Least squares for programmers

Dmitry Sokolov, Nicolas Ray, Etienne Corman

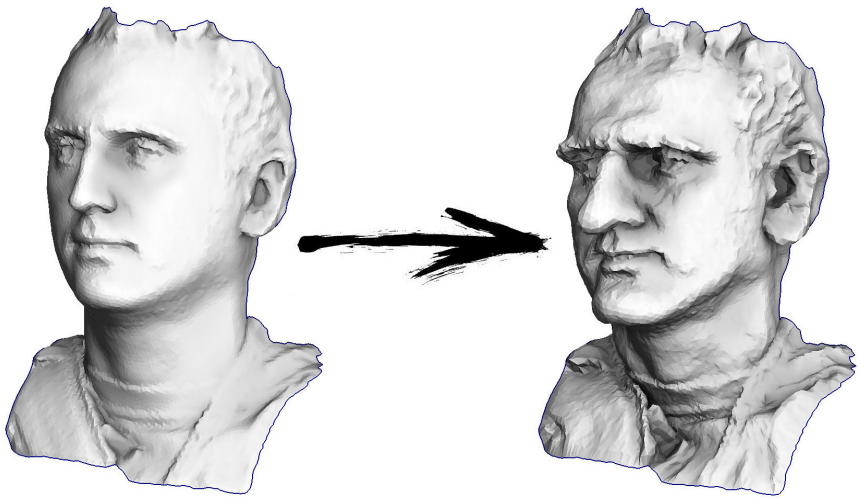
Least squares for programmers

with color plates

`https://github.com/ssloy/least-squares-course/`

Dmitry Sokolov, Nicolas Ray and Étienne Corman

Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France



Chapter 0

Reading guide

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.

The importance of linear regression (LR) cannot be overstated. The most apparent usage of LR is in statistics / data analysis, but LR is much more than that. We propose to discover how the same method (least squares) applies to the manipulation of geometric objects. This first step into the numerical optimization world can be done without strong applied mathematics background; while being simple, this step suffices for many applications, and is a good starting point for learning more advanced algorithms. We strive to communicate the underlying intuitions through numerous examples of classic problems, we show different choices of variables and the ways the energies are built. Over the last two decades, the geometry processing community have used it for computing 2D maps, deformations, geodesic paths, frame fields, etc. Our examples provide many examples of applications that can be directly solved by the least squares method. Note that linear regression is an efficient tool that has deep connections to other scientific domains; we show a few such links to broaden reader's horizons.

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.

You have probably noticed two different text colors of in the table of contents; there are chapters marked as core text and there are chapters marked as optional. Feel free to go and skip chapters depending on the level of granularity you are interested in. For example, if you are eager to see the core, jump right to chapter 3. If, however, you want to zoom in and discover connections to adjacent domains, optional chapters might be of interest.

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. We approach 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[6] 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[1], and it seems to have been validated[9].

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 favorites 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.

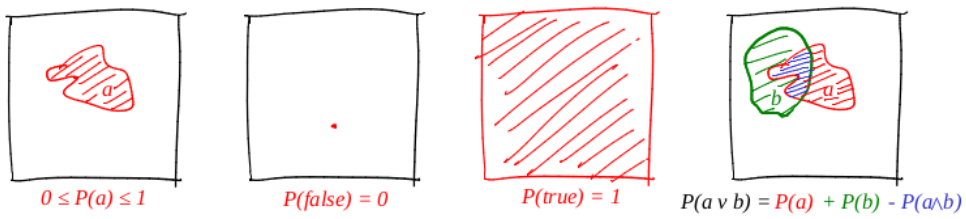


Figure 1.1: A graphical illustration for the Kolmogorov's axioms

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 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 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.

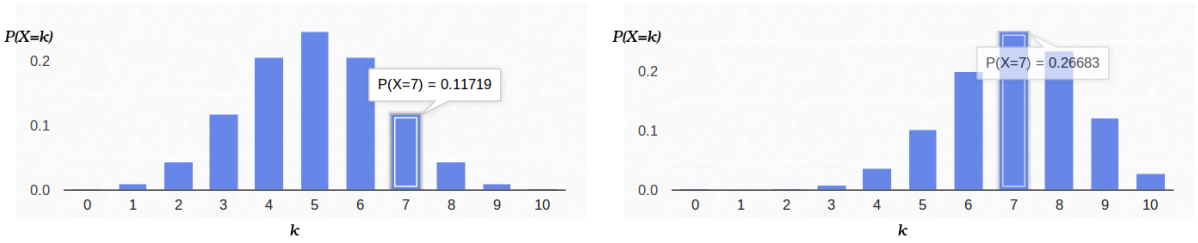


Figure 2.1: **Left:** probability density graph for Bernoulli's scheme with $p = 1/2$. **Right:** probability density graph for Bernoulli's scheme with $p = 7/10$.

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 Bernoulli's formula, leaving p as a free parameter:

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

Then 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 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.

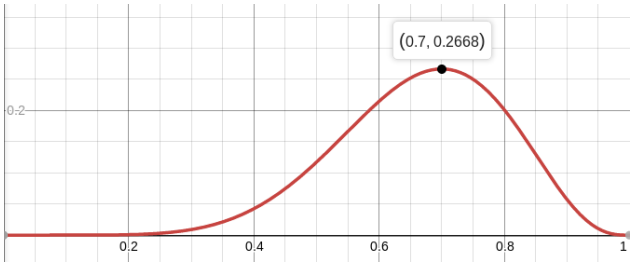


Figure 2.2: The plot of the likelihood function $\mathcal{L}(p)$ for the observation data with 7 tails out of 10 experiments.

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 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]. [2]

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:

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

Let us suppose that each measurement U_j is equal to the real value plus a 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:

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

[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 [8] 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.

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:

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

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.

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

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

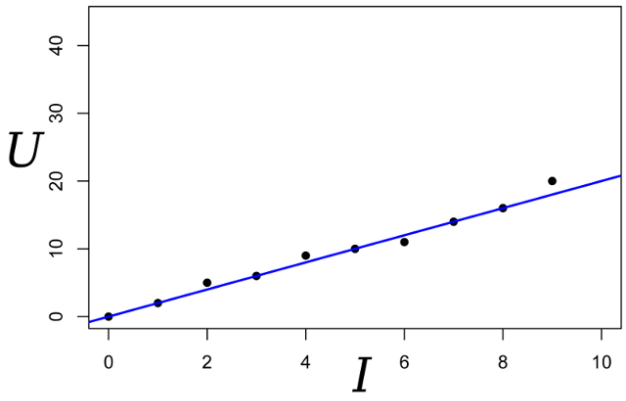


Figure 2.3: Having N meausrements U_j for each reference current I_j, we are looking for the slope of the blue line that approximates the measurements through the Ohm's law.

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:

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

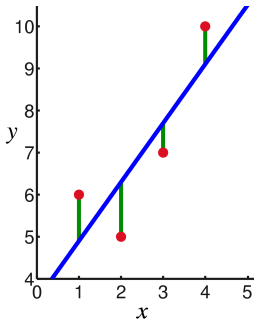


Figure 2.4: To calibrate the spring scale, we can solve the linear regression problem.

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 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 Python 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 _ 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.

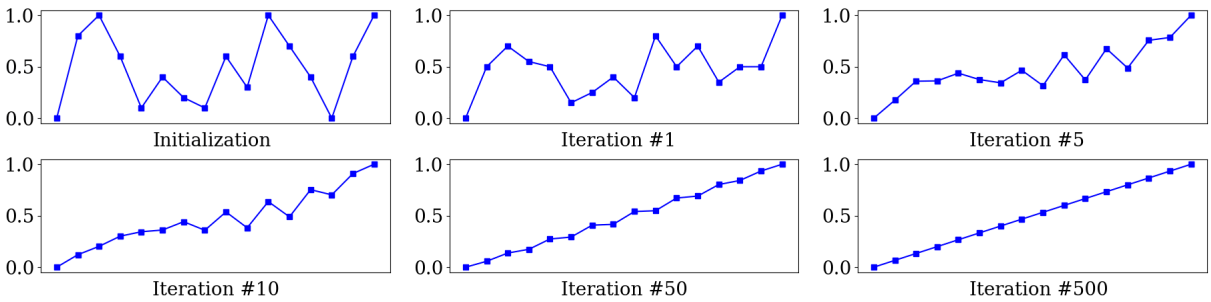


Figure 3.1: Smoothing an array: first 500 iterations of the program from § 3.1.

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 _ 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.

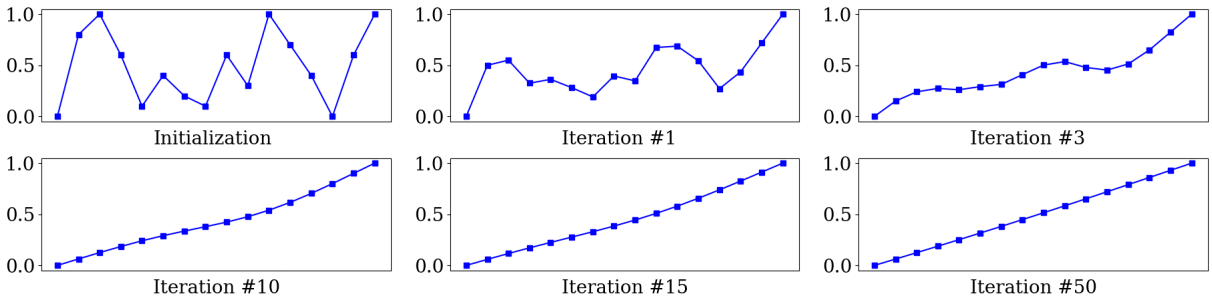


Figure 3.2: Linear function via Gauß-Seidel iteration (§3.2).

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:

```
from mesh import Mesh
import scipy.sparse

m = Mesh("input-face.obj") # load mesh

A = scipy.sparse.lil_matrix((m.nverts, m.nverts))
for v in range(m.nverts): # build a smoothing operator as a sparse matrix
    if m.on_border(v):
        A[v,v] = 1 # fix boundary verts
    else:
        neigh_list = m.neighbors(v)
        for neigh in neigh_list:
            A[v, neigh] = 1/len(neigh_list) # 1-ring barycenter for interior
A = A.tocsr() # sparse row matrix for fast matrix-vector multiplication

for _ in range(8192): # smooth the surface through Gauss-Seidel iterations
    m.V = A.dot(m.V)

print(m) # output smoothed mesh
```

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 illustration.

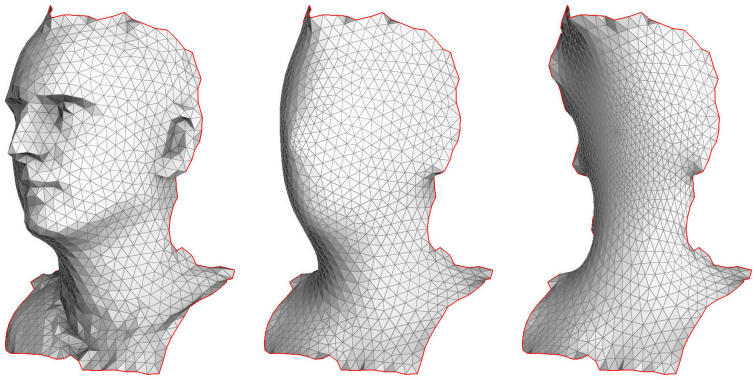


Figure 3.3: Smoothing a 3D surface (§3.3). **Left:** the input surface; **middle:** 10 iterations; **right:** 1000 iterations.

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 _ 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!

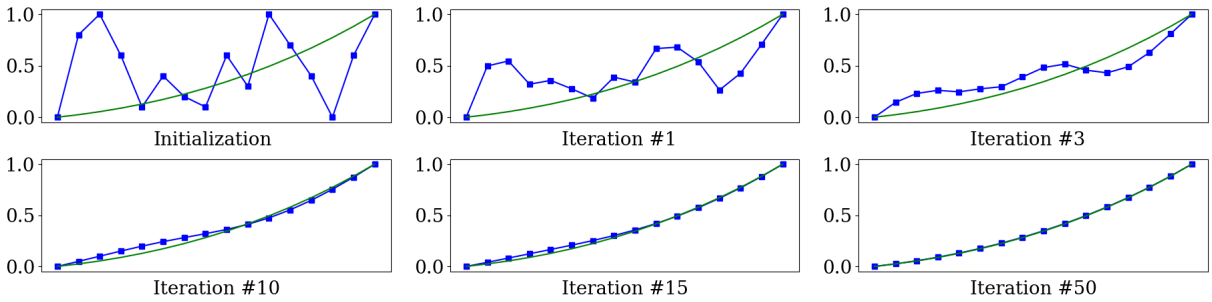


Figure 3.4: Reconstructing a cubic function (§3.4); the ground truth is shown in green.

Chapter 4

Finite elements example

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.

This chapter is closely related both to §3.4 and the chapter that follows. In §4.2 we introduce two fundamental methods for determining the coefficients of the linear system from §3.4. We start with an illustration on 2D vectors, because vectors in vector spaces give a more intuitive understanding than starting directly with approximation of functions in function spaces.

4.1 Approximation of vectors

Suppose we have given a vector $\vec{\varphi}$ and that we want to approximate this vector by a vector aligned with the vector $\vec{w}$. Figure 4.1 vector depicts the situation.

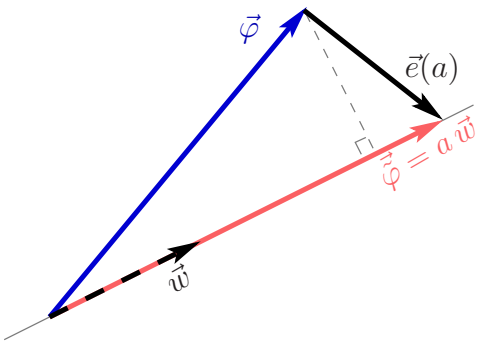


Figure 4.1: Approximation of a vector $\vec{\varphi}$ by a vector $\tilde{\vec{\varphi}}$ constrained to be collinear with a chosen vector $\vec{w}$.

Our aim is to find the vector $\tilde{\vec{\varphi}} = a\vec{w}$ which best approximates the given vector φ. A reasonable criterion for the best approximation could be to minimize the length $\|\vec{e}(a)\|$ of the difference between the approximation $\tilde{\varphi}$ and the given φ. Note that this norm is minimized when $\vec{e}(a) \perp \vec{w}$.

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:

$$\begin{cases} \mathcal{A}\varphi(x) = f(x), \quad 0 < x < 1 \\ \mathcal{A} = \dfrac{d^2}{dx^2}, \quad \varphi(0) = 0, \quad \varphi(1) = 1 \end{cases} \tag{4.1}$$

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. The finite element method does not operate directly on the differential equations; instead the boundary problem is put into equivalent variational form. The solution appears in the integral over a quantity over the domain. The integral of a function over an arbitrary domain can be broken up into the sum of integrals over an arbitrary collection of subdomains called finite elements. As long as the subdomains are sufficiently small, polynomial functions can adequately represent the local behaviour of the solution.

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):

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

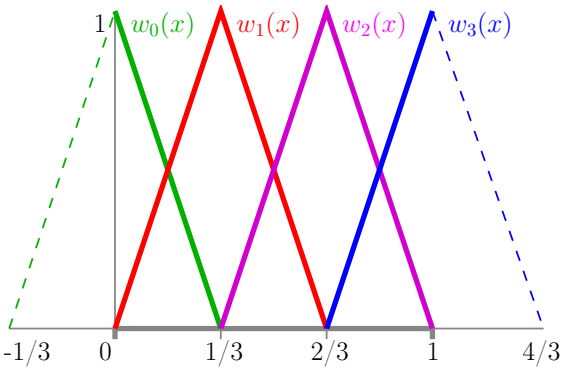


Figure 4.2: Hat function basis for FEM approximation.

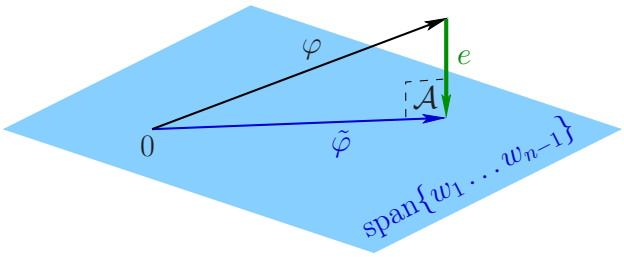


Figure 4.3: The Galerkin method imposes $\mathcal{A}$-orthogonality between the solution space $\text{span}\{w_1, \dots w_{n-1}\}$ and the solution error $e := \tilde{\varphi} - \varphi$.

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

$$\tilde{\varphi}(x) := 0 \cdot w_0(x) + b \cdot w_1(x) + c \cdot w_2(x) + 1 \cdot w_3(x), \tag{4.2}$$

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. In the rest of the section we present two different ways to find the coefficients b and c.

4.2.1 The Galerkin method

At first, let us try to use the classic Galerkin method for finding the approximate solution $\tilde{\varphi}$ of the problem (4.1). Ideally, as we have done for vectors (Figure 4.1), to approximate φ, we would like to project it onto the solution space $\text{span}\{w_1, \dots w_{n-1}\}$, and thus to ask for the error $\tilde{\varphi} - \varphi$ to be orthogonal to $\text{span}\{w_1, \dots w_{n-1}\}$. The problem, however, is that with a differential equation we do not know how to measure the true error (φ is unknown). Nevertheless, we can measure the residual $\mathcal{A}\tilde{\varphi} - f$ and we can project it onto $\text{span}\{w_1, \dots w_{n-1}\}$. This corresponds to $\mathcal{A}$-orthogonality between the true error $\tilde{\varphi} - \varphi$ and the solution space, Figure 4.3 provides an illustration.

The residual being orthogonal to the solution space can be written as the following system of $n-1$ equations:

$$\int_0^1 w_i \cdot (\mathcal{A}\tilde{\varphi} - f)\, dx = 0, \quad i = 1 \dots n-1 \tag{4.3}$$

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.

Warning! 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:

$$\begin{aligned}
\int_{x_{i-1}}^{x_{i+1}} w_i \cdot \frac{d^2\tilde{\varphi}}{dx^2}\, dx &= \int_{x_{i-1}}^{x_i} w_i \cdot \frac{d^2\tilde{\varphi}}{dx^2}\, dx + \int_{x_i}^{x_{i+1}} w_i \cdot \frac{d^2\tilde{\varphi}}{dx^2}\, dx = \\
&= \underbrace{\lim_{\varepsilon\to 0}\left[w_i \cdot \frac{d\tilde{\varphi}}{dx}\right]\Big|_{x_{i-1}+\varepsilon}^{x_i-\varepsilon} + \lim_{\varepsilon\to 0}\left[w_i \cdot \frac{d\tilde{\varphi}}{dx}\right]\Big|_{x_i+\varepsilon}^{x_{i+1}-\varepsilon}}_{=0} - \int_{x_{i-1}}^{x_{i+1}} \frac{dw_i}{dx} \cdot \frac{d\tilde{\varphi}}{dx}\, dx = \\
&= -\int_{x_{i-1}}^{x_{i+1}} \frac{dw_i}{dx} \cdot \frac{d\tilde{\varphi}}{dx}\, dx
\end{aligned}$$

It allows us to rewrite the system (4.3) as follows:

$$\int_{x_{i-1}}^{x_{i+1}} \frac{dw_i}{dx} \cdot \frac{d\tilde{\varphi}}{dx}\, dx + \int_{x_{i-1}}^{x_{i+1}} w_i \cdot f\, dx = 0, \quad i = 1 \dots n-1 \tag{4.4}$$

In our particular example we have two equations ($n = 3$), let us instantiate the functions:

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

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

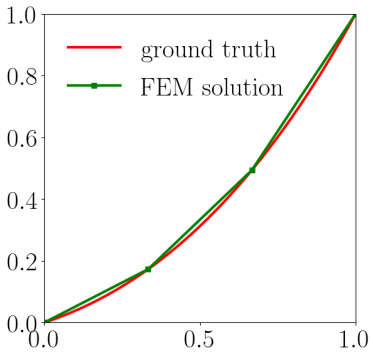


Figure 4.4: The ground truth $\varphi(x) = \frac{1}{6}x^3 + \frac{1}{2}x^2 + \frac{1}{3}x$ and its approximation found by the FEM.

