university of groningen / faculty of mathematics and natural sciences / computing science

Introduction to Scientific Computing

Jos B.T.M. Roerdink

February 2019

Acknowledgement: the picture on the front page shows the result of a flow simulation on the Nvidia logo (W. J. van der Laan et al.: Screen Space Fluid Rendering with Curvature Flow. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 91–98, 2009).

Contents

Chapter 1

What is Scientific Computing?

Scientific Computing (also called Computational Science) deals with computation in the (natural) sciences. It studies the construction of mathematical models of phenomena occurring in nature and human society, and analyses such models quantitatively, both by mathematical techniques and by computer implementation (Heath, 2002; Kutz, 2013). In practice, scientific computing typically involves the application of computer simulation and other forms of computation to problems in various scientific disciplines. Numerical analysis is an important foundation for techniques used in scientific computing.

Scientists and engineers develop computer programs and application software that model the systems under study and run these programs with various sets of input parameters. Often, the computations involve massive amounts of calculations (usually floating-point) which require the use of supercomputers, high performance computing, parallel and distributed computing, or GPU (graphics processing unit) computing (Hager and Wellein, 2010). To get insight in the large amounts of data produced by the simulations, visualization techniques are applied. In fact, the origin of the discipline of Scientific Visualization runs in parallel to the rise of the scientific computing discipline. Its birth is usually associated to the ACM SIGGRAPH panel report *Visualization in Scientific Computing* from 1987.

Here are some typical fields in which scientific computing is applied, with some example problems:

- *Physics*: simulation of elementary particle behaviour.
- *Astronomy*: simulation of star and galaxy formation.
- *Chemistry*: simulation of chemical reactions.
- *Biology*: simulation of pattern formation in organisms, populations, or ecological networks.
- *Earth and Environmental Science*: simulation of earthquakes, simulation of climate dynamics.
- *Medicine*: simulation of X-rays in tissues for radiation therapy planning.
- *Sociology*: simulation of crowd behaviour.

This list can easily be extended with many other examples.

The use of computation in all these areas has given rise to many new subfields, the so-called *computational-X* fields: computational physics, computational chemistry, computational mathematics, computational biology, computational astrophysics, computational neuroscience, computational sociology, computational finance, etc. On the other hand, there are also the *X-informatics* fields, such as bioinformatics, cheminformatics, astroinformatics, or medical informatics, where the focus is more on the use of Information Technology, like database technology, special-purpose programming and script languages, large data handling (transport, compression, and archiving), web services and ontologies, etc. Of course, this distinction is not absolute, and in practice we often see combinations of *computational-X* and *X-informatics* approaches.

Some of the computational methods you will encounter in this course are:

- Solving systems of linear equations in computer tomography (Chapter 2)
- Discrete-time dynamical systems as models of networks and pattern formation (Chapters 3, 4, 5)
- Numerical solutions of (partial) differential equations (Chapters 6, 7, 8)

In this course, we motivate and illustrate the scientific computing techniques by examples from application areas such as medicine, physics, or astronomy. These application domains are used to give context to the computational approaches and to make you aware of the fact that understanding the domain context is an integral part of the task of a computer scientist working in scientific computing. A dialogue between the computer scientist and the domain expert is necessary to develop solutions that are both technically sound and relevant in practice. This does not detract from the fact that the techniques are generic and therefore applicable in many other domains.

Bibliography

Hager, G., Wellein, G., 2010. Introduction to High Performance Computing for Scientists and Engineers. Chapman and Hall.

Heath, M. T., 2002. Scientific Computing: An Introductory Survey (2nd ed.). McGraw-Hill.

Kutz, N., 2013. Data-driven Modeling Scientific Computation. Oxford University Press.

Chapter 2

Computer Tomography

In this chapter we look at the fundamentals of tomographic reconstruction. The algorithms we will consider form the basis of computer tomography, as realized in medical devices such as CT or MRI scanners.

2.1 Introduction

The word *tomography* means 'reconstruction from projections', i.e., the recovery of a function from its line or (hyper)plane integrals (from the Greek $\tau\acute{o}\mu o\sigma$—slice and $\gamma\rho\acute{\alpha}\phi\epsilon\iota\nu$—to write); see Figure 2.1. In the applied sense, it is a method to reconstruct cross sections of the interior structure of an object without having to cut or damage the object. The term often occurs in the combination *computer (computerized, computed) tomography* (CT) or *computer-assisted tomography* (CAT), since for performing the reconstructions in practice one needs the use of a digital computer. Important issues in tomography are existence, uniqueness and stability of reconstruction procedures, as well as the development of efficient numerical algorithms.

The internal property of the object to be reconstructed, such as a density, space-dependent attenuation coefficient, and so on, is generally referred to as the *internal distribution*. The physical agens or *probe* by which to act on this internal distribution may vary from X-rays, gamma rays, visible light, electrons or neutrons to ultrasound waves or nuclear magnetic resonance signals. When the probe is *outside* the object one speaks of *transmission* computer tomography (TCT). In contrast with this stands *emission* computer tomography (ECT), where the probe, such as a radioactive material, is inside the object. This occurs in two variants: SPECT (single particle ECT) where radiation along a half line is detected, and PET (positron emission tomography) where radiation emitted in opposite directions is detected in coincidence. Finally we mention *reflection* tomography, where the object is 'illuminated' by sound waves and the reflected waves are recorded to obtain line integrals of the object's reflectivity function (Kak and Slaney, 1988).

Other forms of tomography exist, such as electric impedance tomography (recovering the conductivity inside a body from electric potential measurements on the surface), biomagnetic imaging (recovering the position of electric currents from magnetic fields induced outside the body), or diffraction tomography, see Herman (1980). Instances of tomography in *three* dimensions are

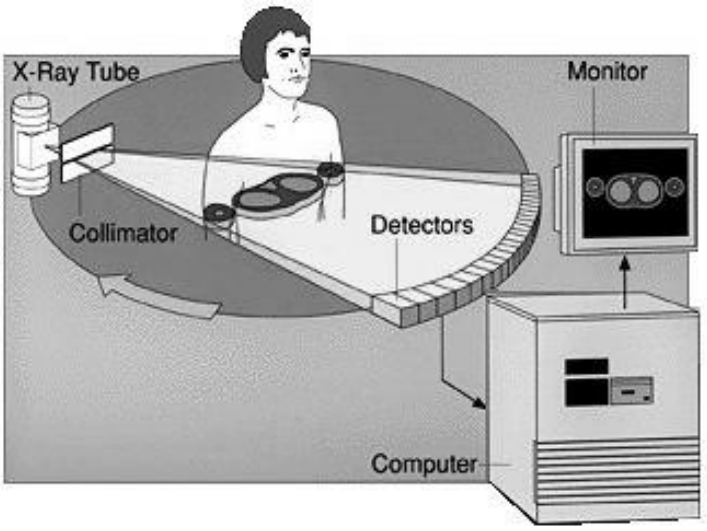


Figure 2.1: *Scanning the body with X-rays to obtain projections. The problem is how to reconstruct a cross-section from the projections.*

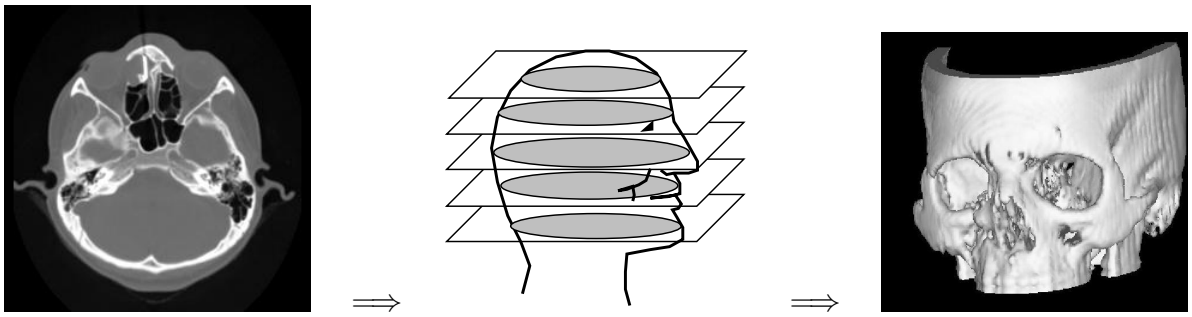


Figure 2.2: *Performing 3D reconstruction by a stack of 2D reconstructions along parallel slices.*

found in radar theory and magnetic resonance imaging. One of the most prominent applications of computer tomography occurs in diagnostic medicine, where the method is used to produce images of the interior of human organs (Shepp and Kruskal, 1978). In 1979 the Nobel prize in physiology or medicine was awarded to G.N. Hounsfield and A.M. Cormack for their fundamental work in the field. Other applications arise in radio astronomy, 3D electron microscopy, soil science, aerodynamics and geophysics, to name a few. In industry the method is used for non-destructive testing.

To obtain 3D reconstruction of objects, the standard approach is to make a stack of 2D reconstructions of parallel cross-sections. These can then be combined into full 3D reconstructions by volume rendering techniques; see Figure 2.2 for the general idea. In the remainder of this chapter we restrict ourselves to the 2D reconstruction process of a single cross-section.

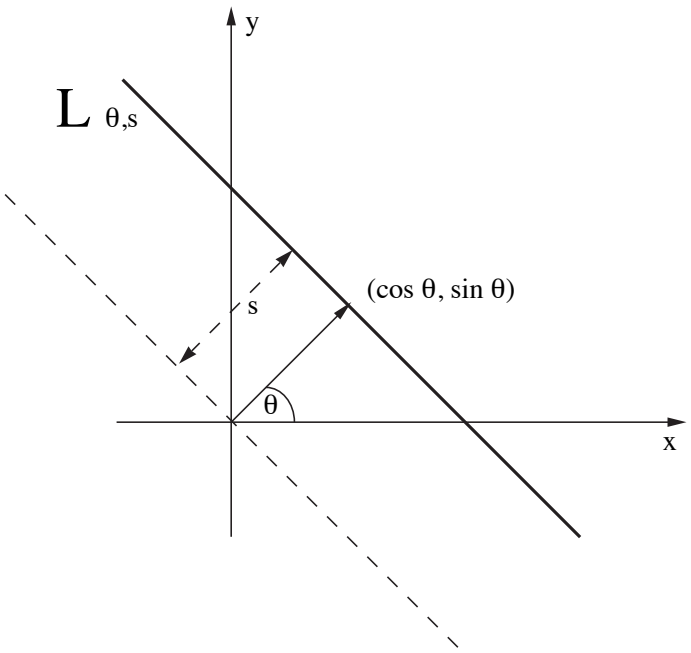


Figure 2.3: *Parameters θ, s defining a line $L_{\theta,s}$.*

2.2 Obtaining the projection data

To describe computer tomography one starts from the following simplified model (in practice, many complications arise). If a beam of X-rays with initial intensity I_0 passes through an object along a straight line L, then the beam intensity is attenuated by a factor which involves the integrated density along this line. This means that the intensity I_1 after having passed the object satisfies

$$\frac{I_1}{I_0} = \exp\Big\{-\int_L f(x,y)\,dx\,dy\Big\}, \tag{2.1}$$

where $f(x, y)$ denotes the X-ray attenuation coefficient of the object at the point (x, y). Hence by measuring the ratio I_1/I_0 line integrals of the unknown distribution f are obtained.

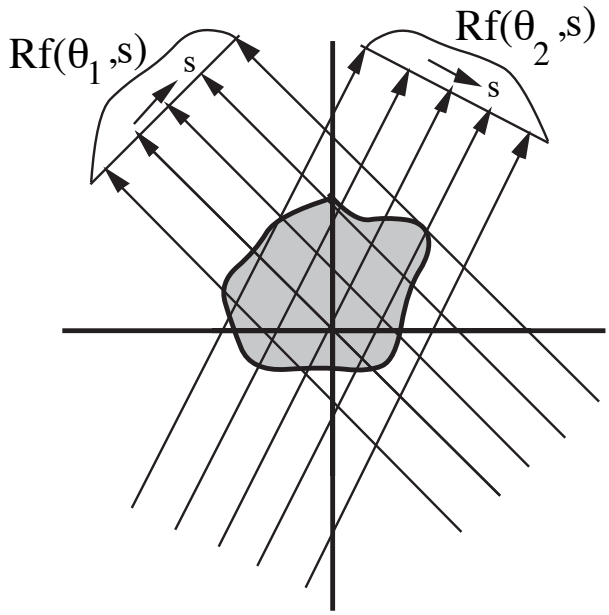


Figure 2.4: *Parallel beam scanning mode in 2D tomography: projections along parallel lines. Shown are projection profiles for two directions with angles θ_1 and θ_2.*

The mathematical concept associated to reconstruction of a distribution from line integrals is the so-called *Radon transform* (Deans, 1983). Let $(\cos\theta, \sin\theta)$ be a vector of length 1 making an angle θ with the x-axis, s a real number, and $L_{\theta,s}$ the line defined by

$$x\,\cos\theta + y\sin\theta = s$$

The line $L_{\theta,s}$ is perpendicular to the vector $(\cos\theta, \sin\theta)$, see Figure 2.3. The integral of f over the line $L_{\theta,s}$

$$\mathbf{R}f(\theta, s) := \int_{L_{\theta,s}} f(x, y)\,dx\,dy$$

is called the *Radon transform* of the function f. The integral $\mathbf{R}f(\theta, s)$ for a single line, i.e., with θ and s fixed, is called a *projection* and the set of projections along parallel lines for a fixed value of θ is called a *projection profile*, or simply a *profile*, cf. Figure 2.4.

In practice, one uses different ways to sample the line integrals of the internal distribution. In *parallel beam scanning*, as described above, parallel line integrals are determined for a fixed direction and the process is repeated for a number of different directions; in *fan-beam scanning* line integrals emanating from a given source point are computed for different directions, which is repeated for a certain number of source points. In this chapter, we only consider the case that projections are collected along parallel lines.

2.2.1 Reconstruction methods

Once we have obtained the projection data, the question is how to recover the original 2D distribution. In computer tomography, the reconstruction methods can be subdivided as follows:

- Direct methods:
  - Filtered backprojection
  - Fourier reconstruction
- Iterative methods:
  - ART: algebraic reconstruction technique
  - SIRT: simultaneous iterative reconstruction technique
  - SART: simultaneous algebraic reconstruction technique

We will consider one of the iterative methods, i.e., ART.

2.3 The algebraic reconstruction technique (ART)

In ART, one considers a discrete version of the projection process, see Figure 2.5. Projections are collected along bundles of parallel strips, where the direction of a bundle is specified by an angle $\theta_k = \pi/k$, $k = 1, 2, \ldots, N_{\text{angles}}$. Each bundle contains a number N_{rays} of parallel lines which are a distance t apart. So the total number of rays is $M = N_{\text{angles}} \cdot N_{\text{rays}}$. Two consecutive parallel lines define a strip. We also assume that the distribution f which we want to reconstruct is defined on a square grid of pixels. Each pixel has width and height d. Each row has n pixels and there are n rows, so the total number of pixels is $N = n^2$.

We assume that the (unknown) density in each pixel is *constant*. The density in pixel j is denoted by f_j, $j = 1, 2, \ldots, N$. So f can be described by a matrix of the form

$$f = \begin{pmatrix} f_1 & f_2 & \cdots & f_n \\ f_{n+1} & f_{n+2} & \cdots & f_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ f_{(n-1)n+1} & f_{(n-1)n+2} & \cdots & f_{n^2} \end{pmatrix} \tag{2.2}$$

Instead of a matrix (or grid) representation of the density, we will also represent the 2D image of dimensions $n \times n$ as a 1D vector $\vec{f}$ of dimension $N = n^2$, where

$$\vec{f} = (f_1, f_2, \ldots, f_n, f_{n+1}, f_{n+2}, \ldots, f_{2n}, \ldots, \ldots, f_{n^2})$$

The projection in strip i is denoted by p_i. Instead of the continuous formula (2.1) we obtain p_i by summing the contributions of all pixels which have a nonzero intersection with strip i. For this reason we will call p_i a **ray sum**. For pixel j, the contribution to ray i will be the percentage

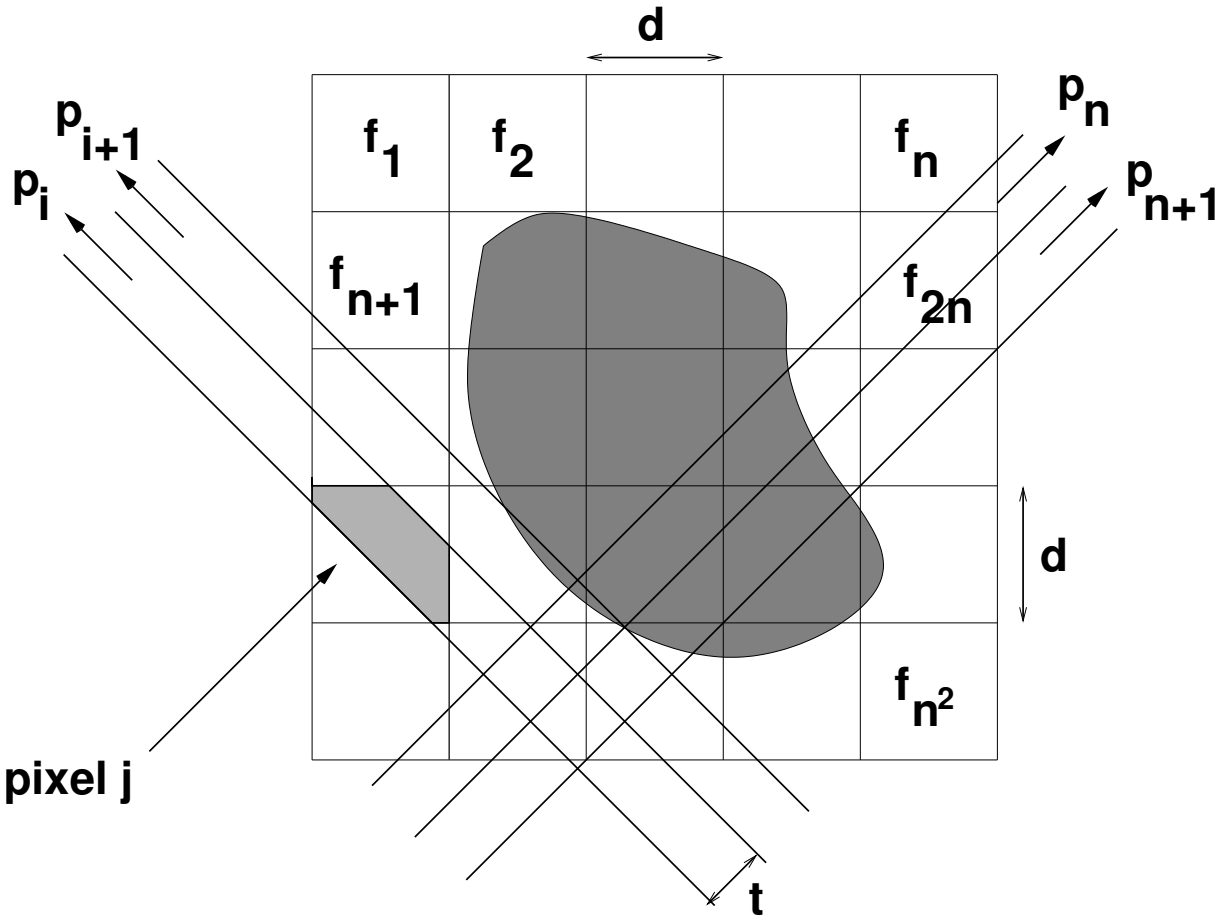


Figure 2.5: *Projections of a distribution f on a grid of size $n \times n$ along bundles of parallel strips of width t. The density in pixel j is denoted by f_j, $j = 1, 2, \ldots, N$, where $N = n^2$. The ray sum for strip i is denoted by p_i.*

of pixel j which is intersected by strip i; this is the shaded area within pixel j in Figure 2.5. We will denote this contribution by w_{ij}. So:

$$w_{ij} = \frac{\text{area of intersection of strip } i \text{ with pixel } j}{d^2}, \tag{2.3}$$

where the denominator d^2 in this formula equals the area of a pixel. The quantity w_{ij} is a weight factor: $0 \leq w_{ij} \leq 1$. It indicates with what weight pixel j is represented in the ray sum p_i. The weight matrix W can be precomputed. However, one problem in practice is that it can become very large, since its size is $M \times N$, and thus requires a lot of storage space.

Sometimes we use the following simpler method to define the weights w_{ij}:

$$w_{ij} = \begin{cases} 1 & \text{if ray } j \text{ intersects pixel } i \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

This makes the implementation easier because we can easily compute the weight factors at run time.

If we sum the contributions of all pixels to the ray sum p_i we find the following *ray equation*:

$$p_i = w_{i1}\, f_1 + w_{i2}\, f_2 + \ldots + w_{iN}\, f_N, \qquad i = 1, 2, \ldots, M \tag{2.5}$$

Since we have M rays we also have M ray equations of the form (2.5). In each equation we have N terms, one for each pixel. So we have M equations in N unknowns.

The question we will consider is how to solve such systems of equations to obtain the unknown densities $f_1, \ldots, f_N$. But first we will look at a simple example to get a good understanding of the reconstruction process.

2.3.1 The case $N = 4, M = 4$: four pixels, four projections

Let us assume we have an unknown grid image f of size 2×2. We take two horizontal and two vertical projections. So we have $N = 4, M = 4$. Assume that the two horizontal ray sums are 3 and 7, respectively, while the vertical ray sums are 4 and 6, respectively. Then we have the following picture:

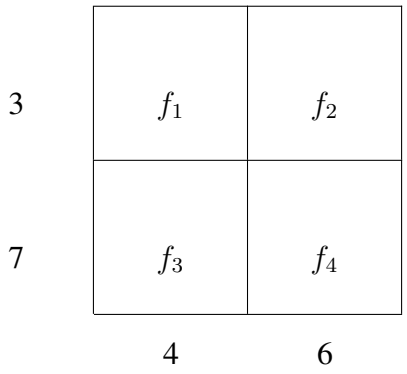


Figure 2.6: *A simple* 2×2 *grid image with known ray sums along rows and columns.*

The ray equations which should be satisfied by the pixel values f_1–f_4 are:

$$f_1 + f_2 = 3 \tag{2.6}$$
$$f_3 + f_4 = 7 \tag{2.7}$$
$$f_1 + f_3 = 4 \tag{2.8}$$
$$f_2 + f_4 = 6 \tag{2.9}$$

We can guess a solution for f_1–f_4. In fact there are many solutions. Here are a few. (We assume that negative pixel values are not allowed.)

1	2
3	4

2	1
2	5

3	0
1	6

1.5	1.5
2.5	4.5

We see that the second solution is found by adding the following image to the first one:

$$n = \begin{array}{|c|c|}\hline 1 & -1 \\ \hline -1 & 1 \\ \hline \end{array}$$

In fact, all solutions can be found by adding a constant times n to the first solution. If we look at the solution n we see that all ray sums are zero. Therefore we will call this a "null image" or "null solution". The existence of such null solutions is a well known phenomenon in linear algebra. In

the Appendix (page 31 etc.) we give some background on linear algebra which explains this in more detail.

How can we ensure that the solution is unique? Well, we can add more projections. For example, suppose that we add a diagonal ray through pixels 2 and 3, with a ray sum of 5:

$$f_2 + f_3 = 5 \tag{2.10}$$

Now we have five equations with four unknowns. This is called an *overcomplete* system of equations. In general, there will be no solution at all, unless the system is *consistent*. In our case, this is indeed true. Consider the equations (2.6), (2.8) and (2.10). This a a system of three equations in three unknowns.

$$f_1 + f_2 = 3 \tag{2.11}$$
$$f_1 + f_3 = 4 \tag{2.12}$$
$$f_2 + f_3 = 5 \tag{2.13}$$

This can easily be solved. From (2.13) we find $f_3 = 5 - f_2$. Substitute this in equation (2.12) to find $f_1 + 5 - f_2 = 4$, that is, $f_2 = f_1 + 1$. Substitute this in equation (2.11) to find $f_1 = 1$. Then we obtain $f_2 = 2$, $f_3 = 3$, and from (2.7) $f_4 = 4$. We can easily verify that the solution $\vec{f} = (1, 2, 3, 4)$ also satisfies equation (2.9). Hence we have found a solution of all equations and in this case there is only *one*.

What we have seen in this simple example also holds in the general case. We have to make sure that we have a sufficient number of rays, otherwise the solution of the reconstruction problem may not be unique. On the other hand, if we have more rays than unknowns the solution may not exist at all, unless the system of equations is consistent. In practice, consistency does not always hold, for example because there is noise in the data or because there are measurement inaccuracies. The solution method we will consider next can also deal with this situation.

2.3.2 Kaczmarz reconstruction method: $M = 2, N = 2$

Kaczmarz developed an algebraic reconstruction technique based on a simple geometric idea. We explain it for the case of two equations in two unknowns ($M = 2, N = 2$). Then the system of equations (2.5) becomes:

$$w_{11}\, f_1 + w_{12}\, f_2 = p_1 \tag{2.14}$$
$$w_{21}\, f_1 + w_{22}\, f_2 = p_2 \tag{2.15}$$

To obtain a solution of these equations we use an *iterative* method. Starting with an initial vector $\vec{f}^{(0)}$, we compute a sequence $\vec{f}^{(0)}, \vec{f}^{(1)}, \vec{f}^{(2)}, \ldots$ which converges to the desired solution $\vec{f}^{(\infty)} = (f_1^{(\infty)}, f_2^{(\infty)})$.

The principle of the Kaczmarz method is as follows.[1] Each of these two equations can be graphically represented as a straight line in the $f_1 - f_2$ plane; see Figure 2.7. The first equation

[1] Please consult the Appendix for some basics about vectors and matrices.

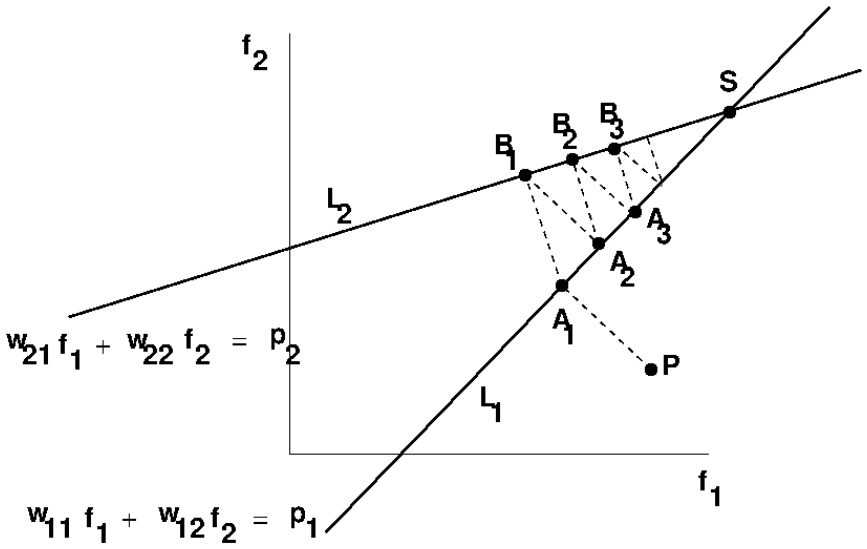


Figure 2.7: *Principle of the Kaczmarz method for two lines. The initial point P is orthogonally projected upon line L_1, then on line L_2, then again on line L_1, etc. The series of points $A_1, B_1, A_2, B_2, \ldots$ converges to the intersection point S of the two lines.*

corresponds to line L_1, the second equation corresponds to line L_2. Choose an initial point P; this corresponds to the initial solution $\vec{f}^{(0)}$. Then project point P perpendicularly upon line L_1; call the projection A_1; this corresponds to the next solution $\vec{f}^{(1)}$. Next project A_1 perpendicularly upon L_2; call the projection B_1; this corresponds to the next solution $\vec{f}^{(2)}$. Then project B_1 again line L_1; call the projection A_2. Continue this process. Then we get a series of points $A_1, B_1, A_2, B_2, \ldots$, corresponding to a sequence of approximations $\vec{f}^{(0)}, \vec{f}^{(1)}, \vec{f}^{(2)}, \ldots$, which converges to the intersection point S of the two lines. This intersection represents the solution $\vec{f}^{(\infty)}$ of the two equations.

We now want to find the formula which computes the first approximation $\vec{f}^{(1)}$ from the initial point P located at position vector $\vec{f}^{(0)}$. We denote the perpendicular projection of P on the line L by Q; the coordinate vector of Q is $\vec{f}^{(1)} = (f_1^{(1)}, f_2^{(1)})$. The equation of line L is $w_{11}\, f_1 + w_{12}\, f_2 = p_1$. Let us write $\vec{w}_1 = (w_{11}, w_{12})$ and $\vec{f} = (f_1, f_2)$. Then we can also write the equation of the line as

$$\vec{f} \cdot \vec{w}_1 = p_1 \tag{2.16}$$

Here $\vec{f} \cdot \vec{w}_1$ denotes the **inner product** of the vectors $\vec{f}$ and $\vec{w}_1$. The definition of inner product of vectors can be found in the Appendix, Section 2.A.1.

The line L intersects the line through $\vec{w}_1$ in a point A. Also, denote the perpendicular projection of P on the line through $\vec{w}_1$ by B. See Figure 2.8 for the situation.

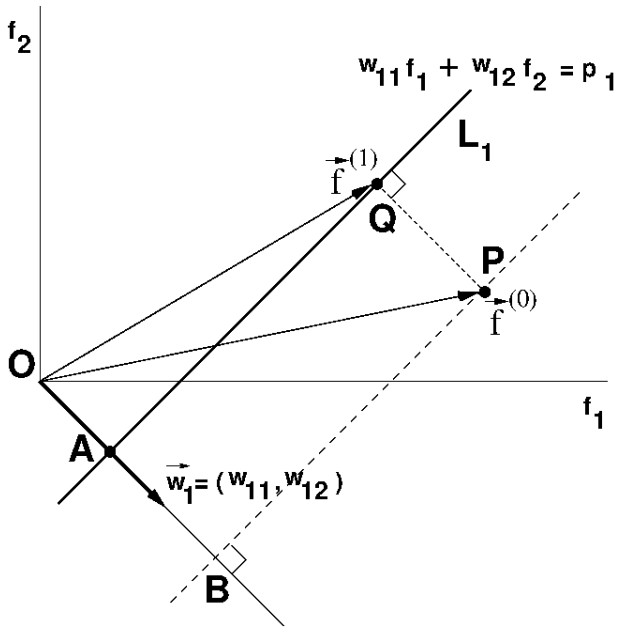


Figure 2.8: *Update step of the Kaczmarz method. The initial point P, representing solution $\vec{f}^{(0)}$, is projected perpendicularly on the line L_1 with equation $w_{11}\, f_{11} + w_{12}\, f_{12} = p_1$. The point of projection Q represents the next approximation $\vec{f}^{(1)}$.*

Then we have

$$\begin{aligned}\vec{f}^{(1)} &= \vec{O}P - \vec{Q}P \\ &= \vec{O}P - \vec{A}B \\ &= \vec{f}^{(0)} - (\vec{O}B - \vec{O}A)\end{aligned}$$

In Section 2.A.3 of the Appendix we prove that the perpendicular projection B of the vector $\vec{f}^{(0)}$ with endpoint P on a line with direction vector $\vec{w}_1$ satisfies

$$\vec{O}B = \lambda\, \vec{w}_1 \qquad \text{with } \lambda = \frac{\vec{f}^{(0)} \cdot \vec{w}_1}{\|\vec{w}_1\|^2}$$

Here $\|\vec{w}_1\|$ is the norm (or length) of the vector $\vec{w}_1$, as defined by formula (A.5) of the Appendix. If we apply the same formula to the vector $\vec{f}^{(1)}$ with endpoint A, we get

$$\vec{O}A = \mu\, \vec{w}_1 \qquad \text{with } \mu = \frac{\vec{f}^{(1)} \cdot \vec{w}_1}{\|\vec{w}_1\|^2}$$

Since point A is on the line L, we know that $\vec{f}^{(1)}$ satisfies equation (2.16), so $\vec{f}^{(1)} \cdot \vec{w}_1 = p_1$. Therefore $\mu = \frac{p_1}{\|\vec{w}_1\|^2}$.

Combining the results so far we find

$$\vec{f}^{(1)} = \vec{f}^{(0)} - \left(\frac{\vec{f}^{(0)} \cdot \vec{w}_1}{\|\vec{w}_1\|^2} \vec{w}_1 - \frac{p_1}{\|\vec{w}_1\|^2} \vec{w}_1 \right)$$

So we have derived the desired update formula:

$$\vec{f}^{(1)} = \vec{f}^{(0)} - \beta_1 \, \vec{w}_1 \tag{2.17}$$

$$\beta_1 = \frac{\vec{f}^{(0)} \cdot \vec{w}_1 - p_1}{\|\vec{w}_1\|^2} \tag{2.18}$$

A similar formula holds when computing the next estimate $\vec{f}^{(2)}$ from $\vec{f}^{(1)}$: we only have to replace $\vec{w}_1$ by $\vec{w}_2$, and p_1 by p_2.

Let us interpret the update formulas (2.17)-(2.18). For the example with $N = 4$, $M = 4$ the weight matrix has the following form (check this):

$$W = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \tag{2.19}$$

The first two rows of the matrix correspond to the two row sums, and the last two rows of the matrix correspond to the two column sums. Let us look at the expression $\vec{f}^{(0)} \cdot \vec{w}_1$. Since $\vec{w}_1 = (1, 1, 0, 0)$ we find, using the definition of inner product in formula (A.6) of the Appendix, that

$$\vec{f}^{(0)} \cdot \vec{w}_1 = 1 \cdot f_1^{(0)} + 1 \cdot f_2^{(0)} + 0 \cdot f_3^{(0)} + 0 \cdot f_4^{(0)} = f_1^{(0)} + f_2^{(0)}$$

If we convert the 1D vector representation $\vec{f}^{(0)}$ back to a 2D matrix representation, this means that $\vec{f}^{(0)} \cdot \vec{w}_1$ is just the first row sum of the matrix $f^{(0)}$; see Figure 2.6. Similarly, $\vec{f}^{(0)} \cdot \vec{w}_2$ is the second row sum of $f^{(0)}$. In the same way we find that $\vec{f}^{(0)} \cdot \vec{w}_3$ and $\vec{f}^{(0)} \cdot \vec{w}_4$ are the first and second column sums of the matrix $f^{(0)}$.

The expression $\|\vec{w}_1\|^2$ takes a very simple form (see formula (A.7)):

$$\|\vec{w}_1\|^2 = \vec{w}_1 \cdot \vec{w}_1 = 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 + 0 \cdot 0 = 2$$

This means that the expression β_1 is obtained as follows: compute the first row sum of the matrix $f^{(0)}$, subtract the row sum p_1 from it, and divide the result by 2. As a result, the first row sum of $f^{(1)}$ will have the correct value p_1, because $f^{(1)}$ corresponds to a point on the line L_1; see Figure 2.8. This can be verified by explicit computation:

$$\begin{aligned} \vec{f}^{(1)} \cdot \vec{w}_1 &= \vec{f}^{(0)} - \beta_1 \vec{w}_1 \cdot \vec{w}_1 = \vec{f}^{(0)} \cdot \vec{w}_1 - \beta_1 \vec{w}_1 \cdot \vec{w}_1 \\ &= \vec{f}^{(0)} \cdot \vec{w}_1 - (\vec{f}^{(0)} \cdot \vec{w}_1 - p_1) = p_1 \end{aligned}$$

Notice that $\beta_1 = 0$ when the first row sum of $f^{(0)}$ is already equal to the row sum p_1.

