Introduction to Computational Physics WS07/08

Lecture of Prof. H. J. Herrmann

Swiss Federal Institute of Technology ETH, Zürich, Switzerland

Script by Dr. H. M. Singer
Computational Physics, IfB, ETH Zürich

October 1, 2008

General Information

Useful addresses and information

All information can be found on the web pages:

http://www.comphys.ethz.ch/index.php/lectures

and

http://www.ifb.ethz.ch/education/IntroductionComPhys

Pdf-files of the slides and exercises can be downloaded from there as well.

	place	time
Lecture	HCI D8	Friday 10:45-12:30
Exercises	HCI D451	Friday 08:45-10:30

For whom is this lecture intended?

The lecture gives an introduction to computational physics for students in the following departments:

- Mathematics and Computer Science (Bachelor and Master course).
- Physics (major course, "Wahlfach").
- Material Science (Master course).
- Civil Engineering (Master course).

Some words about your teacher

Prof. Hans. J. Herrmann is full professor at the Institute of Building Materials (IfB) since April 2006. His field of expertise is computational and statistical physics, in particular granular materials. His present research subjects include dense colloids, the formation of river deltas, quicksand, the failure of fibrous and polymeric composites and complex networks.
Prof. Herrmann can be reached at

hjherrmann@ethz.ch

His office is located in the Institute of Building materials (IfB), HIF E12, ETH Hönggerberg, Zürich.
The personal web page can be found under:

www.icp.uni-stuttgart.de/~hans

and under

www.comphys.ethz.ch

Comments to the script (in particular typos, errors and other mistakes) should be sent to Dr. H. M. Singer: hsinger@ethz.ch

Where to go from here

After having followed this course you will be able to delve deeper into the field of computational physics. Here is a selection of lectures you can take in the summer semester 2008 (SS08):

- *Computational Statistical Physics* by H.J. Herrmann
- *Computational Quantum Physics* by M. Troyer
- *Computational Polymer Physics* by M. Kröger

Outline of this course

28.9.07 General introduction, random numbers (RN)

5.10.07 RN, Percolation

12.10.07 Random Walks (held by Prof. M. Kröger)

19.10.07 Fractals, Cellular Automata

26.10.07 Finite size effects, Monte Carlo

2.11.07 Importance Sampling and Metropolis Algorithm

9.11.07 Differential equations (Euler, Runge Kutta)

16.11.07 Ising model (held by Prof. M. Troyer)

23.11.07 Equations of motion (Newton, Regula Falsi)

30.11.07 Finite Difference Methods, Relaxation

7.12.07 Gradient Methods

14.12.07 Multigrid Methods, Finite Elements Method

21.12.07 Variational FEM, Crank-Nicholson, Wave equations, Navier-Stokes Equation

Introductory examples

In the lecture there will be given examples for different simulations:

- Segregation under vibration (Brazil nut effect)

- Mixing in a cylinder (hard spheres)

- Sedimentation (glass beads descending in silicon oil)

- Motion of Sand Dunes

What should you know already?

- You should have a basic understanding of the UNIX operating system and be able to work with it. This means concepts such as the 'shell', stream redirections, compiling programs and writing small scripts in a scripting language should be familiar.
- You should ideally have some knowledge about a higher level programming language such as (Fortran, C/C++, Java, etc.). In particular you should also be able to write, compile and debug programs yourself.
- It is beneficial to know how to make scientific plots. There are many tools, which can help you with that. For example Matlab, Maple, Mathematica, R, SPlus, gnuplot, etc.
- Requirements in Mathematics:
  - You should know the basics of statistical analysis (averaging, distributions, etc.).
  - Also some knowledge of linear algebra and analysis will be necessary.
- Requirements in Physics
  - You should be familiar with Classical Mechanics (Newton, Lagrange) and Electrodynamics.
  - A basic understanding of Thermodynamics is also beneficial.

What is computational physics?