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

By plugging these instances into the system (4.4) we can rewrite it as follows:

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

Next we compute all the integrals and the problem is reduced to the following system:

$$\begin{cases} 2b - c = -\frac{4}{27} \\ -b + 2c = \frac{22}{27} \end{cases} \tag{4.5}$$

Note that this is exactly the linear system we have solved numerically in §3.4. The analytical solution is $b = \frac{14}{81}$ and $c = \frac{40}{81}$, refer to Figure 4.4 for the plot.

4.2.2 The Ritz method

Let us see another method to solve the problem (4.1). The Ritz method seeks to minimize the total potential energy $\Phi(\tilde{\varphi}) := (\tilde{\varphi}, \mathcal{A}\tilde{\varphi}) - 2(\tilde{\varphi}, f)$. For our particular problem it can be written as follows:

$$\arg\min_{b,c \in \mathbb{R}} \Phi(\tilde{\varphi}), \text{ where } \Phi(\tilde{\varphi}) := \int_0^1 \tilde{\varphi} \frac{d^2}{dx^2} \tilde{\varphi} \, dx - 2 \int_0^1 \tilde{\varphi} f \, dx \tag{4.6}$$

To find the minimum, we set the partial derivatives to zero:

$$\begin{cases} \frac{\partial}{\partial b} \Phi(\tilde{\varphi}) = 0 \\ \frac{\partial}{\partial c} \Phi(\tilde{\varphi}) = 0 \end{cases} \tag{4.7}$$

Let us denote the Dirac delta centered at point y as $\delta_y(x)$; then $\frac{d^2\tilde{\varphi}}{dx^2}$ has the following expression:

$$\frac{d^2\tilde{\varphi}}{dx^2} = 3b(\delta_0(x) - 2\delta_{1/3}(x) + \delta_{2/3}(x)) + 3c(\delta_{1/3}(x) - 2\delta_{2/3}(x) + \delta_1(x)) + 3\delta_{2/3}(x)$$

It is straightforward to compute the potential energy:

$$\begin{aligned}\Phi(\tilde{\varphi}) &= \int_0^1 \tilde{\varphi} \cdot \left(\frac{d^2\tilde{\varphi}}{dx^2} - 2(x+1)\right) dx \\ &= \int_0^1 (bw_1 + cw_2 + w_3) \cdot (3b(\delta_0 - 2\delta_{1/3} + \delta_{2/3}) + 3c(\delta_{1/3} - 2\delta_{2/3} + \delta_1) + 3\delta_{2/3} - 2(x+1))\, dx \\ &= -6b^2 + 6bc - 6c^2 + 6c - 2\left(\frac{5}{9}c + \frac{4}{9}b + \frac{49}{54}\right)\end{aligned}$$

By equating the partial derivatives to zero, the system 4.7 can be reduced to:

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

Note that this is exaclty the same system as (4.5)! It turns out that for equations with self-adjoint positive definite operator ($-\frac{d^2}{dx^2}$ satisfies the conditions), the Galerkin formulation results in the same system of equations as in the Ritz formulation. A proof of this fact is out of scope of the present document; we can, however, demonstrate it for our problem (4.1) approximated with the hat functions.

4.3 DIY PDE solution

All these FEM approaches are very fancy, let us cobble together a simpler solution. Let us represent the solution by $n+1$ samples; we sample the function uniformly over the domain at the points $\{x_i\}_{i=0}^n = \left\{\frac{i}{n}\right\}_{i=0}^n$. So, we want to compute $n+1$ samples $\begin{pmatrix}\tilde{\varphi}(0) & \tilde{\varphi}\left(\frac{1}{n}\right) & \tilde{\varphi}\left(\frac{2}{n}\right) & \dots & \tilde{\varphi}(1)\end{pmatrix}^\top$, where the first and the last one are fixed. Note that for the case of the hat functions basis, computing $\tilde{\varphi}(x_i)$ corresponds exactly to the computation of the weights in Equation (4.2).

We can approximate the differential operator $\mathcal{A}$ by the second-order central finite difference:

$$\frac{d^2}{dx^2}\tilde{\varphi}\left(\frac{i}{n}\right) \approx \frac{\tilde{\varphi}\left(\frac{i-1}{n}\right) - 2\tilde{\varphi}\left(\frac{i}{n}\right) + \tilde{\varphi}\left(\frac{i+1}{n}\right)}{\frac{1}{n^2}}.$$

Since $\tilde{\varphi}(0)$ and $\tilde{\varphi}(1)$ are fixed, all this boils down to the following system of $n-1$ equations:

$$\begin{pmatrix} -2n^2 & n^2 & & & & \\ n^2 & -2n^2 & n^2 & & & \\ & & \ddots & & & \\ & & & n^2 & -2n^2 & n^2 \\ & & & & n^2 & -2n^2 \end{pmatrix} \begin{pmatrix} \tilde{\varphi}\left(\frac{1}{n}\right) \\ \tilde{\varphi}\left(\frac{2}{n}\right) \\ \vdots \\ \tilde{\varphi}\left(\frac{n-1}{n}\right) \end{pmatrix} = \begin{pmatrix} f\left(\frac{1}{n}\right) - n^2\tilde{\varphi}(0) \\ f\left(\frac{2}{n}\right) \\ \vdots \\ f\left(\frac{n-1}{n}\right) - n^2\tilde{\varphi}(1) \end{pmatrix}$$

Let us precompute the right-hand side $f\left(\frac{i}{n}\right) = \frac{i}{n} + 1$, and with minor simplifications our system can be transformed as follows:

$$\underbrace{\begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & & \ddots & & & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix}}_{:=A} \underbrace{\begin{pmatrix} \tilde{\varphi}\left(\frac{1}{n}\right) \\ \tilde{\varphi}\left(\frac{2}{n}\right) \\ \vdots \\ \tilde{\varphi}\left(\frac{n-1}{n}\right) \end{pmatrix}}_{:=x} = \underbrace{\begin{pmatrix} -\frac{1+n}{n^3} + \tilde{\varphi}(0) \\ \frac{2+n}{n^3} \\ \vdots \\ -\frac{n-1+n}{n^2} + \tilde{\varphi}(1) \end{pmatrix}}_{:=b}$$

This is a simple linear system $Ax = b$[1] with a symmetric positive definite matrix A. It is not very surprising to see that this system is again exactly the same as (4.5). The Galerkin method corresponds to

[1] Note that we have redefined b, it is not a scalar anymore but the right-hand side vector.

solving this system directly, whereas the Ritz method corresponds to minimization of the quadratic form $x^\top Ax - 2b^\top x$. It turns out that in the case of a symmetric positive definite matrix A both solving for $Ax = b$ and minimizing the quadratic form $x^\top Ax - 2b^\top x$ are equivalent; the following chapter is dedicated to this remarkable fact.

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.*

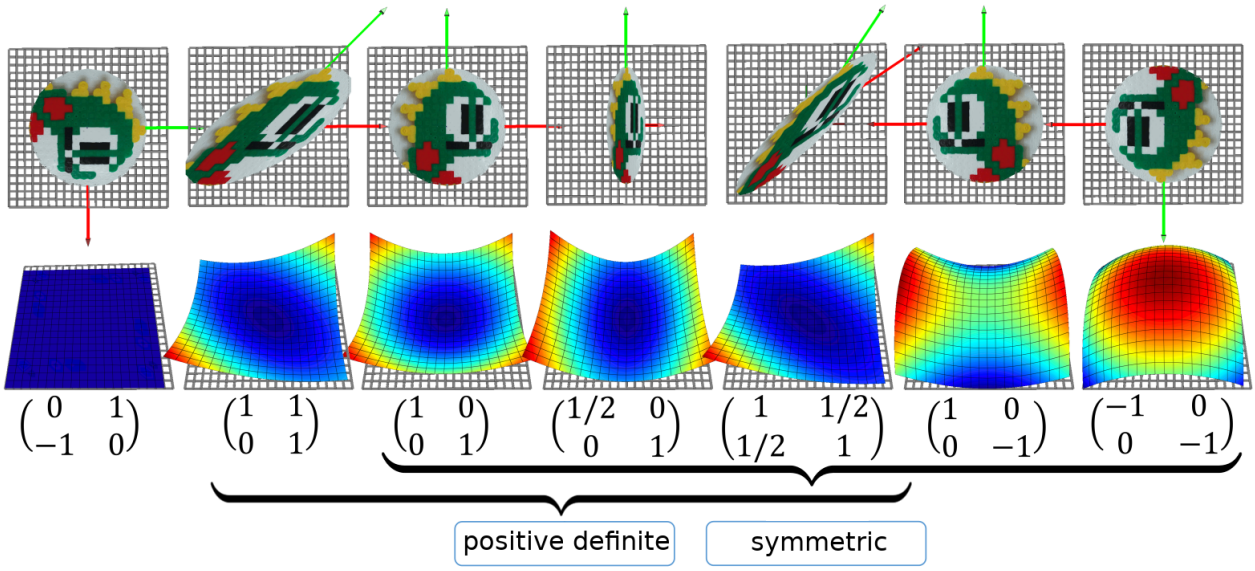


Figure 5.1: Seven examples of 2×2 matrices, some of them are positive definite and/or symmetric. **Top row:** the matrices are interpreted as linear functions $f(x) : \mathbb{R}^2 \to \mathbb{R}^2$. **Middle row:** the matrices are interpreted as quadratic functions $f(x) : \mathbb{R}^2 \to \mathbb{R}$.

This definition looks pretty awkward, but it applies perfectly to 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:

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

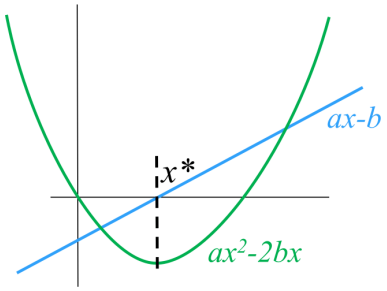


Figure 5.2: In 1d, the solution x^* of the equation $ax - b = 0$ solves the minimization problem $\arg\min_x(ax^2 - 2bx)$ as well.

5.2 Minimizing a quadratic function

Let us return to the unidimensional 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 unidimensional 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:

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

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 gradient 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 $\arg\min_{\alpha,\beta} \|\vec{e}(\alpha, \beta)\| = \arg\min_{\alpha,\beta} \|\vec{e}(\alpha, \beta)\|^2$!

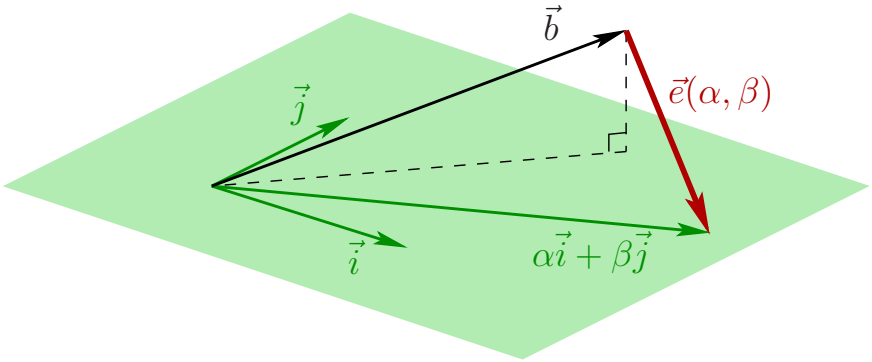


Figure 5.3: Given the vectors $\vec{i}$, $\vec{j}$ and $\vec{b}$, we want to minimize the norm of the error vector $\vec{e}$. Obviously the norm is minimized when the vector is orthogonal to the plane generated by $\vec{i}$ and $\vec{j}$.

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).

Congratulations, we have just found another way to do the spring scale calibration (§2.4). 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:

$$v_{i+1} = v_i + u_i,$$

where the signals are sampled every 1 second, v_i is the car speed and u_i 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 $J(\vec{v}, \vec{u})$:

$$\min J(\vec{v}, \vec{u}) \quad s.t.\ v_{i+1} = v_i + u_i = v_0 + \sum_{j=0}^{i-1} u_j \quad \forall i \in 0..n-1$$

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:

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

Following listing solves the system:

```
import numpy as np
n,v0,vn = 30,0.5,2.3
A = np.matrix(np.tril(np.ones((n,n))))
b = np.matrix([[vn-v0]]*n)
u = np.linalg.inv(A.T*A)*A.T*b
v = [v0 + np.sum(u[:i]) for i in range(0,n+1)]
```

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. The solution is obvious: $u_0 = v_n - v_0$, $u_i = 0\ \forall i > 0$, so in this case the system reaches the final state in one timestep, and it is clearly physically impossible for a car to produce such an acceleration.

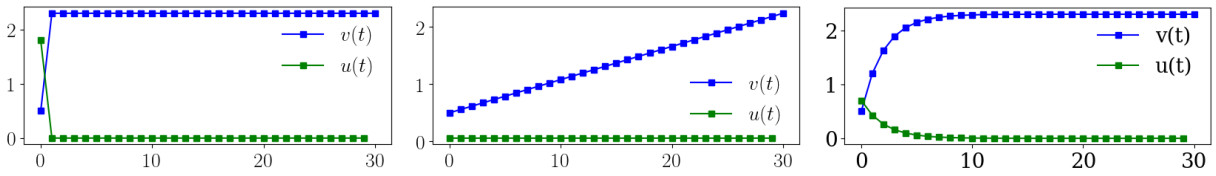


Figure 6.1: 1D optimal control problem. **Left:** lowest settling time goal; **middle:** lowest control signal goal; **right:** a trade-off between the control signal amplitude and the settling time.

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

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

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

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

Following listing solves this system, and the resulting arrays are shown in the middle image of Figure 6.1.

```
import numpy as np
n,v0,vn = 30,0.5,2.3
A = np.matrix(np.vstack((np.diag([1]*n), [1]*n)))
b = np.matrix([[0]]*n + [[vn-v0]])
u = np.linalg.inv(A.T*A)*A.T*b
v = [v0 + np.sum(u[:i]) for i in range(0,n+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:

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

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. Following listing solves this system, and the resulting arrays are shown in the right image of Figure 6.1.

```
import numpy as np
n,v0,vn = 30,0.5,2.3
A = np.matrix(np.vstack((np.tril(np.ones((n,n))), np.diag([2]*n))))
b = np.matrix([[vn-v0]]*n + [[0]]*n)
u = np.linalg.inv(A.T*A)*A.T*b
v = [v0 + np.sum(u[:i]) for i in range(0,n+1)]
```

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

$$u(t) = -F(v(t) - v_{\text{goal}}),$$

This gain is necessary to know in order to build a closed-loop regulator, and 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 Poisson image editing

The next problem is motivated by image editing. Figure 6.3 provides an illustration: we want to replace the baseball from the image **(a)** with the football **(b)**. A direct overlay leads to a unsatisfactory result **(c)**. How to swap the content seamlessly?

Poisson's equation can be of help here. Let us start with a 1D example from Figure 6.2: we are looking for a unknown function $f(x)$ defined on $x \in [0, 2\pi]$ that is as close as possible to $g(x) := \sin x$, but is constrained to have $f(0) = 1$ and $f(2\pi) = 3$. We can formulate it as the Poisson's equation with Dirichlet boundary conditions:

$$\frac{d^2}{dx^2} f = \frac{d^2}{dx^2} g \quad \text{s.t. } f(0) = 1,\ f(2\pi) = 3 \tag{6.1}$$

Two decades ago people used Gauß-Seidel to solve these linear systems [5] (refer to Listing A.1), however with modern conjugate gradients-based solvers [4] it is much easier to reformulate it as a minimization problem. Poisson's equation with Dirichlet boundary conditions corresponds to the following least squares formulation:

$$\min_f \int_0^{2\pi} \|f' - g'\|^2 \quad \text{with } f(0) = 1,\ f(2\pi) = 3$$

Let us say that we represent the functions f and g by n samples, then we can obtain f by solving the following linear system in the least squares sense:

$$\begin{cases} f_1 & & & & = g_1 - g_0 + f_0 \\ -f_1 & +f_2 & & & = g_2 - g_1 \\ & \ddots & \ddots & & \vdots \\ & & -f_{n-3} & +f_{n-2} & = g_{n-2} - g_{n-3} \\ & & & -f_{n-2} & = g_{n-1} - g_{n-2} - f_{n-1} \end{cases} \tag{6.2}$$

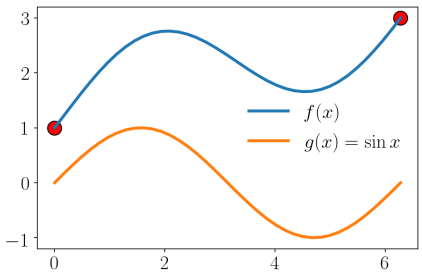


Figure 6.2: 1D Poisson's equation: the function $f(x)$ is constrained at the extremities and it has the same second derivative as $g(x)$.

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

Figure 6.3: 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)**.

Here we approximate the derivative by finite differences; this system has $n-1$ equations (one equation per finite difference) with $n-2$ unknowns, f_0 and f_{n-1} being fixed. Listing A.2 solves this system, and the result is shown in Figure 6.2.

Back to the image editing example (Figure 6.3). All color channels are solved independently one from another, so we can say that we manipulate grayscale images. Let us say that we have two real-valued functions a (sub-image of the baseball photo) and b (the football image) defined over Ω. For the sake of simplicity we consider a rectangular domain Ω. We are looking for the function f who takes its boundary conditions from a and the gradients from b:

$$\min_f \int_\Omega \|\nabla f - \nabla b\|^2 \quad \text{with } f|_{\partial\Omega} = a|_{\partial\Omega}$$

We can discretize the problem exactly as in the previous example: if we have $w \times h$-pixels grayscale images a and b. To compute a $w \times h$-pixels image f, we can solve the following system in the least squares sense:

$$\begin{cases} f_{i+1,j} - f_{i,j} &= b_{i+1,j} - b_{i,j} \quad \forall (i,j) \in [0 \dots w-2] \times [0 \dots h-2] \\ f_{i,j+1} - f_{i,j} &= b_{i,j+1} - b_{i,j} \quad \forall (i,j) \in [0 \dots w-2] \times [0 \dots h-2] \\ f_{i,j} &= a_{i,j} \quad \forall (i,j) \text{ s.t. } i=0 \text{ or } i=w-1 \text{ or } j=0 \text{ or } j=h-1 \end{cases} \tag{6.3}$$

Listing A.3 solves this system, and the result is shown in the rightmost image of Figure 6.3.

6.3 Caricature

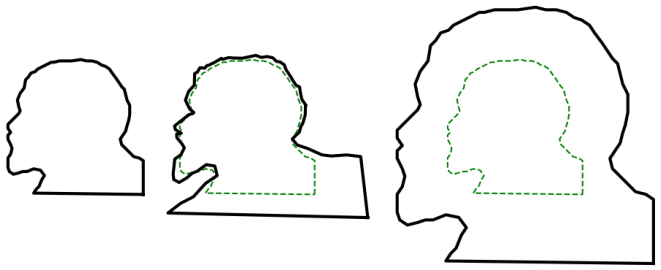
Caricature, a type of exaggerated artistic portrait, amplifies the characteristic traits of human faces. Typically, this task is left to artists, as it has proven difficult for automated methods. Here we show an extremely naive approach, starting with a 2D silhouette. This section is closely related to Poisson image editing described in the previous section. Let us consider the following program:

```
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 _ 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
```

It defines a 2D silhouette as a closed polyline represented by two same length arrays `x` and `y` (Figure 6.4, (a)). The idea is to increase the curvature of the polyline, thus exaggerating the traits. To this end, we compute the curvatures via finite differences and store it in the arrays `cx` and `cy`. Then we want to solve the Poisson's equation with the increased curvature as the right hand side of the equation. So, we perform 1000 Gauß-Seidel iterations to solve the equation. Figure 6.4 shows the evolution of the polyline. After 10 iterations the drawing looks very good, exactly what we had in mind, but what happens next? Well, there is no surprise: the polyline is inflated, because it corresponds exactly to what we have asked for. To scale finite differences is to scale the input signal... How to fix it? Well, we can stop the process after 10 iterations, thus exploiting the fact that Gauß-Seidel has a slow convergence in low frequencies, but it is a unsatisfactory solution. It would much be better if the result was at the true minimum of our optimization routine.



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

Figure 6.4: Gauß-Seidel iterations amplifying the curvature of the closed polyline shown in image **(a)**. Image **(b):** 10 iterations, image **(c):** 1000 iterations.

Figure 6.5: A trade-off between the desirable curvature and data attachment produces the expected effect.

The minimization problem that corresponds to the above listing can be written as follows:

$$\min \sum_{\forall \text{ edge } (i,j)} \left(x'_j - x'_i - c \cdot (x_j - x_i)\right)^2, \tag{6.4}$$

where c corresponds to the scaling coefficient, x_i are the input coordinates and x'_i are the unknowns. The problem is separable in two coordinates, so we list here only the x part.

The simplest way to prevent the "inflation" of the model is to add a data attachment term:

$$\min \sum_{\forall \text{ edge } (i,j)} \left(x'_j - x'_i - c_0 \cdot (x_j - x_i)\right)^2 + \sum_{\forall \text{ vertex } i} c_1^2 \left(x'_i - x_i\right)^2 \tag{6.5}$$

The coefficients c_0 and c_1 allow us to tune the optimization to achieve the desired trade-off between the curvature exaggeration and the data attachment. The result is shown in Figure 6.5. To minimize the energy (Equation (6.5)), we can solve the following system in the least squares sense:

$$\begin{cases}
-x'_0 & +x'_1 & & & & = c_0 \cdot (x_1 - x_0) \\
 & -x'_1 & +x'_2 & & & = c_0 \cdot (x_2 - x_1) \\
 & & \ddots & \ddots & & \vdots \\
 & & & -x'_{n-2} & +x'_{n-1} & = c_0 \cdot (x_{n-2} - x_{n-1}) \\
x'_0 & & & & -x'_{n-1} & = c_0 \cdot (x_{n-1} - x_0) \\
c_1 \cdot x'_0 & & & & & = c_1 \cdot x_0 \\
 & c_1 \cdot x'_1 & & & & = c_1 \cdot x_1 \\
 & & & \ddots & & \vdots \\
 & & & & c_1 \cdot x'_{n-1} & = c_1 \cdot x_{n-1}
\end{cases} \tag{6.6}$$

Refer to Listing A.4 for the corresponding source code. Note that we can solve Equation(6.5) for 3D surfaces as well, an example is shown in Figure 6.6, refer to Listing A.5 for the source code.

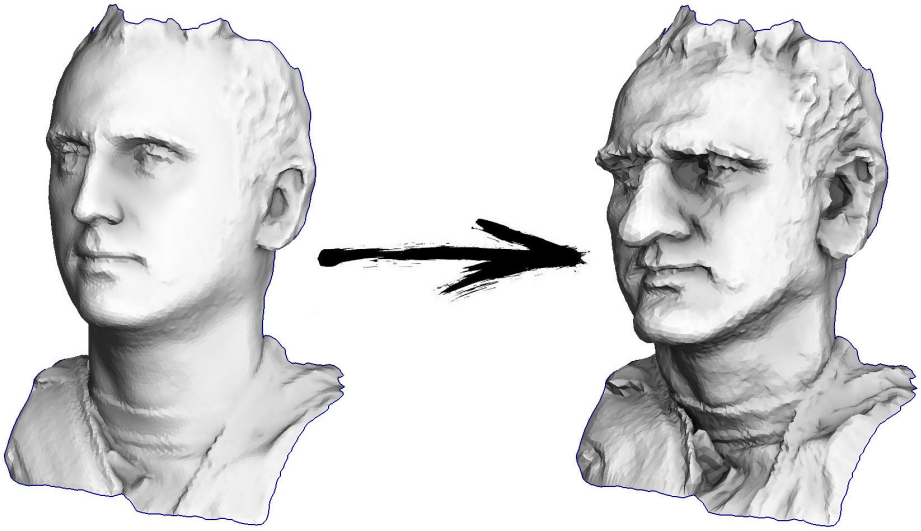


Figure 6.6: We can solve Equation(6.5) for 3D surfaces as well (Listing A.5).

6.4 Cubify it!

Let us compute another deformation of a 3D surface, refer to Figure 6.7 for an illustration. The idea is to deform the surface by aligning triangles with one of the global coordinate planes, thus obtaining a moai statue effect.

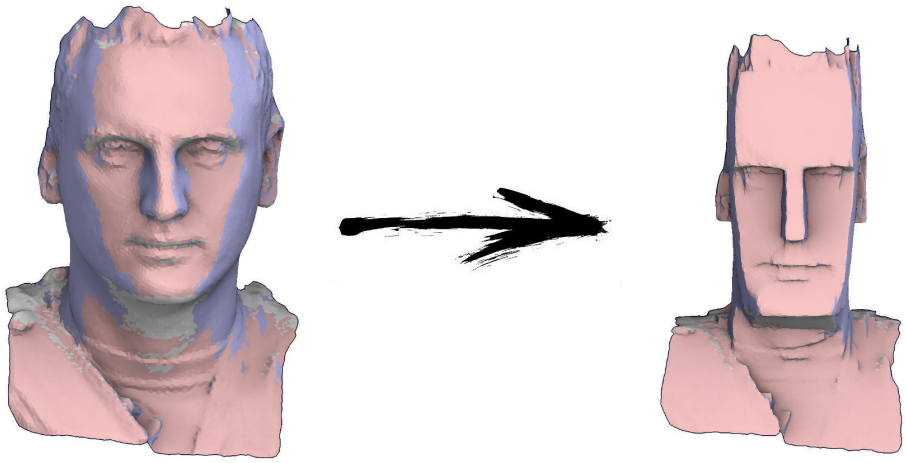


Figure 6.7: If we flatten an input surface in one of three coordinates (flagging shown in color), we can obtain a moai statue effect.

First for each triangle we compute the coordinate axis closest to its normal:

$$\vec{a}_{ijk} := \underset{\vec{u}\in\{(1,0,0),(0,1,0),(0,0,1)\}}{\arg\max} \left|\vec{u}\cdot\vec{N}_{ijk}\right|$$

Three different colors in the left image of Figure 6.7 correspond to the axes. And then we want to deform the surface according to this coloring. It is very easy to do. Let $\vec{e}_{ij}$ denote the vector corresponding to the edge (i,j) in the input data, and $\vec{e'}_{ij}$ be the modified geometry (the unknowns). Here is a small test: what would be the result of the following optimization?

$$\min \sum_{\forall \text{ edge } ij} \left\|\vec{e'}_{ij} - \vec{e}_{ij}\right\|^2$$

Surely you have recognized Equation (6.4), it is the simplest Poisson problem, giving the output geometry equal to the input. We will add few more terms to this energy to obtain the desired effect.

To align the triangles with the coordinate axes, we can define the projection operator by

$$\text{proj}_{\vec{u}}\vec{v} := \frac{\vec{v}\cdot\vec{v}}{\vec{u}\cdot\vec{u}}\vec{u}$$

This operator projects the vector v orthogonally onto the line spanned by vector u. Then the desired geometry can be obtained by minimizing the following energy:

$$\begin{aligned}\min \sum_{\forall \text{ edge } ij} c_0 \left\|\vec{e'}_{ij} - \vec{e}_{ij}\right\|^2 + \sum_{\forall \text{ triangle } ijk} c_1 \Bigg(& \left\|\vec{e'}_{ij} - \vec{e}_{ij} + \text{proj}_{\vec{a}_{ijk}}\left(\vec{e}_{ij}\right)\right\|^2 + \\ & \left\|\vec{e'}_{jk} - \vec{e}_{jk} + \text{proj}_{\vec{a}_{ijk}}\left(\vec{e}_{jk}\right)\right\|^2 + \\ & \left\|\vec{e'}_{ki} - \vec{e}_{ki} + \text{proj}_{\vec{a}_{ijk}}\left(\vec{e}_{ki}\right)\right\|^2 \Bigg)\end{aligned}$$

where c_0 and c_1 represent the trade-off between the flattening and the old data attachment. Note that $\text{proj}_{\vec{a}_{ijk}}\left(\vec{e}_{ij}\right)$ corresponds to the projection of the vector $\vec{e}_{ij}$ to the plane with the normal vector $\vec{a}_{ijk}$. Thus, this energy has two types of terms: attachment to the original geometry and the attachment to the triangles projected to one of the coordinate planes. Listing A.6 provides the source code whose result is visible in Figure 6.7.

6.5 Least squares conformal mapping

A parameterization of a surface consists in finding a way to store the colors of the surface in a texture. The mapping of the points of the surface to the texture is defined by a mapping function $\mathbb{R}^3 \to \mathbb{R}^2$. By inverting this function we can colorize the surface using a flat image drawn by an artist. Parameterization of a surface is a problem equivalent to flattening this surface. Figure 6.8 gives us a pretty intuitive idea of the purpose of parameterization.

Figure 6.8: 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.

In our context, we are manipulating triangulated surfaces and we define a parametrization as a piecewise linear function where the pieces are the triangles. Such functions are stored in texture coordinates which are the 2D coordinates of the vertices of the triangulation.

It is very difficult to define what a good parameterization is, there are many different ways to compare the quality of maps. The distortion of a mapping is defined by the Jacobian matrix. Ideally, it should be an isometric transformation, but this is an unreachable goal. In continuous settings, there only exist maps that preserve angles (conformal) and maps that preserve area (authalic). In this example, we manipulate conformal (angle preserving) maps.

Conformal maps have a very interesting feature: their distortion is locally reduced to a scaling. The stretching of the map is the same in all directions (the map is called isotropic), which makes this type of parameterization relatively simple to manipulate. The conservation of angles implies that the texture (locally) is not elongated. On the other hand, the area is not conserved, implying eventual strong changes in stretching from one place to another on the surface.

Computing such a mapping [3] is a direct instantiation of the Cauchy-Riemann equations on a pair of real-valued functions of two real variables $u(x, y)$ and $v(x, y)$ representing the texture coordinates:

$$\begin{cases} \frac{\partial u}{\partial x} &= \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} &= -\frac{\partial v}{\partial x} \end{cases}$$

In this form, the equations correspond structurally to the condition that the Jacobian matrix is of the form $\begin{pmatrix} a & -b \\ b & a \end{pmatrix}$. Geometrically, such a matrix is always the composition of a rotation with a scaling, and in particular preserves angles. The Jacobian of a function (u, v) takes infinitesimal line segments at the intersection of two curves in the parameter plane (x, y) and rotates them to the corresponding segments in u, v. Consequently, a function satisfying the Cauchy–Riemann equations, with a nonzero derivative, preserves the angle between curves in the plane. That is, the Cauchy–Riemann equations are the conditions for a function to be conformal.

Of course, there will be no exact solutions for a triangle mesh, therefore, as usual, we can sum failure of this relationship to hold over all triangles:

$$\min \sum_{\forall \text{ triangle } ijk} A_{ijk} \left((\nabla u)_{ijk} - N_{ijk} \times (\nabla v)_{ijk} \right)^2,$$

where A_{ijk} is the area of the triangle ijk, and N_{ijk} is the normal vector to the triangle. Our maps are linear

per triangle, therefore the gradients are constant and can be computed as follows: $(\nabla u)_{ijk} = \frac{1}{2A_{ijk}}(u_i\vec{e}_{jk} + u_j\vec{e}_{ki} + u_k\vec{e}_{ij})$.

Of course, we can not do better than $u(x, y) = v(x, y) = 0$, and such a map is unsatisfactory. Therefore we can "pin" two arbitrary vertices to some arbitrary points in the u, v plane. Pinning one vertex determines translation in the plane of the texture, whereas the other determines rotation and scale. This energy is very easy to minimize, just as before we need to solve a linear system. Refer to Listing A.7 for the source code. This is a very simple program: for each triangle we compute its coordinates in a local 2D basis, compute the expression of $(\nabla u)_{ijk}$ and $(\nabla v)_{ijk}$ in this basis and create two rows in the matrix (one row per the Cauchy-Riemann equation). Figure 6.8 shows the result of the execution of the program.

6.6 As-rigid-as possible deformations

How to deform a character in a plausible manner? Usually this task is done by artists expertly rigged 3D models. However simple deformation models can still produce satisfying deformations [7]. Our deformation will be controlled by a set of vertices $\mathcal{I}$ on the surface which will forced to the position p_k (Figure 6.9 left).

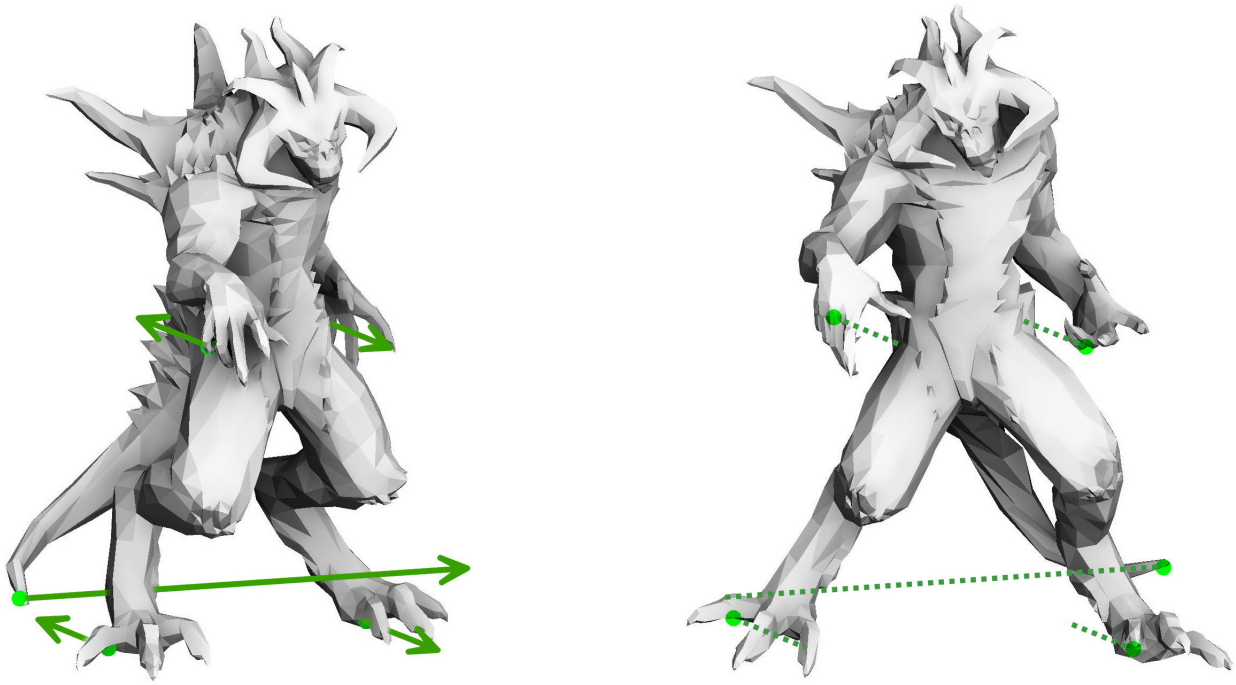


Figure 6.9: **Left:** deformation constraints. **Right:** deformation solution of Equation (6.8).

The first requirement one might have is that the deformation must be smooth. So the first idea is to best preserve the edge of the original surface as best as possible while satisfying the position constraints. The new vertex positions x' are obtained be solving the least squares problem:

$$\min_{x'} \sum_{\forall \text{ edge } (i,j)} \left\| x'_i - x'_j - (x_i - x_j) \right\|^2 \quad \text{with } x'_k = p_k, k \in \mathcal{I}. \tag{6.7}$$

The resulting deformation however does not look realistic at all: the surface is badly stretched near the constraints (Figure 6.10) and the our deformation model is unable to create the global rotation induced by the constraints. How to make the deformation looks like the character is moving? Human(-oid) motions are constraints by a skeleton making any movement a composition of rotations around joints. To simulate this effect we can ask that a vertex neighborhood is the rotation of the vertex neighborhood from the original mesh. This way the deformation will seem more rigid. We assign rotation matrices R_i at each vertex i which will affect all incident edges. The least squares problem has now two sets of unknowns the vertex positions and the rotations:

$$\min_{x',R} \sum_{\forall \text{ edge } (i,j)} \left\| x'_i - x'_j - R_i(x_i - x_j) \right\|^2 \quad \text{with } x'_k = p_k, k \in \mathcal{I}. \tag{6.8}$$

The problem (6.8) is a nonlinear least squares problem but it can be solved efficiently by alternatively solving for the vertex position and solving for the rotation. Solving for the vertex positions is a linear problem that can solved by using Gauß-Seidel iterations. Finding the rotations is a so-called *orthogonal Procrustes problem* which has a closed form solution: Let $U\Sigma V^\top$ be the singular value decomposition of the matrix $\sum_{j \text{ incident } i}(x_i' - x_j')(x_i - x_j)^\top$ then $R_i = UV^\top$. The resulting deformation is able to create a global rotation of the model while the position constraints nicely spread out across the surface (Figure 6.9 and 6.10 right). Refer to Listing A.8 for the source code.

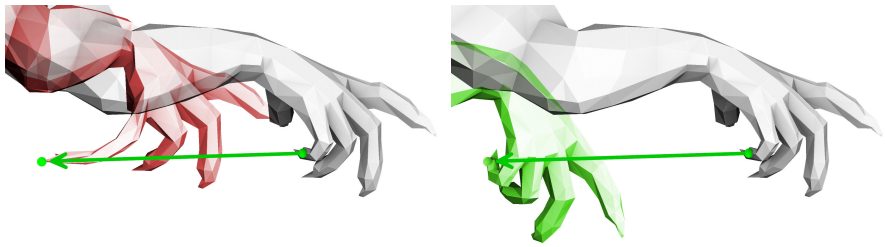


Figure 6.10: **Left:** the deformation solution of Equation (6.7) is very local and overstretch one finger. **Right:** the solution of Equation (6.8) creates a local rotation of the hand.

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 nonlinear function optimization algorithms that solve a series of linear problems.

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,

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

Why reformulating the problem? 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

[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!

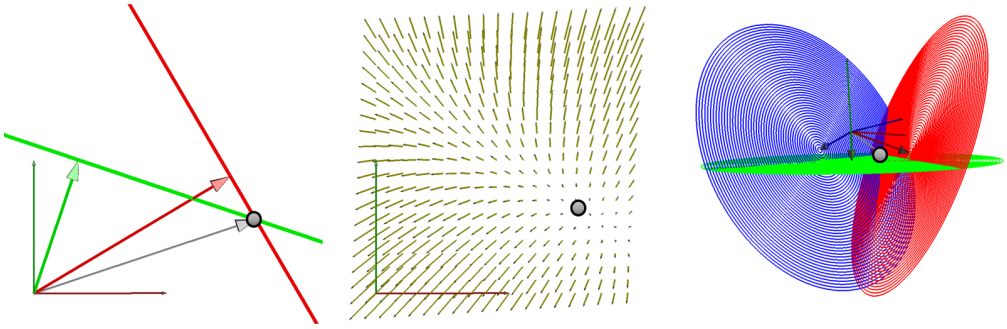


Figure 7.1: Geometric interpretations of the resolution of $Ax - b = 0$. The dot product of x with each row of A is fixed by b. Each of these constraints imposes that x is located on a hyperplane (a straight line in 2D, a plane in 3D). We can define x as the intersection of these hyperplanes: green and 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.

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. As we will see shortly, iterative methods are the technique of the choice for solving such systems.

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 (§7.3)**. We can minimize the quadratic form $x^\top Ax - 2bx$ like any other convex function, however, having a quadratic function allows to calculate an optimal descent step.

Then we present a **conjugation method (§7.4)**. This method works not in the original space, but in the space transformed by some linear map 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 combine two above methods to obtain the **conjugate gradient method (§7.5)**. 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.

7.3 The 1st way: the gradient descent

The gradient of a function gives the direction of fastest increase. The gradient descent consists of, starting from an initial position $x^{(0)}$, to move in the direction opposite to the gradient so as to minimize the function as quickly as possible. Thus, we build a sequence of approximations $x^{(k)}$ that converges to the solution x^*. In the case of a quadratic form, the gradient is easy to compute: $\nabla f(x^{(k)}) = 2Ax^{(k)} - 2b$. Then the iterative descent method can be instantiated as follows:

- Make an initial guess $x^{(0)} \leftarrow \vec{0}$;
- Iterate until convergence:

$$\begin{aligned} r^{(k)} &\leftarrow b - Ax^{(k)} \\ \alpha^{(k)} &\leftarrow \frac{r^{(k)\top} r^{(k)}}{r^{(k)\top} A r^{(k)}} \\ x^{(k+1)} &\leftarrow x^{(k)} + \alpha^{(k)} r^{(k)} \end{aligned}$$

7.3.1 Choosing the step size $\alpha^{(k)}$

Since we are moving in the opposite direction to the gradient, we are sure to decrease f if we make a sufficiently small step. That is being said, in our case, we minimize a quadratic form, which, restricted to the search line, remains a quadratic function (a parabola). It is therefore possible to find directly the optimal step that minimizes $f(x^{(k+1)})$ along the direction $r^{(k)}$. This can be done by canceling the derivative $\frac{df\left(x^{(k+1)}\right)}{d\alpha^{(k)}}$ with $x^{(k+1)} = x^{(k)} + \alpha^{(k)} r^{(k)}$.

We are looking for $\alpha^{(k)}$ such that $x^{(k)} + \alpha^{(k)} r^{(k)}$ minimizes f along the straight line generated by the point $x^{(k)}$ and the vector $r^{(k)}$. To do this, we will cancel the derivative of f along this line: we want the dot product between the gradient and the search direction to vanish in $x^{(k+1)}$.

Here, the search direction $d^{(k)}$ is equal to the residual $r^{(k)}$, however we keep different notations to preserve the semantics, which will be useful later on to combine the different approaches. The step size can be derived from the zero dot product condition as follows:

$$\begin{aligned} d^{(k)\top} r^{(k+1)} &= 0 \\ d^{(k)\top} \left(b - Ax^{(k+1)}\right) &= 0 \\ d^{(k)\top} \left(b - A(x^{(k)} + \alpha^{(k)} r^{(k)})\right) &= 0 \\ \alpha^{(k)} &= \frac{d^{(k)\top} \left(Ax^{(k)} - b\right)}{d^{(k)\top} A r^{(k)}} \\ \alpha^{(k)} &= \frac{d^{(k)\top} r^{(k)}}{d^{(k)\top} A r^{(k)}} \end{aligned}$$

7.3.2 Convergence: stopping criterion