Now let us see what (2.17) means for each of the components:

$$\begin{aligned}(f_1^{(1)}, f_2^{(1)}, f_3^{(1)}, f_4^{(1)}) &= (f_1^{(0)}, f_2^{(0)}, f_3^{(0)}, f_4^{(0)}) - \beta_1(w_{11}, w_{12}, w_{13}, w_{14}) \\ &= (f_1^{(0)}, f_2^{(0)}, f_3^{(0)}, f_4^{(0)}) - \beta_1(1, 1, 0, 0) \\ &= (f_1^{(0)} - \beta_1, f_2^{(0)} - \beta_1, f_3^{(0)}, f_4^{(0)})\end{aligned}$$

In terms of the matrix representation of the image this simply means that β_1 has to be subtracted from each element of the first row of $f^{(0)}$. In the same way we find that:

- matrix $f^{(2)}$ is obtained by subtracting β_2 from each element of the second row of $f^{(1)}$
- matrix $f^{(3)}$ is obtained by subtracting β_3 from each element of the first column of $f^{(2)}$
- matrix $f^{(4)}$ is obtained by subtracting β_4 from each element of the second column of $f^{(3)}$

2.3.3 Kaczmarz reconstruction method: The general case

Now we go back to the general case where we have N pixels and M rays. That is, we have M equations of the form (2.5), one for each ray:

$$\begin{aligned} w_{11}\, f_1 + w_{12}\, f_2 + \ldots + w_{1N}\, f_N &= p_1 \\ w_{21}\, f_1 + w_{22}\, f_2 + \ldots + w_{2N}\, f_N &= p_2 \\ &\vdots \\ w_{M1}\, f_1 + w_{M2}\, f_2 + \ldots + w_{MN}\, f_N &= p_M \end{aligned}$$

If we write $\vec{w}_i = (w_{i1}, w_{i1}, \ldots, w_{iN})$ for $i = 1, 2, \ldots, M$, and $\vec{f} = (f_1, f_2, \ldots, f_N)$, then, using the inner product notation, we can write this system of equations as:

$$\begin{aligned} \vec{w}_1 \cdot \vec{f} &= p_1 \\ \vec{w}_2 \cdot \vec{f} &= p_2 \\ &\vdots \\ \vec{w}_M \cdot \vec{f} &= p_M \end{aligned}$$

Each equation will correspond to a line L_i. We project the initial point on line L_1, then project on line L_2, etc, and finally project on line L_M. Starting with an initial estimate $\vec{f}^{(0)}$, we get successive estimates $\vec{f}^{(1)}$, $\vec{f}^{(2)}$, $\vec{f}^{(3)}$, ..., $\vec{f}^{(M)}$. The formulas to get these estimates have the same form as the formulas (2.17)-(2.18) for the two-dimensional case:

$$\vec{f}^{(i)} = \vec{f}^{(i-1)} - \beta_i\, \vec{w}_i, \qquad i = 1, 2, \ldots, M \tag{2.20}$$

$$\beta_i = \left(\frac{\vec{f}^{(i-1)} \cdot \vec{w}_i - p_i}{\|\vec{w}_i\|^2} \right) \tag{2.21}$$

After all M lines have been processed, we repeat the process, and project again on line L_1, then on L_2, etc.

In practice we don't want to carry out an infinite number of projections to reach the exact solution. Therefore, we introduce so-called **stopping criteria**. That is, we will stop the iteration when a maximum number of iterations `MAX_ITER` has been reached, or at the moment when the relative difference between two successive solutions is smaller than a predefined small number ε, i.e., when

$$\delta(\vec{f}^{((k+1)M)}, \vec{f}^{(kM)}) = \frac{\left\| \vec{f}^{((k+1)M)} - \vec{f}^{(kM)} \right\|}{N} < \varepsilon \tag{2.22}$$

Here $\|\ldots\|$ again denotes the length of a vector (see formula (A.5) of the Appendix).

A pseudocode of the algorithm can be found in Algorithm 2.1.

Algorithm 2.1 *Kaczmarz method for reconstruction of a 2D image with $N = n^2$ pixels from M ray sums.*

INPUT: ray sums $p_1, p_2, \ldots, p_M$ for M rays with weight vectors $\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_M$
INPUT: maximum iteration number `MAX_ITER`; accuracy threshold ε
OUTPUT: vector representation $\vec{f} = (f_1, f_2, \ldots, f_N)$ of the reconstructed image
Initialize vector $\vec{f}^{(0)}$
for $k = 1$ **to** `MAX_ITER` **do**
  for $i = 1$ **to** M **do**
    compute $\vec{f}^{(i)}$ from $\vec{f}^{(i-1)}$ according to formulas (2.20)-(2.21)
    impose constraints (optional)
  end for
  if $\delta(\vec{f}^{(M)}, \vec{f}^{(0)}) < \varepsilon$ **then** {compute δ according to formula (2.22)}
    break
  end if
  $\vec{f}^{(0)} \leftarrow \vec{f}^{(M)}$ {reset $\vec{f}^{(0)}$}
end for

2.3.4 Remarks on the Kaczmarz method

Speed of convergence. The speed at which the projections converge to point S depends on the angles between the lines. If the angle is very small, convergence is slow. For the case of two lines, if the angle is 90° we reach point S in at most two steps (check!). If the angle is 0°, i.e., the lines are parallel, then obviously there is no solution. In this case $B_1 = A_1 = B_2 = A_2$ etc.

Choice of the initial point. We can always choose P to be be located at the origin $(0, 0, \ldots, 0)$ (this means that all elements of the initial image $f^{(0)}$ are zero). But sometimes we have a good initial estimate, which is already quite close to point S. If this is the case, the time to convergence will be decreased significantly.

The case $M < N$. When we have less equations than unknowns the solution wil in general not be unique, but an infinite number of solutions is possible (compare our simple example in

Section 2.3.1). But Kaczmarz' method still works: it will convergence to a solution $\vec{f}^{(s)}$ such that $\left\| \vec{f}^{(0)} - \vec{f}^{(s)} \right\|$ is minimized.

The case $M > N$. In practice we often have more equations than unknowns, where typically the set of equations is not consistent. Graphically, this means that the lines which represent the equations do not have a unique intersection point. Using Kaczmarz' method the "solution" does not converge to a unique point, but will oscillate in the neighbourhood of the intersection points. Nevertheless, using the stopping criterion with a not too small value of ε guarantees that we still get fairly close to the region which contains the intersection points; see Figure 2.9.

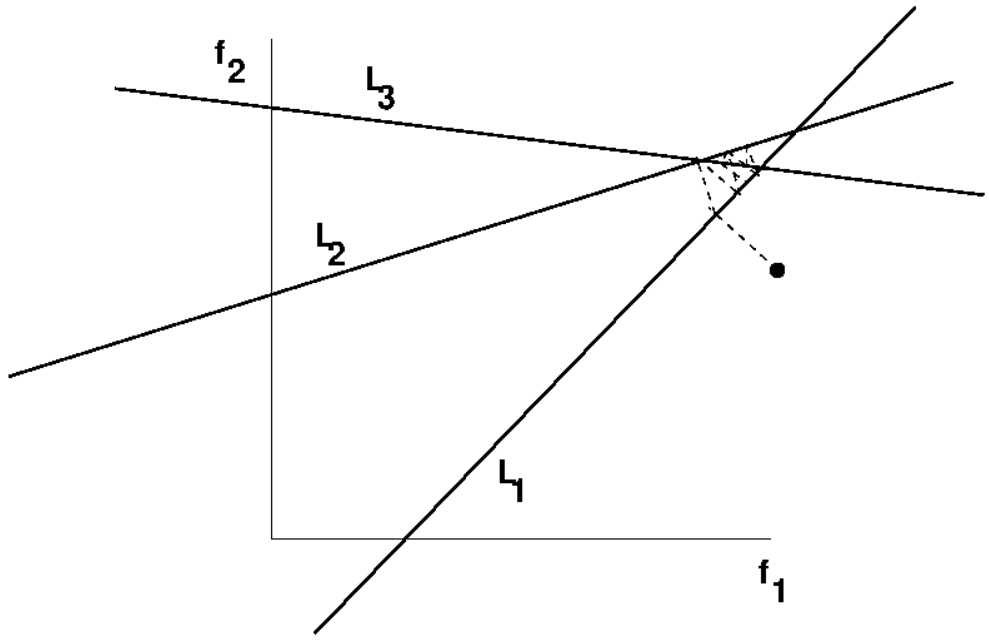


Figure 2.9: *Example of three equations which are inconsistent. The three lines L_1, L_2, L_3 do not intersect in a unique point. The "solution" wanders around in the region of the intersections.*

***A priori* information.** In Kaczmarz' method it is possible to incorporate some *a priori* information about the object one is trying to reconstruct. For example, if we know that the image $f(x, y)$ is nonnegative, then we can easily set each negative component of $\vec{f}^{(k)}$ to zero during the iteration to enforce this non-negativity. This can be done in line 8 of Algorithm 2.1.

2.3.5 Kaczmarz method for binary images

For the case of binary images, where pixel values take only two values, say 0 and 1, we can obtain fairly good reconstruction results with far fewer projections than for the case of grey scale images. Here we will consider projections for only two projections angles: the horizontal and vertical direction. This means that projections correspond to sums of rows or columns of the image.

Let us look at the pseudocode of Algorithm 2.1 and see how it simplifies for the present case.

1. Line 1: The rays sums $p_i, i = 1, 2, \ldots, M$ now consists of two sets: n column sums and n rows sums; so $M = 2n$. Let us assume that the first n values $p_i, i = 1, 2, \ldots, n$ correspond to the sums of the n columns of the image f: we denote these column sums by $p_c^{\text{col}}, c = 1, 2, \ldots, n$. The second n values $p_i, i = n+1, n+2, \ldots, 2n$ correspond to the sums of the rows $i = 1, 2, \ldots, n$ of f: we denote the row sums by $p_r^{\text{col}}, r = 1, 2, \ldots, n$. Since we know that the sums are over rows and columns, we don't have to store the direction vectors explicitly.

2. Line 3: Instead of a vector representation, we will use a matrix representation of images.

3. Line 4: As initialization we can define an $n \times n$ image $f^{(\text{init})}$ with all elements equal to zero.

4. Lines 5-9: This do-loop can now be separated into two separate do-loops: one do-loop over the n columns and one do-loop over the n rows of f.

5. Line 7: Let us look at the update formulas (2.20)-(2.21). Just as we saw in the case $M = 4, N = 4$ above (see end of Section 2.3.2), for $i = 1, 2, \ldots, n$ the vector $\vec{w}_i$ has nonzero elements only for column i of the image, so the expression $\vec{f}^{(i-1)} \cdot \vec{w}_i$ equals the sum of column i of the image $f^{(i-1)}$. Similarly, for $i = n+1, n+2, \ldots, 2n$ the vector $\vec{w}_i$ has nonzero elements only for row i of the image, so the expression $\vec{f}^{(i-1)} \cdot \vec{w}_i$ equals the sum of row i of the image $f^{(i-1)}$.
   The expression $\|\vec{w}_i\|^2$ in the denominator of formula (2.21) is now very simple: since each $\vec{w}_i$ has n ones with all other entries zero, $\|\vec{w}_i\|^2 = \vec{w}_i \cdot \vec{w}_i = n$.
   This means that the expression β_i in (2.21) is obtained as follows: compute the i^{th} column (or row) sum of $f^{(i-1)}$, subtract the i^{th} column sum p_i^{col} (or row sum p_i^{row}) from it, and divide by n. Then (2.20) means the following: subtract β_i from each element of the i^{th} column (or row). (Compare the discussion for the case $M = 4, N = 4$ at the end of Section 2.3.2.)

6. Line 8: we know that pixel values cannot become negative; also, for binary images the values can never be larger than 1. Therefore we can impose these constraints for every pixel (r, c) by the formula

$$f_{r,c}^{(i)} \leftarrow \min(\max(f_{r,c}^{(i)}, 0), 1) \tag{2.23}$$

7. Line 10. For two $n \times n$ binary images f and g, the difference $\|f - g\|$ is simply the number of pixels where f and g differ. As relative difference we can take the *fraction* of pixels where where f and g differ. This means that for the relative difference of f and g we can in our case simply take the sum of the absolute values of $f - g$ over all pixels, divided by n^2:

$$\delta(f, g) = \frac{1}{n^2} \sum_{r=1}^{n} \sum_{c=1}^{n} |f_{r,c} - g_{r,c}|$$

8. After termination of the algorithm we will end up with pixel values between 0 and 1. Since we want a *binary* image as reconstruction we perform a final rounding to the nearest integer.

A pseudocode of the algorithm for the binary case with only horizontal and vertical projections can be found in Algorithm 2.2. Of course, we can extend the algorithm by adding more projection directions, for example along diagonal directions. This means that additional do-loops have to be added.

Algorithm 2.2 *Kaczmarz method for reconstruction of a 2D binary image with $N = n^2$ pixels from $M = 2n$ ray sums (n column sums and n row sums).*

INPUT: column sums $p_1^{\text{col}}, p_2^{\text{col}}, \ldots, p_n^{\text{col}}$ and row sums $p_1^{\text{row}}, p_2^{\text{row}}, \ldots, p_n^{\text{row}}$ of an unknown image f
INPUT: maximum iteration number `MAX_ITER`; accuracy threshold ε
OUTPUT: matrix representation $(f_{r,c}^{(\text{rec})}, r = 1, \ldots, n; c = 1, \ldots, n)$ of the reconstructed image
Initialize matrix $f^{(\text{init})}$ {e.g., by setting all elements to zero}
for $k = 1$ **to** `MAX_ITER` **do**
  $f^{(0)} \leftarrow f^{(\text{init})}$
  for $c = 1$ **to** n **do** {process all columns}
    $\beta_c = \big((\text{sum of column } c \text{ of } f^{(c-1)}) - p_c^{\text{col}}\big)/n$
    $f_{r,c}^{(c)} = f_{r,c}^{(c-1)} - \beta_c \quad \forall r = 1, \ldots, n$ {subtract β_c from each element of column c of $f^{(c-1)}$}
    $f_{r,c}^{(c)} \leftarrow \min(\max(f_{r,c}^{(c)}, 0), 1) \quad \forall r = 1, 2, \ldots, n$ {impose constraints}
  end for
  $f^{(0)} \leftarrow f^{(n)}$ {reset $f^{(0)}$ to output of column loop}
  for $r = 1$ **to** n **do** {process all rows}
    $\beta_r = \big((\text{sum of row } r \text{ of } f^{(r-1)}) - p_r^{\text{row}}\big)/n$
    $f_{r,c}^{(r)} = f_{r,c}^{(r-1)} - \beta_r \quad \forall c = 1, 2, \ldots, n$ {subtract β_r from each element of row r of $f^{(r-1)}$}
    $f_{r,c}^{(r)} \leftarrow \min(\max(f_{r,c}^{(r)}, 0), 1) \quad \forall c = 1, 2, \ldots, n$ {impose constraints}
  end for
  if $\frac{1}{n^2} \sum_{r=1}^{n} \sum_{c=1}^{n} |f_{r,c}^{(n)} - f_{r,c}^{(\text{init})}| < \varepsilon$ **then**
    break
  end if
  $f^{(\text{init})} \leftarrow f^{(n)}$ {reset $f^{(\text{init})}$ to output of row loop}
end for
round elements of output $f^{(k)}$ to nearest integer

2.4 Testing the algorithm

In order to test the accuracy of the reconstruction algorithms, one needs a reference image. This image should consist of simple objects in order to be able to compute the projections of it. The so-called Shepp-Logan "head phantom" as described in Shepp and Logan (1974) is used for this purpose. This image consists of 10 ellipses as shown in Fig. 2.10. The parameters of the ellipses

are given in Table 2.1.[2]

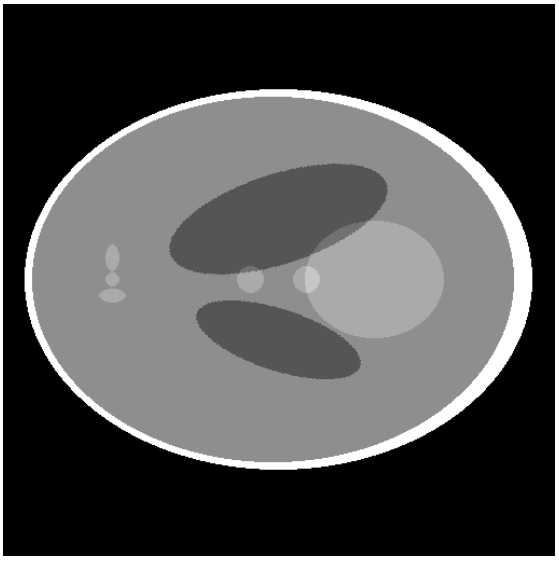


Figure 2.10: *The Shepp-Logan "head phantom".*

Table 2.1: *Parameters of the Shepp-Logan head phantom.*

Centre	Major axis	Minor axis	Rotation angle	Refractive index
$(0, 0)$	0.92	0.69	90	90
$(0, -0.0184)$	0.874	0.6624	90	-40
$(0.22, 0)$	0.31	0.11	72	-20
$(-0.22, 0)$	0.41	0.16	108	-20
$(0, 0.35)$	0.25	0.21	90	10
$(0, 0.1)$	0.046	0.046	0	10
$(0, -0.1)$	0.046	0.046	0	10
$(-0.08, -0.605)$	0.046	0.023	0	10
$(0, -0.605)$	0.023	0.023	0	10
$(0.06, -0.605)$	0.046	0.023	90	10

An advantage of using a phantom as described above is that one can give analytical expressions for the projections. The projections of an image consisting of several ellipses are simply the sum of the projections for each of the ellipses because of the linearity of the projection process. Let $f(x, y)$ be a filled ellipse centered at the origin (see Fig. 2.11),

$$f(x,y) = \begin{cases} \rho & \text{for } \frac{x^2}{A^2} + \frac{y^2}{B^2} \leq 1 \text{ (inside the ellipse)}, \\ 0 & \text{otherwise (outside the ellipse)}, \end{cases} \tag{2.24}$$

[2]Compared to Shepp and Logan (1974) we use larger values of the refractive index.

where A and B denote the half-lengths of the major and minor axis respectively, and ρ denotes the refractive index.

The projections of such an ellipse at an angle θ are given by

$$\mathcal{R}_\theta(t) = \begin{cases} \frac{2\rho AB}{a^2(\theta)}\sqrt{a^2(\theta) - t^2} & \text{for } |t| \leq a(\theta), \\ 0 & \text{for } |t| > a(\theta), \end{cases} \tag{2.25}$$

where $a^2(\theta) = A^2\cos^2\theta + B^2\sin^2\theta$. The projections for an ellipse centred at (x_1, y_1) and rotated over an angle α are easily obtained from this.

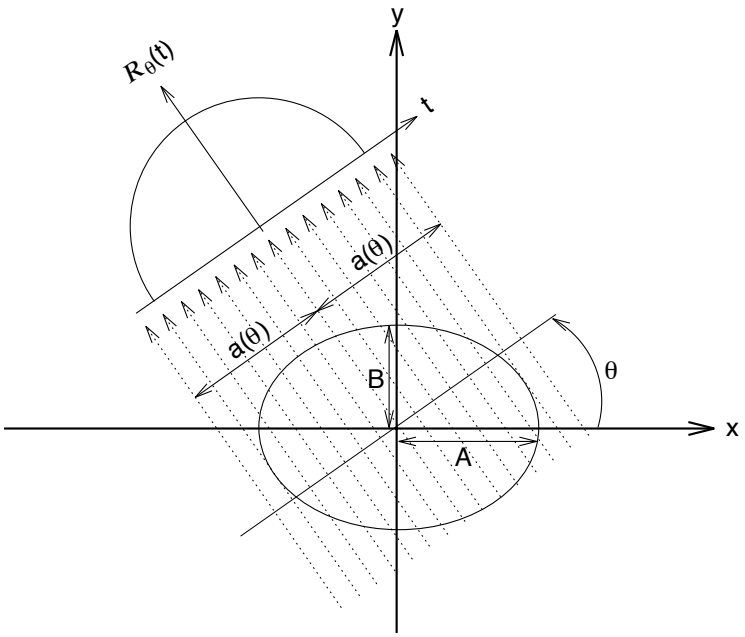


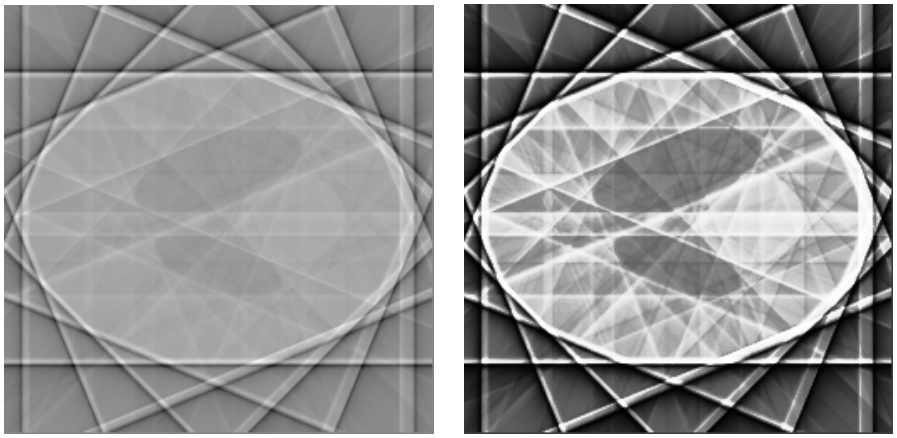
Figure 2.11: *Projection of an ellipse at an angle θ.*

We assume the projections to be available for angles θ_k uniformly distributed over the interval $[0, \pi]$, the total number of angles being denoted by N_{angles}, and a total number of N_{rays} parallel rays per profile with a constant step size. Reconstruction is performed on a 256×256 grid.

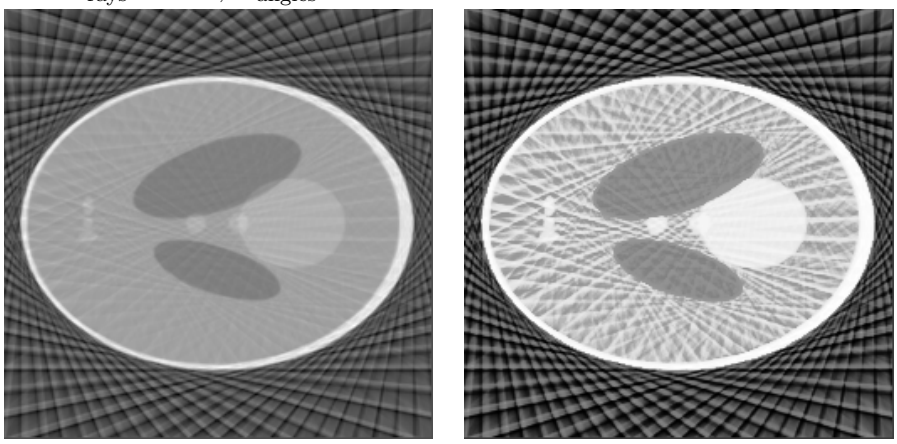
2.4.1 Results

Here we show reconstruction results for the filtered backprojection algorithm, which is much faster than ART and gives higher quality reconstructions. The algorithm was applied to projections of the Shepp-Logan head phantom. The projection data were generated using varying values of N_{angles} and N_{rays}. The reconstruction grid was of size 256×256.

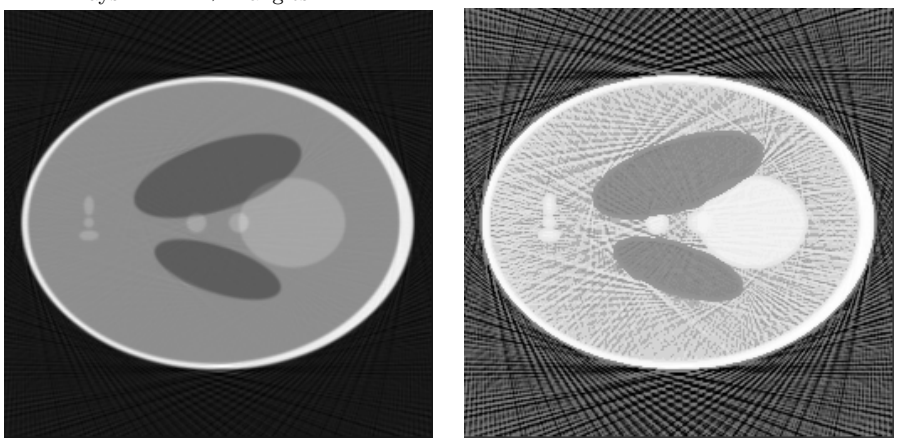
The results for $N_{\text{rays}} = 256$ with varying values of N_{angles} are shown in Figure 2.12. The results for $N_{\text{angles}} = 128$ with varying values of N_{rays} are shown in Figure 2.13. The images on the



$N_{\text{rays}} = 256, N_{\text{angles}} = 8$

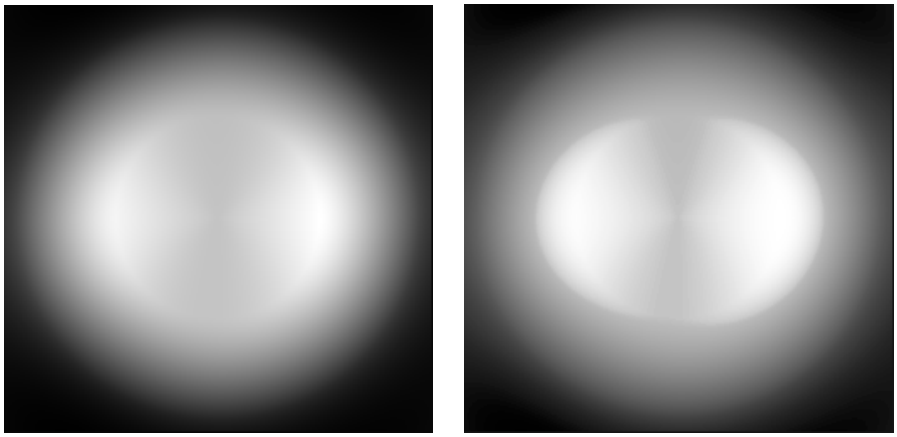


$N_{\text{rays}} = 256, N_{\text{angles}} = 32$

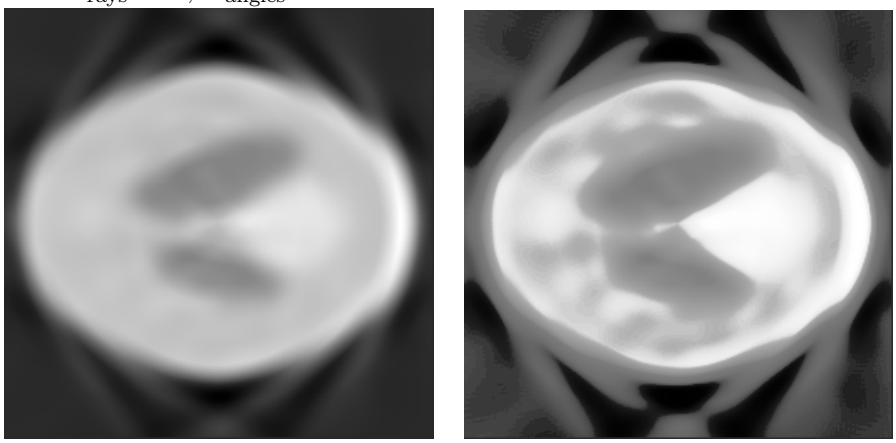


$N_{\text{rays}} = 256, N_{\text{angles}} = 128$

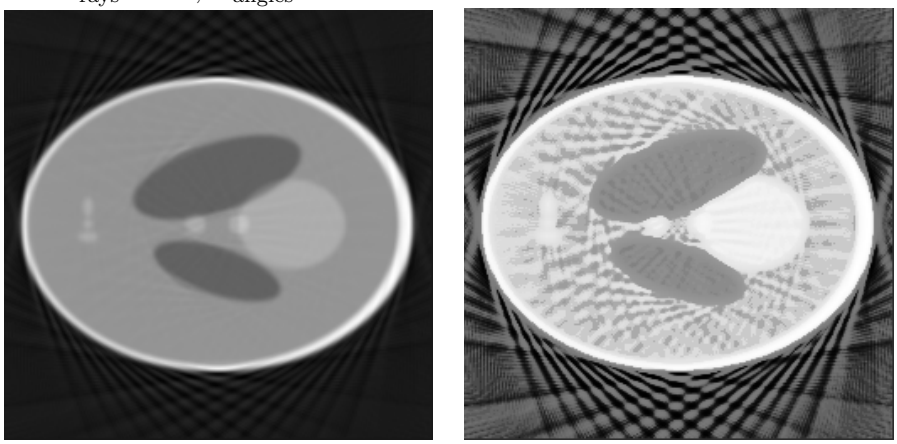
Figure 2.12: *Reconstruction of the Shepp-Logan head phantom. Projection data were generated for varying values of* N_{angles} *and* $N_{\text{rays}} = 256$ *lines per profile. The reconstruction grid was of size* 256×256*. In each row, the right image is a contrast-enhanced version of the reconstruction on the left.*



$N_{\text{rays}} = 8, N_{\text{angles}} = 128$



$N_{\text{rays}} = 32, N_{\text{angles}} = 128$



$N_{\text{rays}} = 128, N_{\text{angles}} = 128$

Figure 2.13: *Reconstruction of the Shepp-Logan head phantom. Projection data were generated for* $N_{\text{angles}} = 128$ *with varying values of* N_{rays}*. The reconstruction grid was of size* 256×256*. In each row, the right image is a contrast-enhanced version of the reconstruction on the left.*

left show the original reconstructions; the right images are a contrast-enhanced versions of the images on the left.

When N_{rays} is very low, the image looks blurred, due to undersampling of the rays per profile. When N_{angles} is very low, the image looks "angular", due to too few profiles. Note how the quality of the reconstruction improves as we increase the value of N_{angles} (more profiles) or increase the value of N_{rays} (more lines per profile). Insufficiency of the data, either by undersampling a profile or by taking the number of profiles too small, causes *aliasing* artifacts such as Gibbs phenomena, streaks (lines which are tangent to discontinuities, like the ellipse boundaries) and Moiré patterns (when the display resolution is too small); see Kak and Slaney (1988).

2.A Linear algebra

2.A.1 Matrix-vector operations

Definition of vector

A vector $\vec{x}$ is a list of elements $x_1, x_2, \ldots, x_N$. The number N is called the *dimension* of the vector. We can write the list in *row* notation or *column* notation (abstractly, both represent the same vector). For example, if $N = 3$:

$$\vec{x} = (x_1, x_2, x_3) \qquad \text{row vector} \tag{A.1}$$

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad \text{column vector} \tag{A.2}$$

Addition of vectors

We can add two vectors by adding the corresponding elements:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \end{pmatrix} \tag{A.3}$$

Multiplication of a vector by a constant

We multiply a vector by a real number λ by doing it for each element:

$$\lambda\, \vec{w} = \lambda \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} \lambda\, w_1 \\ \lambda\, w_2 \\ \lambda\, w_3 \end{pmatrix} \tag{A.4}$$

For example, we can show that every point P on a line through the vector $\vec{w}$ is of the form $\lambda\, \vec{w}$ for some $\lambda \in \mathbb{R}$; see Figure 2.14(a).

Norm of a vector

The norm of a vector $\vec{x} = (x_1, x_2, \ldots, x_N)$, written as $\|\vec{x}\|$, equals its length (see Figure 2.14(b)):

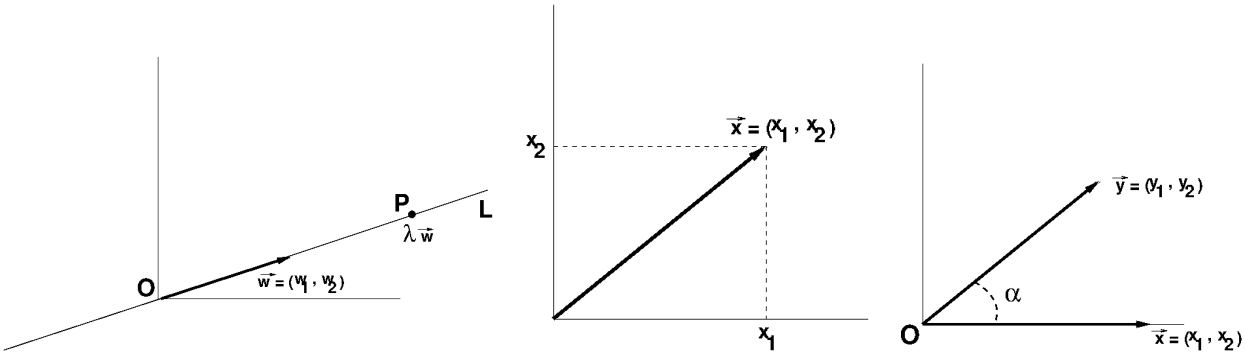
$$\|\vec{x}\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_N^2} \tag{A.5}$$

Inner product of two vectors

If $\vec{x} = (x_1, x_2, \ldots, x_N)$ and $\vec{y} = (y_1, y_2, \ldots, y_N)$ are two vectors of the same length, the inner product[3] of $\vec{x}$ and $\vec{y}$, denoted by $\vec{x} \cdot \vec{y}$, is a real number defined as follows:

$$\vec{x} \cdot \vec{y} = x_1\, y_1 + x_2\, y_2 + \ldots + x_N\, y_N \tag{A.6}$$

[3] Dutch: "inproduct".



(a) (b) (c)

Figure 2.14: *(a): Every point P on a line through the vector $\vec{w}$ is of the form $\lambda\,\vec{w}$ for some $\lambda \in \mathbb{R}$. (b) the norm $\|\vec{x}\|$ of a vector $\vec{x} = (x_1, x_2)$ is equal to its length $\sqrt{x_1^2 + x_2^2}$. (c): the inner product of two vectors $\vec{x}$ and $\vec{y}$ is equal to $\|\vec{x}\|\,\|\vec{y}\|\cos\alpha$.*

For example,

$$(1,0)\cdot(1,1) = 1\cdot 1 + 0\cdot 1 = 1$$
$$(1,1)\cdot(1,1) = 1\cdot 1 + 1\cdot 1 = 2$$
$$(1,0)\cdot(0,1) = 1\cdot 0 + 0\cdot 1 = 0$$

The second and third equations show some general properties of inner products:

- the inner product of a vector with itself is equal to the square of the norm of the vector:

$$\|\vec{x}\|^2 = \vec{x}\cdot\vec{x} \tag{A.7}$$

- if two vectors are perpendicular, their inner product is zero

In fact, it can be shown that

$$\vec{x}\cdot\vec{y} = \|\vec{x}\|\,\|\vec{y}\|\cos\alpha \tag{A.8}$$

where α is the angle between the two vectors $\vec{x}$ and $\vec{y}$.

Definition of matrix

A matrix A is a rectangular array of elements of the form:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{pmatrix} \tag{A.9}$$

The numbers M of rows and the number N of columns are called the *dimensions* of the matrix.

Addition of matrices

We can add two matrices by adding the corresponding elements. For example,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{pmatrix}$$

Also, we can multiply a matrix by a constant λ by doing it for each element, just as in the case of a vector.

Multiplying a matrix and a vector

A matrix A of dimension $M \times N$ can be multiplied with a vector $\vec{x}$ of dimension N, by taking the inner product of each row of the matrix A with the vector $\vec{x}$. For example, if $M = 3$, $N = 2$,

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_{11}\,x_1 + a_{12}\,x_2 \\ a_{21}\,x_1 + a_{22}\,x_2 \\ a_{31}\,x_1 + a_{32}\,x_2 \end{pmatrix}$$

Here the vector is multiplied "on the right". Since a vector can be regarded as a matrix of dimensions $N \times 1$, it is also possible to multiply a vector "on the left". For example,

$$\begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = (x_1\,a_{11} + x_2\,a_{21}, x_1\,a_{12} + x_2\,a_{22}, x_1\,a_{13} + x_2\,a_{23})$$

2.A.2 Solving linear equations

Let us consider again the case of four pixels and four projections of Section 2.3.1.

We can write the equations (2.6)–(2.9) in matrix form:

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} = \begin{pmatrix} 3 \\ 7 \\ 4 \\ 6 \end{pmatrix} \tag{A.10}$$

This is an equation of the general form

$$A\,\vec{x} = \vec{b} \tag{A.11}$$

where A is a matrix, $\vec{x}$ is a vector of unkowns and $\vec{b}$ is a known vector.

As we already know, a possible solution is:

$$\vec{x}^{\,*} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \tag{A.12}$$

The question is: how do we find *all* solutions?

Suppose we can find a vector $\vec{n}$ such that

$$A\,\vec{n} = \mathbf{0} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{A.13}$$

We will say that $\vec{n}$ is a vector in the **null space** of the matrix A.