Computational physics is the study and implementation of numerical algorithms in order to solve problems in physics by means of computers. In particular computational physics solves equations numerically. This is useful, since there are very few systems where an analytical solutions is known. Another field of computational physics is the simulation of many-body/particle systems. This creates a virtual reality, which is sometimes also referred to as the 3rd branch of physics (between experiments and theory).

The evaluation and visualization of large data sets, which can come from numerical simulations or experimental data (for example maps in geophysics) is also part of computational physics.

Yet another part where computers are used in physics is the control of experiments. This is however not treated here in this lecture.

Here is a list of fields, where computational physics plays an important role:

- Computational Fluid Dynamics (CFD): solve and analyze problems that involve fluid flows.
- Classical Phase Transition: Percolation, critical phenomena.
- Solid State Physics (Quantum Mechanics).
- High Energy Physics / Particle Physics: in particular lattice quantum chromodynamics.
- Astrophysics: for example many-body simulations of stars, galaxies etc.
- Geophysics and Solid Mechanics: Earthquake simulations, fracture, rupture, crack propagation etc.
- Agent models (interdisciplinary): for example complex networks in biology, economy, social sciences.

Computer tools:

- On the level of software and programming the concept of object orientation has helped to create codes, which are much more easy to maintain and to extend since classes encapsulate properties of the objects.
- Hardware allows different possibilities how to improve the speed of execution for the simulations:
  - Vector supercomputers: this is a CPU design, which allows the execution of mathematical operations on multiple data elements simultaneously. These computers were popular in the 1980s and the beginning of 1990s (e.g. Cray Supercomputers).

  - Parallel computers: such computers can have different architectures. One possibility is to have different processors, which all share the same memory. This means, that every processor has access to all the data at any time (shared memory). The benefits of this architecture are that there is no overhead for communication for requesting data from other processors. However simulations are limited in the memory capacity of the machine. Another architecture is distributed parallel computing, where each processor has its own memory. This allows for a much higher total amount of memory (sum of all parts) because they don't communicate through the same data bus. The downside of this architecture is that a simulation must coordinate the tasks it wants to run in parallel and communicate between the processors.

- Symbolic Algebra tools such as Mathematica and Maple simplify the derivation of equations, which are optimized for calculations. In particular the derivation for Finite Element equations can be automated by symbolic computations.

- The visualization of the simulated data is very important. Since many simulations are evolutions of a system in time the graphical representation as animations facilitates the understanding of the dynamic processes involved. By means of 3D renderers it is also possible to inspect the systems from any point of view, even from inside out, to obtain more insights in the behavior of the system.

Suggested Literature

Books:

- H. Gould, J. Tobochnik and Wolfgang Christian: „*Introduction to Computer Simulation Methods*“ 3rd edition (Addison Wesley, Reading MA, 2006).
- D. P. Landau and K. Binder: „*A Guide to Monte Carlo Simulations in Statistical Physics*“ (Cambridge University Press, Cambridge, 2000).
- D. Stauffer, F. W. Hehl, V. Winkelmann and J. G. Zabolitzky: „*Computer Simulation and Computer Algebra*“ 3rd edition (Springer, Berlin, 1993).
- K. Binder and D. W. Heermann: „*Monte Carlo Simulation in Statistical Physics*“ 4th edition (Springer, Berlin, 2002).
- N. J. Giordano: „*Computational Physics*“ (Addison Wesley, Reading MA, 1996).
- J. M. Thijssen: “*Computational Physics*”, (Cambridge University Press, Cambridge, 1999).

Book Series:

- „*Monte Carlo Method in Condensed Matter Physics*“, ed. K. Binder (Springer Series).
- „*Annual Reviews of Computational Physics*“, ed. D. Stauffer (World Scientific).
- „*Granada Lectures in Computational Physics*“, ed. J. Marro (Springer Series).
- „*Computer Simulations Studies in Condensed Matter Physics*“, ed. D. Landau (Springer Series).

Journals:

- Journal of Computational Physics (Elsevier).
- Computer Physics Communications (Elsevier).
- International Journal of Modern Physics C (World Scientific).