There is no miracle solution. We can stop when the norm of the gradient becomes too small, or the when the difference $f\left(x^{(k)}\right) - f\left(x^{(k+1)}\right)$ (eventually normalized) is getting too small.

The figures 7.2 and 7.3 show $2D$ and $3D$ examples of the gradient descent behaviour. When all eigenvalues of A are equal, the algorithm converges in one iteration, but when they are very different, the speed of convergence drastically decreases. The maximum ratio between the eigenvalues is called the matrix conditioning, and reflects well the difficulty of the problem.

To sum up, *the gradient descent is a very simple method, and has a very fast convergence rate during first iterations. After few iterations, its behaviour becomes very sensitive to the matrix conditioning.*

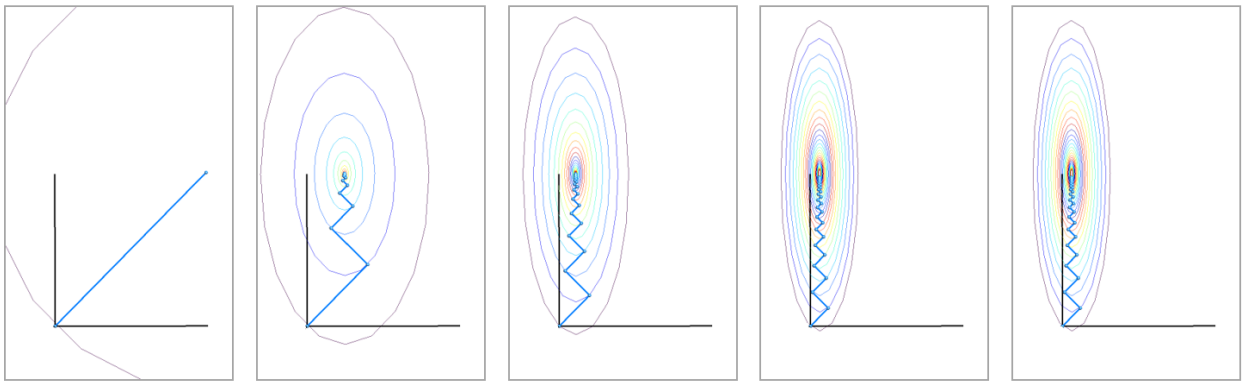


Figure 7.2: Optimal step gradient descent with a diagonal matrix A, where a_{00} takes values from 1 (left) to 5 (right), $a_{11} = 1$ and $a_{10} = a_{01} = 0$. The blue segments connect the $x^{(k)}$ to the $x^{(k+1)}$, and the ellipses show the iso-values of $f(x) = x^\top Ax - 2b^\top x$.

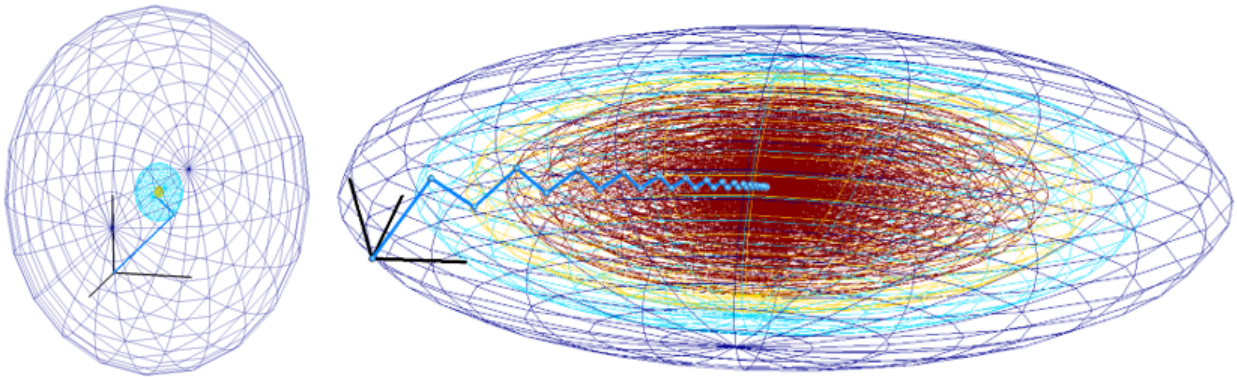


Figure 7.3: A 3D example of gradient descent with a diagonal matrix A. We observe an extremely fast convergence when A is close to the identity (left), and a particularly slow convergence when its diagonal elements are $0.5, 1$ and 2 (right).

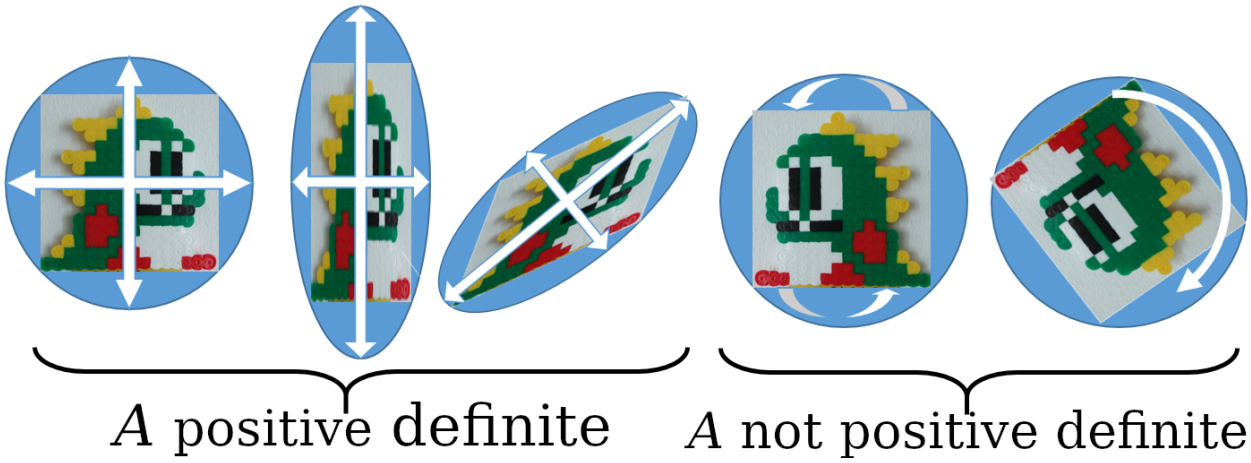


Figure 7.4: Few 2D linear maps transforming an image. From left to right: the identity matrix does not change the image, more generally, a diagonal matrix stretches the image along the coordinate axes. Even more generally, all symmetric positive definite define a stretch along orthogonal directions. Antisymmetric matrix flips axes (negative coefficient stretching), and a rotation matrix "stretches" with a complex coefficient.

7.4 The 2nd way: the (naïve) conjuguation method

Let us put aside the minimization for a moment, and cosider a projection method that solves $Ax = b$ directly. The idea is to build an orthogonal basis and project the solution onto this basis in n iterations. Before doing so, let us build few handy tools first.

7.4.1 A very useful linear map $M = \sqrt{A}$

In this method a central role is played by the map M that we can loosely describe as $M := \sqrt{A}$. The idea is to make orthogonal projections in the space transformed by the map M.

> We define M as a symmetric matrix having the same eigenvectors as A but whose eigenvalues are the square roots of those of A. This matrix is symmetric positive definite with real entries.

The (slightly abusive) notation $M := \sqrt{A}$ comes from the fact that $M^\top M = MM = A$ as you can see:

- either via a *geometrical intuiton*: as we will see shortly, A has a full set of real-valued eigenvectors forming an orthonormal basis $\{v_i\}_{i=1}^n$. We can decompose any vector x in this basis, therefore $Ax = \sum_{i=1}^n (x^\top v_i)\lambda_i v_i$. This means that the linear map defined by A is simply a stretching of space in the directions of each eigenvector, with a factor given by the associated eigenvalue (Figure 7.4).

  Than the linear map M can be seen as two consequtive stretchings of space along the same vectors, but with corresponding factors $\sqrt{\lambda_i}$. This is equivalent to stretching by λ_i (Figure 7.5).

- either *analytically:* since A is symmetric positive definite, it has a full set of real-valued eigenvectors forming an orthonormal basis. Let v_i and v_j be two eigenvectors associated with distinct eigenvalues $\lambda_i \neq \lambda_j$: $Av_i = \lambda_i v_i$ and $Av_j = \lambda_j v_j$. The symmetry of A implies that:

$$\begin{aligned} v_i^\top A v_j &= v_j^\top A v_i \\ v_i^\top \lambda_j v_j &= v_j^\top \lambda_i v_i \\ (\lambda_j - \lambda_i)\left(v_i^\top v_j\right) &= 0. \end{aligned}$$

  Thus, if v_1 and v_2 correspond to two distinct eigenvalues, they are orthogonal. Even if there is an infinity of eigenvectors (due to multiple eigenvalues), it can be shown that A has a full orthonormal basis of eigenvectors.

  Then let us examine the action of M:

$$M^\top Mx = MMx = \sum_{i=1}^n \sum_{j=1}^n \sqrt{\lambda_i}\left(\sqrt{\lambda_j} x^\top v_j\right) v_i = \sum_{i=1}^n \lambda_i v_i \left(x^\top v_i\right) = Ax.$$

  Moreover, as A is positive definite, it means that $x^\top Ax > 0, \forall x \neq 0$. In particular, for an eigenvector v with associated eigenvalue λ (i.e., $Av = \lambda v$), we have $v^\top \lambda v > 0$. This implies that $\lambda > 0$ (and, by the same token, that $\lambda \in \mathbb{R}$). Therefore, all eigenvalues of A are real and positive, and the square roots are well-defined.

7.4.2 A-orthogonality

Note that although we define M, we never compute it explicitly. The interest of defining M is that one can easily compute dot products in the space transformed by M. Indeed, if we want to calculate the dot product between the images of two vectors u and v under the map M, we need to evaluate $(Mu)^\top Mv = u^\top Mv = u^\top Av$. Note that it can be done without having M computed! As a side note, A defines the metric tensor of the linear map associated with M: it is a bilinear map that allows to measure the dot product (thus lengths and angles) in the space transformed by M.

So, we can very efficiently compute $(Mu)^\top Mv = u^\top Av$, the dot product of Mu and Mv, images of any two vectors u and v. It is therefore very easy to know whether Mu and Mv are orthogonal ($u^\top Av$ =0). Moreover, it is also simple to compute the norm of a transformed vector: $\|Mu\| = \sqrt{u^\top Au}$. **Note that even if we do not know the true soluton $x^* := A^{-1}b$, we can efficiently compute $(Mu)^\top Mx^* = u^\top b$ for any vector u,** and this is the key property for recovering the solution itself.

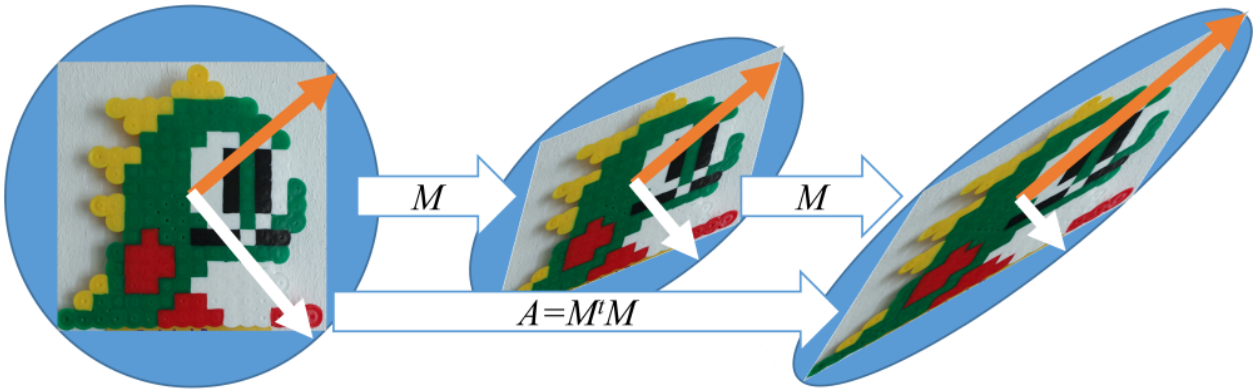


Figure 7.5: The linear map A can be seen as a repeated action of M. The eigenvectors (orange and white arrows) of A and M are the same: only the eigenvalues are different (squared).

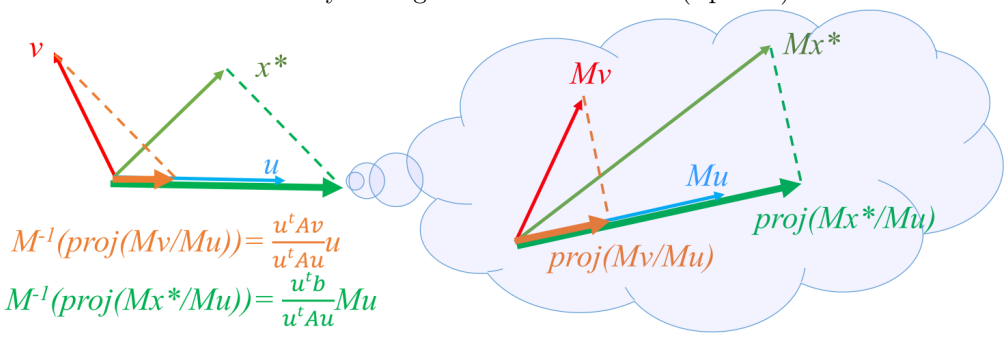


Figure 7.6: Let u and v be two arbitrary vectors and x^* the solution of $Ax - b = 0$. We can easily calculate dot products $Mu \cdot Mv$ and $Mu \cdot Mx^*$. This makes it possible to project v and x^* onto u in a way that would be orthogonal in M.

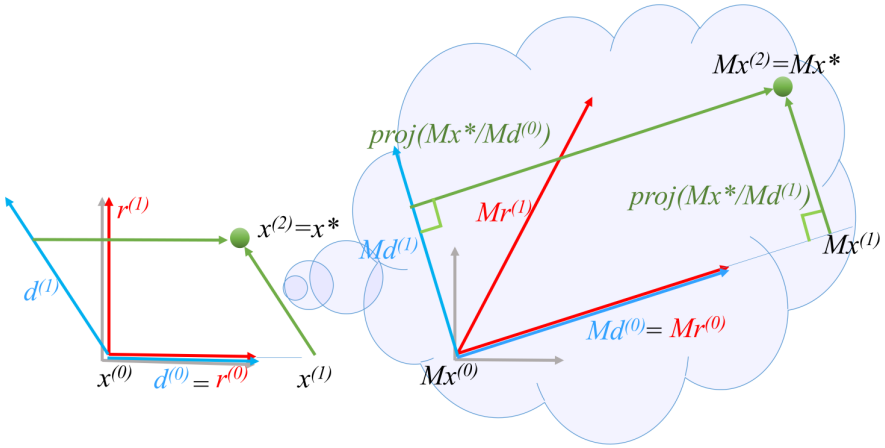


Figure 7.7: The conjuguation method in 2D. We start from two arbitrary linearly independent vectors $r^{(0)}$ and $r^{(1)}$, and then we define $d^{(0)} \leftarrow r^{(0)}$ and $d^{(1)} \leftarrow M^{-1}\left(Mr^{(1)} - proj\left(Mr^{(1)}/Md^{(0)}\right)\right)$. Note that $d^{(0)}$ and $d^{(1)}$ form an A-orthogonal basis. Then the solution x^* can be decomposed in this basis as $x^* \leftarrow \Sigma_k M^{-1} proj\left(Mx^*/Md^{(k)}\right)$.

In practice, we will use the above properties to build an A-orthogonal basis: for any two vectors u and v from this basis, $Mu \perp Mv$. This basis will help us greatly, since we know to project easily the (unknown) solution x^* onto the basis (Figure 7.6).

7.4.3 Back to the naïve conjuguation method

Recall that we have put aside for a moment the minimization of $x^\top Ax - 2b^\top x$, and the goal is to build a direct way to solve for $Ax = b$.

We want to build an orthogonal basis and project the solution onto this basis, what can be done in n iterations. To do so, we start from an arbitrary set of linearly independent vectors $\left\{r^{(k)}\right\}_{k=0}^{n-1}$ and use a variant of the Gram-Schmidt process to build a set of A-orthogonal vectors $\left\{d^{(k)}\right\}_{k=0}^{n-1}$. It can be thought as the ordinary Gram-Schmidt normalization of the set $\left\{Mr^{(k)}\right\}_{k=0}^{n-1}$ that builds an orthogonal set $\left\{Md^{(k)}\right\}_{k=0}^{n-1}$. Recall that we never actually compute M; we find it easier to reason in terms of an orthogonal basis rather than manipulate A-orthogonality, even if we do actually build an A-orthogonal basis $\left\{d^{(k)}\right\}_{k=0}^{n-1}$. Finally, we compute an orthogonal projection of Mx^* onto $\left\{Md^{(k)}\right\}_{k=0}^{n-1}$ and, by the linearity of M, we deduce the contribution of every $d^{(k)}$ to x^*.

So, we need to **(1)** construct the basis and **(2)** project the solution onto it:

1. **Basis construction.** Let us take any set of n linearly independent vectors $\left\{r^{(k)}\right\}_{k=0}^{n-1}$, for example:

$$r_i^{(k)} = \begin{cases} 1 & \text{si } i = k \\ 0 & \text{si } i \neq k \end{cases}$$

Since M is (strictly) positive definite, $\left\{Mr^{(k)}\right\}_{k=0}^{n-1}$ are also linearly independent.

Gram-Schmidt process builds the set incrementally, and only needs to know how to evaluate dot products. Geometrically, this method proceeds as follows: first we define $d^{(0)} := r^{(0)}$ (and thus $Md^{(0)} = Mr^{(0)}$). and then to compute $Md^{(k)}$, first it projects orthogonally $Mr^{(k)}$ onto the subspace span $\left\{Md^{(0)} \dots Md^{(k-1)}\right\}$. The vector $Md^{(k)}$ is then defined to be the difference between $Mr^{(k)}$ and this projection, guaranteed to be orthogonal to all of the vectors in the above subspace.

Refer to Figure 7.6 for an illustration; recall that the projection of a vector v onto the system axis represented by the vector u is the vector u scaled by the factor $v\cos(\angle(u,v))$. We can find it by computing two dot products: $proj(v,u) := \frac{u^\top v}{u^\top u}u$. Here, we use this equation with $v = Mr^{(k+1)}$ and $u = Md^{(i)}$ in order to project the new direction onto the preceding ones and thus decompose $Mr^{(k+1)}$ into $Md^{(0)} \dots Md^{(k)}$.

$$\begin{aligned} Md^{(k+1)} =& Mr^{(k+1)} - \sum_{i=0}^{k} \frac{\left(Mr^{(k+1)}\right)^\top Md^{(i)}}{\left(Md^{(i)}\right)^\top Md^{(i)}} Md^{(i)} \\ =& Mr^{(k+1)} - \sum_{i=0}^{k} \frac{r^{(k+1)\top} Ad^{(i)}}{d^{(i)\top} Ad^{(i)}} Md^{(i)} \end{aligned}$$

This relation is interesting, however our real uknowns are $d^{(k)}$ and not $Md^{(k)}$. So, we multiply both parts of the equation by M^{-1} (ref. to Figure 7.6):

$$d^{(k+1)} \leftarrow r^{(k+1)} - \sum_{i=0}^{k} \frac{r^{(k+1)\top} Ad^{(i)}}{d^{(i)\top} Ad^{(i)}} d^{(i)}. \tag{7.1}$$

This is the formula that will be used in the algorithm because it no longer contains the matrix M we don't know, but only A and the previous vectors $d^{(0)} \dots d^{(k)}$.

2. **Projection of Mx^* onto this basis.** As before, two dot products allow us to find projection on each of the vectors:

$$\begin{aligned} Mx^* &= \sum_{k=0}^{n-1} \frac{(Mx^*)^\top Md^{(k)}}{(Md^{(k)})^\top Md^{(k)}} Md^{(k)} \\ &= \sum_{k=0}^{n-1} \frac{x^* Ad^{(k)}}{d^{(k)\top} Ad^{(k)}} Md^{(k)} \\ &= \sum_{k=0}^{n-1} \frac{b^\top d^{(k)}}{d^{(k)\top} Ad^{(k)}} Md^{(k)}. \end{aligned}$$

Again, we can't do this calculation since we don't know M, but we can find x^* directly by multiplying the formula on either side by M^{-1}:

$$x^* \leftarrow \sum_{k=0}^{n-1} \frac{b^\top d^{(k)}}{d^{(k)\top} Ad^{(k)}} d^{(k)}.$$

Note that we do need to calculate the full set $\left\{d^{(k)}\right\}_{k=0}^{n-1}$ prior projecting Mx^* onto the $Md^{(k)}$. To make the conjugation algorithm closer to the conjugate gradient algorithm, we regroup the two loops into the single one; then the algorithm can be written as follows:

- $d^{(0)} \leftarrow r^{(0)}$
- $x^{(0)} \leftarrow \vec{0} + \frac{b^\top d^{(0)}}{d^{(0)\top} Ad^{(0)}} d^{(0)}$
- for $k \in 1 \dots n-2$:
  - $d^{(k+1)} \leftarrow r^{(k+1)} - \sum_{i=0}^{k} \frac{r^{(k+1)\top} Ad^{(i)}}{d^{(i)\top} Ad^{(i)}} d^{(i)}$
  - $x^{(k+1)} \leftarrow x^{(k)} + \frac{b^\top d^{(k)}}{d^{(k)\top} Ad^{(k)}} d^{(k)}$

To sum up, we can solve $Ax = b$ by building an orthogonal base in M and projecting the image of the solution on it just by computing dot products in M. Figure 7.7 provides an illustration.

This algorithm has a $O(n^3)$ complexity because of the necessity to reiterate Gram-Schmidt every time when we choose a new direction, which is regrettable. This algorithm should not be implemented because it is very inefficient. It forms, however a perfect basement for the conjugate gradient method.

7.5 The 3rd and final algorithm: the conjugate gradient

The conjugate gradient differs from the previous algorithm in the way it constructs the $r^{(k)}$. Instead of taking an arbitrary basis, we define it to be the residual at each iteration $r^{(k)} = b - Ax^{(k)}$. Recall that the gradient of the corresponding quadratic form is collinear with the residual, hence the name:

$$\nabla \left\{x^\top Ax + 2b^\top x\right\} \left(x^{(k)}\right) = -2r^{(k)}.$$

In this way, we combine the gradient descent and the naïve conjugation to obtain the conjugate gradient method. This choice of $r^{(k)}$ offers two advantages: it cancels most of the terms in the equation (7.1) and allows the algorithm to converge numerically long before reaching the n-th iteration. So, while the conjuguate gradient algorithm can be seen as a direct solving method (if n iterations are performed), the most important contributions are made in the beginning, therefore it can also be seen as an iterative method.

We can rewrite the naïve conjuguation algorithm as follows:

line	Direct adaptation	Optimized adaptation
1	$x^{(0)} \leftarrow 0$	$x^{(0)} \leftarrow 0$
2	$r^{(0)} \leftarrow b - Ax^{(0)}$	$r^{(0)} \leftarrow b - Ax^{(0)}$
3	$d^{(0)} \leftarrow r^{(0)}$	$d^{(0)} \leftarrow r^{(0)}$
4	For $k \in 0 \dots n-1$:	For $k \in 0 \dots n-1$:
5	$\quad \alpha^{(k)} \leftarrow \frac{b^\top d^{(k)}}{d^{(k)\top} A d^{(k)}}$	$\quad \alpha^{(k)} \leftarrow \frac{r^{(k)\top} r^{(k)}}{d^{(k)\top} A d^{(k)}}$
6	$\quad x^{(k+1)} \leftarrow x^{(k)} + \alpha^{(k)} d^{(k)}$	$\quad x^{(k+1)} \leftarrow x^{(k)} + \alpha^{(k)} d^{(k)}$
7	$\quad r^{(k+1)} \leftarrow b - Ax^{(k+1)}$	$\quad r^{(k+1)} \leftarrow r^{(k)} - \alpha^{(k)} A d^{(k)}$
8	$\quad \beta_i^{(k+1)} \leftarrow \frac{r^{(k+1)\top} A d^{(i)}}{d^{(i)\top} A d^{(i)}}, \forall i \le k$	$\quad \beta^{(k+1)} \leftarrow -\frac{r^{(k+1)\top} r^{(k+1)}}{r^{(k)\top} r^{(k)}}$
9	$\quad d^{(k+1)} \leftarrow r^{(k+1)} - \sum_{i=0}^{k} \beta_i^{(k+1)} d^{(k)}$	$\quad d^{(k+1)} \leftarrow r^{(k+1)} - \beta^{(k+1)} d^{(k)}$

The right column presents an optimized version of the algorithm that we will describe in details shortly. Note the drastic change in complexity: the performance-killing sum in the last line disappears in the optimization! While the left column is quite straightforward to obtain, the optimized version requires a little bit more analysis. The most striking property allowing us to perform this optimization is the orthogonality of the residuals:

$$\boxed{{r^{(k)}}^\top r^{(i)} = 0 \quad \forall k < i}$$

It is easy to prove: recall that once we take a step in a search direction, we need never step in that direction again, i.e. $\left(Mx^{(k)} - Mx^*\right) \perp \text{span}\left\{Md^{(0)}, \dots Md^{(k-1)}\right\}$, it means that the error $x^{(k)} - x^*$ is A-orthogonal to the subspace $D^{(k-1)} := \text{span}\left\{d^{(0)}, \dots d^{(k-1)}\right\}$. Since the residual $r^{(k)} = A\left(x^* - x^{(k)}\right)$, then $r^{(k)}$ is orthogonal to $D^{(k-1)}$. Let us note that $D^{(k-1)} = \text{span}\left\{r^{(0)}, \dots r^{(k-1)}\right\}$, it can be easily demonstrated by induction using the fact that $d^{(j)}$ is a linear combination of $r^{(j)}$ and $d^{(i)}$ with $i < j$. We can conclude then that the set of residuals $\left\{r^{(k)}\right\}_{k=0}^{n-1}$ form an orthogonal basis, and it is conjugated to the basis $\left\{d^{(k)}\right\}_{k=0}^{n-1}$, who, in its turn, becomes orthogonal under the action of M, refer to Figure 7.8 for an illustration.

Armed with this key property, let us review all the transformations leading to the optimized version of the algorithm:

- **Line 5: step size $\alpha^{(k)}$.** The optimal step size can be computed both via the conjuguation and the gradient descent:

$$\boxed{\alpha^{(k)} = \frac{r^{(k)\top} r^{(k)}}{d^{(k)\top} A r^{(k)}} = \frac{b^\top d^{(k)}}{d^{(k)\top} A d^{(k)}}}$$

  We compute the step $\alpha^{(k)}$ so that the gradient at the new point is orthogonal to the search direction. For the gradient descent we look for $d^{(k)} \perp r^{(k+1)}$, and for the conjugation method we look for $Md^{(k)} \perp Mr^{(k+1)}$. Since placing $x^{(k+1)}$ at the minimum of $\|Ax - b\|^2$ is equivalent to placing $Mx^{(k+1)}$ at the minimum of $\|Ax - b\|^2$ transformed by M, we obtain the same $\alpha^{(k)}$ in both cases.

  It is easy to see that $d^{(k)\top} A r^{(k)} = d^{(k)\top} A d^{(k)}$: to construct $Md^{(k)}$ with Gram-Schmidt, a vector of $MD^{(k-1)}$ was subtracted from $Mr^{(k)}$, leaving the same the dot product with $Md^{(k)}$. We have therefore the equailty of the denominators. Knowing that $x^{(k)} \in D^{(k-1)}$, we also determine the equality of the numerators: $b^\top d^{(k)} = (b - Ax^{(k)})^\top d^{(k)} = r^{(k)\top} d^{(k)} = r^{(k)\top} r^{(k)}$. The last equality comes from the fact