Then any vector of the form

$$\vec{x} = \vec{x}^{\,*} + \lambda\,\vec{n} \qquad \lambda \in \mathbb{R} \tag{A.14}$$

is also a solution of equation (A.10). To prove this, we need the following facts, which you can easily verify:

1. If we multiply a matrix with a sum of two vectors we can get the same result by multiplying the matrix with each vector separately, and then adding the results, that is,

$$A\,(\vec{x}_1 + \vec{x}_2) = A\,\vec{x}_1 + A\,\vec{x}_2$$

2. If we multiply a matrix with a vector, which is itself multiplied with a scalar value λ, we can get the same result by multiplying the matrix with the vector first, and then multiplying the result by λ, that is,

$$A\,(\lambda\,\vec{x}) = \lambda\,A\,\vec{x}$$

Now we can prove that the vector $\vec{x}$ as defined in equation (A.14) is also a solution of equation (A.11):

$$A\,\vec{x} = A\,(\vec{x}^{\,*} + \lambda\,\vec{n}) = A\,\vec{x}^{\,*} + A\,(\lambda\,\vec{n}) = A\,\vec{x}^{\,*} + \lambda\,A\,\vec{n} = A\,\vec{x}^{\,*} + \lambda\,\mathbf{0} = A\,\vec{x}^{\,*} = \vec{b}. \tag{A.15}$$

That is, we can always multiply $\vec{n}$ with some real number λ and add it to $\vec{x}^{\,*}$, and the result will still be a solution of equation (A.10).

Of course, the vector $\vec{n}$ where all elements are zero always satisfies equation (A.13). We call this the *trivial* solution. It does not lead to a solution which is different from the solution $\vec{x}^{\,*}$. To see if there are nontrivial vectors $\vec{n}$ which satisfy equation (A.13) for the case of equation (A.10), we have to solve the following system of equations:

$$\begin{aligned} n_1 + n_2 &= 0 \\ n_3 + n_4 &= 0 \\ n_1 + n_3 &= 0 \\ n_2 + n_4 &= 0 \end{aligned} \tag{A.16}$$

When solving such sets of equations we can always add two equations to get a new one which replaces one of the original equations. If we add equation 1 and 2 we get:

$$n_1 + n_2 + n_3 + n_4 = 0$$

But if we add equation 3 and 4 we get the same equation. This means that the set of equations is *linearly dependent*.

We can take $n_1 = 1$ (because when $n_1 = 0$ we find from the equations that $n_2 = n_3 = n_4 = 0$, which would lead to the trivial solution). Then we find $n_2 = -1$, $n_3 = -1$, $n_4 = 1$. So a nontrivial solution of the equations (A.16) is:

$$\vec{n} = \begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix} \tag{A.17}$$

If we represent this vector as a 2×2 image we get

$$n = \begin{array}{|c|c|} \hline 1 & -1 \\ \hline -1 & 1 \\ \hline \end{array}$$

which is exactly what we already found in section 2.3.1.

2.A.3 Projection

We need a few properties of projection. Consider a line L through the vector $\vec{w}$, and another vector $\vec{v}$, see Figure 2.15(a). The vector $\vec{v}$ is projected on the line L. Let the point of projection be B. Then we know that the length of the projected vector $\vec{OB}$ is equal to

$$\left\| \vec{OB} \right\| = \|\vec{v}\| \cos\alpha \tag{A.18}$$

where α is the angle between $\vec{v}$ and $\vec{w}$. But we also know that

$$\vec{OB} = \lambda \, \vec{w}$$

for some $\lambda \in \mathbb{R}$. So

$$\left\| \vec{OB} \right\| = \|\lambda \, \vec{w}\| = \lambda \, \|\vec{w}\| \, . \tag{A.19}$$

If we equate the two expressions (A.18) and (A.19), we find

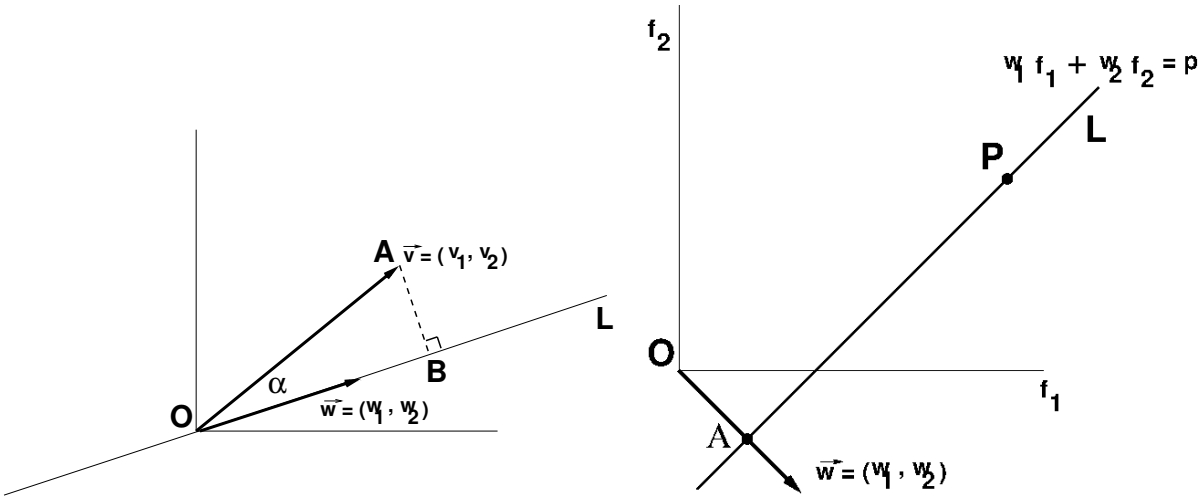
$$\lambda = \frac{\|\vec{v}\| \cos\alpha}{\|\vec{w}\|}$$

From equation (A.8) we know that

$$\cos\alpha = \frac{\vec{v}\cdot\vec{w}}{\|\vec{v}\|\,\|\vec{w}\|}$$

Therefore we find the projection formula:

$$\vec{OB} = \lambda\,\vec{w} \qquad \text{with } \lambda = \frac{\vec{v}\cdot\vec{w}}{\|\vec{w}\|^2} \tag{A.20}$$



(a) (b)

Figure 2.15: *(a): when a vector $\vec{v}$ is projected on a line L through the vector $\vec{w}$, then the length of the projected vector $\vec{OB}$ is equal to $\|\vec{v}\|\cos\alpha$, where α is the angle between $\vec{v}$ and $\vec{w}$. (b): a line L with equation $w_1\,f_1 + w_2\,f_2 = p$ is perpendicular to the vector $\vec{w} = (w_1, w_2)$ and intersects the line through $\vec{w}$ at a distance $\frac{p}{\|\vec{w}\|}$ from the origin.*

Now suppose we have a line L with equation $w_1\,f_1 + w_2\,f_2 = p$, where $\vec{w} = (w_1, w_2)$ is a fixed vector and and $\vec{f} = (f_1, f_2)$ are the coordinates of the points on L; see Figure 2.15(b). We can also write this equation in the form

$$\vec{w}\cdot\vec{f} = p$$

We will show that the vector $\vec{w}$ is perpendicular to the line L. Let A be the intersection of L with the line through $\vec{w}$. Then, since A is a point on the line through $\vec{w}$, the vector $\vec{OA}$ has the form

$$\vec{OA} = \lambda\,\vec{w} = (\lambda\,w_1, \lambda\,w_2)$$

But since A is also on the line L, its coordinates satisfy the equation of the line L, so $w_1\, f_1 + w_2\, f_2 = p$ where $f_1 = \lambda\, w_1$ and $f_2 = \lambda\, w_2$. This implies

$$\begin{aligned} & w_1\, f_1 + w_2\, f_2 = p \\ & w_1\, (\lambda\, w_1) + w_2\, (\lambda\, w_2) = p \\ & \lambda\, (w_1^2 + w_2^2) = p \\ & \lambda\, \|\vec{w}\,\|^2 = p \\ & \lambda = \frac{p}{\|\vec{w}\,\|^2} \end{aligned}$$

On the other hand, take any point P on the line L with coordinates $\vec{f} = (f_1, f_2)$, and project it perpendicularly on the line through $\vec{w}$. Let the point of projection be denoted by B. Then we know from the projection formula (A.20) that

$$\begin{aligned} \vec{O}B &= \mu\, \vec{w} \\ \mu &= \frac{\vec{f} \cdot \vec{w}}{\|\vec{w}\,\|^2} = \frac{p}{\|\vec{w}\,\|^2}. \end{aligned}$$

So we see that $\lambda = \mu$ and in fact the points A and B are the same!

The length of the vector $\vec{O}A$ is given by:

$$\left\|\vec{O}A\right\| = \|\lambda\, \vec{w}\,\| = \left\| \frac{p}{\|\vec{w}\,\|^2}\, \vec{w} \right\| = \frac{p}{\|\vec{w}\,\|^2}\, \|\vec{w}\,\| = \frac{p}{\|\vec{w}\,\|}$$

Conclusion:

> The line with equation $\vec{w} \cdot \vec{f} = p$ is perpendicular to the line through $\vec{w}$ and intersects this line at a distance from the origin given by $\dfrac{p}{\|\vec{w}\,\|}$.

Bibliography

Deans, S. R., 1983. The Radon Transform and Some of Its Applications. J. Wiley.

Herman, G. T., 1980. Image Reconstruction from Projections: the Fundamentals of Computerized Tomography. Academic Press.

Kak, A. C., Slaney, M., 1988. Principles of Computerized Tomographic Imaging. IEEE Press, New York.

Shepp, L. A., Kruskal, J. B., 1978. Computerized tomography: The new medical x-ray technology. Am. Math. Monthly 85, 420–439.

Shepp, L. A., Logan, B. F., 1974. The Fourier reconstruction of a head section. IEEE Transactions on Nuclear Science NS–21, 21–43.

Chapter 3

Stochastic dynamics: Markov chains

In this chapter we study systems that evolve in discrete timesteps according to some stochastic (probabilistic) rules. We will restrict ourselves to a particular type of stochastic system, called *Markov Chains*.

As an application we will discuss Google's PageRank algorithm.

3.1 Markov chains

A Markov chain is a stochastic dynamical system, that is, a system that evolves in time according to some probabilistic rules. We will assume that the system can be in a finite number of states at discrete time steps $t = 0, 1, 2, \ldots$. The characteristic property of a Markov chain is that the probability that the system is in a certain state at time t depends only on the state probabilities at the previous step $t - 1$.

To make things simple we will from now on consider a very simple Markov chain, the so-called *random walk* problem. Assume a *random walker* that can be in two states, 1 and 2. At each discrete time $t = 0, 1, 2, \ldots$ the walker can make a transition to another state or stay in the same state, with a certain probability. We assume that the walker goes from state 1 to state 2 with *transition probability* p, which means the walker stays in state 1 with probability $1 - p$ (where $0 < p < 1$). Similarly, the walker goes from state 2 to state 1 with *transition probability* q, so

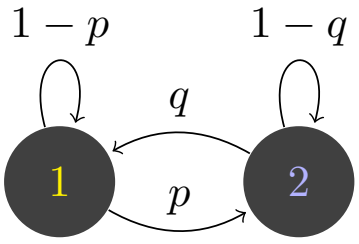


Figure 3.1: *Random walk with two states 1 and 2. The transition probabilities are indicated on the links.*

stays in state 2 with probability $1 - q$ (where $0 < q < 1$).

The probability of the walker to be in state i at time t is denoted by $x_i(t)$, $i = 1, 2$. We combine these two probabilities into a vector $\vec{x}\,(t) = (x_1(t), x_2(t))$, which is called the *state vector* of the Markov chain. Note that the state at time t only depends on the state at the previous time $t - 1$: this is the *Markov property* mentioned above. When $\vec{x}\,(0)$ is given, then the sequence $\vec{x}\,(1)$, $\vec{x}\,(2)$, ... is uniquely defined.

We present now (without proof) an important property of the random walk. When $t \to \infty$ the elements of the state vector become *constant*, *independent of the initial condition*: $x_1(t) \to x_1^\infty$, $x_2(t) \to x_2^\infty$. The probability vector $\vec{x}^{\,\infty} = (x_1^\infty, x_2^\infty)$ is called the *limiting* or *equilibrium distribution* of the walk.

The following equations relate the probabilities at time t to the probabilities at time $t - 1$:

$$(1 - p)\, x_1(t-1) + q\, x_2(t-1) = x_1(t) \tag{3.1}$$
$$p\, x_1(t-1) + (1 - q)\, x_2(t-1) = x_2(t) \tag{3.2}$$

Equation 3.1 can be understood as follows.

1. The walker is in state 1 at time $t - 1$ with probability $x_1(t - 1)$, and can stay in this state at the next step with probability $1 - p$. This gives a contribution $(1 - p)\, x_1(t - 1)$ to the probability $x_1(t)$ to be in state 1 at time t.

2. The walker is in state 2 at time $t - 1$ with probability $x_2(t - 1)$, and can make a transition from state 2 to state 1 at the next step with probability q. This gives a contribution $q\, x_2(t - 1)$ to the probability $x_1(t)$ to be in state 1 at time t.

3. Adding the two contributions gives Eq. 3.1.

Eq. 3.2 can be understood in the same way.

The equations (3.1) and (3.2) that relate the probabilities at time t with the probabilities at time $t - 1$ can be written in matrix form as

$$\vec{x}\,(t) = A\, \vec{x}\,(t-1) \tag{3.3}$$

where

$$A = \begin{pmatrix} 1-p & q \\ p & 1-q \end{pmatrix} \quad \vec{x}\,(t-1) = \begin{pmatrix} x_1(t-1) \\ x_2(t-1) \end{pmatrix} \quad \vec{x}\,(t) = \begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix}$$

We call A the *transition matrix* or *Markov matrix* of the Markov chain.

Note that the sum of elements in each column equals 1: $\sum_i A_{ij} = 1$.

The time evolution of the Markov chain can be studied by looking at iterations of (3.3):

$$\vec{x}(t) = A\,\vec{x}(t-1) = A^2\,\vec{x}(t-2) = A^3\,\vec{x}(t-3) = \dots \tag{3.4}$$

where $A^2 = A\,A$, $A^3 = A\,A\,A$, etc.

The powers of A can be interpreted as follows.

$$\begin{aligned} A_{ij} &= \text{probability to go from state } j \text{ to state } i \text{ in } \textit{one step} \\ (A^2)_{ij} &= \text{probability to go from state } j \text{ to state } i \text{ in } \textit{two steps} \\ \dots &\qquad\qquad \dots \\ (A^n)_{ij} &= \text{probability to go from state } j \text{ to state } i \text{ in } n \ \textit{steps} \end{aligned}$$

3.1.1 Computing the equilibrium distribution

The equations that define the equilibrium probabilities x_i^∞ or simply x_i obey the equations:

$$(1-p)\,x_1 + q\,x_2 = x_1 \tag{3.5}$$
$$p\,x_1 + (1-q)\,x_2 = x_2 \tag{3.6}$$

or in matrix form

$$A\,\vec{x} = \vec{x} \tag{3.7}$$

where

$$A = \begin{pmatrix} 1-p & q \\ p & 1-q \end{pmatrix} \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Now there is the so-called **Perron-Frobenius theorem**, which says that the solution of Eq. 3.7 exists and is *unique up to a scaling factor*. This scaling factor is determined by the condition that the sum of the probabilities equals 1,

$$x_1 + x_2 = 1. \tag{3.8}$$

Rewriting the equations (3.5) and (3.6) that define the equilibrium probabilities we get:

$$-p\,x_1 + q\,x_2 = 0 \tag{3.9}$$
$$p\,x_1 - q\,x_2 = 0 \tag{3.10}$$

Clearly this is a singular (but consistent) linear system of equations, with infinitely many solutions. However, using the normalization equation $x_1 + x_2 = 1$ we get the unique solution (check!):

$$x_1 = \frac{q}{p+q}, \quad x_2 = \frac{p}{p+q}. \tag{3.11}$$

Computing the equilibrium distribution by hand may be difficult or impossible when the number of states of the random walk becomes very large. In that case the computation can be done by numerically iterating the equation $\vec{x}(t) = A\vec{x}(t-1)$, until a desired accuracy is obtained.

3.2 The PageRank algorithm

PageRank is an algorithm used by Google Search to rank websites in their search engine results. This algorithm was developed by Larry Page and Sergey Brin, the founders of Google (Brin and Page, 1988).

PageRank works on the link structure of the World Wide Web. It does not involve the actual content of any Web pages or individual queries. It is frequently recomputed. When a query is submitted, Google finds the pages on the Web that match that query and lists those pages in the order of their PageRank. The basic idea is that a page has high rank if many other pages link to it.

The global way PageRank works is as follows. It counts the number and quality of links to a web page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites. See also `https://en.wikipedia.org/wiki/PageRank`.

The PageRank algorithm can be connected to the concept of random walk in the following way (see Fig. 3.2). Imagine surfing the Web, going from page to page by *randomly* choosing an *outgoing link* from one page to get to the next. To avoid getting stuck in *dead ends* at pages with no outgoing links or *cycles* of interconnected pages, choose a *random page* from the Web a certain fraction of the time. The limiting probability that a random surfer visits any particular page is its **PageRank**. The sum of the PageRanks of all Web pages will be equal to 1.

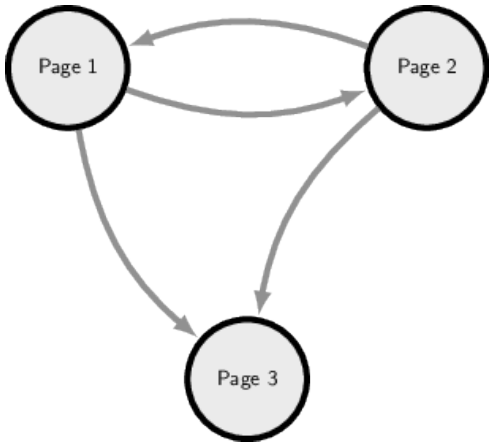


Figure 3.2: *Simple network with three web pages.*

3.2.1 PageRank: defining the Markov chain

The Markov chain that describes the random walker (or "surfer") on a network with n nodes, is defined as follows.

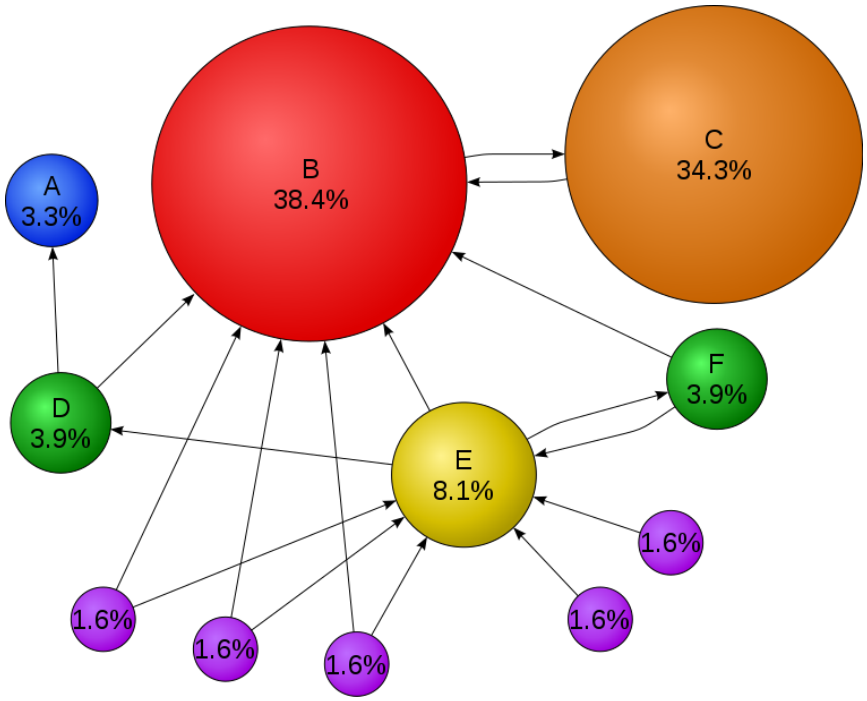


Figure 3.3: *Larger web network. The PageRanks (equilibrium probabilities of the random surfer) are indicated on the nodes.*

First we construct the $n \times n$ *connectivity matrix* G of the network W (set of web pages):

$$G_{ij} = \begin{cases} 1 & \text{if node j is linked to node i} \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

See Fig. 3.3 for an example of such a network. In practice, the value of n (number of web pages) is very large (many billions). The number of nonzeros in G is the total number of hyperlinks in W.

Next define the *probability* p that the random surfer follows an outgoing link. A typical value is $p = 0.85$. This probability is divided equally over all outgoing links of a node. Let c_j be the sum of the j-th column of G (the out-degree of the j-th node, that is, the number of outgoing links) and let $\delta = (1-p)/n$ be the *probability* that the surfer chooses a random page (not following a link). The introduction of δ is needed to avoid that the surfer becomes stuck on a page without outgoing links or gets trapped in a cycle. Most of the elements of A are equal to δ, the probability of jumping from one page to another without following a link. The value of δ is typically very small. For example, when $n = 4 \cdot 10^9$ and $p = 0.85$, then $\delta = 3.75 \cdot 10^{-11}$.

Putting the above ingredients together we can define the *Markov matrix* of the network:

$$A_{ij} = \begin{cases} p\, G_{ij}/c_j + \delta & \text{if } c_j \neq 0 \\ 1/n, & \text{if } c_j = 0 \end{cases} \tag{3.13}$$

3.2.2 Computing the PageRanks

Now that we have determined the Markov matrix A (according to Eq. (3.13)) of the random surfer we can compute the PageRanks by the following steps.

1. Compute the *equilibrium distribution* of the random surfer by solving

$$A\vec{x} = \vec{x}$$

   where A is the Markov matrix of the surfer, and the vector $\vec{x}$ is normalized ($\sum_{i=1}^{n} x_i = 1$). The elements x_i can be initialized arbitrarily, for example uniformly: $x_i = 1/n, i = 1, 2, \ldots, n$.

2. The equilibrium probabilities x_i, $i = 1, 2, \ldots, n$ are the ***PageRanks*** of the nodes i.

3. *Reorder* the nodes in decreasing order of the x_i-values (probabilities), resulting in a permutation $j = perm(i)$, $j = 1, 2, \ldots, n$ of the n indices i.

4. This gives the list of nodes (websites) $j = 1, 2, \ldots, n$ with associated PageRanks x_j in decreasing order (importance).

In practice, the matrices G and A are so big that they are never actually computed directly. Instead, one can take advantage of the particular (sparse) structure of the Markov matrix. Numerical routines dedicated to solving large sparse linear systems of equations exists for this purpose.

Bibliography

Brin, S., Page, L., 1988. The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems 30 (1-7), 107–117.

Chapter 4

Modelling and simulation of pattern formation

In this chapter we will look at models of pattern formation. In particular we will study how models with very simple local interactions between components can lead to very complex and intriguing spatial patterns. Many mathematical techniques are available to formulate such models, for example based on partial differential equations, which treat time and space as *continuous* variables; examples of such equations are discussed in Chapter 8. In this chapter we will study so-called *Cellular Automata*, which are *discrete* in time and space. They were first proposed by Von Neumann and Ulam, see von Neumann (1966). A very popular cellular automaton is *Conway's Game of Life*, invented by the British mathematician John Horton Conway (Gardner, 1970). For introductions and information about the history of cellular automata, see Schiff (2007); Wolfram (2002); Chopard and Droz (1998). Cellular Automata modeling of biological pattern formation is discussed in Deutsch and Dormann (2005).

4.1 Cellular Automata: Introduction

Cellular Automata (CA) provide a way of making simplified models of many natural phenomena. In a CA model we divide the universe into small *cells*. Each cell can be in one of a finite number of states, e.g., "live" or "dead". Then we define simple rules, called *transition rules*, to change the state of each cell, depending on the states of its neighbours. In this way we get a new configuration of cells. Following biological terminology, this new configuration is called the next "generation". This process is then repeated by iterative application of the rules, so we get a succession of generations[1]. In principle this iteration can go on forever; in computer implementations it goes on for a finite (possibly quite large) number of generations.

There is much interest in CA models among ecologists or biologists. This arises from the often observed fact that a simple model with relatively few states and relatively simple rules can often produce very realistic and complicated behaviour. There is also a huge interest amongst

[1] In more mathematical terms, we speak of a *discrete-time dynamical system.*

mathematicians and computer scientists, especially those concerned with the theory of artificial intelligence.

4.2 A simple CA with majority voting

Consider a universe consisting of a subset of the plane, divided up into an n by n array of small cells, indexed by (i, j), for $i, j = 1, \ldots, n$. The states of these cells may be collectively represented by the elements $A(i, j)$ of a matrix (two-dimensional array) A, for $i, j = 1, \ldots, n$. The immediate neighbours of cell (i, j) are defined to be the eight cells that touch it, see Figure 4.1. So a cell (i, j) which is not on the boundary of the universe (that is, $i = 2, \ldots, n-1$ and $j = 2, \ldots, n-1$) has immediate neighbours $(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$, $(i, j-1)$, $(i, j+1)$, $(i+1, j-1)$, $(i+1, j)$ and $(i+1, j+1)$.

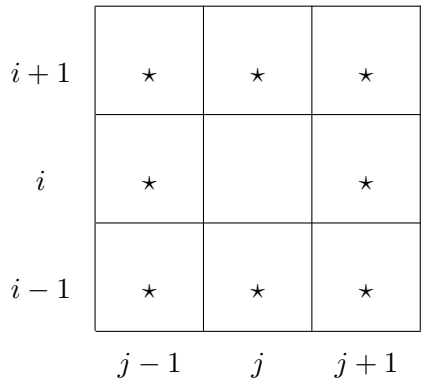


Figure 4.1: *Immediate neighbours (cells with a $\star$) of an interior cell (i, j).*

The next step is to define different types of states for the cells and transition rules for obtaining new generations. We assume that each cell can occupy only two states, called "alive" and "dead". To be specific, we define:

$$A(i,j) = \begin{cases} 1 & \text{if cell } (i,j) \text{ is alive} \\ 0 & \text{if cell } (i,j) \text{ is dead} \end{cases}$$

The transition rules for constructing each generation from the previous one are as follows (note the rules are the same in each generation, that is, they do not depend on time):

- A *live* cell dies in the next generation if more than 4 of its immediate neighbours in the present generation are dead; otherwise it stays alive.

- A *dead* cell comes to life in the next generation if more than 4 of its immediate neighbours in the present generation are alive; otherwise it stays dead.

Note that these rules correspond to a form of *majority voting*.

Since the computer is finite, our universe has to be finite as well, so it will have boundaries. To apply the above rules consistently, we therefore need to define the states of the cells that lie just outside the boundary of the universe. This can be done in various ways. One possibility, which is often used, is to assume the following:

- All cells outside the boundary of our universe are assumed to be dead.

This means that we can effectively extend the boundary of our universe by two rows and two columns to include cells $(0, j)$, $(n+1, j)$, $j = 0, \ldots, n+1$, and $(i, 0)$, $(i, n+1)$, $i = 0, \ldots, n+1$, all of which are defined to be dead.

Remark 4.1 Note that the new state of a cell depends on the states of all neighbouring cells in the *present* generation. So, even if the states of some neighbours of a cell (i, j) already have been updated, these new values should not be used in computing the new state of cell (i, j). ♢

4.2.1 Behaviour of the simple CA

Now we consider various initial cell configurations, and then see what happens in successive generations. We start with very simple initial configurations, and make things more complex as we go on.

Example 4.2 *Initialization*: There is only *one* live cell (i, j), and all other cells are dead.

Then in generation 2 the cell (i, j) will be dead as well. So after generation 2 all cells remain dead. We say that we have reached a *steady state* of the discrete-time dynamical system.[2] ♢

Example 4.3 *Initialization*: *All* cells are alive.

Then in generation 2 all interior cells will be alive as well. Cells on the boundary, but not on the corners, have 5 alive neighbours, so remain alive as well. The four corner cells ($(1, 1)$, $(1, n)$, $(n, 1)$, (n, n)) have only three alive neighbours, so they will be dead in generation 2.

After generation 2 no more changes occur (check!). So again we have reached a *steady state*. ♢

Example 4.4 *Initialization*: Cells are chosen to be alive or dead in a *random* way.

Now we observe that a very irregular random initial state is smoothed out by the birth and death rules, and eventually arrives at a pattern of alive and dead regions with rather smooth interfaces between them. This is illustrated in Fig. 4.2, where alive cells are represented by black pixels and dead cells by white pixels. In this example a steady state is reached after 14 generations. For different initial random distributions with the same fraction of alive cells, the number of generations needed to reach a steady state will in general be different as well. ♢

[2] Or, in mathematical terms, a *stationary solution* or *fixed point*.

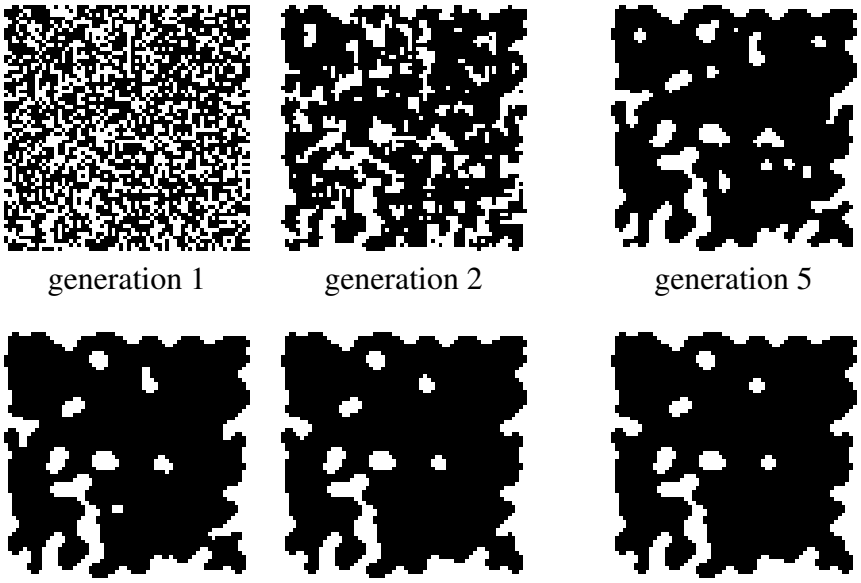


Figure 4.2: *Evolution of cellular automaton with majority rule and $n = 64$, initialized by a random distribution of alive cells (black pixels) and dead cells (white pixels). The initial fraction of alive cells is 0.60, while the fraction in the steady state is 0.79.*

4.3 Conway's Game of Life

Now we consider a CA which has the same states as before, i.e., 1 and 0, standing for "alive" and "dead", respectively. But we modify the transition rules a bit. Again each cell looks at its eight cells immediately around it. Here are the rules.

- an **alive** cell dies if it has **more than 3 or less than 2 alive neighbours**
- a **dead** cell becomes alive if it has **3 alive neighbours**
- in all other cases the state does not change

The first rule is inspired by the biological phenomena of "isolation" and "over-crowding", respectively. Again we may assume that all cells outside the boundary of our universe are dead.

Now something spectacular happens: although the change of transition rules seems not be very large, the number of possible dynamical behaviours of our new CA is extremely rich.

You can have spatial configurations which are *static*, i.e., they never change (stationary solutions or "still lives"). Some configurations lead to *periodic* solutions ("blinkers", "oscillators"), i.e., the configuration repeats itself after a fixed number P of generations; P is called the *period* of the periodic solution. Some configurations glide across the plane ("gliders", "spaceships"); or eat the gliders; or throw off the gliders like waste. Some configurations eventually reach a steady state, some keep on changing, sometimes even growing to infinite sizes.

Let us again consider a number of examples. See also `http://en.wikipedia.org/wiki/Conway's_Game_of_Life` for some nice animations.

Example 4.5 [**Still lives**] *Initialization*: the configurations shown in Fig. 4.3 are all examples of static solutions (check!). ♢

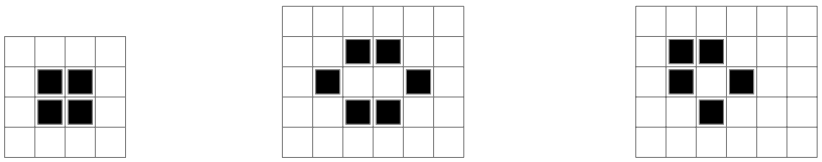
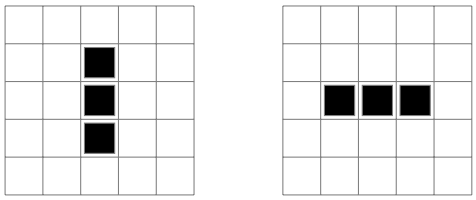


Figure 4.3: ***Still lives****: initial configurations which are static under Game of Life-rules. Black cells are alive, the other cells are dead.*

Example 4.6 [**Oscillators**] The configurations shown in Fig. 4.4 are examples of periodic solutions. After two generations the pattern repeats itself: the solution has **period 2**. ♢

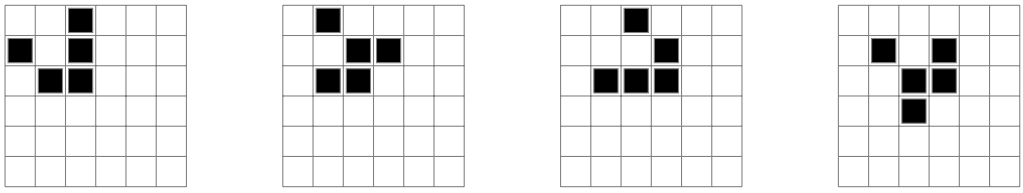


generation 1 generation 2

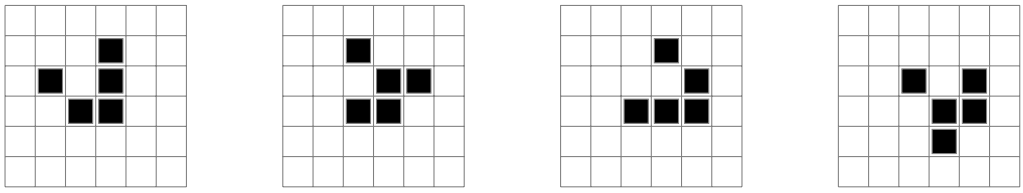
Figure 4.4: ***Blinker****: configuration which repeats itself under Game of Life-rules. Black cells are alive, the other cells are dead. The patterns in generations 1 and 2 are indicated. The period is 2, so the patterns in generation* $2n-1, n = 1, 2, \ldots$ *are the same; the same holds for the patterns in generation* $2n, n = 1, 2, \ldots$.

Example 4.7 [**Glider**] *Initialization*: the initial configuration consists of 5 alive cells in a sea of dead cells, as shown in Fig. 4.5.

The generations 2,3 and 4 are all different. But in generation 5, the pattern is the same as in generation 1, except from a translation in the south-east direction. This means that, when the iteration is continued from this point on, we get the same sequence of 4 patterns as before, apart from the motion in the south-east direction. When this is animated, it looks like a crawling animal slowly sliding down a slope, hence the name "glider". ♢



generation 1 generation 2 generation 3 generation 4



generation 5 generation 6 generation 7 generation 8

Figure 4.5: ***Glider****: evolution of cellular automaton with Game of Life-rule. Black cells are alive, the other cells are dead. After 4 generations the pattern repeats itself, except from a translation in the south-east direction.*

Example 4.8 [**Gosper glider gun**] Conway originally conjectured that no pattern can grow forever, i.e., that for any initial configuration with a finite number of alive cells, the population cannot grow beyond some finite upper limit. However, this conjecture turned out to be false.

In 1970, a team from the Massachusetts Institute of Technology, led by Bill Gosper, came up with a configuration now known as the "Gosper glider gun". It is shown in Fig. 4.6. When this configuration is evolved under the Game of Life-rules, it "produces" an infinite number of gliders. The first glider is produced on the 15th generation, and another glider every 30th generation from then on. ♢

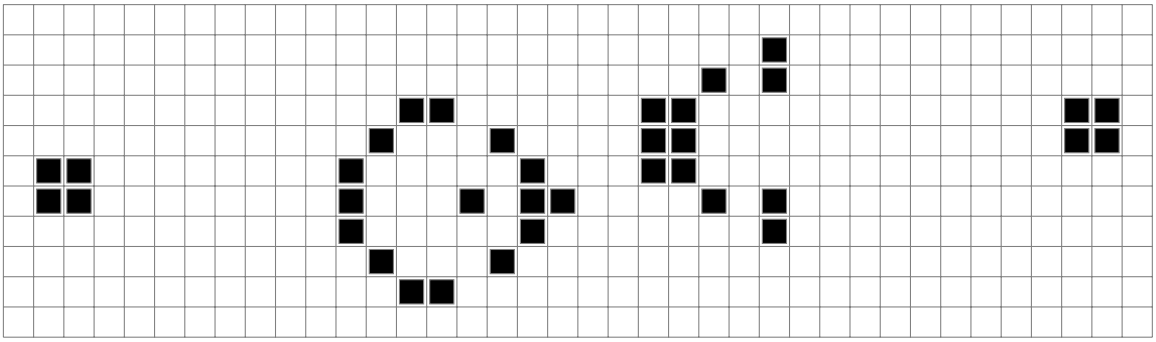


Figure 4.6: ***Gosper glider gun****: produces an infinite number of gliders under Game of Life-rules.*

4.4 Variations and generalizations

Variations on Game of Life. New life-like cellular automata may be obtained by:

- Changing the *transition rules*.
- Changing the *grid*. For example, instead of a square grid, a hexagonal grid can be used.
- Changing the *grid dimension*. Instead of a 2D grid, a 1D or 3D grid can be used.
- Changing the *number of states*. Instead of two states (live and dead), three or more states per cell can be used.