Conferences:

- Annual Conference on Computational Physics (CCP): This year the CCP was held in Brussels (Sept. 5th-8th 2007). Next year the CCP 2008 will be in Brazil.

Chapter 1

Random numbers

Random numbers (RN) are an important tool for scientific simulations. As we will see during this course they are used in many different applications. Here is a list of examples:

- Simulate random events and experimental fluctuations, for example radioactive decay.
- Complement the lack of detailed knowledge (e.g. traffic or stock market simulations).
- Consider many degrees of freedom (e.g. Brownian motion, random walkers).
- Test the stability of a system with respect to perturbations.
- Random sampling.
- Define the temperature.

Special literature about random numbers is given at the end of this chapter.

1.1 Definition of random numbers

Random numbers are a sequence of numbers in random or uncorrelated order. In particular, the probability that a given number occurs next in the sequence is always the same. Physical systems can produce random events for example in electronic circuits ("electronic flicker noise") or in systems where quantum effects play an important role such as for example radioactive decay or the photon emission from a semiconductor.

The algorithmic creation of random numbers is a bit problematic, since the computer is completely deterministic but the sequence should be non-deterministic. Because of that one considers the creation of pseudo-random numbers, which are calculated with a deterministic algorithm, but in such a way that the numbers

are almost homogeneously, randomly distributed. These numbers should follow a well-defined distribution and should have long periods. Additionally they should be calculated quickly and in a reproducible way.

A very important tool in the creation of pseudo-random numbers is the modulo-operator, which determines the remainder of a division of one integer number with another one.

Given two numbers a the dividend and n the divisor then $a \,\mathbf{modulo}\, n$ or $a \,\mathbf{mod}\, n$ is the remainder on division of a by n. The mathematical definition of this operator is: let $q \in \mathbb{Z}$. Then we can write a as

$$a = nq + r \tag{1.1}$$

with $0 \leq r \leq |n|$, where r is the the remainder. The **mod**-operator is useful since from big numbers you can obtain big and small numbers equivalently.

The pseudo-random generators (RNG) can be divided into two classes: the multiplicative and the additive generators. The multiplicative ones are simpler and faster to program and execute but do not produce very good sequences. The additive ones are more difficult to implement and take longer to run but produce much better random sequences.

1.2 Congruential RNG (multiplicative)

The simplest form of a congruential RNG was proposed by Lehmer 1948. The algorithm is based on the properties of the **mod**-operator. Let's assume that we choose two integer numbers c and p and a seed value x_0 with $c, p, x_0 \in \mathbb{Z}$. Then we create the sequence $x_i \in \mathbb{Z}$, $i \in \mathbb{N}$ iteratively by

$$x_i = (cx_{i-1}) \,\mathbf{mod}\, p \tag{1.2}$$

This creates random numbers in the interval $[0, p-1)$ [1]. In order to transform these random numbers to the interval $[0, 1)$ we simply divide by p

$$0 \leq z_i = \frac{x_i}{p} < 1 \tag{1.3}$$

with $z_i \in \mathbb{R}$ (actually $z_i \in \mathbb{Q}$).

Since all integers are smaller than p the sequence must repeat after at least $p-1$ iterations. Thus, the *maximal period* of this RNG is $p-1$. In the special case of the seed value $x_0 = 0$ the sequence sits on a fixed point 0 and therefore cannot be used.

[1] Throughout the manuscript we will adopt the notation that square brackets [ ] in intervals are equivalent to $\leq$ and $\geq$ and parentheses ( ) correspond to $<$ and $>$ respectively. Thus the interval $[0, 1]$ corresponds to $0 \leq x \leq 1$, $x \in \mathbb{R}$ and $(0, 1]$ means $0 < x \leq 1$, $x \in \mathbb{R}$.