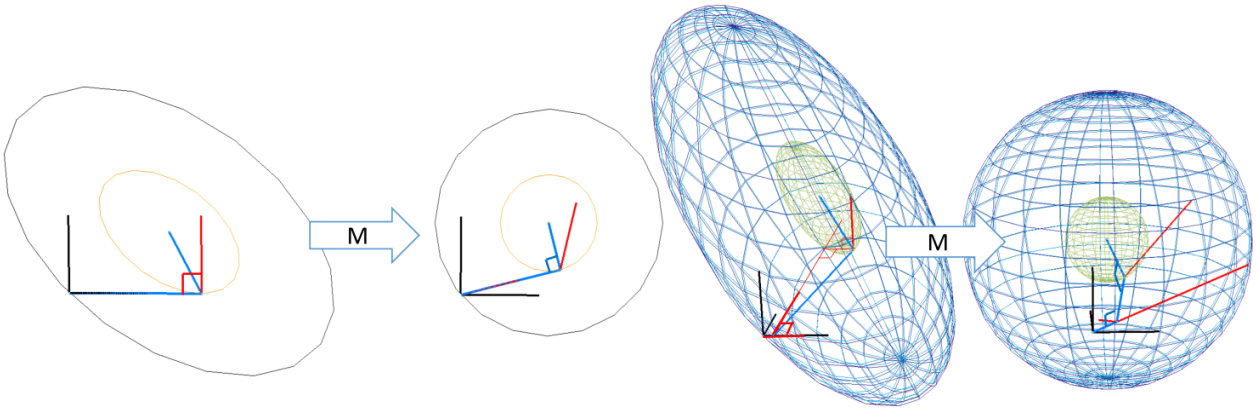


Figure 7.8: The vectors $\left\{r^{(k)}\right\}_{k=0}^{n-1}$ (in red) are mutually orthogonal, as well as the vectors $\left\{Md^{(k)}\right\}_{k=0}^{n-1}$ (in blue) are mutually orthogonal in the space deformed by M.

that the residuals are mutually orthogonal. More presicely, to find $d^{(k)}$ with aid of the Gram-Schmidt process, we have subtracted a vector lying in $D^{(k-1)}$ from $r^{(k)}$; keeping in mind that $r^{(k)} \perp D^{(k-1)}$, it therefore does not modify the dot product $r^{(k)\top} r^{(k)}$, leaving it equal to $d^{(k)\top} r^{(k)}$.

- **Line 7: recursive definition of $r^{(k)}$.** $\boxed{r^{(k+1)} = r^{(k)} + \alpha^{(k)} A d^{(k)}}$

  By definition $r^{(k+1)} = b - Ax^{(k+1)}$. We know that $x^{(k+1)} = x^{(k)} + \alpha^{(k)} d^{(k)}$, thus $r^{(k+1)} = b - A(x_i + \alpha^{(k)} d^{(k)}) = r^{(k)} + \alpha^{(k)} A d^{(k)}$.

- **Lines 8 and 9: massive cancellation of betas.** The first two simplifications are simple and only slightly affect performance. The last simplification is fundamental because it changes each iteration in $O(n)$ instead of $O(n^2)$ (with sparse matrices): it uses the orthogonality of the residuals to simplify the writing of $\beta_j^{(k+1)}$.

  By definition of betas, we have:

$$\begin{aligned}
\beta_i^{(k+1)} &:= \frac{r^{(k+1)\top} A d^{(i)}}{d^{(i)\top} A d^{(i)}} \\
&= \frac{r^{(k+1)\top} (r^{(i)} - r^{(i+1)})}{\alpha^{(i)} d^{(i)\top} A d^{(i)}} \\
&= \frac{r^{(k+1)\top} r^{(i)} - r^{(k+1)\top} r^{(i+1)}}{\alpha^{(i)} d^{(i)\top} A d^{(i)}},
\end{aligned}$$

  where the first transformation is made thanks to the recursive definition of the residuals $r^{(i+1)} = r^{(i)} + \alpha^{(i)} A d^{(i)}$. Then, by orthogonality of the residuals, we have:

$$\boxed{\begin{aligned}
\beta_i^{(k+1)} &= \begin{cases} \frac{-r^{(k+1)\top} r^{(k+1)}}{\alpha^{(k)} d^{(k)\top} A d^{(k)}}, & i = k \\ 0, & i < k \end{cases} \\
&= \begin{cases} \frac{-r^{(k+1)\top} r^{(k+1)}}{r^{(k)\top} r^{(k)}}, & i = k \\ 0, & i < k \end{cases}
\end{aligned}}$$

  The last transformation was obtained by plugging the optimized definition of $\alpha^{(k)}$ into the right-hand side of the equation.

  To sum up, by orthogonality of the residuals, $\beta_i^{(k+1)} = 0$ if $k \neq i$, which makes it possible to simplify the notations by omitting the i: we use $\beta^{(k+1)}$ instead of $\beta_k^{(k+1)}$ because all the other $\beta_i^{(k+1)}$ are null.

7.6 Performance

In the absence of rounding errors, the conjugate gradient converges in n iterations; each iteration requires only one matrix-vector multiplication of five dot products. In practice, the conjugate gradient is used on large sparse matrices as an iterative solver that is less sensitive to matrix conditioning than the gradient descent.

It is interesting to observe the behaviour of the conjugate gradient on this type of matrices. In this section we present two simple cases: a 1D Laplacian and a 2D Laplacian discretized on a regular grid. Although simple, these choices show the impact of a predominant factor for the solvers behaviour: the arrangement of zeros in the matrix, which corresponds to the geometric neighborhoods in the discretized space.

The sparse matrices define graphs that link the indices i and j by non-zero matrix entries a_{ij}. In finite element modeling (FEM) and in geometry processing, these graphs are very close to the structure of the underlying mesh. Indeed, by their definition, derivatives are estimated on neighbouring elements. These two example allow us to see how the algorithms we have just presented behave with this type of data.

7.6.1 1D Laplacian

We represent a function f by the vector x such that $f(i) = x_i$. We can estimate its second derivative by finite differences; the problem is to find a function (the vector x) whose second derivative is null, and which respects boundary constraints shown in Figure 7.9.

This problem is very interesting because it is relatively simple to analyze. In addition, we can visualize the behavior of the various algorithms on a single image (Figure 7.10): we display $x_i^{(k)}$ as a function of the position i and iteration k. Moreover, this problem is extremely complex to solve because the conditioning of the system is very bad: for $n = 5$ it is $13.6 = 1.9/0.14$ and for $n = 20$, it's equal to $200 = 2/.01$. Geometrically, this corresponds to minimizing a quadratic function $x^\top Ax - 2b^\top x$ whose iso-values are ellipses with aspect ratios of 13.6 and 200!

For each algorithm, we observe following behaviour (Figure 7.10):

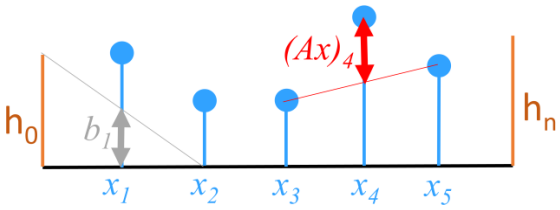
- Gradient descent quickly minimizes the second derivative near the constraints, but struggles to propagate the constraints into the domain.
- The projection method propagates the left-hand constraint as it goes along, but does not take into account the right-hand constraint until the very last iteration. This behavior is certainly due to the choice of $r^{(k)}$, but illustrates the necessity to perform all n iterations.
- The conjugate gradient behaves as if it only sees a small region around the constraints (shown in orange at the edges), where it perfectly solves the problem. This region grows at each iteration to cover the whole domain at $n/2$ iterations, but it is necessary to wait for all the n iterations before each of the two constraints has had time to influence the whole domain.

7.6.2 2D Laplacian

The 2D Laplacian works on the same principle as the 1D Laplacian. We have discretized a function f on a 2D grid and each entry of the vector x is associated with each node of the grid. The Laplacian is estimated by finite difference, so the matrix A is defined by: $a_{ii} = 1$, $a_{ij} = -1/4$ if the nodes i and j are neighbors, and $a_{ij} = 0$ otherwise.

In the 2D case, we can no longer represent the simulation on a single image, so we display the state at certain iterations. Thanks to the Figure 7.11, we can analyze the behavior of the algorithms:

- Gradient descent: like with 1D Laplacian, we observe that the first iterations smooth out the function well, but that the solver struggles to propagate the constraints towards the interior of the domain. This being said, in 100 iterations, the convergence on this 20×20 grid seems similar to that observed on the 1D grid of $n = 20$. The difficulty does not seem to be due to 400 variables, but rather to the distance to the constraints (which is about the same in 1D and 2D).
- Conjugation: the solver has set the x_i one after the other, and did not converge because we stopped it before the necessary 400 iterations.



$$A = \begin{pmatrix} 1 & -1/2 & 0 & 0 & 0 \\ -1/2 & 1 & -1/2 & 0 & 0 \\ 0 & -1/2 & 1 & -1/2 & 0 \\ 0 & 0 & -1/2 & 1 & -1/2 \\ 0 & 0 & 0 & -1/2 & 1 \end{pmatrix} \quad ,b= \begin{pmatrix} h_0/2 \\ 0 \\ 0 \\ 0 \\ h_n/2 \end{pmatrix}$$

Figure 7.9: 1D Laplacian problem. We have a function defined in $n = 5$ points; the goal is to minimize the second derivative estimated by finite differences i.e., $\Sigma_i(x_i - (x_{i-1} + x_{i+1})/2)$. At the boundary we fix values h_0 and h_n, which gives the problem $Ax - b = 0$ with A and b shown in the figure.

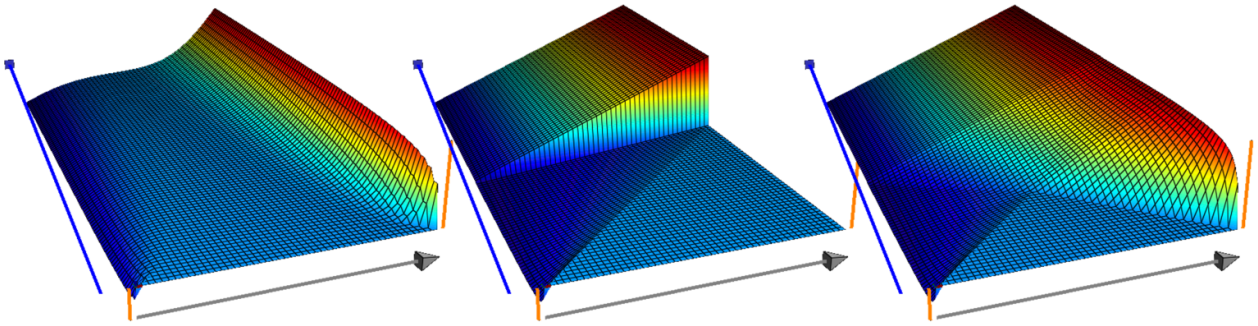


Figure 7.10: Evolution of the solution with $n = 20$: the gray axis represents the function sampled by x, and the blue axis is the iteration axis, the orange segments are the boundary constraints. Three solvers are (from left to right) the gradient descent, the conjugation and the conjugated gradient.

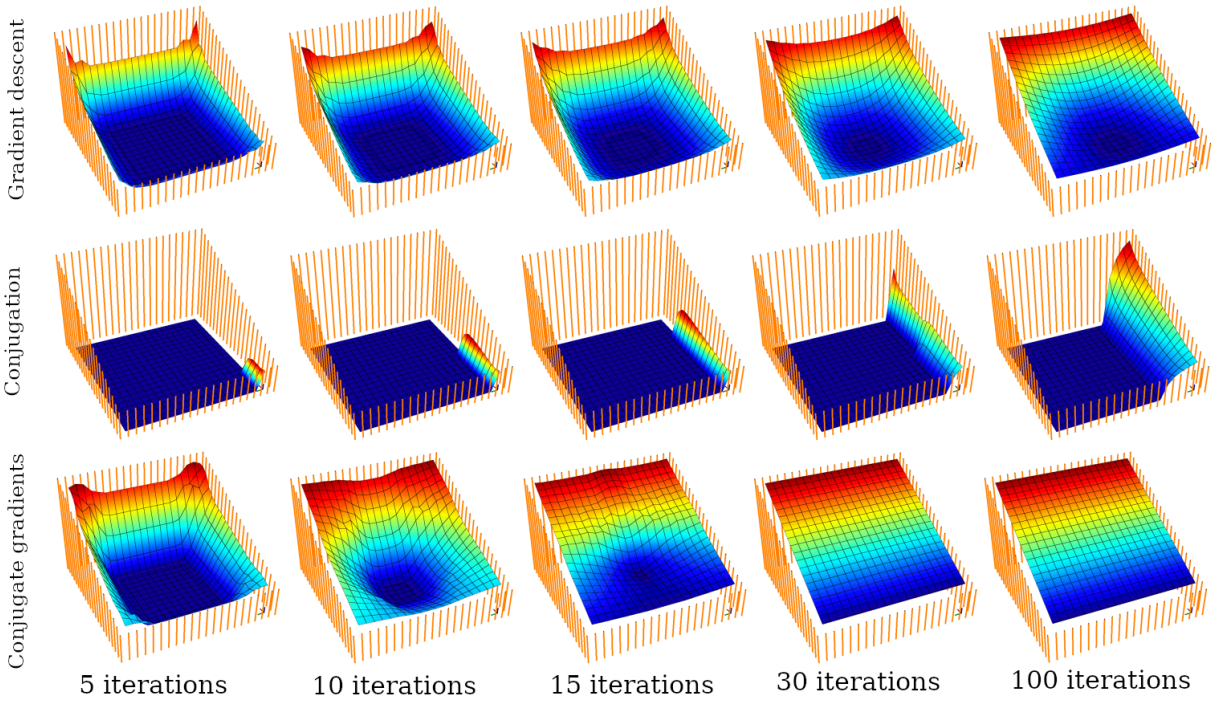


Figure 7.11: Evolution of the 2D Laplacian solution on a 20×20 grid. The solution x is displayed by the height of the function, and the boundary constraints are given by the orange segments.

- Conjugate gradient: it converges in less than 30 iterations, which is far from the 400 iterations necessary (direct method) to guarantee convergence. This behavior can be explained by the time needed for each constraint to influence the whole domain.

7.6.3 Conclusion

The difficulty in solving problems with sparse matrix A is strongly related to the maximum distance (w.r.t. the adjacency graph defined by A) at which a constraint can influence the solution. In fact, for a similar number of variables, 1D problems appear to be more difficult than 2D problems.

7.7 Chapter summary

The conjugate gradient was presented as the fusion of two methods (gradient and conjugation). We saw what makes this method so effective, and observed its behaviour on classical problems in which A is a sparse matrix.

Let us summarize the content of the chapter:

- **Objective:** Solve $Ax = b$ with a symmetric positive definite matrix A
- **Gradient descent:** Solving $Ax - b = 0$ is equivalent to minimization of $x^\top Ax - 2b^\top x$
  - We can minimize with optimal step size.
  - Iterative method extremely dependent on the conditioning of A
- **Conjuguation:** There is M such that $M^\top M = A$.
  - We know how to modify a set of vectors $r^{(k)}$ to construct another set $d^{(k)}$, whose images by M are orthogonal, i.e. $Md^{(k)}.Md^{(i)} = 0$;
  - We know how to decompose the solution x^* into $d^{(k)}$ by projecting Mx^* onto $Md^{(k)}$.;
  - Direct method: requires n iterations, each taking $O(n)$ matrix-vector multiplications.
- **Conjuguated gradient (CG)** can be obntained by merging both methods:
  - CG is a gradient descent, where we rectify the descent direction so as not to return to an already optimized direction;
  - CG is a conjuguation algorithm whose $r^{(k)}$ are carefully chosen (the gradient) to get as close as possible to the solution;
  - CG is therefore a direct and iterative method **at the same time** that usually converges in a several iterations;
  - What makes it an efficient algorithm is that at step k the gradient is already orthogonal (in M) every $d^{(i)}$, except $d^{(k-1)}$. This allows us to do a constant number of matrix/vector multiplications per iteration.

Chapter 8

From least squares to neural networks

Nowadays machine learning is a very trendy topic, almost everyone wants to use it somehow, whether it is reasonable or not. Machine learning seems to be the answer to all the business prayers. It is amazing how people are now diving into the abyss of neural networks without ever looking back. However, it is much more surprising to witness the existence of two irreconcilable camps: those for whom neural networks are the answer to the meaning of life, the universe, and everything, and those who despise neural networks and deny the right to use the tool. We do not advocate for either party; two main points of this chapters are:

- neural networks are not the only machine learning tool;
- there is no clear boundary between least squares methods and neural networks.

Obviously, we cannot fit all of the data with a straight line, or a plane, but with powerful feature extractors, we may be able to reduce our problem to a much simpler one. To put it into perspective, this is what neural networks do effectively, the only difference being that we use some nonlinearity as the activation function in the last layer. If we would remove this, we could look at the last layer of the neural network as a least squares problem, i.e. fitting a plane on the data (activations from previous layers).

Note that this chapter is marked as optional for reading. We do not aim here to teach new tools / techniques, this chapter is here for cultural reasons. Anyhow, let us start with the simplest unidimensional problem.

8.1 Binary classification, the first attempt

The most simple, standard and yet quite common machine learning problem is binary classification. Many very interesting and important problems can be reduced to it. For example, we are given a set of data where n vectors are each marked with a "red" or "green" label. This is the simplest example, but in practice the red/green dataset can be really useful: for example, "spam"/"not spam" email classification, where the input vectors encode various characteristics of the content (e.g. the number of mentions of a certain magic carpet cleaning solution).

Our goal in binary classification is to build a function that takes a vector as an input and predicts its label. First we need to learn from the database: we need to build such a function whose predictions match the database labels. In the end, once the function is constructed, we can discard the database and use the function we have built as an oracle to predict labels for previously unseen vectors.

Long story short, let us consider the simplest example: we have n real numbers and the corresponding colors that we encode as 0 ("red") and 1 ("green"). In other words, we have a sequence $\{(x_i, y_i)\}_{i=1}^n$ as the input, where $x_i \in \mathbb{R}$ and $y_i \in \{0, 1\}$. How do we build the classifying function? Let us start with the basics: we can fit a straight line $y = wx + w_0$, to the set of input points $\{(x_i, y_i)\}_{i=1}^n$, thus performing an ordinary linear regression:

$$\arg\min_{w,w_0 \in \mathbb{R}} \sum_{i=1}^{n} (wx_i + w_0 - y_i)^2$$

Refer to Listing A.9 for the source code that fits a straight line onto the database shown in Figure 8.1.

Next we endow the straight line with the classification rule: if $y(x) > 1/2$ then x is green, otherwise it is red. Left image of Figure 8.1 provides an example: the colored dots show the database; the straight line is

the linear regression result, and the decision boundary is shown by the dashed line. All points on the right of the line are classified as green, and on the left as red. We can see that the database is perfectly learned, and there is no misclassification in the database (for the sake of simplicity, we do not make the distinction between the database we learn on and the database we use to assess the quality of the classifying function).

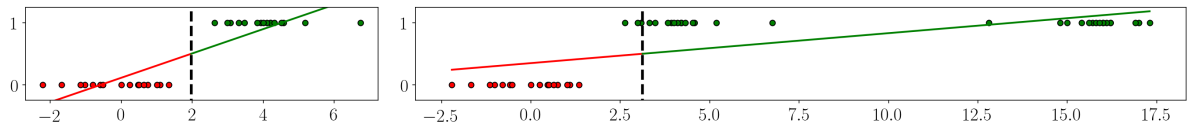


Figure 8.1: Ordinary linear regression as a binary classification function. The decision boundary is shown by the dashed line.

It may sound stupid, but ordinary linear regression is not always a bad choice for a binary classification. Of course, it only works well if certain assumptions about the input data are satisfied (e.g. independent and normally distributed class samples). If, however, these assumptions are not met, we may face problems, as illustrated in right image of Figure 8.1. In this example, we have added to the previous database few more "green" samples. This affects the linear regression, and we encounter misclassified database entries even if the database is perfectly separable. We can fix the situation by fitting a nonlinear model; to do so, first let us meet the logistic growth model before we return to the classification.

8.2 Logistic growth model

Logistic growth functions are useful as models accounting for constraints placed on the growth. An example is a bacteria culture allowed to grow under initially ideal conditions, followed by less favorable conditions that inhibit growth. Imagine a colony of the bacteria *B. dendroides* is growing in a Petri dish. The colony's area a (in square centimeters) can be modeled as a function of t, the elapsed time in hours.

$$a(t) = \frac{c}{1 + e^{-wt - w_0}}, \tag{8.1}$$

where c, w_0 and w are the parameters of the model. Here c is the carrying capacity, w_0 is the initial population size and w is the growth rate. This model was proposed in 1840s by a belgian mathematician Pierre François Verhulst.

Let us say that we want to recover the parameters of the model from an experiment: we have a series of n measurements (a_i, t_i), refer to the left image of Figure 8.2 for the scatter plot of the data.

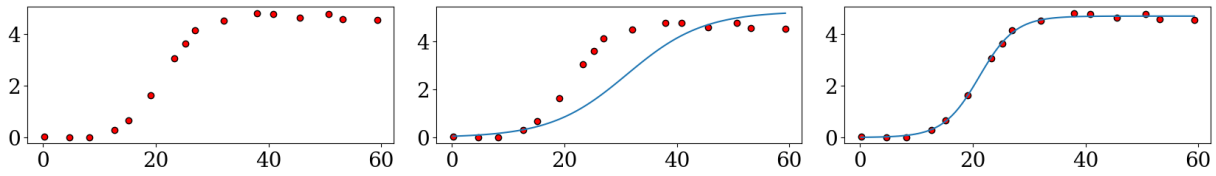


Figure 8.2: Logistic growth least squares fitting. **Left:** experimental data, **middle:** ordinary least squares initial guess, **right:** final fit after a series of least squares problems.

Under the assumption of independent and normally distributed errors, with 0 expectations and common variance, maximizing the likelyhood for the general nonlinear model (8.1) is equivalent to the minimization of the sum of squared errors:

$$S(c, w_0, w) = \sum_{i=1}^{n} (a(t_i) - a_i)^2 \tag{8.2}$$

There are multiple ways to deal with nonlinearities of the model, for example, we can try to apply some

transformation to the model prior the fitting. Let us transform the equation (8.1):

$$\begin{aligned} a(t) &= \frac{c}{1+e^{-wt-w_0}} \\ \frac{c}{a(t)-1} &= e^{-wt-w_0} \\ \log\left(\frac{c-a(t)}{a(t)}\right) &= -wt-w_0 \end{aligned}$$

If we have an estimation $c^{(0)}$ for the parameter c, this model is well suited for an ordinary linear regression. We can find the estimation $w_0^{(0)}$ and $w^{(0)}$ for the parameters w_0 and w by solving the following system in the least squares sense:

$$\begin{pmatrix} 1 & t_1 \\ 1 & t_2 \\ \vdots & \\ 1 & t_n \end{pmatrix} \begin{pmatrix} w_0^{(0)} \\ w^{(0)} \end{pmatrix} = \begin{pmatrix} \log \frac{a_1}{c^{(0)}-a_1} \\ \log \frac{a_2}{c^{(0)}-a_2} \\ \vdots \\ \log \frac{a_n}{c^{(0)}-a_n} \end{pmatrix} \tag{8.3}$$

How can we estimate $c^{(0)}$? Knowing that it corresponds to the carrying capacity, we can do a quick and dirty estimation $c^{(0)} := \max\limits_{i\in 1\ldots n} a_i + \varepsilon$, where a small constant ε is added in order to avoid division by zero in the equation (8.3). Refer to Listing A.10 for the source code; left image of Figure 8.2 shows the logistic curve fitted to our data. Of course, the fitting is far from being ideal: we have estimated the distance from the points a_i to the curve $a(t_i)$ in a very indirect and distoring way. Nevertheless, this fitting can be of use as a first guess for a further optimization. Let us denote by $r_i(b)$ the residual between the input label a_i and the prediction $a(t_i)$:

$$r_i(b) := a(t_i) - a_i,$$

where b is the set of the parameters of the model, in our example $b = (c, w_0, w)$. Then Equation (8.2) can be written in the following way:

$$S(b) = \sum_{i=1}^{n} r_i(b)^2 = \|\vec{r}(b)\|^2 \tag{8.4}$$

We can minimize $S(b)$ by the Gauß–Newton algorithm. The idea is extremely simple: starting from the initial guess $b^{(0)} := \left(c^{(0)}, w_0^{(0)}, w^{(0)}\right)$ that we have built by the ordinary least squares on the transformed model (or from three random values), we build a sequence $b^{(k+1)} = b^{(k)} + \Delta b^{(k)}$, where $\Delta b^{(k)}$ stands for the increment.To find the increment, we can linearize the function $\vec{r}$ at the point $b^{(k)}$:

$$\vec{r}(b) \approx \vec{r}\left(b^{(k)}\right) + J\vec{r}\left(b^{(k)}\right)\left(b - b^{(k)}\right),$$

where $J\vec{r}\left(b^{(k)}\right)$ is the Jacobian $n \times 3$ matrix evaluated at point $b^{(k)}$. Then we are looking for such an increment vector $\Delta b^{(k)}$ that the squared norm of the residual is minimized:

$$\min \left\| J\vec{r}\left(b^{(k)}\right)\Delta b^{(k)} - \vec{r}\left(b^{(k)}\right)\right\|^2$$

This again is an ordinary least squares problem equivalent to solving a 3×3 system of linear equations. Refer to Listing A.10 for the source code; right image of Figure 8.2 shows the logistic curve fitted to our data by solving a series of least squares problems.

8.3 Back to the binary classification: the 2nd attempt

Surely, you have already guessed where all this is going: we will put the logistic function inside the binary classification example from §8.1. So, we have a sequence $\{(x_i, y_i)\}_{i=1}^n$ as the input, where $x_i \in \mathbb{R}$ and $y_i \in \{0, 1\}$. We saw that ordinary linear regression is not an ideal choice for the classifying function, probably the logistic curve would do better. We already know to fit a logistic curve onto a set of points, so we can directly adopt Equation (8.4) with a slight modification: now we have two parameters w and w_0, We are fitting the curve $1/(1 + e^{-wt-w_0})$ that varies between 0 and 1, so we can remove the carrying capacity c from the equation.

Let the initial guess be $b^{(0)} := \left(w_0^{(0)}, w^{(0)}\right)$, where $w_0^{(0)}$ and $w^{(0)}$ are two arbitrary values (usually it makes sense to have small non-zero numbers). Then, just as before, we construct a sequence $b^{(k+1)} = b^{(k)} + \Delta b^{(k)}$, where the increment $\Delta b^{(k)}$ can be found through a least squares problem

$$\min \left\| J\vec{r}\left(b^{(k)}\right) \Delta b^{(k)} - \vec{r}\left(b^{(k)}\right) \right\|^2$$

In this example, $J\vec{r}\left(b^{(k)}\right)$ is the Jacobian $n \times 2$ matrix. Refer to Listing A.11 for the source code, the result is given in Figure 8.3. Compare the result to Figure 8.1: both datasets are well learned without any misclassification, so fitting a sigmoid shows a clear advantage over ordinary linear regression classifier.

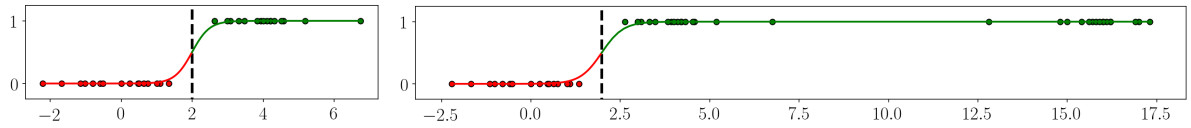


Figure 8.3: Binary classification made by fitting a logistic curve. The decision boundary is shown by the dashed line.

It turns out that we have just built and trained a neural network made of a single neuron, refer to Figure 8.4 for an illustration. 1-neuron perceptron with mean squared error loss function is equivalent to our nonlinear least squares problem.

While the example we have shown here is unidimensional, it is straightforward to generalize it: we can fit the m-dimensional logistic function $\frac{1}{1+e^{-w^\top x}}$ to the data, where w is a $(m+1)$-parameter vector, the last element of x is the constant 1, and $w^\top x$ is the corresponding linear combination. Note that the decision boundary is always linear: it is a point in 1D, a straight line in 2D, a plane in 3D and so forth. Refer to Figure 8.7 for an illustration, where the dashed line shows the decision boundary between two sets of 2D points.

In all our examples the training sets were linearly separable, and thus we did not have any misclassification in the training set. There is a caveat though. It might seem that linear separability is good news for binary classification: linear separability means that the problem is easy in some sense and that learning algorithms have a clear and achievable goal. Consider the fact that the decision boundary in a linear classifier is independent of the scale of the parameters. For instance, in the above example, the decision boundary is given by t such that $\frac{1}{2} = \frac{1}{1+e^{-wt-w_0}}$. Solving this simple equation for t we get $t = \frac{w_0}{w}$. The decision boundary would not move a tiny bit if we multiply both w_0 and w by a constant superior to 1. The boundary does not move, however, the scale does impact the likelihood in logistic regression by causing the logistic function to become more steep. If we have found such a decision boundary that all the samples from the learning database are correctly classified, we can infinitely scale the weights w and w_0, pushing the predictions closer and closer to their correct labels. The logistic function thus converges to the Heaviside step function. It is one of numerous possible instances of what is called overfitting: the model is becoming overconfident about

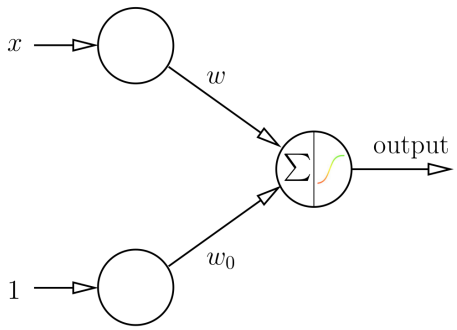


Figure 8.4: Single neuron perceptron shown in this image is equivalent to a nonlinear least squares problem; the decision boundary is always linear despite the dimensionality of the vector x.

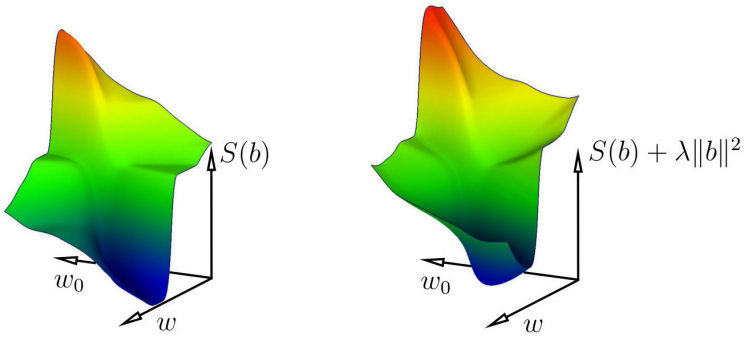


Figure 8.5: Energy plot as a function of parameters $b = (w_0, w)$ for the training database shown in the left image of Figure 8.3. Left image shows a plot of $S(b)$ without any regularization, the right one is a plot of the regularized energy $S(b) + 10^{-3}\|b\|^2$.

the data and uses very large weights to describe it. There is a very simple solution to this problem: regularize the weights. Essentially it means that we are going to find the MLE with a Gaussian prior on the weights:

$$\min \left\| J\vec{r}\left(b^{(k)}\right)\Delta b^{(k)} - \vec{r}\left(b^{(k)}\right)\right\|^2 + \lambda \left\| b^{(k)} + \Delta b^{(k)}\right\|^2$$

Figure 8.5 shows the energy plots with and without the regularization. It is easy to see that the regularized version has a unique minimum. The energy, however, is highly non-convex, making the numerical optimization a difficult problem. There are standard ways to overcome it, e.g., the Tikhonov regularization, whose idea is to make the steps $\Delta b^{(k)}$ small, thus working in a zone where the Jacobian $J\vec{r}\left(b^{(k)}\right)$ is a fair approximation, but these tricks are out of the scope of this document. There is another way to do it: change the loss function in our neural network.

8.4 Logistic regression: the 3rd attempt to binary classification

Logistic regression is another technique borrowed by machine learning from the field of statistics. As before, the basic idea of logistic regression consists in that the space of initial values can be divided by a linear boundary (i.e. a straight line) into two areas corresponding to classes. Simply answering "red" or "green" is pretty crude — especially if there is no perfect rule. Something wich takes a noise into account, and does not just give a binary answer, will often be useful. In short, we want probabilities — which means we need to fit a stochastic model.

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

$$p(y = 1|x, w) := \frac{1}{1 + e^{-w^\top x}}, \tag{8.5}$$

where w is a $(m + 1)$-parameter vector and the last element of x is the constant 1. Here we interpret the sigmoid $p(y = 1|x, w)$ as a conditional distribution of the response y, given the input variables. 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 want to fit the parameters of the sigmoid to match the data. Up to this moment, there are absolutely no changes w.r.t the previous section: we use the same one-neuron perceptron from Figure 8.4. The crucial difference comes from the way we will train it. In previous section we have minimized the mean squared error loss function, implicitly assuming presence of zero-mean Gaussian noise in the labels of the dataset. Our data, however is not real-valued but binary-valued, and thus Bernoulli's scheme is more suited for this situation.

Exactly like we have made it in §2.1, we can write the log-likelihood as

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

where $p_i(w) := p(y_i = 1|x_i, w)$. To fit the sigmoid to the data, we need to maximize the log-likelyhood.

All the hard decisions being made, it's time for basic calculus. 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 $m + 1$ nonlinear equations in w. We can rewrite the system as

$$\frac{\partial \log \mathcal{L}}{\partial w}(w) = X^\top (y - p) = 0, \tag{8.6}$$

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 (8.6) iteratively using Newton-Raphson steps:

$$w^{(k+1)} = w^{(k)} - \left(\frac{\partial^2 \log \mathcal{L}}{\partial w \partial w^\top}\left(w^{(k)}\right)\right)^{-1} \frac{\partial \log \mathcal{L}}{\partial w}\left(w^{(k)}\right) \tag{8.7}$$

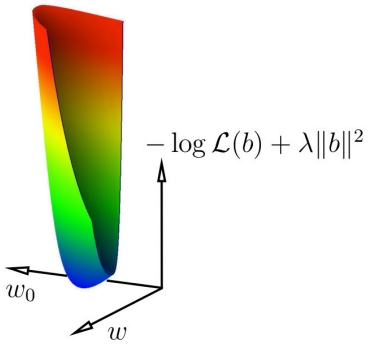


Figure 8.6: In the case of a logistic regression, cross-entropy is convex and very easy to optimize. Compare this plot with Figure 8.5.

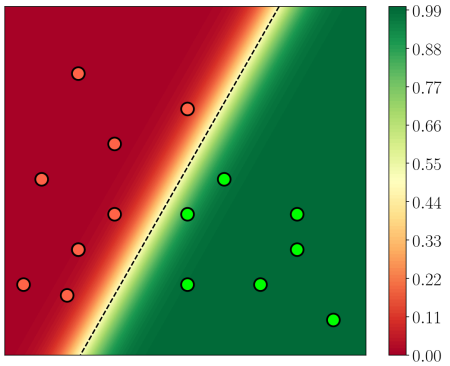


Figure 8.7: Logistic regression is a machine learning technique to fit linear decision boundaries (the dashed line) to the training data (colored dots).

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 (8.7) becomes:

$$\begin{aligned} w^{(k+1)} &= w^{(k)} + \left(X^\top V^{(k)} X\right)^{-1} X^\top (y - p^{(k)}) \\ &= \left(X^\top V^{(k)} X\right)^{-1} X^\top V^{(k)} \left(X w^{(k)} + {V^{(k)}}^{-1} (y - p^{(k)})\right) \\ &= \left(X^\top V^{(k)} X\right)^{-1} X^\top V^{(k)} z^{(k)}, \end{aligned} \tag{8.8}$$

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

Note that $w^{(k+1)}$ as given by (8.8) 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 (8.8) is often called iteratively reweighted least squares. It is easy to see that the Hessian matrix $X^\top V X$ is positive definite (it follows from the fact that $V > 0$), and thus is a convex optimization problem. Figure 8.6 provides a plot of the log-likelyhood function for a uni-dimensional dataset shown in left image of Figure 8.3. Compare it to the right image of Figure 8.5, both plots are made for the same dataset.

Note that like in the previous section, convergence fails if the two classes are linearly separable. In that case we can handle this via a regularization. Instead of maximizing $\log \mathcal{L}(w)$, we can maximize $\log \mathcal{L}(w) - \frac{\lambda}{2} w^\top w$ for some small constant λ. It is easy to see that in this case the gradient and the Hessian matrix take the following form:

$$\frac{\partial\{\log \mathcal{L} - \lambda\|w\|^2\}}{\partial w}(w) = X^\top (y - p) - \lambda w \qquad \frac{\partial^2\{\log \mathcal{L} - \lambda\|w\|^2\}}{\partial w \partial w^\top}(w) = -X^\top V X - \lambda I,$$

where I is the $(m+1) \times (m+1)$ identity matrix.

Listing A.12 contains the source code, whose result is shown in Figure 8.7.

8.5 Nonlinear decision boundaries: 3 neurons perceptron

Okay, now we have mastered the art of binary classification with linear decision boundaries. What about nonlinear boundaries? Refer to the right image of Figure 8.8 for an illustration. A straightforward way to do it is to plug something nonlinear (e.g. a polynomial) instead of $-w^\top x$ into Equation (8.5). It has been done,

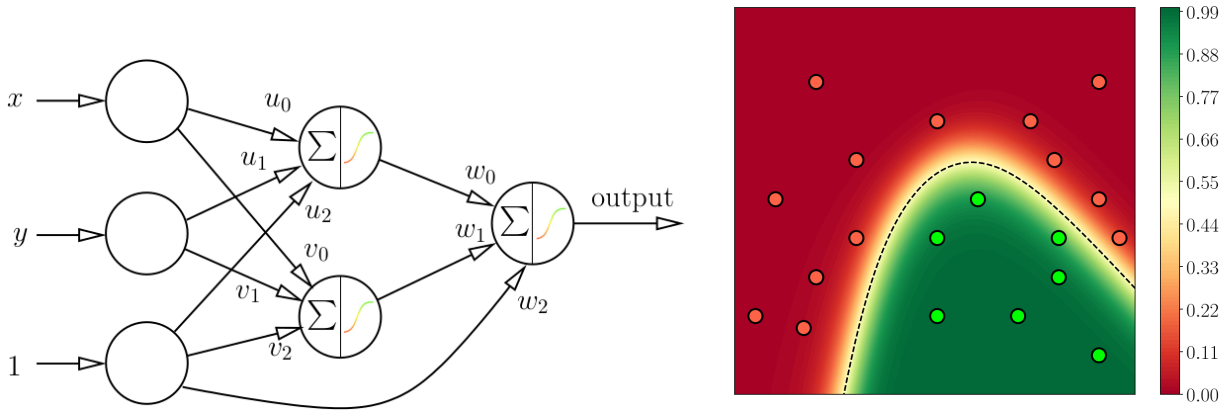


Figure 8.8: Left: 3 neurons perceptron with a hidden layer. Right: learning the database shown by red and green dots, the decision boundary is shown by the dashed line, the color gradient corresponds to the output sigmoid.

however the problem is that such a prior is hard to put on the data; moreover, the optimization scheme is different for every choice of function. Here neural networks come handy.

Our last example in this chapter is a regular perceptron made of 3 neurons: two neurons in the hidden layer and one output neuron. Refer to the left image of Figure 8.8 for the topology and the corresponding notations. The input is the same as in all classification examples throughout this chapter: we have $\{(x_i, y_i)\}_{i=1}^n$ as the input, where $x_i \in \mathbb{R}^m$ and $y_i \in \{0, 1\}$. Let us define the sigmoid function:

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

where w is a $(m+1)$-parameter vector and the last element of x is the constant 1. Then we define a mean squared loss function E as follows:

$$E = \frac{1}{2} \sum_{i=1}^{n} \left(\sigma\left(w^\top x_i'\right) - y_i\right)^2,$$

where

$$x_i' := \begin{pmatrix} \sigma(u^\top x_i) & \sigma(v^\top x_i) & 1 \end{pmatrix}^\top$$

The loss function E is parameterized by three $(m+1)$-component vectors u, v and w. For the lack of better alternative, we can minimize E by a gradient descent. The gradient of the loss function with respect to each weight can be calculated by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule; this is an example of dynamic programming. For our particular example,

$$\begin{aligned}
\frac{\partial E}{\partial w} &= \sum_{i=1}^{n} \left(\sigma\left(w^\top x_i'\right) - y_i\right) \sigma'\left(w^\top x_i'\right) x_i' \\
&= \sum_{i=1}^{n} \left(\sigma\left(w^\top x_i'\right) - y_i\right) \sigma\left(w^\top x_i'\right) \left(1 - \sigma\left(w^\top x_i'\right)\right) x_i' \\
\frac{\partial E}{\partial u} &= \sum_{i=1}^{n} \left(\sigma\left(w^\top x_i'\right) - y_i\right) \sigma\left(w^\top x_i'\right) \left(1 - \sigma\left(w^\top x_i'\right)\right) \sigma\left(u^\top x_i\right) \left(1 - \sigma\left(u^\top x_i\right)\right) x_i \\
\frac{\partial E}{\partial v} &= \sum_{i=1}^{n} \left(\sigma\left(w^\top x_i'\right) - y_i\right) \sigma\left(w^\top x_i'\right) \left(1 - \sigma\left(w^\top x_i'\right)\right) \sigma\left(v^\top x_i\right) \left(1 - \sigma\left(v^\top x_i\right)\right) x_i
\end{aligned}$$

The right image of Figure 8.8 is computed by the program given in Listing A.13. All the training data is correctly classified by a nonlinear decision boundary.

Let us deconstruct the network to understand how the nonlinearity of the boundary is obtained, Figure 8.9 provides an illustration. A perceptron with a hidden layer combines sigmoids from the hidden layer and

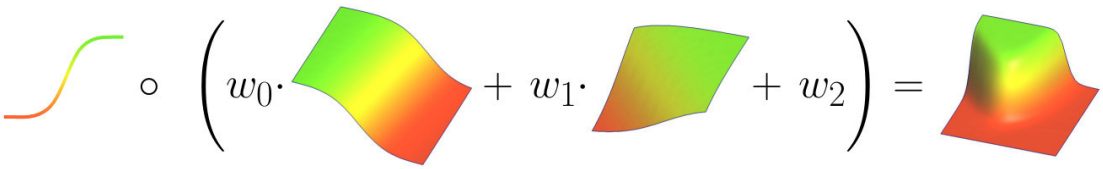


Figure 8.9: A perceptron with a hidden layer combines sigmoids from the hidden layer and applies a sigmoid over it. The idea is to be able to represent nonlinear decision boundaries by stacking neurons.

applies a sigmoid over it. In our example the data is two-dimensional; for the sake of simplicity, imagine that the hidden layer neurons produce unit steps instead of sigmoids. Let us suppose that all the weights are equal to 1, then the sum of these steps separates the data plane into four quadrants where the sum is equal to 0, 1 and 2. By applying a unit step over this result, we can isolate one particular quadrant, effectively creating a non-linear decision boundary. It is the last layer activation function that makes the difference. If we would remove this, the decision boundary would remain linear.

So, we can choose the type of the decision boundary by adapting the topology of the network, but be aware that you are entering the black magic realm! Generally speaking, there is no whatsoever guarantee of convergence; and even this simplest network has too much degrees of freedom for our training dataset. We can negate all the weights without affecting the decision boundary...

Appendix A

Program listings

Listing A.1: 1D Poisson image editing (§6.2). This program solves Equation (6.1).