1-D Cellular Automata. In 1-D, there are $2^3 = 8$ possible configurations for a cell and its two immediate neighbors. The rule defining the cellular automaton must specify the resulting state for each of these possibilities, so there are $2^8 = 256$ possible 1-D cellular automata.

Chaotic behaviour. Under certain transition rules initial patterns evolve in a pseudo-random or chaotic manner.

Self-replication. A self-constructing pattern called "Gemini" was invented by Andrew J. Wade in 2010, which creates a copy of itself while destroying its parent. This pattern replicates in 34 million generations. See `http://conwaylife.com/wiki/index.php?title=Gemini`.

Reversibility. A cellular automaton is called *reversible* if for every current configuration of the cellular automaton there is exactly one past configuration. For one-dimensional cellular automata algorithms exist for deciding whether a rule is reversible or not. For cellular automata in two or more dimensions reversibility is *undecidable*: there is no algorithm that is guaranteed to determine correctly whether the automaton is reversible.

Universal Turing machine. It has been shown that Game of Life is a very powerful computational machine. In fact, Game of Life is theoretically as powerful as any computer with unlimited memory and no time constraints: it is a universal Turing machine.

Bibliography

Chopard, B., Droz, M., 1998. Cellular Automata Modeling of Physical Systems. Cambridge University Press.

Deutsch, A., Dormann, S., 2005. Cellular Automaton Modeling of Biological Pattern Formation. Birkhäuser Boston, Cambridge, MA.

Gardner, M., October 1970. Mathematical games. The fantastic combinations of John Conway's new solitaire game "life". Scientific American 223, 120–123.

Schiff, J. L., 2007. Cellular Automata: A Discrete View of the World. John Wiley & Sons, New York, NY.

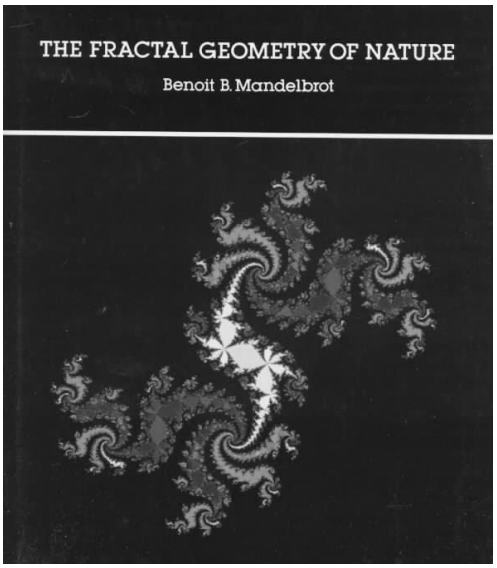
von Neumann, J., 1966. The Theory of Self-reproducing Automata (A. Burks, ed.). Univ. of Illinois Press, Urbana, IL.

Wolfram, S., 2002. A New Kind of Science. Wolfram Media.
URL `www.wolframscience.com/nksonline`

Chapter 5

Dynamics in the complex plane

In this chapter we study dynamical systems generated by transformations (or "mappings") of the complex plane. Certain types of such transformations can be used to generate pictures of geometrical structures that are known as *fractals*. The term "fractal" was coined by Benoit Mandelbrot in his book *The Fractal Geometry of Nature* (Mandelbrot, 1982), see also Barnsley (1988). Since then fractals have become a playground for Computer Graphics, and many amazing pictures have been produced by this method.



The theory behind fractals requires some deep mathematics, involving the calculus of complex numbers. We will restrict ourselves here to the most elementary aspects needed. Therefore, let us first have a brief look at complex numbers.

5.1 Intermezzo: Complex numbers

A complex number z has the form

$$z = a + i\,b, \quad i^2 = -1 \tag{5.1}$$

where a and b are real numbers, called the real part and the imaginary part of z, respectively, and i is the complex unit, defined by $i^2 = -1$. The number z can be visualized in the complex plane as a point (a, b), where the horizontal axis represents the real part and the vertical axis represents the imaginary part; see Fig. 5.1. To each complex number z is associated the *complex conjugate* number z^* defined by

$$z^* = a - i\,b, \qquad \text{(complex conjugate)} \tag{5.2}$$

Geometrically z^* is obtained by reflecting the point z w.r.t. the horizontal axis.

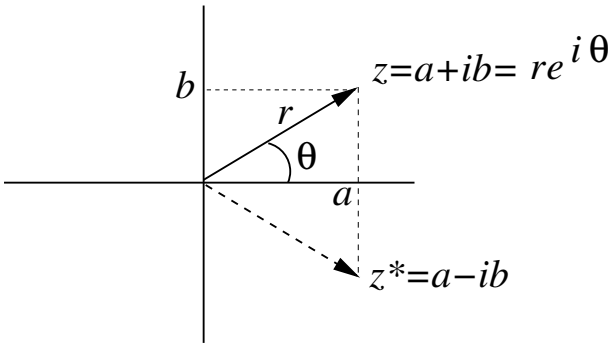


Figure 5.1: *Point z and its complex conjugate z^* in the complex plane.*

A complex number has also a *polar representation* (see Fig. 5.1):

$$z = r\,e^{i\theta}, \qquad r = |z| = \sqrt{z\,z^*} = \sqrt{a^2 + b^2} \text{ (magnitude)}$$
$$\theta = \arctan\Big(\frac{b}{a}\Big) \text{ (phase)}$$

5.2 Mappings in the complex plane

Consider a mapping $f(z)$ that maps points to points in the complex plane:

$$z' = f(z)$$

We can *iterate* this mapping (see Fig. 5.2):

$$z_{k+1} = f(z_k), \qquad k = 0, 1, 2, \ldots \tag{5.3}$$

To save writing let us define

$$f^{(n)}(z) = \overbrace{f(f(f(\ldots)))}^{n\ times}(z)$$

So:

$$z_k = f^{(k)}(z_0)$$

where z_0 is the starting point.

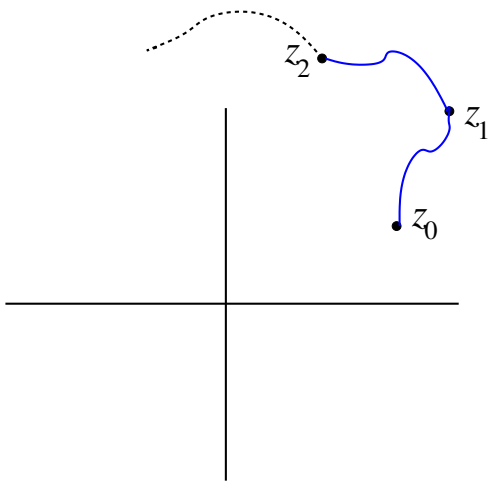


Figure 5.2: *Iterated mapping in the complex plane.*

The sequence $z_0, z_1, z_2, \ldots$, is called the *trajectory* or *orbit* of the dynamical system (5.3).

We can now ask the following question: what kind of behaviours are possible when the number of iterations becomes *(infinitely) large*? In general, the answer is not easy to give. Therefore, in the following we consider a very special mapping, the *quadratic equation*.

5.3 The homogeneous quadratic equation

Consider the mapping $f(z) = z^2$. So

$$z_{k+1} = z_k^2 \qquad k = 0, 1, 2, \ldots$$

To analyze what happens when this mapping is iterated, consider the polar representation $z = r\, e^{i\,\phi}$. Then the equation becomes:

$$r_{k+1}\, e^{i\,\phi_{k+1}} = (r_k\, e^{i\,\phi_k})^2 = r_k^2\, e^{2\,i\,\phi_k}$$

Taking the magnitude on both sides we get:

$$r_{k+1} = r_k^2$$

This implies, for any $r_k \neq 0$, that

$$\phi_{k+1} = 2\phi_k$$

Let us consider a number of cases for z_0; see Fig. 5.3.

1. z_0 is in Region I. Then $r_0 < 1$, so $r_k = r_0^{2k} \to 0$. All points inside the unit circle eventually move to the origin.

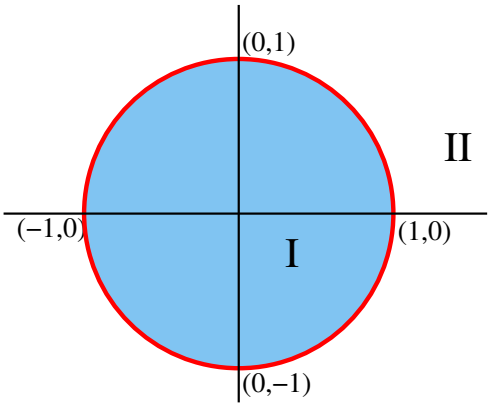


Figure 5.3: *Regions of the complex plane.*

2. z_0 is in Region II. Then $r_0 > 1$, so $r_k = r_0^{2k} \to \infty$. All points outside the unit circle eventually move to infinity.

3. z_0 is on the unit circle. Then $r_0 = 1$, so $r_k = r_0^{2k} = 1$. All points stay on the unit circle.
   Moreover, $\phi_k = 2^k\,\phi_0$. Examples:

   (a) $\phi_0 = 0$. Then $\phi_k = 0$. So the trajectory of $z_0 = 1 + 0\,i$ is a *fixed point*.

   (b) $\phi_0 = \pi/2$. Then $\phi_1 = \pi$, $\phi_2 = 2\,\pi$.
   So the trajectory of $z_0 = 0 + i$ *ends at a fixed point*.

   (c) $\phi_0 = 2\pi/3$. Then $\phi_1 = 4\pi/3$, $\phi_2 = 8\pi/3 = 2\pi/3$.
   So the trajectory of $z_0 = -\frac{1}{2} + i\,\frac{1}{2}\sqrt{3}$ is a *periodic orbit (cycle) of period 2*.

So for the homogeneous quadratic equation the possible behaviours are very simple. More interesting behaviours occur when the equation is *inhomogeneous*.

5.4 The inhomogeneous quadratic equation

Consider the mapping $f(z) = z^2 + c$. So,

$$z_{k+1} = z_k^2 + c \qquad k = 0, 1, 2, \ldots$$

To analyze what type of dynamical behaviour is described by this equation requires very advanced mathematics.

Two types of question can be asked:

1. *Fix the initial point* $z_0 = 0$, and study the behaviour of the orbit $z_0, z_1, z_2, \ldots, z_k, \ldots$ *for various values of the parameter* c. This leads to the concept of the **Mandelbrot set**.

2. *Fix the parameter* c, and study the behaviour of the orbit $z_0, z_1, z_2, \ldots, z_k, \ldots$ *for various values of the initial point* z_0. This leads to the concept of the **Julia set**.

5.4.1 The Mandelbrot set

We start by the iteration with fixed initial condition $z_0 = 0$:

$$\begin{aligned} z_{k+1} &= z_k^2 + c \\ z_0 &= 0 \end{aligned} \tag{5.4}$$

The first iterates are:

$$\begin{aligned} z_1 &= c \\ z_2 &= c^2 + c \\ z_3 &= (c^2 + c)^2 + c \end{aligned}$$

Question: *What is the set M of parameter values c in the complex plane for which the orbit $z_0 = 0, z_1, z_2, \ldots, z_k, \ldots$ remains bounded as $k \to \infty$?* This set M is called the Mandelbrot set.

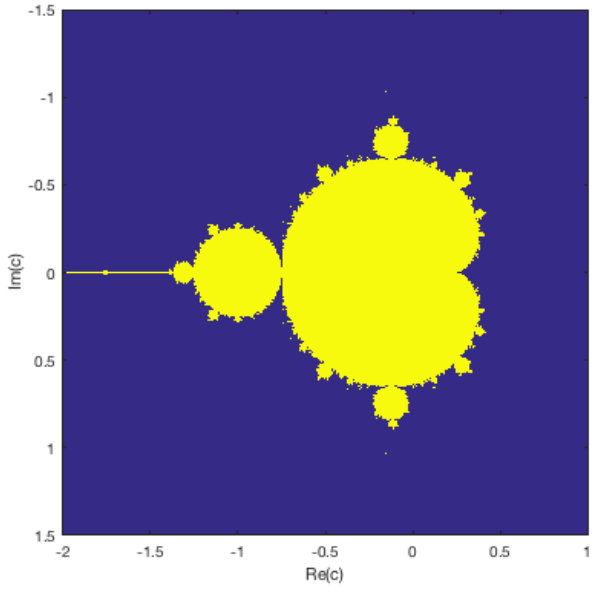


Figure 5.4: *Picture of the Mandelbrot set (yellow region).*

The set M is called the *Mandelbrot set*. It can be displayed as an image in the complex c-plane (yellow region in Fig. 5.4.).

As we can see, the Mandelbrot set has a very complicated structure.

To get a better idea, let us consider some points in / outside the Mandelbrot set M. For this we have to determine whether for a given value of c the iterates of (5.4) remain bounded or not.

Some points *inside* M:

1. $c = 0$:
   orbit $z_1 = 0$, $z_2 = 0$, etc. 0 is a *fixed point*, so $c = 0$ is in M.

2. $c = -1$:
   orbit $z_1 = -1$, $z_2 = 0$, $z_3 = -1$, $\ldots$, so $c = -1$ is in M. The orbit is a *cycle of period 2*.

3. $c = -2$:
   orbit $z_1 = -2$, $z_2 = 2$, $z_3 = 2$, etc. The orbit settles on a *fixed point*, so $c = -2$ is in M.

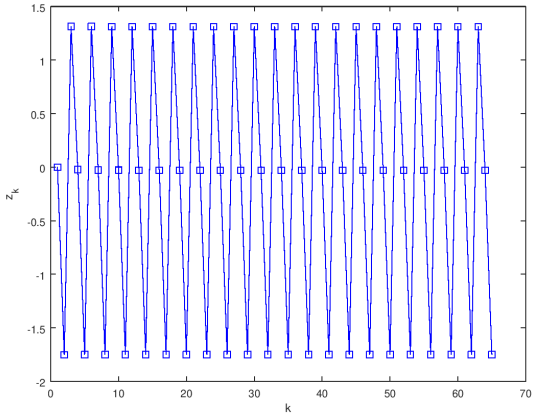
4. $c = i$:
   orbit $z_1 = i$, $z_2 = -1 + i$, $z_3 = -i$, $z_4 = -1 + i$, $\ldots$. The orbit settles on a *cycle of period 2*, so $c = i$ is in M.

Some points *outside* M:

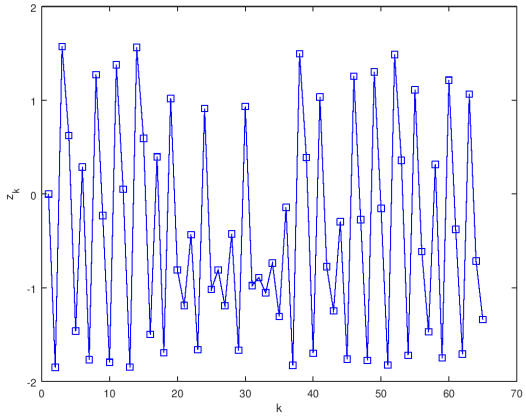
1. $c = 1$:
   orbit $z_1 = 1$, $z_2 = 2$, $z_3 = 5$, $z_4 = 26$, $\ldots$.
   This grows forever, so $c = 1$ is *not* in M.

2. $c = 2i$:
   orbit $z_1 = 2i$, $z_2 = -4 + 2i$, $z_3 = 12 - 14i$, $z_4 = -52 - 334i$, $\ldots$.
   This grows forever, so $c = 2i$ is *not* in M.

Here are more orbits for real c, one orbit which is periodic and one which is chaotic:



$c = -1.751$. The orbit is a *period 3* cycle.



$c = -1.85$. The orbit is *chaotic*.

Real numbers in the Mandelbrot set

Let us now consider points on the real axis of the complex c-plane, and see which of these points fall inside the Mandelbrot set.

So assume c is a *real* number, $z_{k+1} = z_k^2 + c$, $z_0 = 0$. Then all iterates z_k are real as well. The following statement can be proved: *The set of all real numbers c in the Mandelbrot set is the interval* $[-2, 0.25]$.

This can be seen as follows.

1. We have seen that $c = -2$ is in the Mandelbrot set. If $c < -2$ then z_k increases in each iteration without bound. So $c = -2$ is the left limit of real points in the Mandelbrot set.
2. If c is positive, each iteration z_k is greater than the one before. For the orbit z_k to reach a *limit*, say z, as $k \to \infty$, it must be the case that

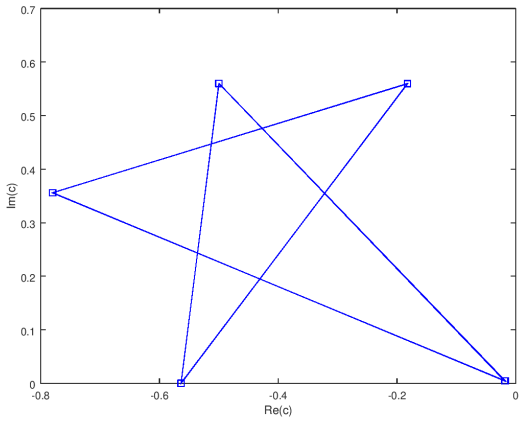
   $$z = z^2 + c$$

   This equation has the solution

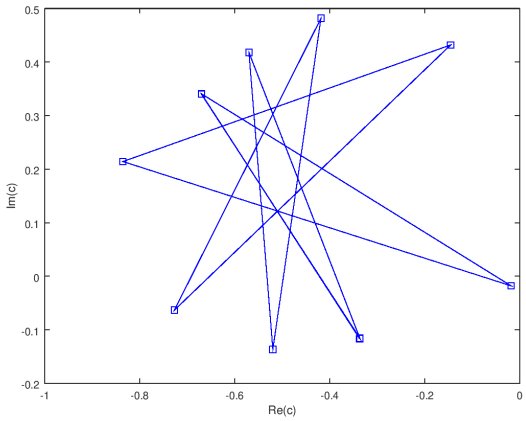
   $$z = \frac{1 \pm \sqrt{1 - 4c}}{2} \tag{5.5}$$

   which becomes imaginary at $c > \frac{1}{4}$. So $c = \frac{1}{4}$ is the largest real number in the Mandelbrot set, with the limit of the orbit equal to $z = \frac{1}{2}$. (For $c < \frac{1}{4}$ the solution z is the one with the minus sign in (5.5).)

Here are a few more orbits for complex c:



$c = -0.5 + 0.56i$. A *period 5* cycle.



$c = -0.67 + 0.34i$. A *period 9* cycle.

Periods of points inside the Mandelbrot set

The periods of periodic orbits are *constant* for any c inside the so-called *primary bulbs* of the Mandelbrot set; see Fig. 5.5. For more information, see Robert L. Devaney, `http://math.bu.edu/DYSYS/FRACGEOM/FRACGEOM.html`

Computing the Mandelbrot set

Let us now consider how the Mandelbrot set can be computed numerically.

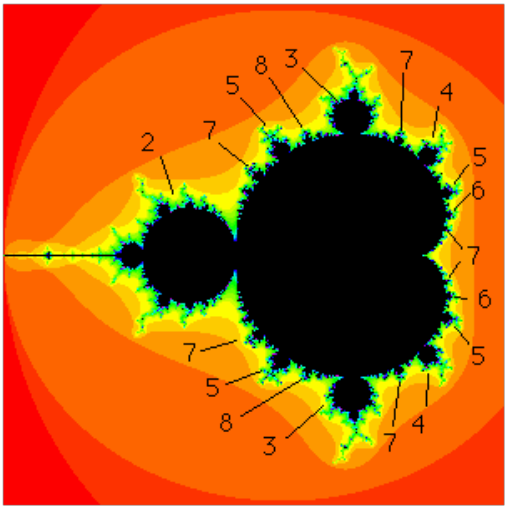


Figure 5.5: *The Mandelbrot set and its primary bulbs (labeled by integers).*

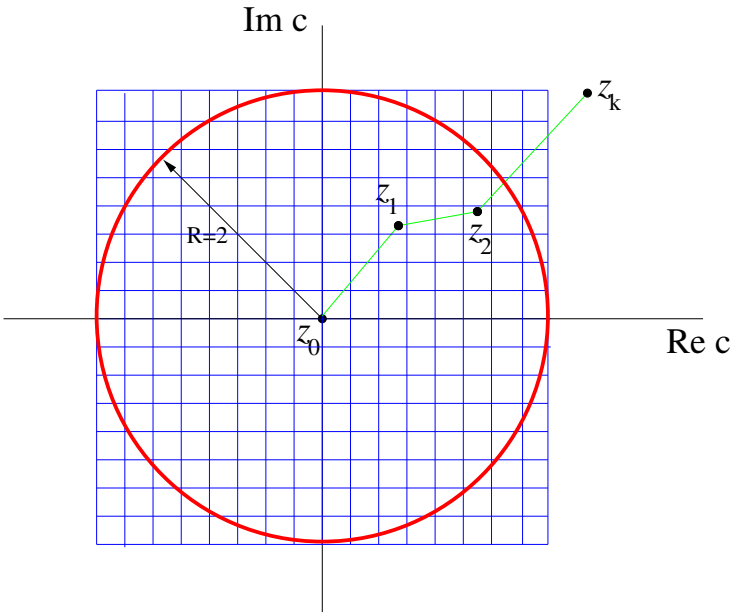


Figure 5.6: *Grid in the plane with escape circle with radius* $R = 2$.

First, define a grid of pixels covering the area within the square $-2 \leq c \leq 2$, $-2 \leq c \leq 2$ of the complex c-plane. For each pixel, the coordinates of one of the corner points (say, the lower-left corner) are used to define the value of c to be used in the iteration. (In Matlab, the `meshgrid` function can be used to define a 2D grid.) This is repeated for all values of c on the grid.

To check (for a given value of c) whether the orbit $z_k, k = 0, 1, 2, \ldots$ goes to infinity we use an *escape circle* of radius $R = 2$; see Fig. 5.6. It has been proven that if $|z_k|$ passes this circle, it will never return to a point within this circle, but will rapidly escape to infinity.

The maximum number of iterations needed can be quite low, say $n_{\max} = 100$. Actually, it depends on the *resolution* of the image. If we zoom the Mandelbrot set near its boundary, it takes many iterations before it can be decided whether the orbit passes the escape circle, so the maximum iteration number has to be increased.

Computing the Mandelbrot set: the fringe

The most interesting behaviours occur near the boundary of the Mandelbrot set. To get a better picture of this region, we adapt the algorithm as follows.

Instead of determining whether a point escapes to infinity or not, we compute the *number of iterations* k required for z_k to escape the disc of radius 2. The value of k will be between 0 and the maximum number of iterations $n_{\max}$ ("depth"), which we set as a parameter of the algorithm. Typical values are several hundreds to a few thousands.

Two cases can be distinguished:

1. If $k = n_{\max}$ then z_k did not get larger than 2, so the current c is part of the Mandelbrot set. We can colour the pixel black (or any colour we want).

2. If $k < n_{\max}$, however, then we know that this c does not belong to the Mandelbrot set. Large values of k (but smaller than $n_{\max}$) indicate that c is in the *fringe*, that is, the area *just outside the boundary*. We map the value of k to a colour and draw the pixel with that colour using a colourmap.

Figure 5.7 gives an example of the Mandelbrot set with its fringe, computed by implementing the above algorithm in Matlab.

By varying the colourmap different artistic effects can be achieved, as is shown in Fig. 5.8.

Computing the Mandelbrot set: zooming in

A very interesting property of the Mandelbrot set is that it is self-similar. This can be shown if we zoom in to a small part near the boundary if the Mandelbrot set.

So consider the region:

$$\begin{aligned} -0.133 &\leq Re(c) \leq -0.123 \\ -0.9925 &\leq Im(c) \leq -0.9825; \end{aligned}$$

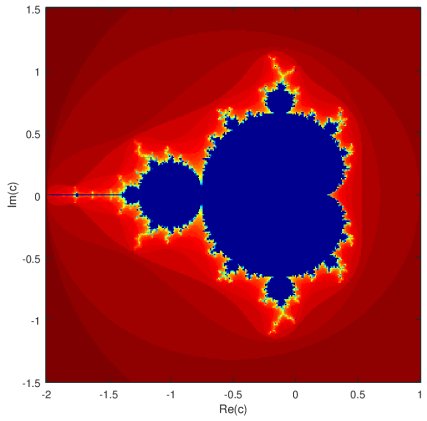
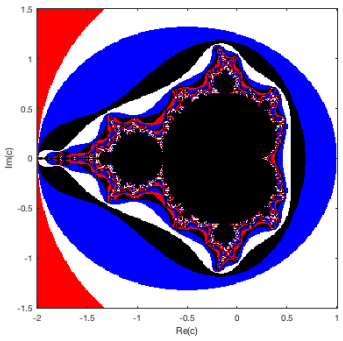
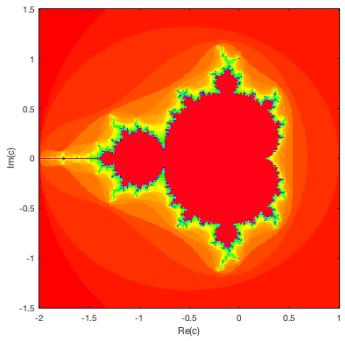


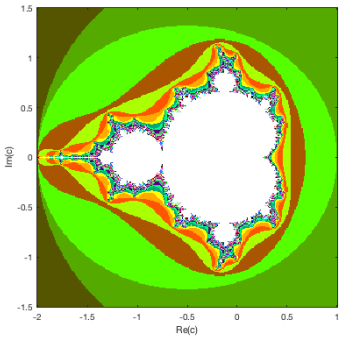
Figure 5.7: *Picture of the Mandelbrot set with fringe. The flipudjet colourmap has been used*



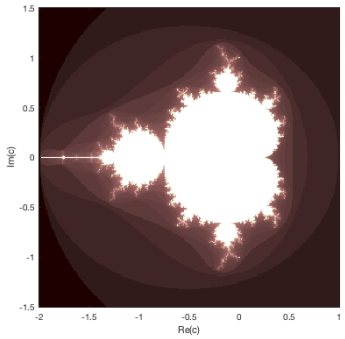
flag colourmap



hsv colourmap



colourcube colourmap



pink colourmap

Figure 5.8: *Same as Fig. 5.7, but with different colourmaps.*

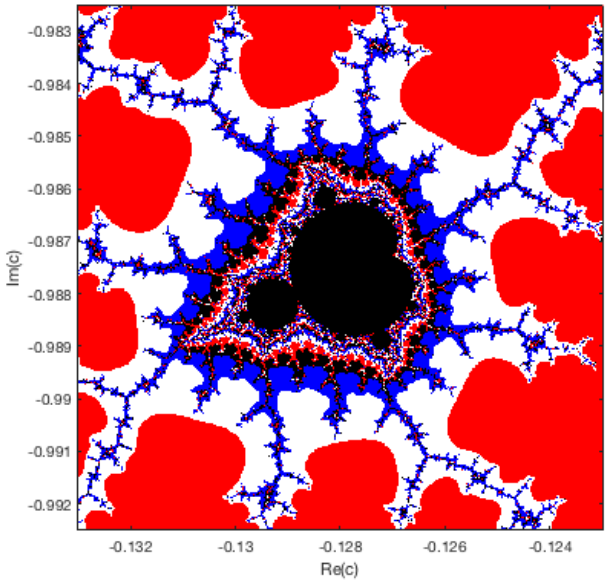


Figure 5.9: *A small region near the boundary of the Mandelbrot set: a "baby-Mandelbrot set" appears. Colourmap: "flag".*

A picture of the Mandelbrot set for this region is shown in Fig. 5.9. We have used the grid size: 512^2, and have set the number of iterations to $n_{\max} = 512$. We observe *self-similarity* under zooming: a mini-version of the Mandelbrot set (a "baby-Mandelbrot set") appears! This process can be repeated, by further zooming in. At all zoom levels new copies of the Mandelbrot set appear.

Many other regions have been found, where interesting visual patterns occur when zooming in. Here are two examples.

Valley of the Seahorses Figure 5.10 gives a picture for the region

$$\begin{aligned} -0.82 &\leq Re(c) \leq -0.72 \\ -0.18 &\leq Im(c) \leq -0.08; \end{aligned}$$

Grid size: 1024^2; number of iterations: $n_{\max} = 512$.

Left tip of the Mandelbrot set Figure 5.11 gives a picture for the region

$$\begin{aligned} -2.000 &\leq Re(c) \leq -1.998 \\ -0.001 &\leq Im(c) \leq 0.001 \end{aligned}$$

Grid size: 512^2; number of iterations: $n_{\max} = 100$.

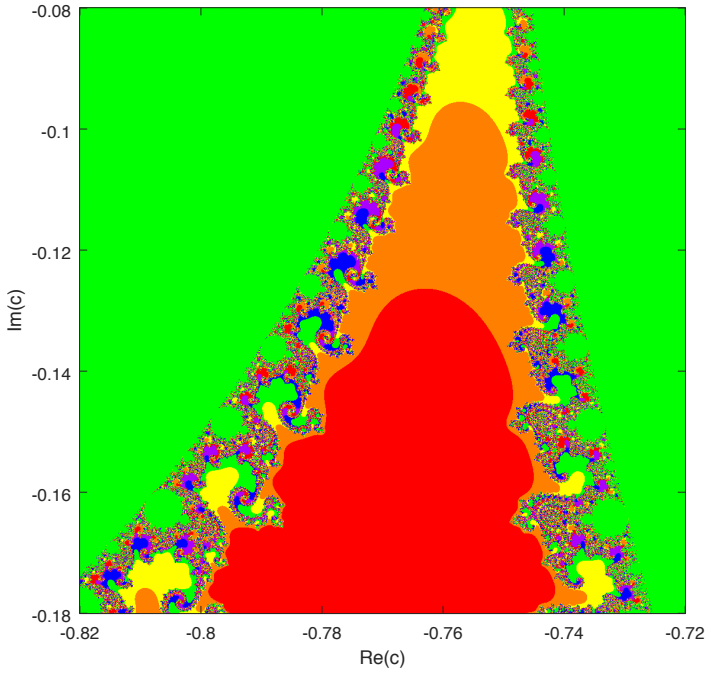


Figure 5.10: *Valley of the Seahorses. Colourmap:"prism".*

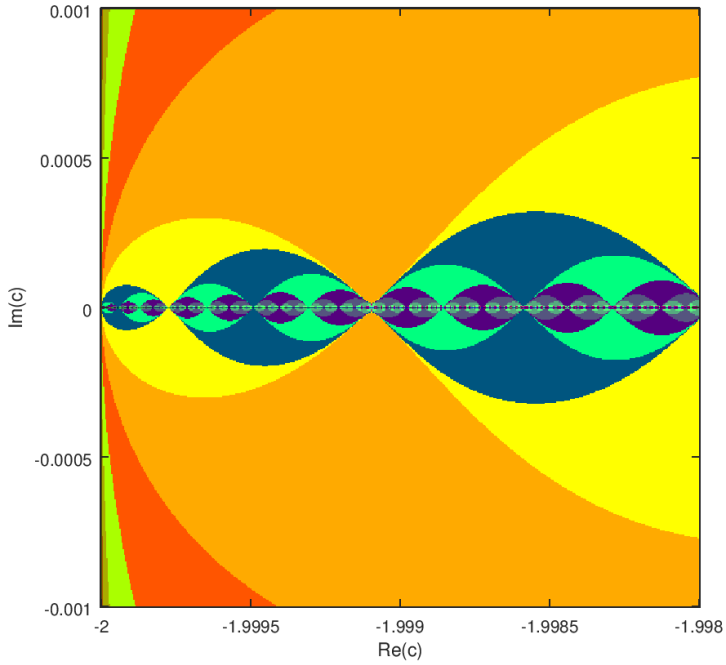


Figure 5.11: *Left tip of the Mandelbrot set. Colourmap:"colorcube".*

Properties of the Mandelbrot set M

Here are a few properties of the Mandelbrot set, given without proof.

1. M is *connected*: there is a path within M from each point of M to any other point of M. In other words, there are no disconnected "islands".
2. The *area* of M is *finite* (it fits inside a circle of radius 2).
3. The *length of the border* of M is *infinite*. Any part of the border of M also has infinite length. The border has "infinite details", no part of the border is smooth. Zooming into any part of the border always produces something interesting (when resolution and number of iterations are high enough).
4. M is *self-similar* under zooming.

5.4.2 The filled Julia set

Again, consider the iteration

$$z_{k+1} = z_k^2 + c \qquad k = 0, 1, 2, \ldots$$

This time, we *fix* c and vary the initial point z_0. The **filled Julia set** [1] J_c is the set of points z_0 for which the orbit $z_0, z_1, z_2, \ldots, z_k, \ldots$ remains bounded.

There are filled Julia sets for each value of c. The algorithm for computing filled Julia sets is almost the same as for the Mandelbrot set, with a few small adaptations.

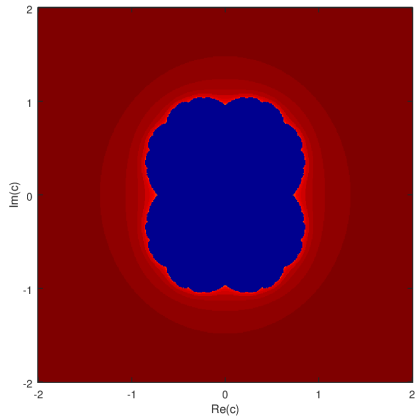


Figure 5.12: *Filled Julia set: $c = 0.2$. Colourmap: flipud(jet).*

[1] The "Julia set" is the boundary of the filled Julia set. Named after Gaston Julia, 1893–1978.

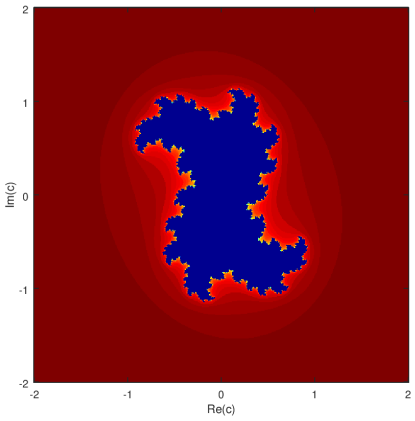


Figure 5.13: *Filled Julia set:* $c = 0.353 + 0.288i$*. Colourmap: flipud(jet).*

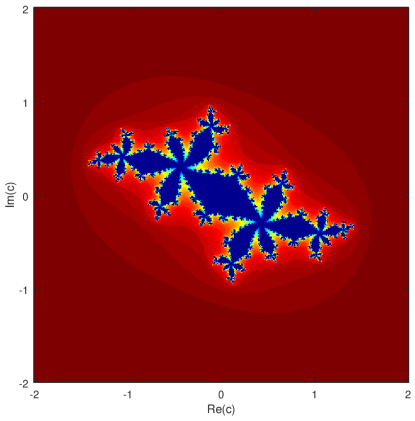


Figure 5.14: *Filled Julia set:* $c = -0.52 + 0.57i$*. Colourmap: flipud(jet).*

Computing the filled Julia set: example 1 Figure 5.12 gives an example of the filled Julia set, for $c = 0.2$. Region of the z_0-plane: $[-2, 2] \times [-2, 2]$. Grid size: 512^2, number of iterations: $n_{\max} = 50$.

Computing the filled Julia set: example 2 Figure 5.13 gives an example of the filled Julia set, for $c = 0.353 + 0.288i$. Region of the z_0-plane: $[-2, 2] \times [-2, 2]$. Grid size: 512^2, number of iterations: $n_{\max} = 50$.

Computing the filled Julia set: example 3 Figure 5.14 gives an example of the filled Julia set, for $c = -0.52 + 0.57i$. Region of the z_0-plane: $[-2, 2] \times [-2, 2]$. Grid size: 512^2, number of iterations: $n_{\max} = 50$.

The filled Julia set: a theoretical result

In 1919, *Gaston Julia* and *Pierre Fatou* proved the following theorem.

Theorem 5.1 *For each c-value, the filled Julia set J_c for the mapping $f(z) = z^2 + c$ is either a* connected set *or a* Cantor set.

A connected set is a set that consists of just one piece. A Cantor set consists of infinitely (in fact, uncountably) many pieces (actually, points). An example if shown in Fig. 5.15.