In 1910 R. D. Carmichael proved that the maximal period can be obtained if p is a Mersenne prime number[2] and the smallest integer number for which the condition holds that

$$c^{p-1} \bmod p = 1. \tag{1.4}$$

Park and Miller presented in 1988 the following numbers, which produce a relatively long sequence of pseudo-random numbers, here in pseudo-C code:

```
const int p=2147483647;
const int c=16807;
int rnd=42;  // seed
rnd=(c*rnd)%p;
print rnd;
```

The number p is of course a Mersenne prime with the maximal length of an integer (32 bit): $2^{31} - 1$.

As was shown in the course the distribution of pseudo-random numbers calculated with a congruential RNG can be represented in a plot of consecutive random numbers (x_i, x_{i+1}), where there will form some patterns, mostly lines, depending on the chosen parameters. It is of course also possible to do this kind of visualization for three consecutive numbers (x_i, x_{i+1}, x_{i+2}) in 3D. There even exists a theorem, which quantifies the patterns observed. Let us call the normalized numbers of the pseudo-random sequence $\{z_i\} = \{x_i\}/p$, $i \in \mathbb{N}$. Let then $\pi_1 = (z_1, ..., z_n)$, $\pi_2 = (z_2, ..., z_{n+1})$, $\pi_3 = (z_3, ..., z_{n+3})$, ... be the points of the unit n-cube formed from n successive z_i.

Theorem 1 (Marsaglia, 1968)**.** *If $a_1, a_2, ..., a_n \in \mathbb{Z}$ is any choice of integers such that*

$$a_1 + a_2 c + a_3 c^2 + ... + a_n c^{n-1} \equiv 0 \bmod p,$$

then all of the points $\pi_1, \pi_2, ...$ will lie in the set of parallel hyperplanes defined by the equations

$$a_1 y_1 + a_2 y_2 + ... + a_n y_n = 0, \pm 1, \pm 2, ..., \quad y_i \in \mathbb{R},\ 1 \leq i \leq n$$

There are at most

$$|a_1| + |a_2| + ... + |a_n|$$

of these hyperplanes, which intersect the unit n-cube and there is always a choice of $a_1, ..., a_n$ such that all of the points fall in fewer than $(n!p)^{1/n}$ hyperplanes.

Proof. (Abbreviated) The theorem is proved in four steps:
Step 1: If

$$a_1 + a_2 c + a_3 c^2 + ... + a_n c^{n-1} \equiv 0 \bmod p$$

[2] A *Mersenne number* is defined as $M_n = 2^n - 1$. If this number is also prime, it is called *Mersenne prime*.

then one can prove that

$$a_1 z_i + a_2 z_{i+1} + ... + a_n z_{i+n-1}$$

is an integer for every i and thus

Step 2: The point $\pi_i = (z_i, z_{i+1}, .., z_{i+n-1})$ must lie in one of the hyperplanes

$$a_1 y_1 + a_2 y_2 + ... + a_n y_n = 0, \pm 1, \pm 2, ..., \;\; y_i \in \mathbb{R},\, 1 \leq i \leq n.$$

Step 3: The number of hyperplanes of the above type, which intersect the unit n-cube is at most

$$|a_1| + |a_2| + ... + |a_n|,$$

and

Step 4: For every multiplier c and modulus p there is a set of integers $a_1, ..., a_n$ (not all zero) such that

$$a_1 + a_2 c + a_3 c^2 + ... + a_n c^{n-1} \equiv 0 \,\mathbf{mod}\, p$$

and

$$|a_1| + |a_2| + ... + |a_n| \leq (n! p)^{1/n}.$$

This is of course only the outline of the proof. The exact details can be read in G. Marsaglia, Proc. Nat. Sci. U.S.A. **61**, 25 (1968). The interested student can obtain a copy of this short article from the author of the manuscript (hsinger@ethz.ch).