```
import numpy as np
n,f0,fn = 32,1.,3.
g = [np.sin(x) for x in np.linspace(0, 2*np.pi, n)]
f = [f0] + [0]*(n-2) + [fn]
for _ in range(512):
    for i in range(1, n-1):
        f[i] = ( f[i-1] + f[i+1] + (2*g[i]-g[i-1]-g[i+1]) )/2.
```

Listing A.2: 1D Poisson image editing (§6.2). This program solves Equation (6.2).

```
import numpy as np
n,f0,fn = 32,1.,3.
g = [np.sin(x) for x in np.linspace(0, 2*np.pi, n)]
A = np.matrix(np.zeros((n-1,n-2)))
np.fill_diagonal(A,      1)
np.fill_diagonal(A[1:], -1)
b = np.matrix([[g[i]-g[i-1]] for i in range(1,n)])
b[ 0,0] = b[ 0,0] + f0
b[-1,0] = b[-1,0] - fn
f = [f0] + (np.linalg.inv(A.T*A)*A.T*b).T.tolist()[0] + [fn]
```

Listing A.3: Poisson image editing (§6.2). This program solves Equation (6.3).

```
import matplotlib.image as mpimg
import scipy.sparse
from scipy.sparse.linalg import lsmr

base = mpimg.imread('baseball.png')
foot = mpimg.imread('football.png')
w,h = len(foot[0]), len(foot)
ox,oy = 100, 60 # glue the football here

A = scipy.sparse.lil_matrix((2*(w-1)*(h-1)+2*w+2*h, w*h))
for i in range(0,w): # top data fitting
    A[i, i] = 1
for i in range(0,w): # bottom data fitting
    A[w+i, i+(h-1)*w] = 1
for j in range(0,h): # left data fitting
    A[2*w+j, j*w] = 1
for j in range(0,h): # right data fitting
    A[2*w+h+j, w-1+j*w] = 1
cnt = 2*w+2*h
for j in range(0,h-1): # gradient matrix
    for i in range(0,w-1):
        A[cnt,   i   + j*w] = -1
        A[cnt,   i+1 + j*w] =  1
        A[cnt+1, i +  j   *w] = -1
        A[cnt+1, i + (j+1)*w] =  1
        cnt += 2
A = A.tocsc()

for channel in range(3):
    b = A.dot(foot[:,:,channel].flatten()) # fill the gradient part of the r.h.s.
    b[0:w]     = base[oy,ox:ox+w,channel]   # top data fitting
    b[w:2*w]   = base[oy+h,ox:ox+w,channel] # bottom data fitting
    b[2*w  :2*w+h]   = base[oy:oy+h, ox, channel]   # left data fitting
    b[2*w+h:2*w+2*h] = base[oy:oy+h, ox+w, channel] # right data fitting

    x = lsmr(A, b)[0] # call the least squares solver
    base[oy:oy+h,ox:ox+h, channel] = x.reshape((h, w)) # glue the football
mpimg.imsave('poisson_ls.png', base)
```

Listing A.4: A caricature on a 2D silhouette (§6.3). This program solves Equation (6.6).