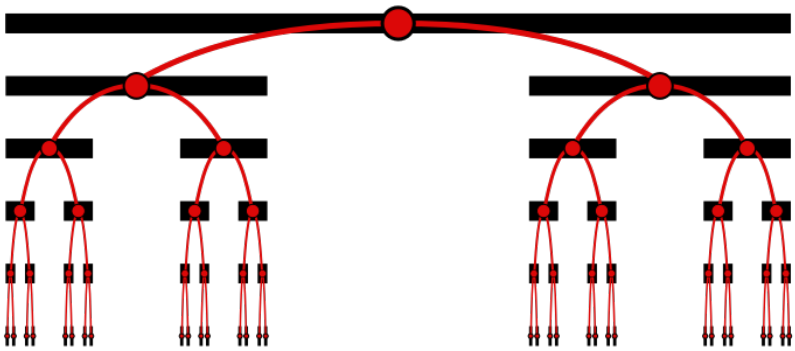


Figure 5.15: *Cantor ternary set.*

A related theorem is the following.

Theorem 5.2 *Let M be the Mandelbrot set and J_c be the filled Julia set for parameter c. If c lies in M then J_c is connected. If c does not lie in M then J_c is a Cantor set.*

To end we show in Figure 5.16 the Mandelbrot set and various Julia sets for various values of c.

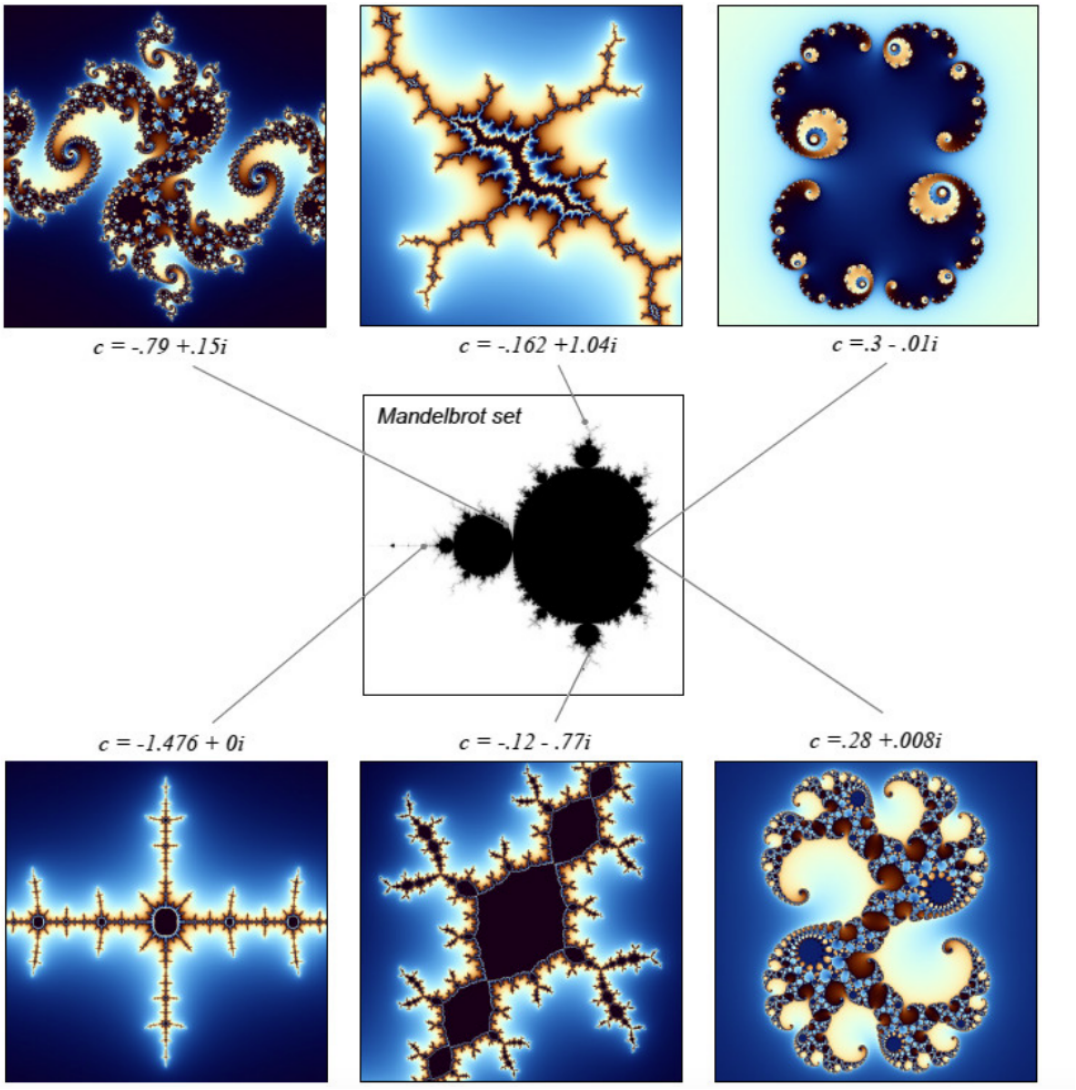


Figure 5.16: *The Mandelbrot set and various Julia sets. See* `http://www.karlsims.com/julia.html`.

Bibliography

Barnsley, M. F., 1988. Fractals Everywhere.

Mandelbrot, B., 1982. The Fractal Geometry of Nature. Times Books.

Chapter 6

Differential equations

Many dynamical phenomena in nature and technology can be modelled by differential equations. In this chapter we look at two different types of such equations, *linear* and *nonlinear ordinary differential equations*. An *ordinary* differential equation (ODE) is a differential equation containing one or more functions of *one* independent variable (often the time variable, denoted by t) and its derivatives.

In a later chapter we will consider *partial* differential equations, which involve more than one independent variable.

6.1 Linear ordinary differential equations

Let us start with a very simple example.

6.1.1 Exponential growth

The following linear first order differential equation with one dependent variable describes exponential growth:

$$\dot{y} = \frac{dy}{dt} = k\, y(t)$$
$$y(0) = y_0$$

Here k is the *growth rate*, $y(0) = y_0$ the *initial condition*. The solution of this equation is easy to find:

$$y(t) = y_0\, e^{k\, t} \tag{6.1}$$

Figure 6.1 shows a graph of the solution $y(t)$.

6.1.2 Circular motion

Next, we consider a case with *two* dependent variables, described by a system of two linear first order differential equations:

$$\dot{x} = y \qquad \dot{y} = -x \tag{6.2}$$

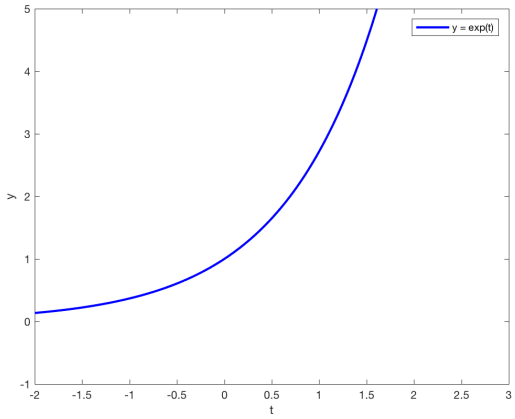


Figure 6.1: *Graph of the exponential function.*

Initial conditions $x(0) = 1$, $y(0) = 0$.

A general approach for solving linear ODEs is to try a solution of exponential form:

$$x(t) = C\,e^{\lambda t}, \quad y(t) = D\,e^{\lambda t}$$

Here the parameter λ can be a real or complex number. Substitute the assumed solution in (6.2):

$$C\,\lambda\,e^{\lambda t} = D\,e^{\lambda t},\ D\,\lambda\,e^{\lambda t} = -C\,e^{\lambda t}$$

This gives $C\,\lambda = D$, $D\,\lambda = -C$, so $\lambda^2 = -1$. This equation has two *roots*: $\lambda_1 = i$ or $\lambda_2 = -i$. When $\lambda_1 = i$ we find that $D_1 = i\,C_1$. When $\lambda_2 = -i$ then $D_2 = -i\,C_2$.

So the solution is

$$x(t) = C_1\,e^{it} + C_2\,e^{-it}$$
$$y(t) = i\,C_1\,e^{it} - i\,C_2\,e^{-it}$$

Apply the initial conditions:

$$x(0) = 1,\ \text{so } C_1 + C_2 = 1$$
$$y(0) = 0,\ \text{so } i\,C_1 - i\,C_2 = 0$$

Then we see that $C_1 = C_2$, $2C_1 = 1$, $C_1 = C_2 = \frac{1}{2}$. Then we find the final form of the solution:

$$x(t) = \frac{1}{2}\,(e^{it} + e^{-it}) = \cos t$$
$$y(t) = i\,\frac{1}{2}\,(e^{it} - e^{-it}) = -\sin t$$

These equations describe *circular motion*: the curve (orbit) followed by the point $(x(t), y(t))$ in the x-y plane when t increases is a circle.

In the two cases above we were able to compute the solution analytically. However, in many cases this is not possible and we have to turn to computing the solution numerically. A common type of numerical algorithms are the so-called *finite difference schemes*, which we consider next.

6.1.3 Finite difference schemes

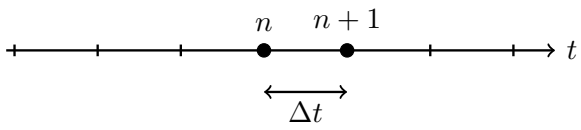


Figure 6.2: *Discretization of the time-axis.*

Finite difference schemes work as follows:

- Divide the time axis into cells of size Δt (the *step size*).
- Replace derivatives by *finite differences*.
- *Update (integration) step*: compute values at step $n+1$ from values at step n.

Various methods exist:

1. **Explicit** (forward) Euler method
2. **Implicit** (backward) Euler method
3. **Symplectic** (semi-implicit) Euler method

Let us now look at these methods in detail for the case of circular motion described by the equations (6.2).

6.1.4 Explicit (forward) Euler method

The steps of this method are as follows:

- Divide the time axis into cells of size Δt (the *step size*); see Fig. 6.2. The n^{th} time point is

$$t_n = t_0 + n\Delta t, \ldots n = 1, \ldots, N$$

- Each derivative is replaced by a finite difference:

$$\dot{x} = \frac{dx(t)}{dt} \approx \frac{x^{n+1} - x^n}{\Delta t} \qquad \dot{y} = \frac{dy(t)}{dt} \approx \frac{y^{n+1} - y^n}{\Delta t}$$

  where $x^{n+1} = x(t_{n+1})$, $x^n = x(t_n)$ and $y^{n+1} = y(t_{n+1})$, $y^n = y(t_n)$.

- Each finite difference at step n equals the value of the right-hand-side of the corresponding equation *at the current step* n:

$$x^{n+1} - x^n = \Delta t\, y^n \qquad y^{n+1} - y^n = -\Delta t\, x^n$$

  Reordering we get *explicit* expressions for x^{n+1}, y^{n+1}:

$$\boxed{x^{n+1} = x^n + \Delta t\, y^n \qquad y^{n+1} = y^n - \Delta t\, x^n}$$

The following Matlab code shows an implementation of the explicit Euler method for the equations (6.2) describing circular motion.

```
% initial values
x0=1; y0=0;
% step size
Delta_t = 2*pi/60;
% number of steps
num_steps = 60;
% Initialization
x = zeros(1,num_steps+1);
y = zeros(1,num_steps+1);
% insert initial values
x(1) = x0; y(1) = y0;
% Simulate
for i=1:num_steps
    % Compute derivatives
    dxdt = y(i);
    dydt = - x(i);
    % Update
    x(i+1) = x(i) + Delta_t*dxdt;
    y(i+1) = y(i) + Delta_t*dydt;
end
```

Figure 6.3 shows a plot of $y(t)$ against $x(t)$. Such a plot is called a *phase-space plot*. As we can observe, the plot is not a circle, but an outwardly moving spiral. We say that the Euler method is *unstable*. This can be attributed to insufficient (numerical) *damping*.

6.1.5 Implicit (backward) Euler method

This method attempts to provide a solution for the instability of the explicit Euler method. The steps of this method are as follows:

- Divide the time axis into cells of size Δt, and replace each derivative by a finite difference, as before.

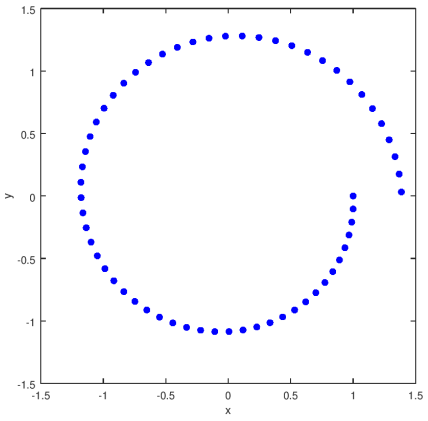


Figure 6.3: *Phase-space plot of the explicit Euler method for the equations* (6.2).

- Each finite difference at step n equals the value of the right-hand-side of the corresponding equation *at the next step* $n+1$, leading to *implicit* expressions for x^{n+1}, y^{n+1}:

$$x^{n+1} - x^n = \Delta t\, y^{n+1}$$
$$y^{n+1} - y^n = -\Delta t\, x^{n+1}$$

  Reordering we get a 2×2 system of equations to be solved for (x^{n+1}, y^{n+1}):

$$x^{n+1} - \Delta t\, y^{n+1} = x^n$$
$$y^{n+1} + \Delta t\, x^{n+1} = y^n$$

The following Matlab code shows an implementation of the implicit Euler method for the equations (6.2) describing circular motion.

```
% initial values
x0=1; y0=0;
% step size
Delta_t = 2*pi/60;
% number of steps
num_steps = 60;
% Initialization
x = zeros(1,num_steps+1);
y = zeros(1,num_steps+1);
% insert initial values
x(1) = x0; y(1) = y0;
% Simulate
```

```
for i=1:num_steps
    % Solve linear system
    M=[1, -Delta_t; Delta_t, 1];
    soln=M\[x(i);y(i)];  % matrix inversion
    % Update
    x(i+1)=soln(1);
    y(i+1)=soln(2);
end
```

Figure 6.4 shows the *phase-space plot*. As we can see, the method is *stable*, but the trajectory spirals inwards. There is *too much damping*.

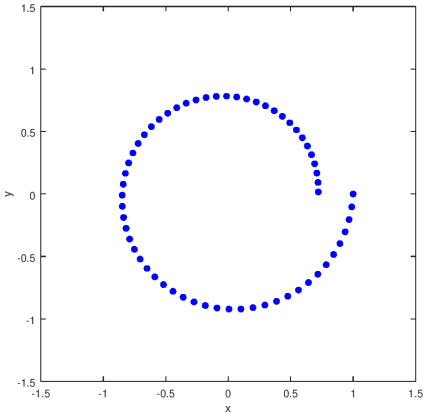


Figure 6.4: *Phase-space plot of the implicit Euler method for the equations* (6.2).

6.1.6 Symplectic (semi-implicit) Euler method

A further improvement can be obtained by the so-called symplectic (semi-implicit) Euler method. The steps of this method are as follows:

- Divide the time axis into cells of size Δt, and replace each derivative by a finite difference, as before.
- *Two half steps*: first use the current value y^n in the finite difference for x^n, then use the *updated value* x^{n+1} in the finite difference for y^n:

$$x^{n+1} - x^n = \Delta t\, y^n$$
$$y^{n+1} - y^n = -\Delta t\, x^{n+1}$$

  Reordering, we get:

$$x^{n+1} = x^n + \Delta t\, y^n \tag{6.5}$$
$$y^{n+1} = y^n - \Delta t\, x^{n+1} \tag{6.6}$$

So, given (x^n, y^n), first (6.5) is used to update x^n, producing the new estimate x^{n+1}. Then y^n and x^{n+1} are used in (6.6) to update y^n, producing the new estimate y^{n+1}.

The following Matlab code shows an implementation of the symplectic Euler method for the equations (6.2) describing circular motion.

```
% initial values
x0=1; y0=0;
% step size
Delta_t = 2*pi/60;
% number of steps
num_steps = 60;
% Initialization
x = zeros(1,num_steps+1);
y = zeros(1,num_steps+1);
% insert initial values
x(1) = x0; y(1) = y0;
% Simulate
for i=1:num_steps
    % Compute derivative
    dxdt = y(i);
    % Update
    x(i+1) = x(i) + Delta_t*dxdt;
    % Compute derivative
    dydt = - x(i+1);
    % Update
    y(i+1) = y(i) + Delta_t*dydt;
end
```

Figure 6.5 shows the *phase-space plot*. As we can see, the method is *stable*, although the trajectory is not a perfect circle. To further improve the accuracy, higher order numerical schemes can be used. However, these are beyond the scope of this course. For further information, see Quarteroni et al. (2006, 2014).

Constant of motion

One way to determine the quality of numerical algorithms is to see to what extent they preserve so-called constants of motion, that is, functions of the dependent variables that remain constant over time.

For the equations (6.2) describing circular motion, we can define the following quantity:

$$E = x^2 + y^2 \tag{6.7}$$

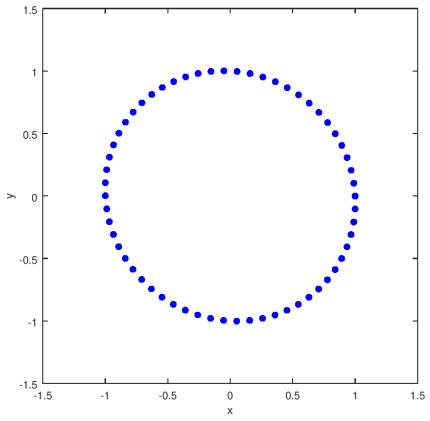


Figure 6.5: *Phase-space plot of the symplectic Euler method for the equations* (6.2).

The function $E(t) = x^2(t) + y^2(t)$ is independent of time. Here $(x(t), y(t))$ is the solution to the system of differential equations. To prove this, let us compute the derivative of $E(t)$ with respect to time:

$$\begin{aligned} \dot{E} &= 2\,x(t)\,\dot{x}(t) + 2\,y(t)\,\dot{y}(t) \\ &= 2\,x(t)\,y(t) + 2\,y(t)\,(-x(t)) = 0 \end{aligned}$$

We see that the time derivative is zero, so $E(t)$ is constant in time. We call E a *constant of motion* or *integral of motion* or *conserved quantity*. It is a function of the variables which is *constant along a trajectory in phase space*.

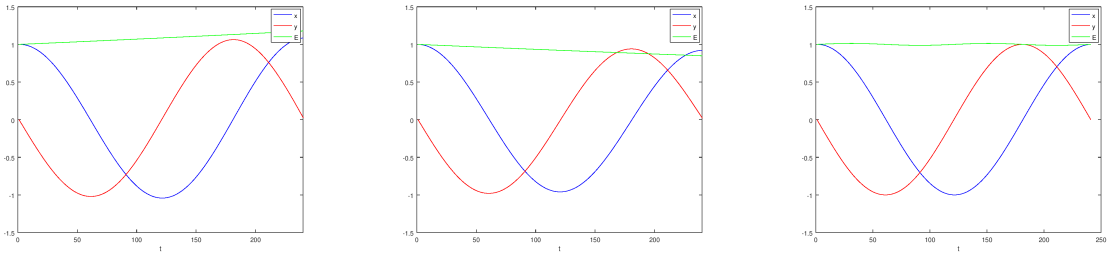
Figure 6.6 shows simultaneous plots of x (blue), y (red), and E (green) as function of time for the three Euler methods. Only the *symplectic method* preserves the constant of motion E to a large degree.

6.2 Nonlinear ordinary differential equations

Now we look at two examples of nonlinear ODEs, the logistic model (one dependent variable) and the Lotka-Volterra model (two dependent variables).

6.2.1 Logistic model

The logistic model is a modification of the exponential growth model which takes into account that in reality resources in the environment are limited. We say that the environment has a finite "carrying capacity".



explicit Euler implicit Euler symplectic Euler

Figure 6.6: *Plots of x (blue), y (red), and E (green) as function of time for the three Euler methods.*

The nonlinear first order differential equation that describes this is the *logistic equation*:

$$\dot{y} = k\, y(t)\, (1 - \frac{y(t)}{\mu})$$
$$y(0) = y_0$$

Here k is the *growth rate*, μ the *carrying capacity*, and $y(0) = y_0$ the *initial condition*. The solution of this model (called the logistic function) can be obtained analytically:

$$y(t) = \frac{\mu\, y_0\, e^{k\,t}}{y_0\, e^{k\,t} + \mu - y_0} \tag{6.8}$$

Steady states (equilibrium points) of this model are solutions which are constant in time. To find these, we set $\dot{y} = 0$. Two steady state solutions are found: $y(t) = 0$ or $y(t) = \mu$.

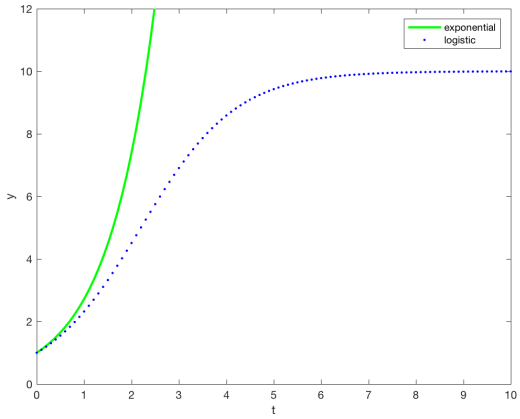


Figure 6.7: *Plot of the* exponential *function (solid line) and the* logistic *function (dotted line).*

Plots of the *exponential* function (green) and the *logistic* (or *sigmoid*) function (blue) are shown in Fig. 6.7. We can observe that for small values of y the system shows exponential growth. For larger values, saturation due to *food depletion* or *crowding* occurs.

Logistic model: simulation

We can simulate the logistic model by computing the finite difference solution by explicit Euler integration. We set $k = 1$, $\mu = 10$, $y_0 = 1$, $\Delta t = 0.1$, and use 100 timepoints.

Here is the Matlab code:

```
for i=1:num_steps
    % Compute derivative
    dydt = k*y(i)*(1-(y(i)/m));
    % Update
    y(i+1) = y(i) + Delta_t*dydt;
end
```

Figure 6.8 shows plots of the exact solution and the numerical solution. As we can see, the two curves show good agreement.

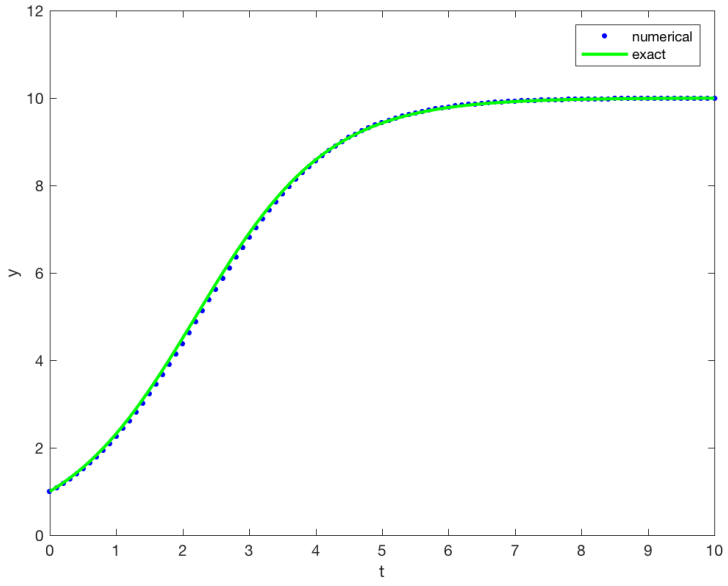


Figure 6.8: *Plots of the solution of the logistic model. Dotted line: numerical solution. Solid line: the exact solution.*

6.2.2 Lotka-Volterra model

The Lotka-Volterra model is an example of a predator-prey model. Consider two species, a *predator* (say foxes) and a *prey* (say hares), which are in *competition*. If there are *no predators* the prey will grow exponentially. If there is *no prey* the predators die out exponentially. If there is much prey the number of predators will grow. So the prey decreases, therefore the predators decrease. Then the prey can grow again and the *process repeats itself*.

The Lotka-Volterra model is defined by a pair of nonlinear first order differential equations. Let $y_1(t)$ be the number of prey and $y_2(t)$ be the number of predators at time t. Then the equations are:

$$\dot{y}_1 = k_1\, y_1(t)\, (1 - \frac{y_2(t)}{\mu_2})$$
$$\dot{y}_2 = -k_2\, y_2(t)\, (1 - \frac{y_1(t)}{\mu_1})$$

Here k_i are the net *growth rates*, μ_i the *carrying capacities*. Two *initial conditions* are needed: $y_1(0) = y_{1,0}$, $y_2(0) = y_{2,0}$.

Again we can look at the *steady states* (equilibrium points), defined by $\dot{y}_1 = \dot{y}_2 = 0$. Two solutions are obtained: $(y_1(t), y_2(t)) = (0, 0)$ or $(y_1(t), y_2(t)) = (\mu_1, \mu_2)$.

Lotka-Volterra model: simulation

Let us use the parameter values: $(k_1, k_2) = (1, 1)$; $(\mu_1, \mu_2) = (300, 200)$; $(y_{1,0}, y_{2,0}) = (400, 100)$. Duration = 3 periods (period = 6.5357), $\Delta t =$ period$/60$.

Finite difference solution by *symplectic Euler integration* is given by the following Matlab code:

```
% Matlab code snippet
% Simulate
for i=1:num_steps
```

```
    % Compute derivative
    dy1dt = k1*y1(i)*(1-(y2(i)/m2));
    % Update
    y1(i+1) = y1(i) + Delta_t*dy1dt;
    % Compute derivative
    dy2dt = -k2*y2(i)*(1-(y1(i+1)/m1));
    % Update
    y2(i+1) = y2(i) + Delta_t*dy2dt;
end
```

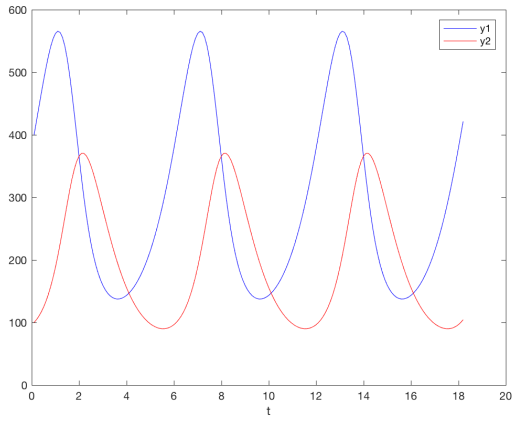


Figure 6.9: *Plot of the solution of the Lotka-Volterra model.*

Figure 6.9 shows plots of the numerical solutions for $y_1(t)$ and $y_2(t)$. As we can see, the solution is *periodic*. Detailed analysis (not carried out here) shows that the period depends on the values of k_1, k_2, μ_1, μ_2 and *on the initial conditions* $y_{1,0}, y_{2,0}$.

If we plot y_2 against y_1 we get a *phase-space plot*, which is a closed curve, confirming that the solution is periodic; see Fig. 6.10.

Lotka-Volterra model: Motion Invariant

Consider the following quantity:

$$G(y_1, y_2) = k_2\,(\frac{y_1}{\mu_1} - \ln\frac{y_1}{\mu_1}) + k_1\,(\frac{y_2}{\mu_2} - \ln\frac{y_2}{\mu_2})$$

It can be verified that $\frac{dG(y_1(t),y_2(t))}{dt} = 0$. In other words, G is time-independent: it is a *motion invariant* or *constant of motion*. Different values of G correspond to different closed contours in the phase-space; see Fig. 6.11. These contours are called *iso-lines*, since the value of G is identical for all points on the contour. For initial conditions $(y_{1,0}, y_{2,0})$ close to the steady state (μ_1, μ_2), the iso-lines are ellipses, but for initial conditions farther away from the steady state point the iso-lines start to take the shape of something like an hand axe.

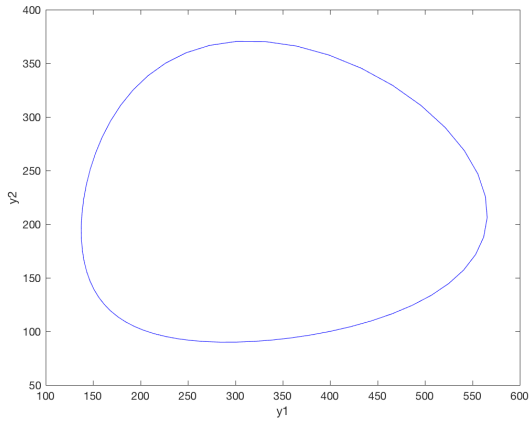


Figure 6.10: *Phase-space plot of the Lotka-Volterra model. Parameters:* $(k_1, k_2) = (1, 1)$*;* $(\mu_1, \mu_2) = (300, 200)$*;* $(y_{1,0}, y_{2,0}) = (400, 100)$.

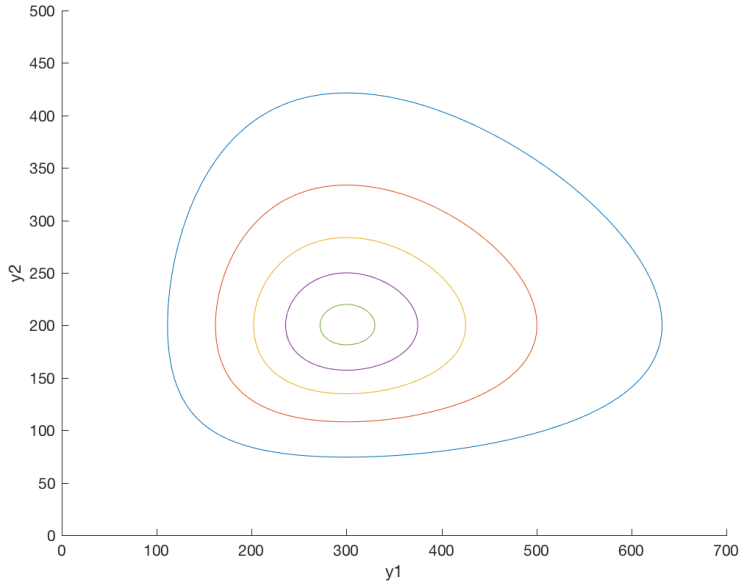


Figure 6.11: *Plots of* $G(y_1, y_2) = C$ (iso-lines) *for various values of the constant* $C = G(y_{1,0}, y_{2,0})$ *determined by the initial conditions* $y_{1,0}, y_{2,0}$*. For all plots* $(k_1, k_2) = (1, 1)$*;* $(\mu_1, \mu_2) = (300, 200)$.

Bibliography

Quarteroni, A., Sacco, R., Saleri, F., 2006. Numerical Mathematics (2nd ed.). No. 37 in Text in Applied Mathematics. Springer.

Quarteroni, A., Saleri, F., Gervasio, P., 2014. Numerical Mathematics (4th ed.). No. 2 in Texts in Computational Science and Engineering. Springer.

Chapter 7

N-body simulations

Understanding the motions of the planets and the stars has been important throughout human history. According to Newton's law of universal gravitation (as described in his *Philosophiae Naturalis Principia Mathematica* of 1687) states that a (point) particle attracts every other particle in the universe using a force that is directly proportional to the product of the masses of the particles and inversely proportional to the square of the distance between them. That is, the force F satisfies the equation

$$F = G\frac{m_1\, m_2}{r^2}, \tag{7.1}$$

where m_1, m_2 are the masses of the particles, r is the distance between the centers of the two masses, and G is the gravitational constant[1]. The force on each particle is *central*, that is, it is directed along the line that connects the two masses. Also, the force on m_1 has a direction that is opposite to that of the force on m_2. For spherical bodies with a uniform mass distribution the same formula holds, where the masses can be replaced by point particles at their centers; see Figure 7.1.

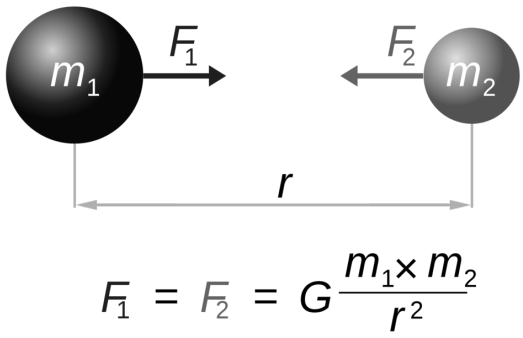


Figure 7.1: *Mutual attraction of two massive bodies according to Newton's law of universal gravitation.*

[1]The value of G is $6.67 \times 10^{-11}\mathrm{m}^3/\mathrm{kg\,s}^2$

Since Newton's discoveries it became possible to try and analyze motions of planets and other heavenly bodies in a principled, accurate, and recognizably modern fashion. Despite Einstein's general relativity providing an even more accurate model, most modern simulations of heavenly bodies essentially still use the Newtonian law at their core. However, such simulations can of course handle enormous numbers of bodies/particles[2]. Modern simulations often also consider more than just gravity (for example cooling and heating).

7.1 Navigation

Historically, one of the most important N-body problems to have been studied was the *three-body problem* involving the Sun, the Earth, and the Moon. Navigation at sea was a major reason to be interested in predicting the relative positions of the Sun, Earth, and Moon. In particular, although latitude was relatively easy to determine, longitude was much more difficult, and this was where the thee-body problem turned up. The main idea is that if you can predict in advance where the moon should be in the sky relative to other heavenly bodies (like the Sun or certain bright stars) at particular times, then by measuring the position of the moon relative to some other heavenly body, you can figure out the time in Greenwich (for example). If you then also know the local time (by observing the Sun, for example), then longitude follows easily (15 degrees per hour difference).

Unfortunately, unlike a two-body problem, a three-body problem is quite difficult to treat analytically (Diacu, 1996; Šiuvakov and Dmitrašinović, 2013). As a result, one has little choice but to use numerical methods and/or other kinds of approximations to approach the problem. Clearly, this was no easy task before the advent of computers, and this largely explains the wide variety of famous mathematicians who concerned themselves with the three-body problem in one way or another (Brown, 1896; Diacu, 1996; Wilson, 2010): Newton, Euler, Laplace, Lagrange, Poisson, Poincaré, among others.

7.2 Planetary system formation

The general idea for how the solar system came to be is that small particles coalesced into bigger ones, and then into even bigger ones, and so on until they ended up as the planets and moons we currently have. But is this idea accurate? One way to test this is to use an N-body simulation of the process to see what would happen. The main trick is that collisions can and will happen, and that these need to be modelled, but this is possible.

One question that can be investigated this way is the phenomenon known as the Late Heavy Bombardment of the terrestial planets. It is hypothesized, based on analyses of lunar rocks, that there was a spike in the number of impacts about 700 million years after the planets were formed. However, the basic picture just discussed does not really explain how this could happen. Using simulations of the gravitational interactions between the Sun, the giant planets (Jupiter, Saturn,

[2]The so-called "Millenium Simulation" was a simulation of over 10 billion particles, which ran for more than a month (in 2005) on a supercomputer (Springel et al., 2005).

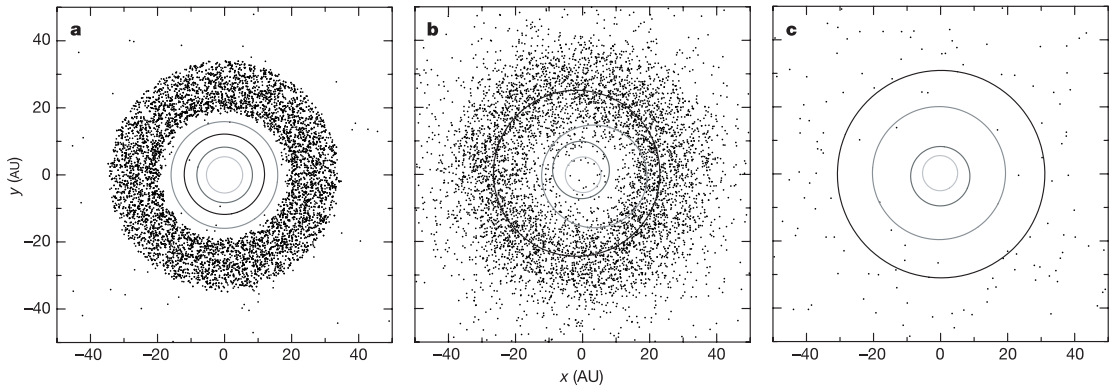


Figure 7.2: *Snapshots from a simulated planet migration (Gomes et al., 2005, Fig. 2). a) Just before the migration. b) The interaction of the planets is scattering debris all through the solar system (causing the Late Heavy Bombardment). c) Afterwards, the planets are in their final orbits.*

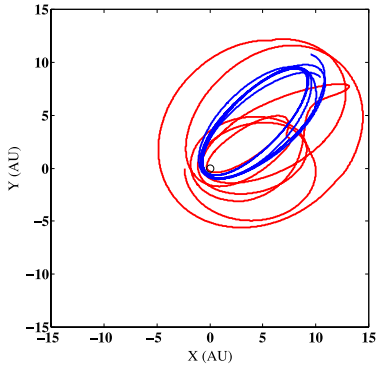


Figure 7.3: *A simulation of the orbits in a proposed detection of a planetary system with two planets (Horner et al., 2013, Fig. 3). As the orbits clearly are not very stable, it is extremely unlikely that the planets exist (as proposed).*

Uranus, and Neptune), and disks of debris, a plausible scenario can be found where an instability led to the migration of (in particular) Uranus and Neptune, in turn changing the orbit of part of the debris to cross into the interior of the solar system (Gomes et al., 2005), as illustrated in Fig. 7.2.