□

In a very similar way it is possible to show that for congruential RNGs the distance between the planes must be larger than

$$\sqrt{\frac{p}{n}} \tag{1.5}$$

1.3 Lagged Fibonacci RNG (additive)

A more complicated version of a RNG is the Lagged Fibonacci algorithm proposed by Tausworth 1965. The idea with this generator is to replace the multiplicative part with a generalized Fibonacci-sequence. Let us assume that we have b random bits x_i. Then we can write the next bit in the sequence as

$$x_i = (\sum_{j \in \Im} x_{i-j}) \, \mathbf{mod} \, 2 \tag{1.6}$$

with $\Im \subset [1, .., b]$. As an example a two element lagged Fibonacci generator is discussed. Let now $a, b \in \mathbb{N}$ with $b < a$. Then the sequence is recursively defined as

$$x_{i+1} = (x_{i-a} + x_{i-b}) \mathbf{mod} \, 2. \tag{1.7}$$

Thus in order to actually start producing new pseudo-random bits an initial seed sequence of at least a bits must exists already. The usual procedure is to use a congruential generator for this seed sequence.

There are conditions for the choice of the numbers a and b as in the case of congruential generators. This time a, b must satisfy the Zierler-Trinomial condition that

$$T_{a,b}(z) = 1 + z^a + z^b \tag{1.8}$$

cannot be factorized in subpolynomials, where z is a binary number. The number a is chosen up to 100000 and it can be shown that the maximal period is $2^a - 1$, which is much larger than for congruential generators. The smallest numbers satisfying the Zierler conditions are $(a, b) = (250, 103)$. The generator is named after the discoverers of the numbers Kirkpatrick and Stoll (1981). Here is a list of known pairs

(a, b)			
(250	,	103)	**Kirkpatrick – Stoll (1981)**
(4187	,	1689)	**J.R. Heringa et al. (1992)**
(132049	,	54454)	
(6972592	,	3037958)	**R.P. Brent et al. (2003)**

1.4 How good is a RNG?

There are many possibilities to test how random the sequence of an RNG really is. Here we present a short list of possible tests of a sequence $\{s_i\}$, $i \in \mathbb{N}$:

1. Square test: the plot of two consecutive numbers $(s_i, s_{i+1}) \, \forall i$ should be distributed homogeneously. Any sign of lines or clustering shows the non-randomness and correlation of the sequence $\{s_i\}$.

2. Cube test: this test is similar to the square test, but this time the plot is three-dimensional with the tuples (s_i, s_{i+1}, s_{i+2}). Again the tuples should be distributed homogeneously.

3. The average value: the arithmetic mean of all the numbers in the sequence $\{s_i\}$ should correspond to the analytical mean value. Let us assume here that the numbers s_i are rescaled to be in the interval $s_i \in [0, 1)$. Then the arithmetic mean should be

$$\bar{s} = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} s_i = \frac{1}{2} \tag{1.9}$$

   Thus the more numbers are averaged the better 1/2 will be approximated.

4. Fluctuation of the mean value (χ^2-Test): The distribution around the mean value should behave like an Gaussian distribution.

5. Spectral analysis (Fourier analysis): By assuming the $\{s_i\}$ to be values of a function, it is possible to perform a Fourier transform by means of the Fast Fourier Transform (FFT). If the frequency distribution corresponds to white noise (uniform distribution) then the randomness is good, otherwise peaks will be seen (resonances).

6. Correlation test: Analysis of the correlation such as

$$< s_i * s_{i+d} > - < s_i^2 > \tag{1.10}$$

   for different d.

Of course this list is not complete. There are many other tests that can be used for testing the pseudo-random sequences.

Very famous are Marsaglia's "Diehard" tests for random numbers. These Diehard tests are a battery of statistical tests for measuring the quality of a set of random numbers. They were developed over many years and published for the first time by Marsaglia on a CD-ROM with random numbers in 1995. These tests are[3]:

- Birthday spacings: If random points are chosen in a large interval, then the spacing between the points should be asymptotically Poisson distributed. The name stems from the birthday paradox[4].

[3]The tests are no better or worse than the ones stated previously. They have become famous though, since the names of the tests use rather funny names.