```
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)
```

Listing A.5: 3D surface caricature (§6.3). Refer to Equation (6.5) for the problem definition.

```
from mesh import Mesh
from scipy.sparse.linalg import lsmr
from scipy.sparse import lil_matrix

m = Mesh("input-face.obj") # load mesh

A = lil_matrix((m.nverts+m.ncorners, m.nverts))
for v in range(m.nverts): # per-vertex attachment to the original geometry
    if m.on_border(v):    # hard on the boundary
        A[v,v] = 10
    else:                 # light on the interior
        A[v,v] = .29
for c in range(m.ncorners): # per-half-edge discretization of the derivative
    A[m.nverts+c, m.org(c)] = -1
    A[m.nverts+c, m.dst(c)] =  1
A = A.tocsr() # sparse row matrix for fast matrix-vector multiplication

for dim in range(3): # the problem is separable in x,y,z; the matrix A is the same, the right hand side changes
    b = [m.V[v][dim]*10 if m.on_border(v) else m.V[v][dim]*.29 for v in range(m.nverts)] + \
        [2.5*(m.V[m.dst(c)][dim]-m.V[m.org(c)][dim]) for c in range(m.ncorners)]
    x = lsmr(A, b)[0] # call the least squares solver
    for v in range(m.nverts): # apply the computed distortion
        m.V[v][dim] = x[v]

print(m) # output the deformed mesh
```

Listing A.6: “Cubification” of a 3d surface (§6.4).

```
import numpy as np
from mesh import Mesh
from scipy.sparse        import lil_matrix
from scipy.sparse.linalg import lsmr

def normalize(v):
    return v / np.linalg.norm(v)

def cross(v1, v2):
    return [v1[1]*v2[2] - v1[2]*v2[1], v1[2]*v2[0] - v1[0]*v2[2], v1[0]*v2[1] - v1[1]*v2[0]]

def nearest_axis(n):
    axes = [[1,0,0],[0,1,0],[0,0,1]]
    nmin = -1
    imin = -1
    for i,a in enumerate(axes):
        if np.abs(np.dot(n,a))>nmin:
            nmin = np.abs(np.dot(n,a))
            imin = i
    return imin

m = Mesh("input-face.obj") # load mesh

for dim in range(3): # the problem is separable in x,y,z
    A = lil_matrix((m.ncorners*2, m.nverts))
    b = [m.V[m.dst(c)][dim]-m.V[m.org(c)][dim] for c in range(m.ncorners)] + [0]*m.ncorners
    for c in range(m.ncorners):
        A[c, m.org(c)] = -1 # per-half-edge discretization of the derivative
        A[c, m.dst(c)] =  1

        t = c//3 # triangle id from halfedge id
        n = normalize(cross(m.V[m.T[t][1]]- m.V[m.T[t][0]], m.V[m.T[t][2]]- m.V[m.T[t][0]])) # normal
        if nearest_axis(n)==dim:  # flatten the right dimension of each half-edge
            A[c+m.ncorners, m.org(c)] = -2
            A[c+m.ncorners, m.dst(c)] =  2
    A = A.tocsr() # sparse row matrix for fast matrix-vector multiplication
    x = lsmr(A, b)[0] # call the least squares solver
    for v in range(m.nverts): # apply the computed distortion
        m.V[v][dim] = x[v]

print(m) # output the deformed mesh
```

Listing A.7: Least squares conformal mapping (§6.5).

```
from mesh import Mesh
import numpy as np
import scipy.sparse
from scipy.sparse.linalg import lsmr

def normalize(v):
    return v / np.linalg.norm(v)

def project_triangle(p0, p1, p2):
    X = normalize(np.subtract(p1, p0)) # construct an orthonormal 3d basis
    Z = normalize(np.cross(X, np.subtract(p2, p0)))
    Y = np.cross(Z, X)
    z0 = np.array([0,0]) # project the triangle to the 2d basis (X,Y)
    z1 = np.array([np.linalg.norm(np.subtract(p1, p0)), 0])
    z2 = np.array([np.dot(np.subtract(p2, p0), X), np.dot(np.subtract(p2, p0), Y)])
    return [z0, z1, z2]

m = Mesh("input-face.obj") # load mesh

# build the system                                          # 2 eq per triangle + 4 eq for pinning verts
A = scipy.sparse.lil_matrix((2*m.ntriangles+4, 2*m.nverts)) # the variables are packed as u0,v0,u1,v1, ...
lock1, lock2 = 10324%m.nverts, 35492%m.nverts  # select two arbitrary vertices to pin
for (t,[i,j,k]) in enumerate(m.T): # for each triangle ijk
    zi,zj,zk = project_triangle(m.V[i], m.V[j], m.V[k]) # project the triangle to a local 2d basis
    ejk = zk-zj # edges of the projected triangle:
    eki = zi-zk # the gradients are computed
    eij = zj-zi # as a function of the edges
    A[t*2+0, i*2  ] =  ejk[0] # (grad u)[0] = (grad v)[1]
    A[t*2+0, j*2  ] =  eki[0]
    A[t*2+0, k*2  ] =  eij[0]
    A[t*2+0, i*2+1] =  ejk[1]
    A[t*2+0, j*2+1] =  eki[1]
    A[t*2+0, k*2+1] =  eij[1]

    A[t*2+1, i*2  ] =  ejk[1] # (grad u)[1] = -(grad v)[0]
    A[t*2+1, j*2  ] =  eki[1]
    A[t*2+1, k*2  ] =  eij[1]
    A[t*2+1, i*2+1] = -ejk[0]
    A[t*2+1, j*2+1] = -eki[0]
    A[t*2+1, k*2+1] = -eij[0]
A[-1,lock2*2+1] = A[-2,lock2*2+0] = A[-3,lock1*2+1] = A[-4,lock1*2+0] = 10 # quadratic penalty

A = A.tocsr() # convert to compressed sparse row format for faster matrix-vector muliplications
b = [0]*(2*m.ntriangles) + [0,0,10,10] # one pinned to (0,0), another to (1,1)
x = lsmr(A, b)[0] # call the least squares solver

for v in range(m.nverts): # apply the computed flattening
    m.V[v] = np.array([x[v*2], x[v*2+1], 0])
print(m) # output the deformed mesh
```

Listing A.8: As-rigid-as possible deformations (§6.6).

```
from mesh import Mesh
import numpy as np
from scipy.sparse import lil_matrix
from scipy.sparse.linalg import lsmr
from scipy.linalg import svd

m = Mesh("diablo.obj")
eij = [np.matrix(m.V[m.dst(c)] - m.V[m.org(c)]).T for c in range(m.ncorners)] # reference for each half-edge

lock = [1175, 1765, 381, 2383, 1778] # id of the vertices to constrain
disp = [[0,0,-0.5], [0,0,0.5], [0,0,-0.5], [0,0,0.5], [1.5,0,0]] # displacement for the constrained vertices
for v,d in zip(lock, disp): # apply the displacement
    m.V[v] = m.V[v] + d

A = lil_matrix((m.ncorners+len(lock), m.nverts))
for c in range(m.ncorners): # Least-squares verion of Poisson's problem
    A[c, m.dst(c)] =  1
    A[c, m.org(c)] = -1
for i,v in enumerate(lock):  # the vertices are locked
    A[m.ncorners+i, v] = 100 # via quadratic penalty
A = A.tocsr() # convert to compressed sparse row format for faster matrix-vector muliplications

for _ in range(100):
    R = [] # rotation per vertex
    for v in range(m.nverts): # solve for rotations
        M = np.zeros(3)
        c = m.v2c[v] # half-edge departing from v
        while True: # iterate through all half-edges departing from v
            M = M + np.matrix(m.V[m.dst(c)] - m.V[m.org(c)]).T*eij[c].T
            c = m.c2c[c] # next around vertex
            if c==m.v2c[v]: break
        U, s, VT = svd(M)
        R.append(np.dot(U,VT)) # rotation matrix for the neighborhood of vertex v

    for dim in range(3): # the problem is separable in x,y,z
        b = [ (R[m.org(c)]*eij[c])[dim,0] for c in range(m.ncorners) ] + \
            [ 100*m.V[v][dim] for v,d in zip(lock, disp) ]
        x = lsmr(A, b)[0] # call the least squares solver
        for v in range(m.nverts): # apply the computed deformation
            m.V[v][dim] = x[v]
print(m) # output the deformed mesh
```

Listing A.9: Ordinary linear regression for binary classification (§8.1).

```
import numpy as np
samples = [[0.47,1],[0.24,1],[0.75,1],[0.00,1],[-0.80,1],[-0.59,1],[1.09,1],[1.34,1],
           [1.01,1],[-1.02,1],[0.50,1],[0.64,1],[-1.15,1],[-1.68,1],[-2.21,1],[-0.52,1],
           [3.93,1],[4.21,1],[5.18,1],[4.20,1],[4.57,1],[2.63,1],[4.52,1],[3.31,1],
           [6.75,1],[3.47,1],[4.32,1],[3.08,1],[4.10,1],[4.00,1],[2.99,1],[3.83,1]]
n = len(samples)
m = len(samples[0])
labels = [0]*(n//2) + [1]*(n//2)

A = np.matrix(np.zeros((n,m)))
b = np.matrix(np.zeros((n,1)))
for i in range(n):
    A[i,:] = samples[i]
    b[i,0] = labels[i]
W = np.linalg.inv(A.transpose()*A)*A.transpose()*b
```

Listing A.10: Logistic growth least squares fitting (§8.2).

```
import numpy as np

X = [0.2,37.9,32.0,12.7,23.3,8.2,25.2,27.0,40.9,4.7,19.1,50.7,53.2,59.3,15.2,45.5]
Y = [0.04,4.79,4.51,0.30,3.05,0.01,3.61,4.14,4.77,0.01,1.64,4.77,4.56,4.53,0.67,4.61]
n = len(X)

guess_c = np.max(Y)*1.1 # 1.1 to avoid division by zero

A = np.matrix(np.zeros((n, 2)))
b = np.matrix(np.zeros((n, 1)))
for i in range(n):
    A[i,0] = 1
    A[i,1] = X[i]
    b[i,0] = np.log(Y[i]/(guess_c - Y[i]))

guess_w0, guess_w = (np.linalg.inv(A.T*A)*A.T*b).T.tolist()[0]

U = np.matrix([[guess_c],[guess_w0],[guess_w]])
for _ in range(5):
    JR = np.matrix(np.zeros((n, 3)))
    R  = np.matrix(np.zeros((n, 1)))
    for i in range(n):
        ei = np.exp(-U[1,0] - X[i]*U[2,0])
        R[i,0] = U[0,0]/(1+ei) - Y[i]
        for j in range(3):
            JR[i, 0] = 1/(1+ei)
            JR[i, 1] = U[0,0]*ei/(1+ei)**2
            JR[i, 2] = X[i]*U[0,0]*ei/(1+ei)**2
    U = U - np.linalg.inv(JR.T*JR)*JR.T*R
```

Listing A.11: Binary classification: logistic curve fitting by least squares (§8.3).

```
import numpy as np
samples = [[0.47,1],[0.24,1],[0.75,1],[0.00,1],[-0.80,1],[-0.59,1],[1.09,1],[1.34,1],
           [1.01,1],[-1.02,1],[0.50,1],[0.64,1],[-1.15,1],[-1.68,1],[-2.21,1],[-0.52,1],
           [3.93,1],[4.21,1],[5.18,1],[4.20,1],[4.57,1],[2.63,1],[4.52,1],[3.31,1],
           [6.75,1],[3.47,1],[4.32,1],[3.08,1],[4.10,1],[4.00,1],[2.99,1],[3.83,1]]
n = len(samples)
m = len(samples[0])
labels = [0]*(n//2) + [1]*(n//2)

W = np.matrix([[1],[1]])
for _ in range(5):
    JR = np.matrix(np.zeros((n+2, 2)))
    R  = np.matrix(np.zeros((n+2, 1)))
    for i in range(n):
        ei = np.exp(-W[1,0] - samples[i][0]*W[0,0])
        R[i,0] = -1/(1+ei) + labels[i]
        for j in range(3):
            JR[i, 0] = samples[i][0]*ei/(1+ei)**2
            JR[i, 1] = ei/(1+ei)**2
    l = .001 # regularization
    JR[n,0] = JR[n+1, 1] = 1.*l
    R[n  ,0] = -W[0]*l
    R[n+1,0] = -W[1]*l
    W = W + np.linalg.inv(JR.T*JR)*JR.T*R
```

Listing A.12: Binary classification: logistic regression with crossentropy loss function (§8.4).

```
import numpy as np
import math

def p(x, w):
    return 1./(1.+math.exp(-np.dot(x,w)))

samples = [[.5,.7,1.],[.1,.5,1.],[.3,.6,1.],[.2,.8,1.],
           [.17,.17,1.],[.2,.3,1.],[.3,.4,1.],[.05,.2,1.],
           [.2,.3,1.],[.8,.3,1.],[.5,.2,1.],[.7,.2,1.],
           [.9,.1,1.],[.8,.4,1.],[.6,.5,1.],[.5,.4,1.]]
labels = [0.,0.,0.,0.,0.,0.,0.,0.,1.,1.,1.,1.,1.,1.,1.]

n = len(samples)
X = np.matrix(samples)
y = np.matrix(labels).T
wk = np.matrix([[.3], [.7], [-.02]]) # small random numbers

l = 0.001 # regularization coefficient
for _ in range(5):
    pk = np.matrix([p(xi,wk) for xi in samples]).T
    Vk = np.matrix(np.diag([pk[i,0]*(1.-pk[i,0]) for i in range(n)]))
    wk += np.linalg.inv(X.T*Vk*X + l*np.matrix(np.identity(len(samples[0]))))*(X.T*(y-pk) - l*wk)
print(wk)
```

Listing A.13: Binary classification: training a neural network made of 3 neurons (§8.5).

```
import numpy as np

samples = [[.5,.7,1.],[.1,.5,1.],[.3,.6,1.],[.2,.8,1.],[.17,.17,1.],[.2,.3,1.],
           [.3,.4,1.],[.05,.2,1.], [.2,.3,1.],[.8,.3,1.],[.5,.2,1.],[.7,.2,1.],
           [.9,.1,1.],[.8,.4,1.],[.6,.5,1.], [.5,.4,1.], [.9, .5, 1.],[.79, .6, 1.],
           [.73, .7, 1.],[.9, .8, 1.],[.95, .4, 1.]]
labels = [0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0]

def neuron(x, w):
    return 1./(1.+np.exp(-np.dot(x,w)))

u = np.array([0.814, 0.779, 0.103]) # small random values
v = np.array([0.562, 0.310, 0.591])
w = np.array([0.884, 0.934, 0.649])

alpha = 1. # learning rate
for _ in range(0,3000):
    E = 0
    for x, label in zip(samples,labels):
        E += (label - neuron([neuron(x, u), neuron(x, v), 1.],w))**2
    print("E =",E)

    for x, label in zip(samples,labels):
        out_u = neuron(x, u)
        out_v = neuron(x, v)
        out_w = neuron([out_u, out_v, 1.], w)
        u += alpha*(label-out_w)*out_w*(1.-out_w)*out_u*(1.-out_u)*np.array(x)
        v += alpha*(label-out_w)*out_w*(1.-out_w)*out_v*(1.-out_v)*np.array(x)
        w += alpha*(label-out_w)*out_w*(1.-out_w)*np.array([out_u, out_v, 1.])
print(u,v,w)
```

Bibliography

[1] Cerda, H., and Wright, D. Modeling the spatial and temporal location of refugia to manage resistance in bt transgenic crops. *Agriculture, Ecosystems & Environment 102*, 2 (2004), 163 – 174.

[2] Gauss, C. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium.* Carl Friedrich Gauss Werke. Hamburgi sumptibus Frid. Perthes et I.H.Besser, 1809.

[3] Lévy, B., Petitjean, S., Ray, N., and Maillot, J. Least squares conformal maps for automatic texture atlas generation. In *ACM transactions on graphics (TOG)* (2002), vol. 21, ACM, pp. 362–371.

[4] Lévy, B. aceessed August 17, 2021. `http://alice.loria.fr/index.php/software/4-library/23-opennl.html`.

[5] Pérez, P., Gangnet, M., and Blake, A. Poisson image editing. *ACM Trans. Graph. 22*, 3 (July 2003), 313–318.

[6] Popper, K. *Logik der Forschung: Zur Erkenntnistheorie der Modernen Naturwissenschaft.* Schriften zur Wissenschaftlichen Weltauffassung, Hrsg., von Philipp Frank... und Moritz Schlick. Springer Vienna, 1935.

[7] Sorkine, O., and Alexa, M. As-rigid-as-possible surface modeling. In *Proceedings of EUROGRAPHICS/ACM SIGGRAPH Symposium on Geometry Processing* (2007), pp. 109–116.

[8] Stigler, S. M. Gauss and the invention of least squares. *Ann. Statist. 9*, 3 (05 1981), 465–474.

[9] Tabashnik, B., Gassmann, A., Crowder, D., and Carriere, Y. Insect resistance to bt crops: Evidence versus theory. *Nature biotechnology 26* (03 2008), 199–202.