Another interesting use of N-body simulations is to get a better handle on the *exoplanetary systems* identified using the Kepler instrument [3]. Given that even stars appear to be not much more than pinpricks even on images captured by telescopes, exoplanet hunting is clearly a challenging business. In fact, exoplanets are far too small and dim to image directly, so we can only detect by a very faint dimming of their host star when (and *if*) they pass in front of it. Still, the data can be used to inform and validate N-body simulations of general planetary system formation,

[3] `https://www.nasa.gov/mission_pages/kepler`

giving us a better idea of the possible characteristics of the planets we detect. In some cases, it is even possible to use simulations for (in)validating detections, see Fig. 7.3.

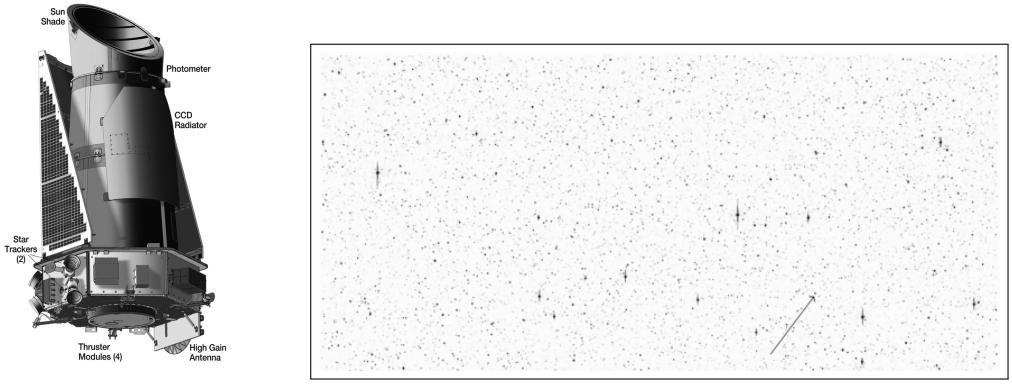


Figure 7.4: Left: *The Kepler instrument used for exoplanet hunting.* Right: *Image data from Kepler (a single frame comprises 42 of these images). The star pointed to by the arrow (in the lower-right quadrant) has an exoplanet called TrES-2 (a so-called "hot Jupiter"). Credit: NASA/Kepler/Ball Aerospace*

7.3 Dark matter and the cosmic web

More exotic, and more challenging (computationally) at the same time, are attempts to simulate the evolution of the universe using N-body simulations of (predominantly) "dark matter". The idea is to start with a large number of particles that are distributed not quite evenly over space in a virtual universe, and then let this evolve under the influence of (Newtonian) gravity. Clearly, these particles are not individual atoms or other microscopic particles, each particle is simply assumed to be a kind of "blob" of matter with a particular mass. One can then examine, for example, the influence of subtle changes to the models on voids (Bos et al., 2012), or the evolution of galaxies (Schaye et al., 2015; Barber et al., 2016).

The reason such simulations are computationally demanding is that, in principle, every particle interacts (gravitationally) with every other particle. So for N particles there are $N(N-1)/2$ interactions. For example, for 100 particles, $(100 \cdot 99)/2 = 4950$ interactions need to be taken into account. While for the 300 billion $= 3 \times 10^{11}$ particles used in the Millenium-XXL simulation (Angulo et al., 2012)[4], we need to consider almost 4.5×10^{22} interactions. Even on the world's largest supercomputer[5], assuming we only need a single operation for each interaction (you typically need more), and ignoring any communication overhead (which you will have), a single computation of all forces would take almost 100 hours[6]. For a practical simulation, up to $11\,000$ timesteps can be needed (Springel et al., 2005), leading to simulation times of up to 125

[4]An interactive visualisation of this data is available at `http://galformod.mpa-garching.mpg.de/mxxlbrowser/`.

[5]As of Nov. 2016, according to top500.org.

[6]This is based on the peak performance of 125,435.9 TFlop/s of the machine listed at top500.org.

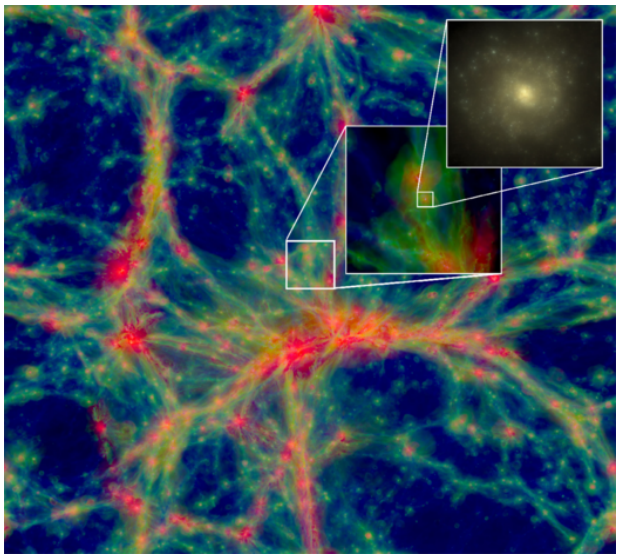


Figure 7.5: *A visualization of part of the EAGLE simulation designed to study the evolution of galaxies. Intensity shows gas density, while colour encodes temperature (blue:* $< 10^{4.5}$K*, green:* $10^{4.5}$–$10^{5.5}$K*, red:* $> 10^{5.5}$K*). The insets show progressively zoomed-in views of a galaxy (the top-right inset uses a different type of visualization that roughly mimics what you would see through a telescope).*

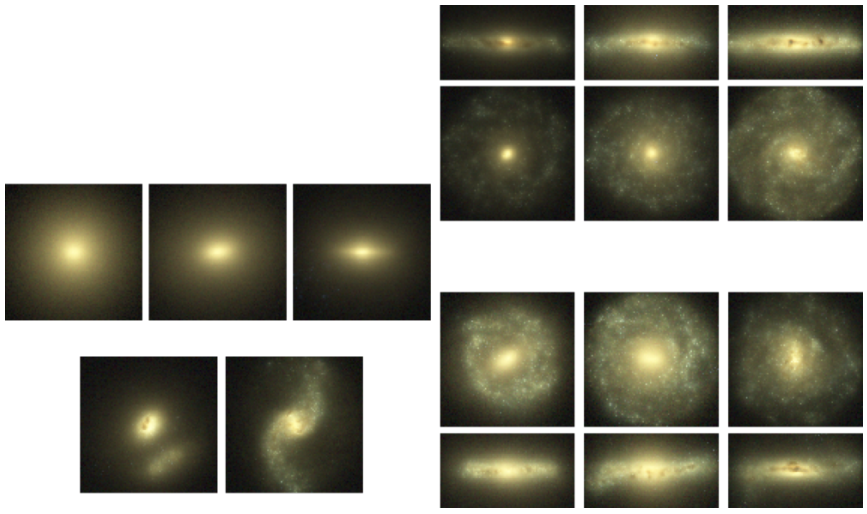


Figure 7.6: *Examples of galaxies present in the EAGLE simulation*

years. Clearly this is impractical, so alternative simulation techniques have been developed.

7.4 Two-body problem

Let us consider the simple case of the two-body problem ($N = 2$). In addition, we assume that the two bodies or particles move in circular orbits under the influence of each other's gravitational attraction, each with the same angular velocity ω. The bodies have masses m and M, respectively, where we assume that $M \geq m$. Now we want to study the equations for the positions, velocities, and accelerations of the two masses.

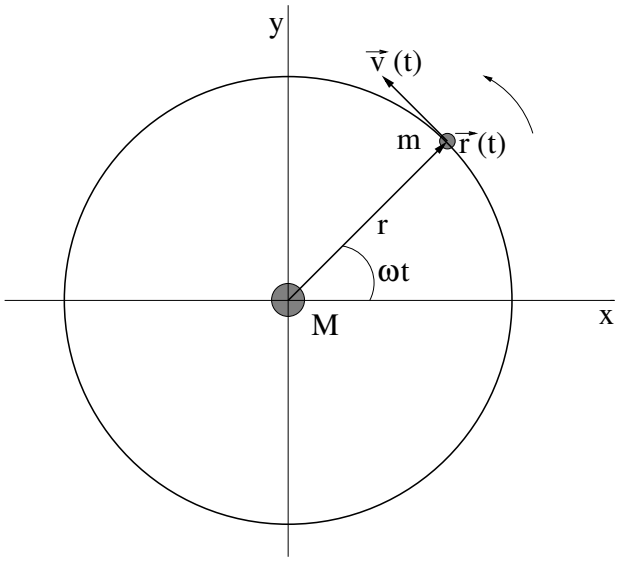


Figure 7.7: *Body with mass m moving in a circular orbit under the influence of the gravitational attraction of a mass M fixed at the origin.*

7.4.1 One particle moving in a circular orbit

We will start with the case that one of the masses is fixed at the origin of the coordinate center. This model is a good approximation for the case when the heavy mass M is much larger than the light mass m. So assume a particle of mass m moves in a circular orbit of radius r around the origin with angular velocity ω; see Fig. 7.7. Writing the x-coordinate of the particle as $x(t)$ and the y-coordinate as $y(t)$, where t denotes time, we have that

$$x(t) = r\,\cos\omega\,t \qquad y(t) = r\,\sin\omega\,t$$

We will write the position of the particle as a *column vector* $\vec{r}(t)$:

$$\vec{r}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} r\,\cos\omega\,t \\ r\,\sin\omega\,t \end{pmatrix} \tag{7.2}$$

We can also use the *row vector* form, where the vector is written as $(x(t), y(t))$. We say that the column vector is the *transpose* of the row vector, which is denoted by the superscript T: $\vec{r}(t) = (x(t), y(t))^T$.

Since the particle moves in a circle of radius r the length of the vector $\vec{r}(t)$ is constant and equal to r. We can check this as follows. For any vector $\vec{x} = (x_1, x_2)$ its *magnitude* or *length* is equal to $\|\vec{x}\| = \sqrt{x_1^2 + x_2^2}$. This is also called the *norm* of the vector $\vec{x}$. Applying this formula to the vector $\vec{r}(t)$ we find

$$\|\vec{r}(t)\| = \sqrt{(r\,\cos\omega\,t)^2 + (r\,\sin\omega\,t)^2} = r\,\sqrt{\cos^2\omega\,t + \sin^2\omega\,t} = r \tag{7.3}$$

Next we compute the *velocity vector* $\vec{v}(t) = (v_x(t), v_y(t))^T$ of the particle, by taking the time derivatives of the position coordinates:

$$\vec{v}(t) = \begin{pmatrix} v_x(t) \\ v_y(t) \end{pmatrix} = \begin{pmatrix} \frac{dx(t)}{dt} \\ \frac{dy(t)}{dt} \end{pmatrix} = \begin{pmatrix} -\omega\,r\,\sin\omega\,t \\ \omega\,r\,\cos\omega\,t \end{pmatrix} \tag{7.4}$$

The *speed* of the particle is defined as the magnitude of the velocity vector:

$$v = \|\vec{v}(t)\| = \sqrt{v_x^2(t) + v_y^2(t)} = \sqrt{(-\omega\,r\,\sin\omega\,t)^2 + (\omega\,r\,\cos\omega\,t)^2} = \omega\,r \tag{7.5}$$

So we see that the speed is *constant*: $v = \omega\,r$.

Also, the position vector $\vec{r}(t)$ and the velocity vector $\vec{v}(t)$ are *perpendicular* at any time. This can be shown by computing the inner product (see the Appendix) of these two vectors:

$$\begin{aligned} \vec{r}(t)\cdot\vec{v}(t) &= x(t)\cdot v_x(t) + y(t)\cdot v_y(t) = \\ &= (r\,\cos\omega\,t)\cdot(-\omega\,r\,\sin\omega\,t) + (r\,\sin\omega\,t)\cdot(\omega\,r\,\cos\omega\,t) = 0 \end{aligned}$$

So the inner product is zero, which means the vectors are perpendicular.

Next we compute the *acceleration vector* $\vec{a}(t) = (a_x(t), a_y(t))$ by taking the time derivatives of the components of the velocity vector:

$$\vec{a}(t) = \frac{d\vec{v}(t)}{dt} = \begin{pmatrix} \frac{dv_x(t)}{dt} \\ \frac{dv_y(t)}{dt} \end{pmatrix} = \begin{pmatrix} -\omega^2\,r\,\cos\omega\,t \\ -\omega^2\,r\,\sin\omega\,t \end{pmatrix} = -\omega^2\,\begin{pmatrix} r\,\cos\omega\,t \\ r\,\sin\omega\,t \end{pmatrix} = -\omega^2\,\vec{r}(t) \tag{7.6}$$

Note that we can also write the acceleration as the second order time derivative of the position:

$$\vec{a}(t) = \frac{d\vec{v}(t)}{dt} = \frac{d^2\vec{r}(t)}{dt^2} = \begin{pmatrix} \frac{d^2x(t)}{dt^2} \\ \frac{d^2y(t)}{dt^2} \end{pmatrix} \tag{7.7}$$

Formula (7.6) shows that the acceleration has the constant magnitude $a = \omega^2\,r$ and is directed in the opposite direction of the position vector $\vec{r}(t)$. The *force* $\vec{F}(t)$ which is needed to keep the particle in the circular orbit is given by Newton's second law of motion:

$$\vec{F}(t) = m\,\vec{a}(t) = -m\,\omega^2\,\vec{r}(t)$$

This means that the force vector is pointing towards the center of the circular motion (the origin), which is why it is called the *centripetal ("center seeking") force*. The magnitude of the centripetal force, which we denote by F_c, is equal to

$$F_c = \left\| \vec{F}(t) \right\| = m\,\omega^2\ \|\vec{r}(t)\| = m\,\omega^2\,r = \frac{m\,v^2}{r}.$$

This force has to be provided by the gravitational attraction force of the mass M at the origin, which according to Newton's gravitational law (Eq. (7.1)) has magnitude

$$F_g = \frac{G\,m\,M}{r^2} \tag{7.8}$$

and is directed from the position of the mass m to the origin, that is, along the unit vector $-\frac{\vec{r}(t)}{\|\vec{r}(t)\|}$. So in vector form the gravitational force is equal to (remember that $\|\vec{r}(t)\| = r$)

$$\vec{F}_g(t) = -\frac{G\,m\,M}{r^2}\,\frac{\vec{r}(t)}{\|\vec{r}(t)\|} = -\frac{G\,m\,M}{r^3}\,\vec{r}(t)$$

By setting $F_c = F_g$ we find

$$\frac{m\,v^2}{r} = \frac{G\,m\,M}{r^2}$$

So

$$v = \sqrt{\frac{G\,M}{r}} \qquad \omega = v/r = \sqrt{\frac{G\,M}{r^3}} \tag{7.9}$$

In other words, circular motion at radius r can only happen if the speed v, c.q. the angular speed ω, satisfy formula (7.9). If this special relation is not satisfied other types of orbits are possible. The study of the general case of two-body motion is called the *Kepler problem*.[7] The analysis of this problem shows that the following types of orbits are possible: circle, ellipse, parabola, and hyperbola.

7.4.2 Two particles moving in circular orbits

Let us now relax the assumption that one of the masses is fixed and consider the case that both bodies are moving around one another in circular orbits; see Fig. 7.8. The heavy mass M moves on a circle of radius R and the lighter mass on a circle of radius r. Both circles have the same center, which is the center of mass (or barycenter) C, which lies on the line connecting the two masses.

Let $\vec{r}_1(t)$ be the position vector of particle 1 (the body with mass M) and $\vec{r}_2(t)$ be the position vector of particle 2 (the body with mass m). The center of mass has the position $\vec{R}_c(t)$ given by

$$\vec{R}_c(t) = \frac{M\,\vec{r}_1(t) + m\,\vec{r}_2(t)}{M+m}$$

[7] Johannes Kepler (1571–1630) discovered experimentally that the orbits of the planets are ellipses with the Sun in one focal point of the ellipse.

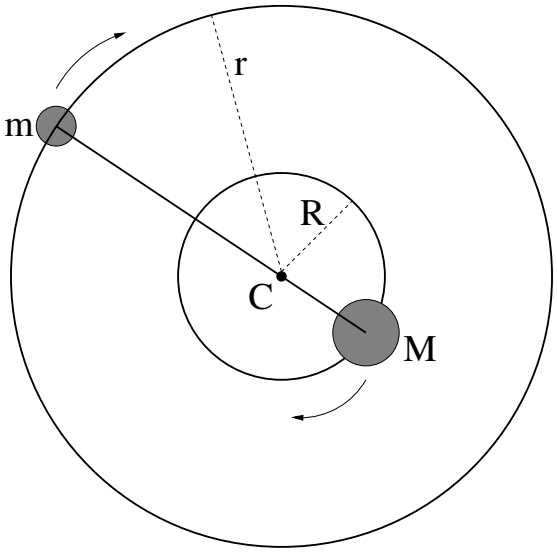


Figure 7.8: *Two bodies with masses m and M moving in circular orbits under the influence of their mutual gravitational attraction.*

By applying Newton's law of gravitation to the two masses it can be shown that the barycenter C moves with constant velocity. So we can put C at the origin of the coordinate system, that is, $\vec{R}_c(t) = (0, 0)$. This means that $M\,\vec{r}_1(t) = -m\,\vec{r}_2(t)$. Remembering that the magnitude of $\vec{r}_1(t)$ equals R and the magnitude of $\vec{r}_2(t)$ equals r, we thus find that the masses and radii are related by the equation

$$M\,R = m\,r \tag{7.10}$$

So indeed the radius of the orbit of the heavy mass is smaller than that of the lighter mass.

As we have seen in Section 7.4.1, the magnitude of the centripetal force on particle 1 is equal to $M\,\omega^2\,R$ and the magnitude of the centripetal force on particle 2 is equal to $m\,\omega^2\,r$. In view of Eq. (7.10) these two magnitudes are equal. From Newton's law of gravitation we know that the magnitude of the gravitational force exerted by mass m on mass M and the magnitude of the gravitational force exerted by mass M on mass m both have the value $\frac{G\,m\,M}{(R+r)^2}$, because the distance between the two masses is $R + r$. So we have found the formula

$$\frac{G\,m\,M}{(R+r)^2} = m\,\omega^2\,r$$

This means that the angular speed ω is equal to

$$\omega = \sqrt{\frac{G\,M}{r(R+r)^2}}$$

By putting $R = 0$ (heavy mass fixed at the origin) we recover formula (7.9).

7.5 Simulation techniques

The simplest technique for performing N-body simulations is *direct simulation*. The idea is simple: for each particle i the mass m_i is considered given and constant, and we have 6 variables describing the position and velocity of each particle in 3D: $x_i, y_i, z_i, v_{x,i}, v_{y,i}, v_{z,i}$; Newton's second law of motion ($\vec{F} = m\,\vec{a}$) is then used to determine the acceleration of each particle based on Eq. (7.1). Write $\vec{r}_i = (x_i, y_i, z_i)^T$ and $\vec{v}_i = (v_{x,i}, v_{y,i}, v_{z,i})^T$. Equation (7.1) gives the magnitude of the force acting between particles i and j. Now, keeping in mind that the direction of the force is towards the other body, dividing by the mass m_i to get acceleration rather than force, and summing over all particles other than i, the acceleration of particle i is given by

$$\vec{a}_i = \sum_{j \neq i} G \frac{m_j}{\|\vec{r}_j - \vec{r}_i\|^2} \frac{\vec{r}_j - \vec{r}_i}{\|\vec{r}_j - \vec{r}_i\|} = \sum_{j \neq i} G \frac{m_j\,(\vec{r}_j - \vec{r}_i)}{\|\vec{r}_j - \vec{r}_i\|^3}. \tag{7.11}$$

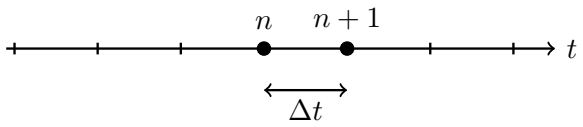


Figure 7.9: *Discretization of the time domain. The grid has cells of size Δt along the time axis. Two discrete time instants indexed by n and $n+1$ are shown.*

Considering the evolution of positions $\vec{r}_i(t)$ and velocities $\vec{v}_i(t)$ over time t, we get

$$\frac{d\vec{r}_i(t)}{dt} = \vec{v}_i(t) \tag{7.12}$$

$$\frac{d\vec{v}_i(t)}{dt} = \vec{a}_i(t) \tag{7.13}$$

The solution trajectory, $(\vec{r}_i(t), \vec{v}_i(t))$, $t \geqslant 0$, is determined by the initial positions $\vec{r}_i(0)$ and the initial velocities $\vec{v}_i(0)$ of all the particles $i = 1, \ldots, N$.

To compute the solution numerically, these differential equations have to be *discretized*. This can be done using *finite difference schemes*, as we discussed before in Chapter 6. We do this by dividing the time axis into cells of size Δt; see Fig. 7.9. The n^{th} time point is

$$t_n = t_0 + n\Delta t, \ldots n = 1, \ldots, N.$$

Each derivative is replaced by a finite difference. For example,

$$\frac{d\vec{r}_i(t)}{dt} \approx \frac{\vec{r}_i^{\,n+1} - \vec{r}_i^{\,n}}{\Delta t}, \tag{7.14}$$

where $\vec{r}_i^{\,n+1} = \vec{r}_i(t_{n+1})$ and $\vec{r}_i^{\,n} = \vec{r}_i(t_n)$. The symbol $\approx$ means "approximately".

After discretization the equations (7.12)-(7.13) become:

$$\vec{r}_i^{\,n+1} = \vec{r}_i^{\,n} + \Delta t\ \vec{v}_i^{\,n} \tag{7.15}$$

$$\vec{v}_i^{\,n+1} = \vec{v}_i^{\,n} + \Delta t\ \vec{a}_i^{\,n} \tag{7.16}$$

When implementing these equations in Matlab it is convenient to put the vectors $\vec{x}_i$ and $\vec{v}_i$ as column vectors in a matrix (see the Appendix for more information on vectors and matrices). For example, for $N = 3$ we can write the equations in matrix form as follows. Equation (7.15) becomes:

$$\begin{pmatrix} x_1^{n+1} & x_2^{n+1} & x_3^{n+1} \\ y_1^{n+1} & y_2^{n+1} & y_3^{n+1} \\ z_1^{n+1} & z_2^{n+1} & z_3^{n+1} \end{pmatrix} = \begin{pmatrix} x_1^{n} & x_2^{n} & x_3^{n} \\ y_1^{n} & y_2^{n} & y_3^{n} \\ z_1^{n} & z_2^{n} & z_3^{n} \end{pmatrix} + \Delta t \begin{pmatrix} v_{x,1}^{n} & v_{x,2}^{n} & v_{x,3}^{n} \\ v_{y,1}^{n} & v_{y,2}^{n} & v_{y,3}^{n} \\ v_{z,1}^{n} & v_{z,2}^{n} & v_{z,3}^{n} \end{pmatrix} \tag{7.17}$$

In the same way, Eq. (7.16) becomes:

$$\begin{pmatrix} v_{x,1}^{n+1} & v_{x,2}^{n+1} & v_{x,3}^{n+1} \\ v_{y,1}^{n+1} & v_{y,2}^{n+1} & v_{y,3}^{n+1} \\ v_{z,1}^{n+1} & v_{z,2}^{n+1} & v_{z,3}^{n+1} \end{pmatrix} = \begin{pmatrix} v_{x,1}^{n} & v_{x,2}^{n} & v_{x,3}^{n} \\ v_{y,1}^{n} & v_{y,2}^{n} & v_{y,3}^{n} \\ v_{z,1}^{n} & v_{z,2}^{n} & v_{z,3}^{n} \end{pmatrix} + \Delta t \begin{pmatrix} a_{x,1}^{n} & a_{x,2}^{n} & a_{x,3}^{n} \\ a_{y,1}^{n} & a_{y,2}^{n} & a_{y,3}^{n} \\ a_{z,1}^{n} & a_{z,2}^{n} & a_{z,3}^{n} \end{pmatrix} \tag{7.18}$$

where, according to (7.11), we can write the following matrix expression for the accelerations at time n:

$$\begin{pmatrix} a_{x,1}^{n} & a_{x,2}^{n} & a_{x,3}^{n} \\ a_{y,1}^{n} & a_{y,2}^{n} & a_{y,3}^{n} \\ a_{z,1}^{n} & a_{z,2}^{n} & a_{z,3}^{n} \end{pmatrix} = G \begin{pmatrix} \frac{m_2\,(x_2^{\,n} - x_1^{\,n})}{\|\vec{r}_2^{\,n} - \vec{r}_1^{\,n}\|^3} + \frac{m_3\,(x_3^{\,n} - x_1^{\,n})}{\|\vec{r}_3^{\,n} - \vec{r}_1^{\,n}\|^3} & \frac{m_1\,(x_1^{\,n} - x_2^{\,n})}{\|\vec{r}_1^{\,n} - \vec{r}_2^{\,n}\|^3} + \frac{m_3\,(x_3^{\,n} - x_2^{\,n})}{\|\vec{r}_3^{\,n} - \vec{r}_2^{\,n}\|^3} & \frac{m_1\,(x_1^{\,n} - x_3^{\,n})}{\|\vec{r}_1^{\,n} - \vec{r}_3^{\,n}\|^3} + \frac{m_2\,(x_2^{\,n} - x_3^{\,n})}{\|\vec{r}_2^{\,n} - \vec{r}_3^{\,n}\|^3} \\ \frac{m_2\,(y_2^{\,n} - y_1^{\,n})}{\|\vec{r}_2^{\,n} - \vec{r}_1^{\,n}\|^3} + \frac{m_3\,(y_3^{\,n} - y_1^{\,n})}{\|\vec{r}_3^{\,n} - \vec{r}_1^{\,n}\|^3} & \frac{m_1\,(y_1^{\,n} - y_2^{\,n})}{\|\vec{r}_1^{\,n} - \vec{r}_2^{\,n}\|^3} + \frac{m_3\,(y_3^{\,n} - y_2^{\,n})}{\|\vec{r}_3^{\,n} - \vec{r}_2^{\,n}\|^3} & \frac{m_1\,(y_1^{\,n} - y_3^{\,n})}{\|\vec{r}_1^{\,n} - \vec{r}_3^{\,n}\|^3} + \frac{m_2\,(y_2^{\,n} - y_3^{\,n})}{\|\vec{r}_2^{\,n} - \vec{r}_3^{\,n}\|^3} \\ \frac{m_2\,(z_2^{\,n} - z_1^{\,n})}{\|\vec{r}_2^{\,n} - \vec{r}_1^{\,n}\|^3} + \frac{m_3\,(z_3^{\,n} - z_1^{\,n})}{\|\vec{r}_3^{\,n} - \vec{r}_1^{\,n}\|^3} & \frac{m_1\,(z_1^{\,n} - z_2^{\,n})}{\|\vec{r}_1^{\,n} - \vec{r}_2^{\,n}\|^3} + \frac{m_3\,(z_3^{\,n} - z_2^{\,n})}{\|\vec{r}_3^{\,n} - \vec{r}_2^{\,n}\|^3} & \frac{m_1\,(z_1^{\,n} - z_3^{\,n})}{\|\vec{r}_1^{\,n} - \vec{r}_3^{\,n}\|^3} + \frac{m_2\,(z_2^{\,n} - z_3^{\,n})}{\|\vec{r}_2^{\,n} - \vec{r}_3^{\,n}\|^3} \end{pmatrix}$$

Note that the positions and velocities are updated simultaneously, and that we just use their values at time step n to compute their values at time step $n + 1$. This method of discretization is called the *Euler method*, which is an example of an *explicit* numerical scheme. In practice a slightly different method (the colourfully named *leapfrog method*) is usually used to improve the accuracy of the simulation; this method alternately updates the positions and velocities (for example, it starts by updating the positions, then it updates the velocities based on the new positions, then it updates the positions based on the last computed velocities, etc., etc.). This method is also *explicit*.

Direct simulation works very well, and is used in practice for small systems, but for large systems it becomes prohibitively expensive, due to having to compute the forces between all pairs of particles (which scales quadratically with the number of particles). There are two commonly used ways of speeding up the computations, and they can even be used together. The first is to store the points in a *tree* (or hierarchy) that recursively clusters together particles based on how close they are to each other; each cluster of particles is then treated as a single (heavier)

particle when computing its contribution to the force acting on some other, distant, particle (the further away the other particle is, the better the approximation works). The second method is a little more complicated to explain, but finds the force acting on a particle through computing a potential, which can be done efficiently with some further tricks (this is called the *particle mesh* method). Finally, to further speed up the simulation as a whole, different step sizes are usually used for different particles, depending on how regular their trajectories are.

Bibliography

Angulo, R. E., Springel, V., White, S. D. M., Jenkins, A., Baugh, C. M., Frenk, C. S., 2012. Scaling relations for galaxy clusters in the Millennium-XXL simulation. MNRAS 426 (3), 2046–2062.

Barber, C., Schaye, J., Bower, R. G., Crain, R. A., Schaller, M., Theuns, T., 2016. The origin of compact galaxies with anomalously high black hole masses. MNRAS 460 (1), 1147–1161.

Bos, E. G. P., van de Weygaert, R., Dolag, K., Pettorino, V., 2012. The darkness that shaped the void: dark energy and cosmic voids. MNRAS 426 (1), 440–461.

Brown, E. W., 1896. An Introductory Treatise On The Lunar Theory. Cambridge. The University Press.

Diacu, F., 1996. The solution of the n-body problem. The Mathematical Intelligencer 18 (3), 66–70.

Gomes, R., Levison, H. F., Tsiganis, K., Morbidelli, A., 2005. Origin of the cataclysmic Late Heavy Bombardment period of the terrestrial planets. Nature 435 (7041), 466–469.

Horner, J., Wittenmyer, R. A., Hinse, T. C., Marshall, J. P., Mustill, A. J., Tinney, C. G., 2013. A detailed dynamical investigation of the proposed QS Virginis planetary system. MNRAS 435 (3), 2033–2039.

Schaye, J., Crain, R. A., Bower, R. G., Furlong, M., Schaller, M., Theuns, T., Dalla Vecchia, C., Frenk, C. S., McCarthy, I. G., Helly, J. C., Jenkins, A., Rosas-Guevara, Y. M., White, S. D. M., Baes, M., Booth, C. M., Camps, P., Navarro, J. F., Qu, Y., Rahmati, A., Sawala, T., Thomas, P. A., Trayford, J., 2015. The EAGLE project: simulating the evolution and assembly of galaxies and their environments. MNRAS 446 (1), 521–554.

Springel, V., White, S. D. M., Jenkins, A., Frenk, C. S., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J. A., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., Pearce, F., 2005. Simulations of the formation, evolution and clustering of galaxies and quasars. Nature 435 (7042), 629–636.

Šiuvakov, M., Dmitrašinović, V., 2013. Three Classes of Newtonian Three-Body Planar Periodic Orbits. Phys. Rev. Lett. 110, 114301+.

Wilson, C., 2010. Lunar Theory from the 1740s to the 1870s A Sketch. In: The Hill-Brown Theory of the Moon's Motion. Sources and Studies in the History of Mathematics and Physical Sciences. Springer New York, pp. 9–30.

Chapter 8

Simulation of reaction-diffusion processes

In this chapter we first will look at the physical process of *diffusion*. We will see how to mathematically model the diffusion process and how to solve the corresponding mathematical equations on a grid. Then diffusion will be combined with reaction processes into so-called *reaction-diffusion processes*, which can describe pattern formation in *continuous* space (in contrast to the cellular automata models of Chapter 4 which are *discrete* in time and space.)[1]

8.1 Diffusion processes

Diffusion is the process whereby particles of liquids, gases, or solids mix due to spontaneous movement caused by thermal agitation. An example is the diffusion of a drop of ink in water. More ink particles will move from a region of higher to one of lower concentration than the other way around. The net result is that differences in the local ink concentration are gradually smoothed out.

The diffusion process can be mathematically modelled by the so-called *diffusion equation*. The same equation also describes how heat dissipates in a material, therefore another name is the *heat equation*. We will first formulate the equation when the spatial domain is one-dimensional. Then this is generalised to the 2D case.

8.2 The diffusion equation in 1D

The concentration of diffusing material at position x at time t is denoted by $f(x,t)$, where $0 \leq t \leq T$. T is the maximal time the diffusion process is running. We may however also study the process for an infinitely long time; then $T = \infty$. Let us also assume that the spatial domain is an interval $[a,b]$, so $a \leq x \leq b$. Again, the spatial interval can also be infinite ($a = -\infty$, $b = \infty$) or half infinite ($a = -\infty$ or $b = \infty$, but not both).

The diffusion equation relates the change in time of the concentration $f(x,t)$ at position x to the second order concentration difference at position x. (If the concentration is a linear function

[1] The material of Section 8.4 was developed by Jasper van de Gronde.

of x, the first order concentration difference is constant, while the second order concentration difference is zero.) Mathematically, the change in time of $f(x,t)$ at x is given by the derivative $\frac{\partial f(x,t)}{\partial t}$. This derivative is computed by differentiating the expression of $f(x,t)$ with respect to t while keeping x fixed. Similarly, the second order change in space of $f(x,t)$ at position x is given by the second derivative $\frac{\partial^2 f(x,t)}{\partial x^2}$. This derivative is computed by differentiating the expression of $f(x,t)$ two times with respect to x while keeping t fixed.

The diffusion equation now takes the form of an equation (a so-called *partial differential equation* or PDE) which says that the temporal derivative and second order spatial derivative of $f(x,t)$ are proportional, where the proportionality constant D is called the *diffusion coefficient*:

$$\frac{\partial f(x,t)}{\partial t} = D\,\frac{\partial^2 f(x,t)}{\partial x^2} \tag{8.1}$$

We need to define an initial condition in time when the process starts, that is at $t = 0$. This takes the form $f(x,0) = f_0(x)$, where $f_0(x)$ describes the spatial shape of the initial concentration profile. For the boundary condition in space, several choices are possible. For example, one may set $f(a,t) = c$, $f(b,t) = d$, where c and d are given constants.[2]

Note that the initial condition is a formula which holds at $t = 0$ for any x, while the boundary conditions are formulas that hold for specific spatial locations a and b at any time t.

8.2.1 Analytical solution of the diffusion equation in 1D

The equation (8.1) can be analytically solved. The methods to do so in the most general case are beyond the scope of this chapter. However, the solution can be easily given in the following special case. Let the spatial interval be infinite and assume that the concentration is zero at both ends $x = -\infty$ and $x = \infty$ for all times t. Also, assume that at $t = 0$ the concentration has a peak of height M at $x = 0$ and is zero everywhere else. The constant M can be interpreted as the total mass of diffusing material. Then it can be shown that the solution of equation (8.1) is given by the formula:

$$f(x,t) = \frac{M}{\sqrt{4\pi D t}}\, e^{-\frac{x^2}{4Dt}} \tag{8.2}$$

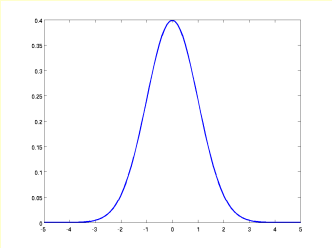
Considered as a function of x for fixed t, this is an example of a so-called normal or Gaussian distribution, which occurs in many fields of mathematics and statistics; see Figure 8.1. If we compare to the general formula for the normal distribution in equation 8.3, we see that the solution of the diffusion equation is a normal distribution at any time t, with mean $\mu = 0$ and standard deviation $\sigma = \sqrt{2Dt}$. This means that as time proceeds, the normal distribution becomes wider while the peak value becomes smaller. Plots of the solution (8.2) for a number of time points are given in Fig. 8.2. As is seen from the figure, shortly after the start of the diffusion process ($t = 0.01$), the peak has already broadened a bit. At $t = 0.05$ the broadening is very clear, while at $t = 1$ the curve starts to flatten out.