[4]The birthday paradox states that the probability of two randomly chosen persons having the same birthday in a group of 23 (or more) people is more than 50%. In case of 57 or more people the probability is already more than 99%. Finally for at least 366 people the probability is exactly 100%. This is not paradoxical in a logical sense, it is called paradox nevertheless since naive intuition estimates the chance much lower than 50%.

- Overlapping permutations: When analyzing five consecutive random numbers then the 120 possible orderings should occur with statistically equal probability.
- Ranks of matrices: Some number of bits from some number of random numbers are formed to a matrix over {0,1}. Then the rank of this matrix is determined and the ranks are counted.
- Monkey test: Sequences of some number of bits are taken as words. Then the number of overlapping words in a stream is counted. The number of words not appearing should follow a known distribution. The name is based on the infinite monkey theorem[5].
- Parking lot test: Randomly place unit circles in a 100 x 100 square. If the circle overlaps an existing one, try again. After 12,000 tries, the number of successfully "parked" circles should follow a certain normal distribution.
- Minimum distance test: Find the minimum distance of 8000 randomly placed points in a 10000 x 10000 square. The square of this distance should be exponentially distributed with a certain mean.
- Random spheres test: put 4000 randomly chosen points in a cube of edge 1000. Now a sphere is placed on every point with a radius corresponding to the minimum distance to another point. The smallest sphere's volume should then be exponentially distributed.
- Squeeze test: 231 random float in $[0, 1)$ are multiplied until 1 is reached. After 100000 repetitions the number of floats needed to reach 1 should follow a certain distribution.
- Overlapping sums test: Sequences of 100 consecutive floats are summed up in a very long sequence of random floats in $[0, 1)$. The sums should be normally distributed with characteristic mean and standard deviation.
- Runs test: Ascending and descending runs in a long sequence of random floats in $[0, 1)$ are counted. The counts should follow a certain distribution.
- Craps test: 200000 games of craps[6] are played. The number of wins and the number of throws per game should follow a certain distribution.

[5]The infinite monkey theorem states that a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely (i.e. with probability 1) type a particular chosen text, such as the complete works of William Shakespeare.

[6]Casino dice game

1.5 Other distributions

Up to now we have considered only the uniform distribution of pseudo-random numbers. The congruential and lagged Fibonacci RNG produce numbers in $\mathbb{N}$. These can be easily mapped to the interval $[0, 1)$ or any other interval by simple shifts and multiplications. If the goal is, however, to produce random numbers, which are distributed according to a certain distribution, then the algorithms presented so far are not very well suited. There are however tricks, how to transform uniform pseudo-random numbers to other distributions. In essence there are two different ways how to perform this transformation. For a given distribution, for which an analytic description is known, it is under certain conditions possible to apply a mapping. For data, which does not follow a known distribution or for distributions, to which such mathematical transformation cannot be applied the so called rejection method is used. These methods are explained in the following sections.

1.5.1 Transformation methods of special distributions

For a certain class of distributions it is possible to create pseudo-random numbers from the uniformly distributed random numbers by finding a mathematical transformation. In particular the transformation method works nicely for the most common distributions: exponential, Poisson and normal. The transformation is rather straightforward, however, depending on the analytical description of the distribution not always feasible. Essentially the idea is to find the equivalence between area slices of the uniform distribution P_u and the distribution of interest. The uniform distribution is written as

$$P_u(z) = \begin{cases} 1 & \text{for } z \in [0,1] \\ 0 & \text{otherwise} \end{cases} \tag{1.11}$$

Let now the distribution of interest be $P(y)$. In comparing the areas of the integration we find

$$z = \int_0^y P(y')dy' = \int_0^z P_u(z')dz' \tag{1.12}$$

where z is a uniformly distributed random variable and y a random variable distributed according to the desired distribution. Let's rewrite the integral of $P(y)$ as $I_P(y)$ then we find $z = I_P(y)$ and therefore

$$y = I_P^{-1}(z). \tag{1.13}$$

This shows that a transformation between the two distributions can only be found under the conditions that

1. The integral $I_P(y) = \int_0^y P(y')dy'$ can be solved analytically in a closed form.

2. There exists an analytic inverse of $z = I_P(y)$ such that $y = I_P^{-1}(z)$.

Of course, these conditions can be overcome up to certain extent by precalculating/tabulating and inverting $I_P(y)$ numerically, if the integral is well-behaved i.e. no singularities. Then, with a slight overhead of looking up in the precalculated tables, it is possible to transform the uniform numbers numerically.

