2.3 - 0.5
for i in range(0,n): # u_i = 0
    A[n+i, i] = 1*2.

u = np.linalg.inv(A.T*A)*A.T*b
v = [.5 + np.sum(u[:i]) for i in range(0,n+1)]
```

Listing 3: LQR example (§6.1). This program minimizes the criterion $J(\vec{v},\vec{u}) := \sum_{i=1}^{n}(v_i - v_n)^2 + 4\sum_{i=0}^{n-1} u_i^2$.

```
#include "OpenNL_psm.h"
#include "model.h"

int main(int argc, char** argv) {
    Model m("../input-face.obj");

    for (int d=0; d<3; d++) { // solve for x, y and z separately
        nlNewContext();
        nlSolverParameteri(NL_NB_VARIABLES, m.nverts());
        nlSolverParameteri(NL_LEAST_SQUARES, NL_TRUE);
        nlBegin(NL_SYSTEM);
        nlBegin(NL_MATRIX);

        for (int v=0; v<m.nverts(); v++) { // attachment to the original geometry
            double scaling = (m.is_boundary_vert(v) ? 10. : 0.29);
            nlRowScaling(scaling); // the boundary is fixed,
            nlBegin(NL_ROW);       // the interior has a light attachment
            nlCoefficient(v, 1);
            nlRightHandSide(m.point(v)[d]);
            nlEnd(NL_ROW);
        }

        for (int h=0; h<m.nhalfedges(); h++) { // amplify the gradients
            int v1 = m.from(h);
            int v2 = m.to(h);
            nlRowScaling(1.);
            nlBegin(NL_ROW);
            nlCoefficient(v1,  1);
            nlCoefficient(v2, -1);
            nlRightHandSide(2.5*(m.point(v1)[d] - m.point(v2)[d]));
            nlEnd(NL_ROW);
        }

        nlEnd(NL_MATRIX);
        nlEnd(NL_SYSTEM);
        nlSolve();

        for (int i=0; i<m.nverts(); i++)
            m.point(i)[d] = nlGetVariable(i);
    }
    std::cout << m;
    return 0;
}
```

Listing 4: Least squares conformal mapping.