[2]This choice is known as Dirichlet boundary conditions.

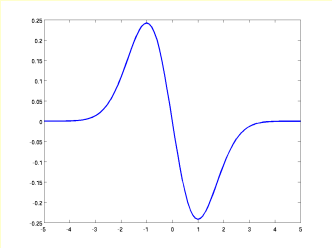
The one-dimensional normal distribution or Gaussian function $n_{\mu,\sigma}(x)$ is defined by

$$n_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{8.3}$$

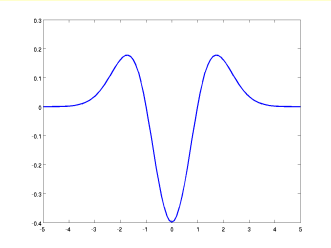
The parameter μ is the mean and σ the *width* or *standard deviation*. Plots of $n_{\mu,\sigma}(x)$ and its first and second derivatives are shown in the next figures:



$n_{\mu,\sigma}(x)$



$\frac{dn_{\mu,\sigma}(x)}{dx}$



$\frac{d^2 n_{\mu,\sigma}(x)}{dx^2}$

Figure 8.1: *The 1D normal distribution.*

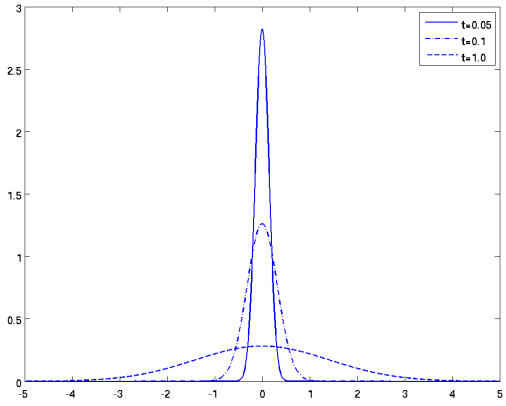


Figure 8.2: *Plots of the solution (8.2) to the diffusion equation for three time points. The parameter values are:* $D = 1$, $M = 1$.

Now suppose the initial concentration has not one, but a number of peaks at different locations. It can be shown that in this case the solution of the diffusion equation is a combination of normal distributions, each centered at the location of the corresponding peak. This is an illustration of the so-called *superposition principle*.

8.2.2 Numerical solution of the diffusion equation in 1D

To compute the solution numerically, the diffusion equation has to be discretized. As in Chapter 7 this can be done using *finite difference schemes*. Now there are two domains that have to be discretized, the spatial and the temporal domain. We will do that by considering time and space as two axes of a 2D space-time domain, and discretize this plane into cells just as we did in the case of cellular automata (there the 2D space had two spatial axes); see Figure 8.3.

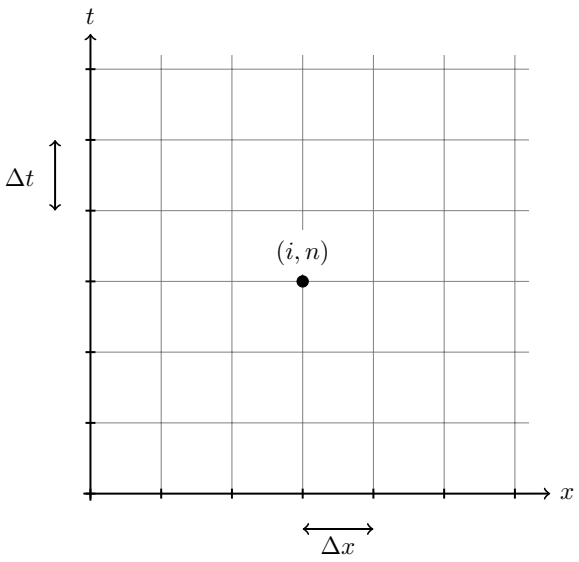


Figure 8.3: *Discretization of the space-time domain on which diffusion takes place. The grid has cells of size Δx along the (horizontal) space axis, and cells of size Δt along the (vertical) time axis. The concentration at time t_n at location x_i is f_i^n.*

In other words, we divide the time axis into cells of size Δt, so the n^{th} time point is

$$t_n = t_0 + n\Delta t, \ldots n = 1, \ldots, N.$$

Similarly, the spatial axis is divided into cells of size Δx, so the i^{th} space point is

$$x_i = x_0 + i\Delta x, i = 1, \ldots, I. \tag{8.4}$$

The concentration of material at time t_n at location x_i is $f(t_n, x_i)$. We abbreviate this by f_i^n.

Now look at the original diffusion equation (8.1). The left-hand side is replaced by a finite difference,

$$\frac{\partial f(x_i, t_n)}{\partial t} \approx \frac{f_i^{n+1} - f_i^n}{\Delta t} \tag{8.5}$$

The symbol $\approx$ means "approximately". Next consider the right-hand side of equation (8.1). This contains a second order derivative with respect to x. We replace this by a second-order difference:

$$\frac{\partial^2 f(x_i, t_n)}{\partial x^2} \approx \frac{f_{i+1}^n + f_{i-1}^n - 2f_i^n}{(\Delta x)^2} \tag{8.6}$$

Combining the results so far we find the discrete diffusion equation:

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} = D\left(\frac{f_{i+1}^n + f_{i-1}^n - 2f_i^n}{(\Delta x)^2}\right). \tag{8.7}$$

Of course, initial and boundary conditions have to be defined, just as in the continuous case.

We can rewrite (8.7) as follows:

$$f_i^{n+1} = f_i^n + \frac{D\Delta t}{(\Delta x)^2}\left(f_{i+1}^n + f_{i-1}^n - 2f_i^n\right). \tag{8.8}$$

This expresses the concentration at time step $(n+1)$ in terms of quantities at time step n. This is called an *explicit* numerical scheme.[3]

In Figure 8.4 we show bar plots of the solution to the discrete equation (8.7) for a number of time steps. We have used the same initial distribution as in the continuous case, that is, a peak of height M at $x = 0$. The following parameter values were used: $I = 21$, $M = 100$, $\Delta x = 1$, $\Delta t = 1$, $D = 0.25$, $t_0 = 0$.

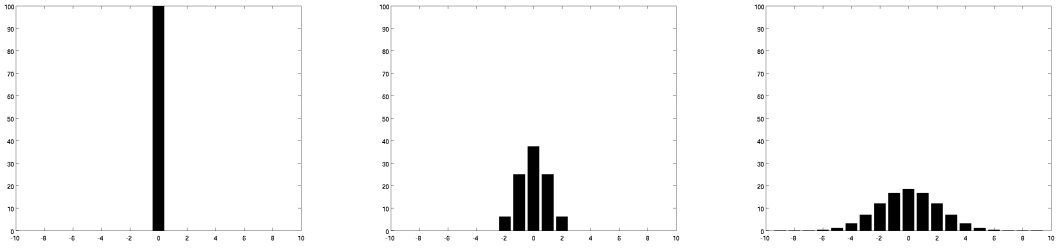


Figure 8.4: *Plots of the solution to the finite difference approximation (8.7) of the 1D diffusion equation for three time steps* $n = 1$, $n = 3$, *and* $n = 10$.

As we can see, the behaviour is similar to the continuous case, with the peak rapidly broadening. Also, the profiles of the concentration show a clear similarity to the normal distributions in Fig. 8.2.

We have not been very explicit about the exact meaning of the approximation symbol $\approx$. Various questions may be asked:

1. Is the behaviour of the finite-difference equation (8.7) similar to the continuous version (8.1)?

[3] Also *implicit* numerical schemes exist.

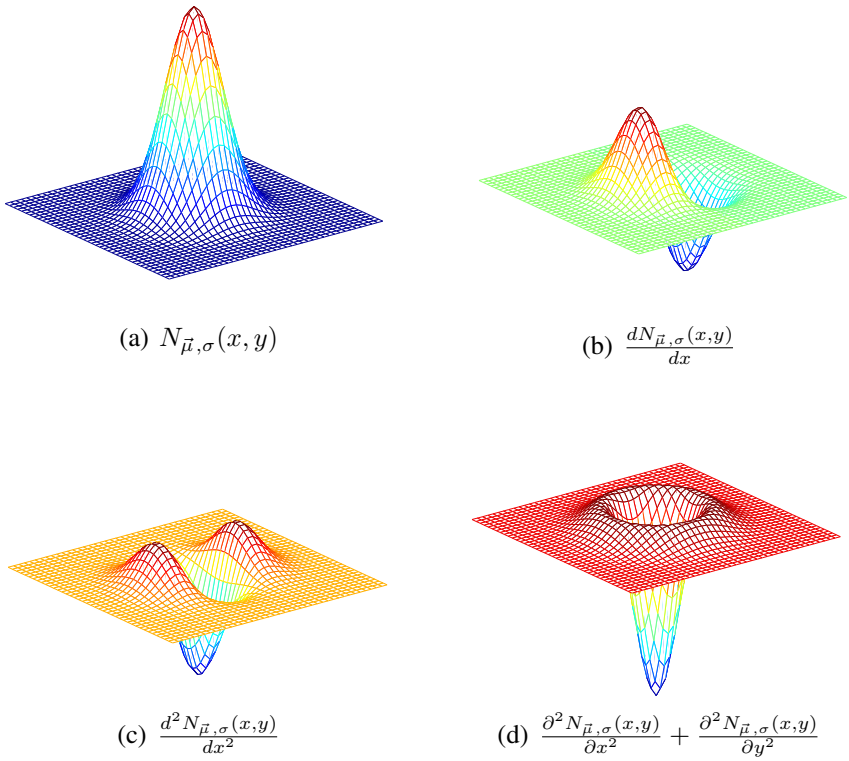
2. Given a certain choice for the time and space intervals Δt and Δx, how large is the error in the solution we make by replacing the continuous equation by a finite-difference equation?

3. If we let Δt and Δx go to zero, do the solutions of the finite-difference equation become identical to the solutions of the continuous version?

Studying these questions in detail is beyond the scope of this chapter. Such a study belongs to a branch of mathematics called *numerical analysis*. One result is worth mentioning here. It has been found that the finite-difference approximation may lead to instabilities in the solution if the step size Δt and Δx are not carefully chosen. The following *stability criterion* guarantees that such instabilities do not occur:

$$\frac{2D\Delta t}{(\Delta x)^2} \leq 1 \tag{8.9}$$

The interpretation of this formula is that the maximum allowed time step is, up to a factor, equal to the diffusion time across a cell of width Δx.



(a) $N_{\vec{\mu},\sigma}(x,y)$ (b) $\frac{dN_{\vec{\mu},\sigma}(x,y)}{dx}$

(c) $\frac{d^2N_{\vec{\mu},\sigma}(x,y)}{dx^2}$ (d) $\frac{\partial^2 N_{\vec{\mu},\sigma}(x,y)}{\partial x^2} + \frac{\partial^2 N_{\vec{\mu},\sigma}(x,y)}{\partial y^2}$

Figure 8.5: *Surface plots of the 2D isotropic Gaussian distribution $N_{\vec{\mu},\sigma}(x,y)$ and some of its derivatives ($\sigma = 3.0$).*

8.3 The diffusion equation in 2D

We will not discuss the 2D case in the same detail as the 1D case. When the space domain is two-dimensional, the concentration depends on three parameters: time t, and two spatial variables x and y. So we write $f(x, y, t)$ for the concentration. The diffusion equation now has the form:

$$\frac{\partial f(x,y,t)}{\partial t} = D\left(\frac{\partial^2 f(x,y,t)}{\partial x^2} + \frac{\partial^2 f(x,y,t)}{\partial y^2}\right) \tag{8.10}$$

Compare this to equation (8.1). The form of the equation is the same, but we now have two second order derivatives on the right-hand side, one with respect to x and one with respect to y.

Solutions to this equation are again a superposition of normal distributions, but now in two dimensions. Plots of a multivariate 2D isotropic Gaussian function $N_{\vec{\mu},\sigma}(x, y) = n_{\mu_1,\sigma}(x)\, n_{\mu_2,\sigma}(y)$ (mean $\vec{\mu} = (\mu_1, \mu_2)$, variance σ in both dimensions) and some of its derivatives are shown in Figure 8.5.

8.3.1 Numerical solution of the diffusion equation in 2D

The discretization is done in a similar way as for the 1D case. Now we have a three-dimensional space-time domain, with one time axis and two space axes. The time axis is again divided into cells of size Δt, and the two spatial axes into cells of sizes Δx and Δy, respectively. The concentration of material at time t_n and location (x_i, y_j) is $f(t_n, x_i, y_j)$. We abbreviate this by writing $f^n_{i,j}$.

The left-hand side of equation (8.10) is replaced by a finite difference,

$$\frac{\partial f(x_i, y_j, t_n)}{\partial t} \approx \frac{f^{n+1}_{i,j} - f^n_{i,j}}{\Delta t} \tag{8.11}$$

Next consider the right-hand side of equation (8.10). The first term is a second order derivative with respect to x. We replace this by a second-order difference:

$$\frac{\partial^2 f(x_i, y_j, t_n)}{\partial x^2} \approx \frac{f^n_{i+1,j} + f^n_{i-1,j} - 2f^n_{i,j}}{(\Delta x)^2} \tag{8.12}$$

Doing the same for the second order derivative with respect to y and combining all the results so far we find the discrete diffusion equation

$$\frac{f^{n+1}_{i,j} - f^n_{i,j}}{\Delta t} = D\left(\frac{f^n_{i+1,j} + f^n_{i-1,j} - 2f^n_{i,j}}{(\Delta x)^2} + \frac{f^n_{i,j+1} + f^n_{i,j-1} - 2f^n_{i,j}}{(\Delta y)^2}\right). \tag{8.13}$$

This equation can again be written in explicit form, as in equation (8.8).

Next we will look at biological vision and show how we can use simulated diffusion to create so-called *linear scale spaces* which are used in computer vision.

8.4 Reaction-diffusion systems

In general it is still an open question how exactly various patterns seen in nature arise (see Fig. 8.6 for some examples). In 1952 Alan Turing (also known for his involvement in defeating Enigma, as well as the Turing test and Turing machines already mentioned in Chapter 4) published a fairly simple (and still relevant (Maini et al., 2012)) model that at least shows how many such patterns could develop. The model has seen countless tweaks and variations, but the essence has remained the same: multiple "substances" that diffuse and react can (under certain conditions) give rise to stable patterns that always look similar but are never exactly the same. Note that the "substances" were originally considered to be chemicals, but you can also imagine different types of cells or even more macroscopic objects.

Figure 8.6: *Leopard spots, as well as stripes on fish could plausibly be explained by the kind of reaction-diffusion equations discussed in this section. (The leopard image was provided by user Karamash on the English Wikipedia and available under the CC BY 3.0 license.)*

We have already seen diffusion of a single "substance". Now we just add another substance that undergoes diffusion. Let us denote the concentration of the two diffusing substances by $f(x, y, t)$ and $g(x, y, t)$, respectively. If there is no interaction between the two substances then each of them will satisfy the diffusion equation, so we get:

$$\begin{aligned}\frac{\partial f(x,y,t)}{\partial t} &= D_f \left(\frac{\partial^2 f(x,y,t)}{\partial x^2} + \frac{\partial^2 f(x,y,t)}{\partial y^2} \right) \\ \frac{\partial g(x,y,t)}{\partial t} &= D_g \left(\frac{\partial^2 g(x,y,t)}{\partial x^2} + \frac{\partial^2 g(x,y,t)}{\partial y^2} \right),\end{aligned}$$

Here the diffusion constants D_f and D_g determine the rates of diffusion for f and g, respectively.

Now we take an essential step, which is to let the two substances *react* with one another. Then we get the following pair of equations, the *reaction-diffusion equation*:

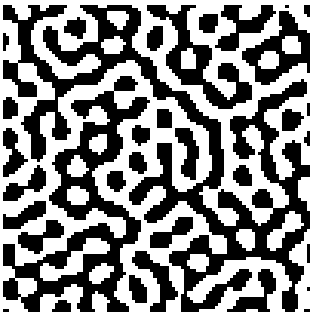
$$\begin{aligned}\frac{\partial f(x,y,t)}{\partial t} &= D_f \left(\frac{\partial^2 f(x,y,t)}{\partial x^2} + \frac{\partial^2 f(x,y,t)}{\partial y^2} \right) + \phi_f(f(x,y,t), g(x,y,t)) \\ \frac{\partial g(x,y,t)}{\partial t} &= D_g \left(\frac{\partial^2 g(x,y,t)}{\partial x^2} + \frac{\partial^2 g(x,y,t)}{\partial y^2} \right) + \phi_g(f(x,y,t), g(x,y,t)).\end{aligned} \tag{8.14}$$

Here ϕ_f and ϕ_g are two functions that both depend on the concentrations $f(x, y, t)$ and $g(x, y, t)$ and which take care of the reaction rate between the two substances.

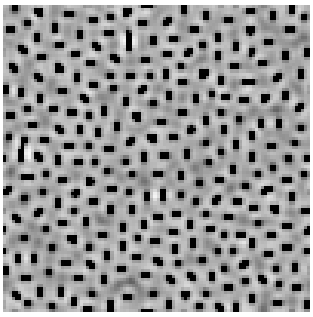
A simple case is the *linear model*, where the reaction rates are just simple linear functions of the concentrations (for simplicity we write f and g instead of $f(x, y, t)$ and $g(x, y, t)$):

$$\begin{aligned} D_f &= 1/10, & D_g &= 1/5, \\ \phi_f(f, g) &= \frac{1}{2}f - \frac{97}{128}g, & \phi_g(f, g) &= \frac{1}{2}f - \frac{3}{4}g. \end{aligned} \tag{8.15}$$

Figure 8.7(a) shows an example of a simulation of the linear model.



(a) Linear reaction rates



(b) Non-linear reaction rates

Figure 8.7: *Left: a simple example using linear reaction rates (with unbounded "concentrations", white being positive and black negative), see Eq.* (8.15)*. Right: an example with non-linear reaction rates from Turing's original paper (black is 0, white is 8), see Eq.* (8.16)*. In both cases the concentration of the second substance (g) is shown, and periodic boundary conditions are used.*

We are by no means limited to linear functions. The following *nonlinear model* was suggested by Turing (1952, p. 65):

$$\begin{aligned} D_f &= 1/4, & D_g &= 1/16, \\ \phi_f(f, g) &= \frac{1}{16}(16 - f \cdot g), & \phi_g(f, g, \beta) &= \begin{cases} \frac{1}{16}(f \cdot g - g - \beta) & \text{if } g > 0 \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{8.16}$$

Here ϕ_g depends not just on the local values of f and g, but also on that of β, which actually is a function $\beta(x, y)$ depending on space but not on time. Figure 8.7(b) shows an example of a simulation of the nonlinear model. In the given example $\beta(x, y)$ is normally distributed around 12 with a standard deviation of 0.1 and constant in time. One can think of β as giving the local concentration of a particular enzyme/catalyst, and its local fluctuations are what drives the system to break away from a constant solution and produce a pattern. In particular, it can be checked that $f(x, y) = g(x, y) = 4$ and $\beta(x, y) = 12$ for all x and y in the domain is a stable solution of the reaction-diffusion equations with the choices specified in Eq. (8.16).

All examples shown here involve fairly isotropic patterns, looking like spots or a kind of “maze”, but more elaborate systems do exist that allow generating a variety of patterns (Bard, 1981; Meinhardt, 1982; Turk, 1991).

Bibliography

Bard, J. B. L., 1981. A model for generating aspects of zebra and other mammalian coat patterns. Journal of Theoretical Biology 93 (2), 363–385.

Maini, P. K., Woolley, T. E., Baker, R. E., Gaffney, E. A., Lee, S. S., 2012. Turing's model for biological pattern formation and the robustness problem. Interface Focus 2 (4), 487–496.

Meinhardt, H., 1982. Models of biological pattern formation. Academic Press, London.

Turing, A. M., 1952. The Chemical Basis of Morphogenesis. Philosophical Transactions of the Royal Society of London B: Biological Sciences 237 (641), 37–72.

Turk, G., 1991. Generating Textures on Arbitrary Surfaces Using Reaction-diffusion. SIGGRAPH Computer Graphics 25 (4), 289–298.

Chapter 9

Sequence alignment

This chapter will study algorithms for sequence alignment. In the field of bioinformatics several new algorithms for sequence alignment have been developed. Therefore, we first explain in very general terms what biological sequence analysis is. Then we study some computer algorithms that biologists use to perform sequence analysis[1].

9.1 Biological background

We first present some basic biological background which is needed to understand the terms used. This will also equip you with the required knowledge for understanding sequence alignment, which is the topic of the remainder of this chapter.

9.1.1 DNA, RNA, proteins

The genetic information in living organisms is encoded in their DNA (or RNA, as in some viruses). A DNA molecule is a long, linear, chain molecule (a linear polymer) consisting of four *nucleotides*: deoxyAdenosine monophosphate, deoxyThymidine monophosphate, deoxyGuanosine monophosphate and deoxyCytidine monophosphate. Each nucleotide sub-unit consists of a phosphate, a deoxyribose sugar and one of the 4 nitrogenous nucleobases (usually simply called "bases"): adenine (abbreviated A), guanine (abbreviated G), cytosine (abbreviated C) and thymine (abbreviated T); see Figure 9.1.

These four nucleotides can be considered as a *four-letter alphabet*. RNA is a very similar polymer of Adenosine monophosphate, Guanosine monophosphate, Cytidine monophosphate, and Uridine monophosphate. Uridine monophosphate is a nucleotide functionally equivalent to Thymidine monophosphate; it contains the base uracil (abbreviated U). Since it is the bases which distinguish the different nucleotides from each other, one often uses "base" as a synonym of the corresponding nucleotide.

[1]Part of this text was adapted from Robert Giegerich and David Wheeler, Pairwise sequence alignment, `http://www.techfak.uni-bielefeld.de/bcd/Curric`, VSNS-BCD©.

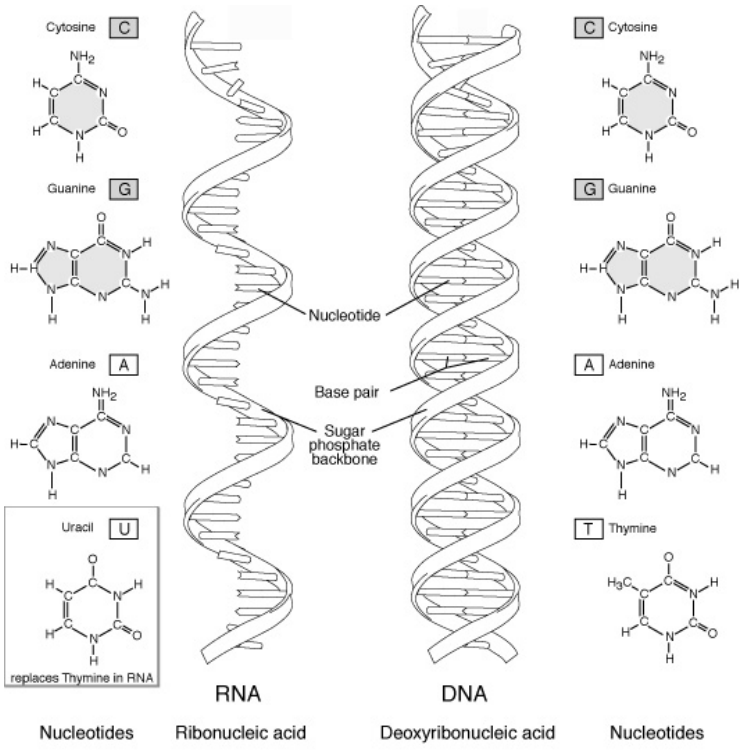


Figure 9.1: *The structure of DNA vs. RNA (©National Human Genome Research Institute, National Institutes of Health, USA).*

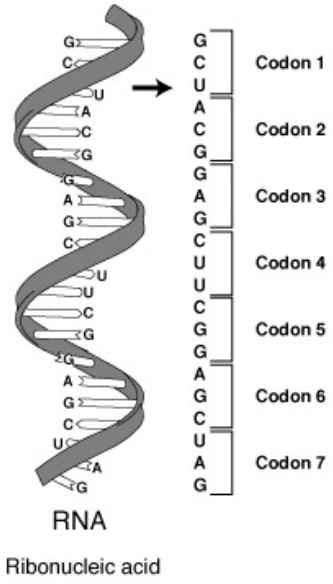


Figure 9.2: *Codons are triplets of bases from the RNA sequence (©National Human Genome Research Institute, National Institutes of Health, USA).*

A triplet of letters (i.e., bases) from the DNA sequence (or complementary RNA sequence) is called a *codon*. Each codon specifies an *amino-acid*; see Fig. 9.2. Successive triplets specify successive amino acids, and sequences of amino acids form *proteins*.

Since there are four different nucleotides, we can form $4^3 = 64$ different triplet codes. The way that these 64 codes are mapped onto 20 amino acids is called the *genetic code*. Since there are only around 20 amino acids, the genetic code is redundant. Actually, not all triplets code for an amino acid: 3 of the 64 codes, called *stop codons*, specify "end of amino acid sequence". The *standard genetic code* is presented in Table 9.1.

Table 9.1: *The standard genetic code.*

TTT Phe	TCT Ser	TAT Tyr	TGT Cys
TTC Phe	TCC Ser	TAC Tyr	TGC Cys
TTA Leu	TCA Ser	TAA STOP	TGA STOP
TTG Leu	TCG Ser	TAG STOP	TGG Trp
CTT Leu	CCT Pro	CAT His	CGT Arg
CTC Leu	CCC Pro	CAC His	CGC Arg
CTA Leu	CCA Pro	CAA Gln	CGA Arg
CTG Leu	CCG Pro	CAG Gln	CGG Arg
ATT Ile	ACT Thr	AAT Asn	AGT Ser
ATC Ile	ACC Thr	AAC Asn	AGC Ser
ATA Ile	ACA Thr	AAA Lys	AGA Arg
ATG Met	ACG Thr	AAG Lys	AGG Arg
GTT Val	GCT Ala	GAT Asp	GGT Gly
GTC Val	GCC Ala	GAC Asp	GGC Gly
GTA Val	GCA Ala	GAA Glu	GGA Gly
GTG Val	GCG Ala	GAG Glu	GGG Gly

9.1.2 Sequence similarity

Proteins and DNA can be similar with respect to their function, their structure, or their primary sequence of amino or nucleic acids. The general rule is that sequence determines protein shape, and shape determines function. So when we study sequence similarity, we eventually hope to discover or validate similarity in shape and function. This approach is often successful. However, there are many examples where two sequences have little or no similarity, but still the molecules fold into the same shape and share the same function. In this chapter we do not speak of shape or function. Sequences are seen as strings of characters. In fact, the ideas and techniques we discuss have important applications in text processing, too.

Similarity has both a quantitative and a qualitative aspect: A similarity measure gives a quantitative answer, saying that two sequences show a certain degree of similarity. A sequence alignment

is a mutual arrangement of two sequences which is a sort of qualitative answer; it exhibits where the two sequences are similar, and where they differ. An optimal alignment is one that exhibits the most correspondences, and the least differences.

9.2 Definition of sequence alignment

Given two (nucleotide or amino acid) sequences, we want to:

- measure their similarity;
- determine the correspondences between elements of the sequences.

Once this is possible one can take a given sequence and look in databanks for related sequences. This can then be used to make biological inferences:

- observe patterns of sequence conservation between related biological species and variability of sequences over time;
- infer evolutionary relationships.

We will consider two types of sequences, each with their own alphabet, which we denote by $\mathcal{A}$.

- the DNA alphabet with four letters: $\mathcal{A}$={A,C,T,G}.
- the amino acid alphabet with 20 letters.

The techniques we will describe are independent of the particular alphabet used. So we introduce the following general definition.

Definition 9.1 *Given an alphabet $\mathcal{A}$, a **string** is a finite sequence of letters from $\mathcal{A}$. **Sequence alignment** is the assignment of letter-letter correspondences between two or more strings from a given alphabet.*

DNA and protein molecules evolve mostly by three processes: point mutations (exchange of a single letter for another), insertions, and deletions. The process of sequence alignment aims at identifying locations of a gene or DNA sequence that are derived from a common ancestral locus (DNA location).

The following simple example is taken from Lesk (2005).
Consider two nucleotide strings GCTGAACG and CTATAATC. Some possible alignments are:

An alignment without gaps:

```
G C T G A A C G
C T A T A A T C
```

An alignment with gaps:

```
G C T G A – A – – C G
– – C T – A T A A T C
```

Another alignment with gaps:

```
G C T G – A A – C G
– C T A T A A T C –
```

The first alignment simply aligns each position of sequence 1 with the corresponding position of sequence 2. The second alignment introduces gaps, but without any resulting match. The third alignment introduces gaps at strategic positions in order to maximize the number of matches.

Sometimes one uses vertical bars to indicate exact matches in an alignment. For example, the last alignment would then be written as follows:

```
G C T G – A A – C G
  | |     | |   |
– C T A T A A T C –
```

Clearly, to decide which alignment is the best of all possibilities, we need:

1. A way to systematically examine all possible alignments;
2. A score for each possible alignment, which reflects the *similarity* of the two sequences.

The **optimal alignment** will then be the one with the highest similarity score. Note that there may be more than one optimal alignment.

The example above illustrates *pairwise* sequence alignment. A mutual alignment of more than two sequences is called *multiple* sequence alignment. Such multiple alignments are more informative than pairwise alignments in terms of revealing patterns of conservation. In this chapter we will restrict ourselves to pairwise alignment.

9.3 Dotplot

The dotplot is a simple graphical way to give an overview of pairwise sequence similarity. The dotplot is a table or matrix, where the rows correspond to the characters of one sequence and the columns to the characters of the other sequence. In each cell (i, j), corresponding to row i and column j, some graphical symbol (letter, color, dot, etc.) is inserted when the character at position i in sequence 1 matches the character at position j in sequence 2. If there is no match, the cell is left blank. Stretches of similar characters will show up in the dotplot as diagonals in the upper-left to lower-right direction.

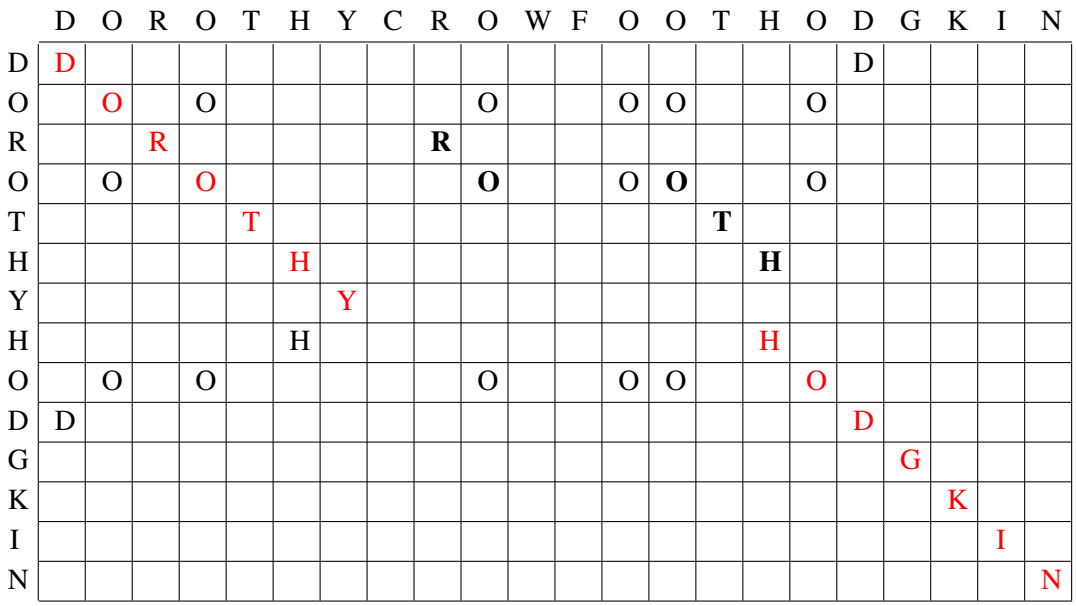


Figure 9.3: *Dotplot showing identities between short name (DOROTHYHODGKIN) and full name (DOROTHYCROWFOOTHODGKIN) of a famous protein crystallographer.*

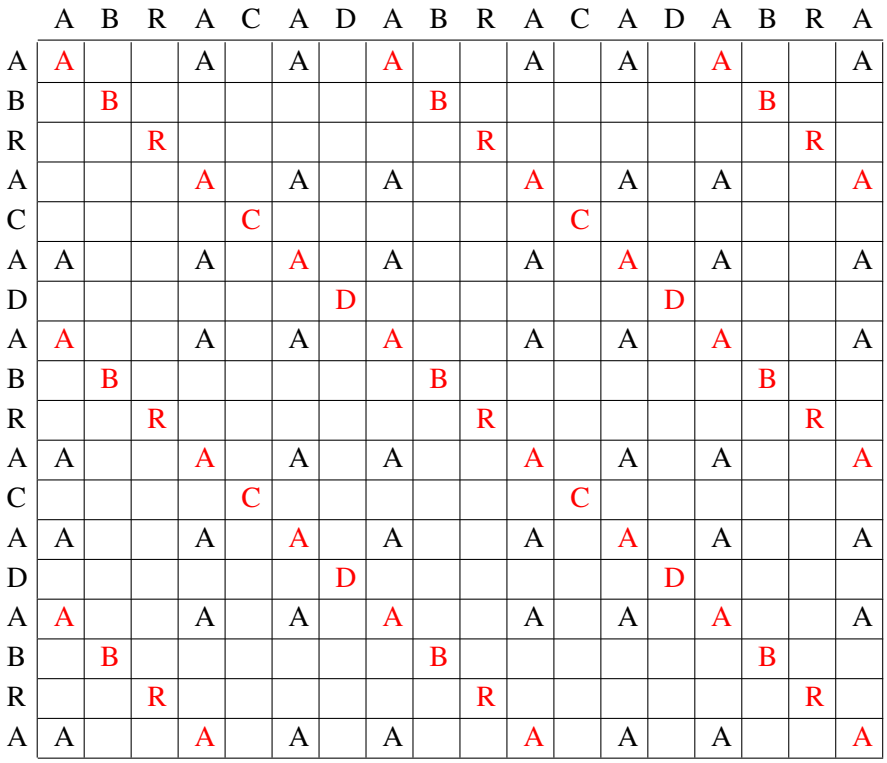


Figure 9.4: *Dotplot showing identities between a repetitive sequence (ABRACADABRA-CADABRA) and itself.*

The illustrations in Figures 9.3-9.4 of a dotplot are taken from (Lesk, 2005, Example 4.1). Here we use letters as symbols in the cells: whenever there is a match of two letters we plot that letter in the cell; otherwise we leave it blank. In Figure 9.3 letters corresponding to isolated matches are shown in non-bold type. The longest matching regions, shown in red, are the first and last names DOROTHY and HODGKIN. Shorter matching regions, such as the OTH of dorOTHy and crowfoOTHodgkin are noise; these are indicated in bold.

Figure 9.4 contains a dotplot showing identities between a repetitive sequence and itself: ABRACADABRACADABRA. The repeats appear on several subsidiary diagonals parallel to the main diagonal. Repetitive sequences are very common in DNA sequences.

When the sequences become very large it is impractical to use letters in the cells where matches occur. Instead we simply put a dot. This is the 'real' dotplot.

Filtering. To remove very short stretches and avoid many small gaps along stretches of matches one may use the filtering parameters *window* and *threshold*. This means that a dot will appear in a cell of the dotplot if that cell is in the center of a stretch of characters of length *window* such that the number of matches is larger than or equal to the value of *threshold*. Typical values would be a window of size 15 with a threshold of 6. Another option is to give the cell a color (or grey value), such that the higher the number of matches in the window, the more intense the color becomes.

9.4 Measures of sequence similarity

Two ways are used to quantify similarity of two sequences:

1. By a **similarity measure**. This is a function that associates a numeric value with a pair of sequences, such that a higher value indicates greater similarity.

2. By a **distance function**. This is somewhat dual to similarity. A distance measure is a function that also associates a numeric value with a pair of sequences, such that the larger the distance, the smaller the similarity, and vice-versa (so a distance function is a *dissimilarity* measure). Distance measures usually satisfy the mathematical axioms of a metric. In particular, distance values are never negative.

In most cases, distance and similarity measures are interchangeable in the sense that a small distance means high similarity, and vice-versa. Two often-used distance measures are the following.

Hamming distance. For two strings of equal length their Hamming distance is the number of character positions in which they differ. For example:

s :	A	G	T	C
t :	C	G	T	A

Hamming distance = 2

s :	A	G	C	A	C	A	C	A
t :	A	C	A	C	A	C	T	A

Hamming distance = 6

The Hamming distance measure is very useful in some cases, but in general it is not flexible enough. First of all, sequences may have different length. Second, there is generally no fixed correspondence between their character positions. In the mechanism of DNA replication, errors like deleting or inserting a nucleotide are not unusual. Although the rest of the sequences may be identical, such a shift of position leads to exaggerated values in the Hamming distance. Look at the second example above. The Hamming distance says that s and t are apart by 6 characters (out of 8). On the other hand, by deleting G from s and T from t, both become equal to ACACACA. In this sense, they are only two characters apart! This observation leads to the concept of *edit distance*.

Edit distance. For two strings of not necessarily equal length a distance can be based on the number of 'edit operations' required to change one string to the other. Here an edit operation is a *deletion*, *insertion* or *alteration* of a single character in either sequence.

Given two sequences s and t, we consider the following one-character edit operations. We introduce a gap character "–" and say that the pair:

- (a, a) denotes a match (no change from s to t)
- $(a, -)$ denotes deletion of character a (in s); it is indicated by inserting a "–" symbol in t
- (a, b) denotes replacement of a (in s) by b (in t), where $a \neq b$
- $(-, b)$ denotes insertion of character b (in s); it is indicated by inserting a "–" symbol in s.

Since the problem is symmetric in s and t, a deletion in s can be seen as an insertion in t, and vice-versa. An alignment of two sequences s and t is an arrangement of s and t by position, where s and t can be padded with gap symbols to achieve the same length. For the last two sequences s and t mentioned above this yields:

Table 9.2: *Two examples of alignment of two sequences.*

s :	A	G	C	A	C	A	C	–	A
t :	A	–	C	A	C	A	C	T	A

or

s :	A	G	–	C	A	C	A	C	A
t :	A	C	A	C	A	C	T	–	A

If we read the alignment column-wise, we have a protocol of edit operations that lead from s to t.

Left:	Match	(A, A)	Right	Match	(A, A)
	Delete	(G, –)		Replace	(G, C)
	Match	(C, C)		Insert	(–, A)
	Match	(A, A)		Match	(C, C)
	Match	(C, C)		Match	(A, A)
	Match	(A, A)		Match	(C, C)
	Match	(C, C)		Replace	(A, T)
	Insert	(–, T)		Delete	(C, –)
	Match	(A, A)		Match	(A, A)

The left-hand alignment shows one Delete, one Insert, and seven Matches. The right-hand alignment shows one Insert, one Delete, two Replaces, and five Matches.

Unit cost model. We turn the edit protocol above into a measure of distance by assigning a "cost" or "weight" w to each operation. For example, for arbitrary characters a, b from the alphabet $\mathcal{A}$ we may define:

$$w(a, a) = 0 \tag{9.1}$$
$$w(a, b) = 1 \text{ for } a \neq b \tag{9.2}$$
$$w(a, -) = w(-, b) = 1 \tag{9.3}$$

This scheme is known as the **Levenshtein Distance**, also called **unit cost model**. Its predominant virtue is its simplicity. In general, more sophisticated cost models must be used (see section 9.4.1).

Now we are ready to define the most important notion for sequence analysis.

Definition 9.2 ***Edit distance***

- *The **cost** of an alignment of two sequences s and t is the sum of the costs of all the edit operations needed to transform s to t.*
- *An **optimal alignment** of s and t is an alignment which has minimal cost among all possible alignments.*
- *The **edit distance** of s and t is the cost of an optimal alignment of s and t under a cost function w. We denote it by $d_w(s; t)$.*

Using the unit cost model for w for the example in Table 9.2, we obtain a cost of 2 for the left alignment and a cost of 4 for the right alignment. Here it is easily seen that the left-hand assignment is optimal under the unit cost model, and hence the edit distance $d_w(s; t) = 2$.

9.4.1 Scoring functions

For applications in molecular biology, it is important to realize that some changes in nucleotide or amino acid sequences are more likely than others. For example, amino acid substitutions tend to be conservative. This means that it is likely that an amino acid is replaced by another amino acid with similar physicochemical properties. Also, the deletion of a contiguous sequence of elements (bases or amino acids) is more probable that the independent deletion of the same number of elements at non-contiguous positions. Therefore we want to assign variable weights to different edit operations.

This leads to the concept of *scoring functions* or *substitution matrices*. A substitution matrix is a square array of values which indicate the scores associated to possible transitions (substitutions, insertions, deletions). One uses either:

- *Similarity scores*. Here substitutions that are more likely get a higher score.
- *Dissimilarity scores*. Here substitutions that are more likely get a lower score. The edit distance falls in this category: more likely transitions get lower scores (costs).

Similarity scoring schemes for nucleid acid sequences. In the case of nucleid acid sequence comparison, one uses a *Percent Identity* substitution matrix. For example, a 99% and a 50% identity matrix have the following forms, respectively[2]:

	A	T	G	C
A	+1	-3	-3	-3
T	-3	+1	-3	-3
G	-3	-3	+1	-3
C	-3	-3	-3	+1

99% identity matrix

	A	T	G	C
A	+3	-2	-2	-2
T	-2	+3	-2	-2
G	-2	-2	+3	-2
C	-2	-2	-2	+3

50% identity matrix

When we use the 99% identity matrix, mismatches have a large penalty, so we look for strong alignments. For the case of the 50% identity matrix, weaker alignments have higher scores. For example, for the two sequences

```
C A G G T A G C A A G C
| |   | | | | | |   | |
C A T G T A G C A C G C
```

the 99% score is $10 \cdot 1 - 2 \cdot 3 = 4$, but the 50% score is higher, i.e., $10 \cdot 3 - 2 \cdot 2 = 26$.

More complicated matrices may be used which reflect the fact that transitions A↔G and T↔C (transition mutations) are more common than (A or G)↔(T or C) (transversion mutations).

In addition to the substitution matrix, one has to specify values for creating and extending a gap in the alignment.

[2]The derivation of these matrices is outside the scope of this introduction.

Similarity scoring schemes for amino acid sequences. In the case of amino acid sequence comparison, the two most common schemes are:

- **PAM** (Percent Accepted Mutation) matrix, developed by Margaret Dayhoff in the 1970s. This estimates the rate at which each possible residue in a sequence changes to each other residue over time. 1 PAM = 1 percent accepted mutation. For example PAM30 corresponds to 75% sequence similarity, PAM250 to 20% similarity.

  As an example we show the PAM250 scoring matrix in Table 9.3. This PAM 250 matrix has a built-in gap penalty of -8, as seen in the * column (of course, other gap penalties may be used)[3]. There are 24 rows and 24 columns. The first 20 are the amino acids, represented by the one letter code. B represents the case where there is ambiguity between aspartate or asparigine, and Z is the case where there is ambiguity between glutamate or glutamine. X represents an unknown, or nonstandard, amino acid.

- **BLOSUM** (BLOck SUbstitution Matrix), developed by S. Henikoff and J.G. Henikoff. A BLOSUM-X matrix identifies sequences that are X% similar to the query sequence, based on a more realistic model of amino acid substitutions. For example, one often uses the BLOSUM50 for 50% or BLOSUM62 matrix for 62% sequence identity. BLOSUM62 is used for closer sequences than BLOSUM50. The BLOSUM50 matrix has the form given in Table 9.4. The coding is the same as for the PAM250 matrix (J is the case where there is ambiguity between Leucine or Isoleucine).

```
I G R H R Y H I G – G
  : |   | |   | |   |
– S – – R Y – I G R G
```

9.5 Dotplots and sequence alignment

It is helpful to return to the dotplot , because it captures not only sequence similarity, but also the complete set of possible alignments. Consider again the simple example of Figure 9.3. Any path through this dotplot from upper left to lower right, moving at each cell only East, South or Southeast, corresponds to a possible alignment. This is visualized in Figure 9.5.

[3] A PAM matrix calculator is available at `http://www.bioinformatics.nl/tools/pam.html`.

Table 9.3: *The PAM250 scoring matrix (source:* `http://www.bioinformatics.nl/tools/pam.html`*).*

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	2	-2	0	0	-2	0	0	1	-1	-1	-2	-1	-1	-3	1	1	1	-6	-3	0	0	0	0	-8
R	-2	6	0	-1	-4	1	-1	-3	2	-2	-3	3	0	-4	0	0	-1	2	-4	-2	-1	0	-1	-8
N	0	0	2	2	-4	1	1	0	2	-2	-3	1	-2	-3	0	1	0	-4	-2	-2	2	1	0	-8
D	0	-1	2	4	-5	2	3	1	1	-2	-4	0	-3	-6	-1	0	0	-7	-4	-2	3	3	-1	-8
C	-2	-4	-4	-5	12	-5	-5	-3	-3	-2	-6	-5	-5	-4	-3	0	-2	-8	0	-2	-4	-5	-3	-8
Q	0	1	1	2	-5	4	2	-1	3	-2	-2	1	-1	-5	0	-1	-1	-5	-4	-2	1	3	-1	-8
E	0	-1	1	3	-5	2	4	0	1	-2	-3	0	-2	-5	-1	0	0	-7	-4	-2	3	3	-1	-8
G	1	-3	0	1	-3	-1	0	5	-2	-3	-4	-2	-3	-5	0	1	0	-7	-5	-1	0	0	-1	-8
H	-1	2	2	1	-3	3	1	-2	6	-2	-2	0	-2	-2	0	-1	-1	-3	0	-2	1	2	-1	-8
I	-1	-2	-2	-2	-2	-2	-2	-3	-2	5	2	-2	2	1	-2	-1	0	-5	-1	4	-2	-2	-1	-8
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6	-3	4	2	-3	-3	-2	-2	-1	2	-3	-3	-1	-8
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5	0	-5	-1	0	0	-3	-4	-2	1	0	-1	-8
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	6	0	-2	-2	-1	-4	-2	2	-2	-2	-1	-8
F	-3	-4	-3	-6	-4	-5	-5	-5	-2	1	2	-5	0	9	-5	-3	-3	0	7	-1	-4	-5	-2	-8
P	1	0	0	-1	-3	0	-1	0	0	-2	-3	-1	-2	-5	6	1	0	-6	-5	-1	-1	0	-1	-8
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	2	1	-2	-3	-1	0	0	0	-8
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3	-5	-3	0	0	-1	0	-8
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17	0	-6	-5	-6	-4	-8
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-4	-2	7	-5	-3	-3	0	10	-2	-3	-4	-2	-8
V	0	-2	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	-1	0	-6	-2	4	-2	-2	-1	-8
B	0	-1	2	3	-4	1	3	0	1	-2	-3	1	-2	-4	-1	0	0	-5	-3	-2	3	2	-1	-8
Z	0	0	1	3	-5	3	3	0	2	-2	-3	0	-2	-5	0	0	-1	-6	-4	-2	2	3	-1	-8
X	0	-1	0	-1	-3	-1	-1	-1	-1	-1	-1	-1	-1	-2	-1	0	0	-4	-2	-1	-1	-1	-1	-8
*	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	1

Table 9.4: *The BLOSUM50 scoring matrix (source:* `http://www.ncbi.nlm.nih.gov/IEB/ToolBox/C_DOC/lxr/source/data/BLOSUM50`*).*

```
# Entries for the BLOSUM50 matrix at a scale of ln(2)/3.0.
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  J  Z  X  *
A  5 -2 -1 -2 -1 -1 -1  0 -2 -1 -2 -1 -1 -3 -1  1  0 -3 -2  0 -2 -2 -1 -1 -5
R -2  7 -1 -2 -4  1  0 -3  0 -4 -3  3 -2 -3 -3 -1 -1 -3 -1 -3 -1 -3  0 -1 -5
N -1 -1  7  2 -2  0  0  0  1 -3 -4  0 -2 -4 -2  1  0 -4 -2 -3  5 -4  0 -1 -5
D -2 -2  2  8 -4  0  2 -1 -1 -4 -4 -1 -4 -5 -1  0 -1 -5 -3 -4  6 -4  1 -1 -5
C -1 -4 -2 -4 13 -3 -3 -3 -3 -2 -2 -3 -2 -2 -4 -1 -1 -5 -3 -1 -3 -2 -3 -1 -5
Q -1  1  0  0 -3  7  2 -2  1 -3 -2  2  0 -4 -1  0 -1 -1 -1 -3  0 -3  4 -1 -5
E -1  0  0  2 -3  2  6 -3  0 -4 -3  1 -2 -3 -1 -1 -1 -3 -2 -3  1 -3  5 -1 -5
G  0 -3  0 -1 -3 -2 -3  8 -2 -4 -4 -2 -3 -4 -2  0 -2 -3 -3 -4 -1 -4 -2 -1 -5
H -2  0  1 -1 -3  1  0 -2 10 -4 -3  0 -1 -1 -2 -1 -2 -3  2 -4  0 -3  0 -1 -5
I -1 -4 -3 -4 -2 -3 -4 -4 -4  5  2 -3  2  0 -3 -3 -1 -3 -1  4 -4  4 -3 -1 -5
L -2 -3 -4 -4 -2 -2 -3 -4 -3  2  5 -3  3  1 -4 -3 -1 -2 -1  1 -4  4 -3 -1 -5
K -1  3  0 -1 -3  2  1 -2  0 -3 -3  6 -2 -4 -1  0 -1 -3 -2 -3  0 -3  1 -1 -5
M -1 -2 -2 -4 -2  0 -2 -3 -1  2  3 -2  7  0 -3 -2 -1 -1  0  1 -3  2 -1 -1 -5
F -3 -3 -4 -5 -2 -4 -3 -4 -1  0  1 -4  0  8 -4 -3 -2  1  4 -1 -4  1 -4 -1 -5
P -1 -3 -2 -1 -4 -1 -1 -2 -2 -3 -4 -1 -3 -4 10 -1 -1 -4 -3 -3 -2 -3 -1 -1 -5
S  1 -1  1  0 -1  0 -1  0 -1 -3 -3  0 -2 -3 -1  5  2 -4 -2 -2  0 -3  0 -1 -5
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  2  5 -3 -2  0  0 -1 -1 -1 -5
W -3 -3 -4 -5 -5 -1 -3 -3 -3 -3 -2 -3 -1  1 -4 -4 -3 15  2 -3 -5 -2 -2 -1 -5
Y -2 -1 -2 -3 -3 -1 -2 -3  2 -1 -1 -2  0  4 -3 -2 -2  2  8 -1 -3 -1 -2 -1 -5
V  0 -3 -3 -4 -1 -3 -3 -4 -4  4  1 -3  1 -1 -3 -2  0 -3 -1  5 -3  2 -3 -1 -5
B -2 -1  5  6 -3  0  1 -1  0 -4 -4  0 -3 -4 -2  0  0 -5 -3 -3  6 -4  1 -1 -5
J -2 -3 -4 -4 -2 -3 -3 -4 -3  4  4 -3  2  1 -3 -3 -1 -2 -1  2 -4  4 -3 -1 -5
Z -1  0  0  1 -3  4  5 -2  0 -3 -3  1 -1 -4 -1  0 -1 -2 -2 -3  1 -3  5 -1 -5
X -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -5
* -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5  1
```

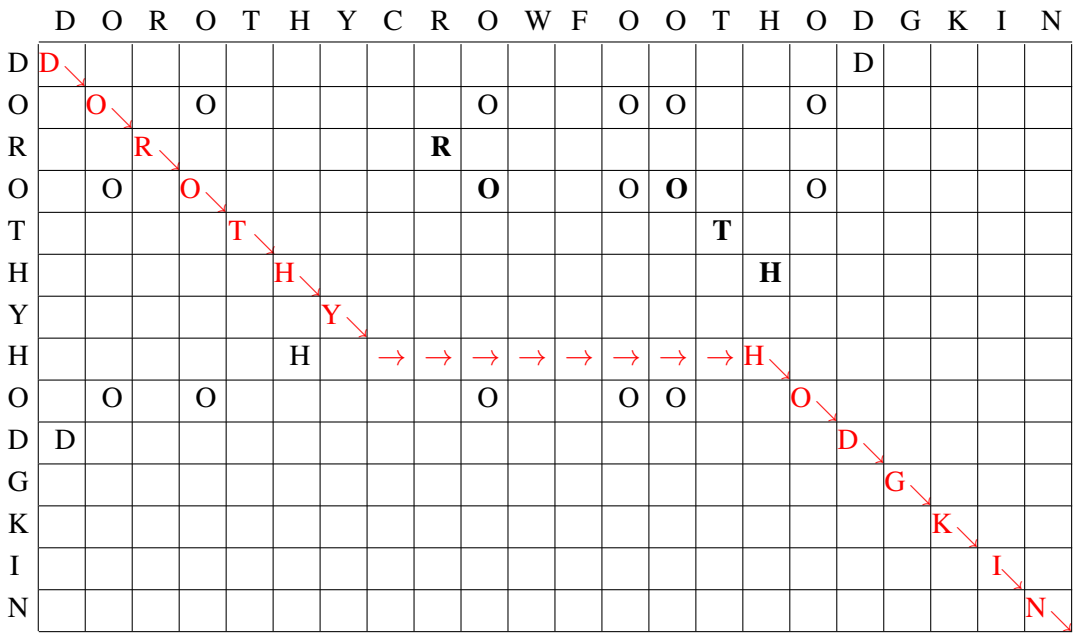


Figure 9.5: *Any path through this dotplot from upper left to lower right, moving at each cell only East, South or Southeast, corresponds to a possible alignment.*

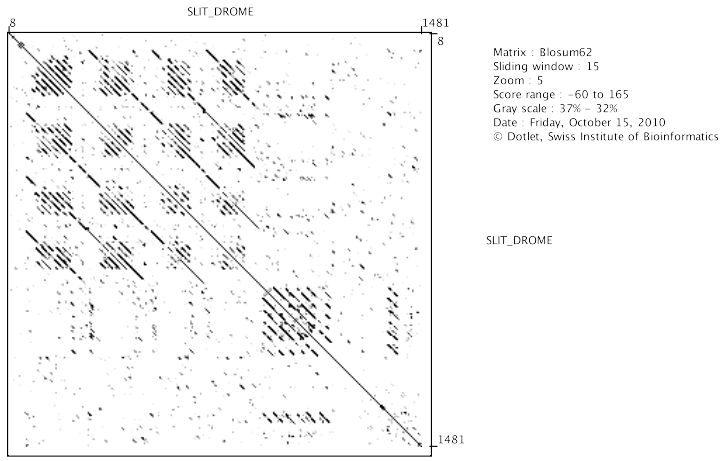
The path consists of a succession of cells, each of which:

- pairs a character from the row with a character from the column;
- or indicates a gap in one of the sequences.

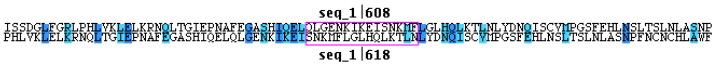
If the direction of a move between successive cells is diagonal, two pairs of successive characters appear in the alignment without an insertion between them. If the move is horizontal (vertical), a gap is introduced in the sequence indexing the rows (columns). Note that the path need not pass through filled-in points. However, the more filled-in points are on the path, the more matching characters the alignment will contain. This is for example the case for Figure 9.5. The path in this figure corresponds to the alignment:

D	O	R	O	T	H	Y	–	–	–	–	–	–	–	–	H	O	D	G	K	I	N
D	O	R	O	T	H	Y	C	R	O	W	F	O	O	T	H	O	D	G	K	I	N

An example of a dotplot of the amino acid sequence of the SLIT protein of Drosophila melanogaster (the fruit fly) is given in Figure 9.6. This protein is necessary for the development of certain brain structures of the fruit fly.



(a)



(b)

Figure 9.6: *(a): Dotplot of the amino acid sequence of Drosophila melanogaster SLIT protein. (b): alignment in a local region of the dotplot (plots made by the web tool* `Dotlet` *available at* `http://myhits.isb-sib.ch/cgi-bin/dotlet`*).*

9.6 Pairwise alignment via dynamic programming

Now we come to an important question: how do we *compute* sequence alignments?

The number of possible alignments between two large DNA or amino acid sequences is gigantic, and unless the weight function is very simple, it may seem difficult to pick out an optimal alignment. But fortunately, there is an easy and systematic way to find it. The method described now is very famous in mathematical optimization and computer programming. It is usually called "the dynamic programming algorithm", and was developed by Richard Bellman[4]. This algorithm is a method for solving complex problems by breaking them down into simpler steps.

The idea to use dynamic programming to solve the global pairwise sequence alignment problem was first put forward by Needleman and Wunsch (1970). For this reason the method we will describe below is known as the *Needleman-Wunsch algorithm*[5].

But first some words of caution.

- The algorithm is guaranteed to give a global optimum: it will find the best alignment score, i.e., the minimal cost, given the weight parameters.

[4] A special case of this is the shortest path algorithm of the Dutch computer scientist E.W. Dijkstra.

[5] Actually, it is a simplified version of this algorithm.

- However, many alignments may give the same optimal score, and none of these may actually correspond to the biologically correct one. Many alignments may exist with scores close to the optimum, and one of these may be the correct one.

- The time to align two sequences of lengths n and m is proportional to $n \times m$. So this method is not suitable for matching one sequence against an entire database of sequences.

We will only outline the basic idea of the algorithm. A detailed treatment of this topic will have to wait until the course on Algorithms & Datastructures.

9.6.1 Optimal substructure property

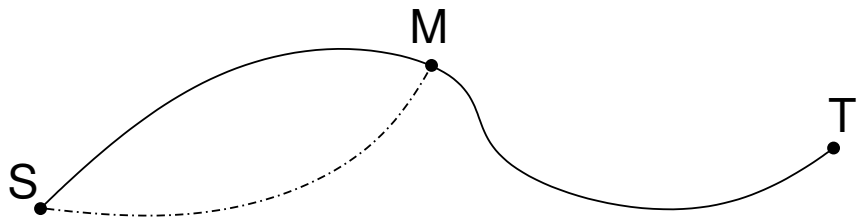


Figure 9.7: *If M is a point on an optimal path $\pi_{[S\to T]}$ from point S to point T (solid line), then the paths $\pi_{[S\to M]}$ and $\pi_{[M\to T]}$ along the solid line are also optimal paths. The cost of the dotted path from S to M cannot be smaller than the cost of the solid path from S to M.*

The main observation is the following, which is usually called the *Optimal substructure* property; see Figure 9.7.

Observation (Optimal substructure). *Consider a path $\pi_{[S\to T]}$ between two points S and T which is optimal, i.e., has the lowest cost. Let M be a point on the path $\pi_{[S\to T]}$. Then the path $\pi_{[S\to M]}$ from S to M which everywhere follows the path from S to T is also an optimal path, i.e., has lowest cost of all paths from S to M.*

To show that this observation is correct, we use a proof by contradiction[6]. So let us assume that the path $\pi_{[S\to M]}$ from S to M is not an optimal path. Then there would be another path $\pi'_{[S\to M]}$ from S to M with a lower cost than the path $\pi_{[S\to M]}$. But then the path $\pi'_{[S\to M]}$ followed by the path $\pi_{[M\to T]}$ would be a path from S to T with a smaller cost than the original path $\pi_{[S\to T]}$. But this is impossible, since we assumed that $\pi_{[S\to T]}$ was optimal.

The main observation can be used to systematically subdivide the optimal alignment problem in parts which are slightly smaller. The dynamic programming method is based on this idea.

[6] In Dutch: Bewijs uit het ongerijmde.

9.6.2 Recursive computation of the edit distance

Remember that the **edit distance** $d_w(s;t)$ of two sequences $s = a_1a_2\ldots a_n$ and $t = b_1b_2\ldots b_m$ is the cost of an optimal alignment of s and t under a cost function w. That is, $d_w(s;t)$ is the minimum[7] of all sequences of edit operations that convert s and t into a common sequence.

In the context of the dotplot matrix, the alignment problem can be reformulated as follows. Find a path through the matrix from the upper left to the lower right (with moves to the East, South or Southeast only) that has the lowest cost. We can do this by creating a matrix by D, with elements $D(i,j)$, $i = 1,2,\ldots,n$ and $j = 1,2,\ldots,m$, such that $D(i,j)$ is the minimal edit distance between the sequences that consist of the first i characters of s and the first j characters of t. Then $D(n,m)$ will be the minimal edit distance between the full sequences s and t.

Sequences of edit operations correspond to paths in the dotplot matrix of the form

$$(i_0,j_0) = (0,0) \rightarrow (i_1,j_1) \rightarrow \cdots (n,m)$$

Each step in the matrix which arrives in cell (i,j) can be of three types, i.e., East (previous cell was $(i,j-1)$), South (previous cell was $(i-1,j)$), and SouthEast (previous cell was $(i-1,j-1)$). This corresponds to three possible edit operations with associated costs, as follows:

edit operation	**step in matrix**	**cost**
substitution of $a_i \rightarrow b_j$	$(i-1,j-1) \rightarrow (i,j)$	$w(a_i,b_j)$
deletion of a_i from sequence s	$(i-1,j) \rightarrow (i,j)$	$w(a_i,-)$
deletion of b_j from sequence t	$(i,j-1) \rightarrow (i,j)$	$w(-,b_j)$

The algorithm computes $D(i,j)$ by **recursion**. We compare the costs of the following three paths which arrive at cell (i,j):

1. The optimal path from the start $(0,0)$ to cell $(i-1,j-1)$, followed by the step $(i-1,j-1) \rightarrow (i,j)$. The cost of this path is $D(i-1,j-1) + w(a_i,b_j)$.
2. The optimal path from the start $(0,0)$ to cell $(i-1,j)$, followed by the step $(i-1,j) \rightarrow (i,j)$. The cost of this path is $D(i-1,j) + w(a_i,-)$.
3. The optimal path from the start $(0,0)$ to cell $(i,j-1)$, followed by the step $(i,j-1) \rightarrow (i,j)$. The cost of this path is $D(i,j-1) + w(-,b_j)$.

If we take the *minimum* of these three costs we get the cost $D(i,j)$ of the optimal path from the start $(0,0)$ to cell (i,j) (compare the *Optimal substructure* property above).

So we have derived the recursion:

$$D(i,j) = \min\{D(i-1,j-1) + w(a_i,b_j), D(i-1,j) + w(a_i,-), D(i,j-1) + w(-,b_j)\}$$

[7]Note that we have to take the minimum because edit distance is a *dissimilarity* score.

In order to retrieve the optimal *path* (and not only the optimal *cost*) after the calculation we also store a pointer (an arrow) to one of the three cells $(i-1, j-1)$, $(i-1, j)$ or $(i, j-1)$ that provided the minimal value. This cell is called the **predecessor** of (i, j). If there are more cells that provided the minimal value (remember that optimal paths need not be unique) we store a pointer to each of these cells.

On the top row and left column of the matrix we have no North or West neighbours, respectively. So here we have to initialize values:

$$D(i, 0) = \sum_{k=0}^{i} w(a_k, -), \qquad D(0, j) = \sum_{k=0}^{j} w(-, b_k)$$

which impose the gap penalty on unmatched characters at the beginning of either sequence.

In practice one often uses a constant gap penalty:

$$w(a_k, -) = w(-, b_k) = g$$

The pseudo-code for the algorithm to compute the D-matrix is given in Algorithm 9.1.

As an example, let us align the sequences s=GGAATGG and t=ATG, with scoring scheme:

$$w(a, a) = 0 \text{ (match)}$$
$$w(a, b) = 4 \text{ for } a \neq b \text{ (mismatch)}$$
$$w(a, -) = w(-, b) = 5 \text{ (gap insertion)}$$

After initialization and the first diagonal step the matrix looks as follows:

s \ t	–	A	T	G
–	0	5	10	15
G	5	**4**		
G	10			
A	15			
A	20			
T	25			
G	30			
G	35			

The value **4** was entered at position $(1, 1)$ since it is the smallest of the three possibilities $5 + 5$ (horizontal move), $5 + 5$ (vertical move), $0 + 4$ (diagonal move). This also means that cell (0,0) is the predecessor of cell (1,1).

After the complete calculation has finished, the matrix, including pointers to the predecessor(s) of each cell, looks as follows:

s \ t	–	A	T	G
–	0	5 ←	10 ←	15
	↑ ↖	↖	↖	
G	5	4 ←	9	10
	↑ ↖	↑ ↖	↖	
G	10	9	8	9
	↑ ↖	↖	↑ ↖	
A	15	10	13	12
	↑ ↖	↑ ↖	↖	↑
A	20	**15**	14	17
	↑	↑ ↖	↖	
T	25	20	15	18
	↑	↑	↑ ↖	
G	30	25	20	15
	↑	↑	↑ ↖	↑
G	35	30	25	**20**

The red arrows indicate trace-back paths of optimal alignment, starting at the lower right and moving back to upper left. There are two cells (with cost value drawn in bold) where the trace-back path branches. This gives four optimal alignments with equal score:

G	G	A	A	T	G	G
–	–	–	A	T	G	–

G	G	A	A	T	G	G
–	–	–	A	T	–	G

G	G	A	A	T	G	G
–	–	A	–	T	G	–

G	G	A	A	T	G	G
–	–	A	–	T	–	G

Algorithm 9.1 *Needleman-Wunsch algorithm for global pairwise sequence alignment.*

INPUT: two sequences $s = a_1 a_2 \dots a_n$ and $t = b_1 b_2 \dots b_m$; cost function w with gap penalty g
OUTPUT: matrix D containing the minimal edit distance between the sequences s and t
for $i = 0$ **to** n **do**
    $D(i, 0) \leftarrow g \cdot i$
end for
for $j = 0$ **to** m **do**
    $D(0, j) \leftarrow g \cdot j$
end for
for $i = 1$ **to** n **do**
    for $j = 1$ **to** m **do**
        Match $\leftarrow D(i-1, j-1) + w(a_i, b_j)$
        Delete $\leftarrow D(i-1, j) + g$
        Insert $\leftarrow D(i, j-1) + g$
        $D(i, j) \leftarrow \min$(Match, Insert, Delete)
    end for
end for

9.7 Variations and generalizations

What we have discussed so far is *global alignment* of two sequences. There are a number of variants and generalizations of this scheme that we briefly mention.

- **Local alignment**. Here we look for a region in one sequence that matches a region in another sequence. An algorithm for this purpose was developed by Smith and Waterman (1981). Or we probe a database with one sequence, regarding the database itself as one very long sequence. A well-known algorithm for this is PSI-BLAST ("Position Specific Iterative Basic Local Alignment Search Tool").

- **Motif match**. Here we look for matches of a short sequence fragment (the "motif") in one or more regions of a long sequence.

- **Multiple alignment**. Here we do a mutual alignment of more than two sequences.
- **Significance of alignments**. Suppose we find an alignment between two sequences with a high similarity. Then we want to know whether this result is significant or could have arisen by chance. To answer this question, one looks at random permutations of one of the sequences, aligns each of them with the other sequence, and computes the distribution of the scores. If the randomized sequences score as well as the original sequence the alignment is unlikely to be significant. Statistical measures of this significance are the *Z-score=(score−mean)/standard deviation*, or the *p-value*, which is the probability that the observed match could have happened by chance.

9.8 Sequence logos

An interesting way to visualize multiple sequence alignments are the sequence logos (Schneider and Stephens, 1990).

The idea is to use a graphical display of multiple alignment, with colored stacks of letters representing nucleotides or amino acids at successive positions, such that **height** of a letter at a position increases with increasing **frequency** of amino acids at that position in the different sequences.

This means that letters in stacks with single amino acids—conserved positions— are taller than those in stacks with multiple amino acids— where there is more variation. An example is given in Figure 9.8. This logo shows a small sample of human exon-intron splice boundaries on the DNA (Stephens and Schneider, 1992). Splicing is a modification of mRNA after transcription, in which introns are removed and exons are joined, which is needed before the RNA can be used to produce a correct protein through translation. Looking at Figure 9.8, we see that at position 0 all sequences have the same base G; at position 4 all four bases occur, with G most frequently, T less frequently, etc.

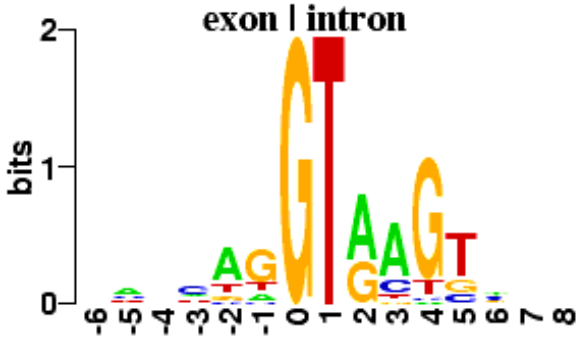


Figure 9.8: *Sequence logo of human exon-intron splice boundaries.*
© `http://weblogo.berkeley.edu`

A sequence logo is an alternative to a so-called **consensus sequence**, which would display a

single sequence as the representative of all the multiple sequences. Sequence logos have the advantage that information on the frequency of occurrence of the different letters is maintained.

9.9 Circular visualization

Another way to visualize alignments is through circular arrangements. The *Circos* tool (`http://mkweb.bcgsc.ca/circos`) was developed for this purpose. An example for the human genome is given in Figure 9.9. This figure shows the chromosomes arranged in a circular orientation, shown as wedges, marked with a length scale. Data placed outside of the chromosome ring represent small- and large-scale variations at a given genome position found between different populations.

Data placed on top of the chromosome ring highlight positions of genes implicated in disease, such as cancer, diabetes, and glaucoma. Data placed inside the ring link disease-related genes found in the same biochemical pathway (grey) and the degree of similarity for a subset of the genome (colored).

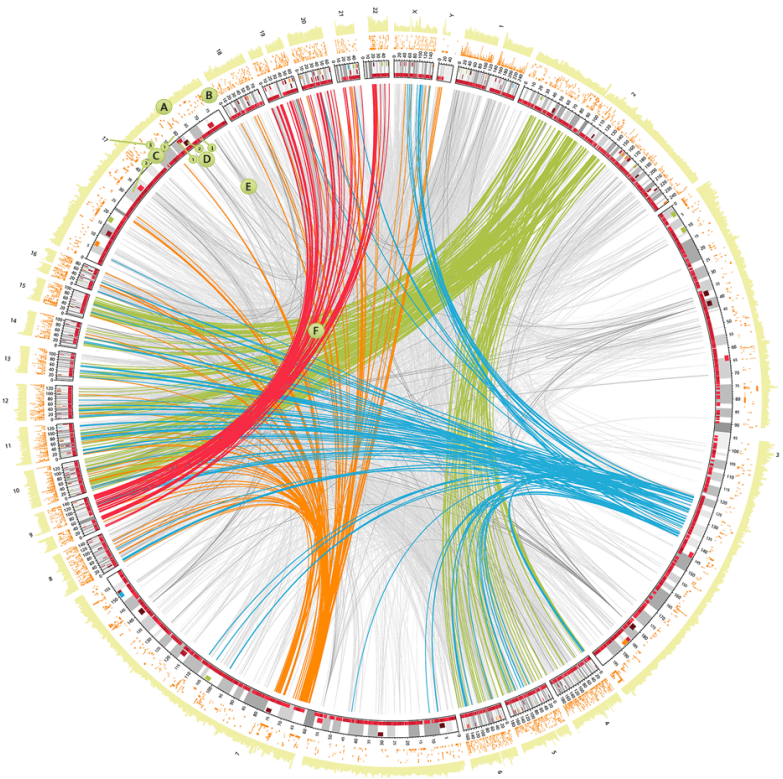


Figure 9.9: *An illustration of the human genome showing location of genes implicated in disease. Taken from* `http://mkweb.bcgsc.ca/circos/intro/genomic_data` *(original picture: The Condé Nast Portfolio* `http://www.portfolio.com/news-markets/national-news/portfolio/2007/10/15/23andMe-Web-Site`*).*

Bibliography

Lesk, A. M., 2005. Introduction to Bioinformatics (2nd ed.). Oxford University Press, New York, NY.

Needleman, S. B., Wunsch, C. D., 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology 48 (3), 443–453.

Schneider, T. D., Stephens, R. M., 1990. Sequence logos: A new way to display consensus sequences. Nucleic Acids Res. 18, 6097–6100.
URL `http://www.ccrnp.ncifcrf.gov/~toms/paper/logopaper/`

Smith, T. F., Waterman, M. S., 1981. Identification of common molecular subsequences. Journal of Molecular Biology 147 (1), 195–197.
URL `http://www.sciencedirect.com/science/article/B6WK7-4DN3Y5S-24/2/b00036bf942b543981e4b5b7943b3f9a`

Stephens, R. M., Schneider, T. D., 1992. Features of spliceosome evolution and function inferred from an analysis of the information at human splice sites. J. Mol. Biol. 228, 1124–1136.

Index