We will now demonstrate this method for the two most commonly used distributions: the Poisson distribution and the Gaussian distribution. As we will see for the Gaussian distribution there is much more work involved to create such a transformation.

The Poisson distribution

The Poisson distribution is defined as

$$P(y) = ke^{-\frac{y}{k}}. \tag{1.14}$$

By applying the area equality of eq. (1.12) we find

$$z = \int_0^y ke^{-\frac{y'}{k}}dy' = \int_0^z P_u(z')dz' \tag{1.15}$$

thus

$$z = -e^{-\frac{y'}{k}}|_0^y = 1 - e^{-\frac{y}{k}}. \tag{1.16}$$

Now, inverting the integral leads to

$$y = -k\ln(1-z). \tag{1.17}$$

The normal distribution

Analytical methods of generating normally distributed random number are very useful, since there are many applications and examples where such numbers are needed.

The Gaussian or normal distribution is written as

$$P(y) = \frac{1}{\sqrt{\pi\sigma}}e^{-\frac{y^2}{\sigma}} \tag{1.18}$$

Unfortunately only the limit for $y \to \infty$ can be solved analytically:

$$\int_0^\infty \frac{1}{\sqrt{\pi\sigma}}e^{-\frac{y'^2}{\sigma}}dy' = \frac{\sqrt{\pi}}{2}. \tag{1.19}$$

However Box and Muller[7] have introduced following elegant trick to circumvent this restriction. Let's assume we take two (uncorrelated) uniform random variables

[7] G. E. P. Box and Mervin E. Muller, A Note on the Generation of Random Normal Deviates, The Annals of Mathematical Statistics (1958), Vol. **29**, No. 2 pp. 610-611

z_1 and z_2. Of course we can apply now again the area equality of eq. (1.12) but this time we write it as a product of the two random variables

$$z_1 \cdot z_2 = \int_0^{y_1} \frac{1}{\sqrt{\pi\sigma}} e^{-\frac{y_1'^2}{\sigma}} dy_1' \cdot \int_0^{y_2} \frac{1}{\sqrt{\pi\sigma}} e^{-\frac{y_2'^2}{\sigma}} dy_2' = \int_0^{y_2} \int_0^{y_1} \frac{1}{\pi\sigma} e^{-\frac{y_1'^2+y_2'^2}{\sigma}} dy_1' dy_2'. \tag{1.20}$$

This integral can now be solved by transforming the variables y_1' and y_2' into polar coordinates:

$$r^2 = y_1^2 + y_2^2 \tag{1.21}$$

$$\tan\phi = \frac{y_1}{y_2} \tag{1.22}$$

with

$$dy_1' dy_2' = r' dr' d\phi' \tag{1.23}$$

Substituting in eq. (1.20) leads to

$$z_1 \cdot z_2 = \frac{1}{\pi\sigma} \int_0^{\phi} \int_0^{r} e^{-\frac{r'^2}{\sigma}} r' dr' d\phi' \tag{1.24}$$

$$= \frac{\phi}{\pi\sigma} \int_0^{r} e^{-\frac{r'^2}{\sigma}} r' dr' \tag{1.25}$$

$$= \frac{\phi}{\pi\sigma} \cdot \frac{\sigma}{2} \left(1 - e^{-\frac{r^2}{\sigma}}\right) \tag{1.26}$$

$$z_1 \cdot z_2 = \underbrace{\frac{1}{2\pi} \arctan\left(\frac{y_1}{y_2}\right)}_{\equiv z_1} \cdot \underbrace{\left(1 - e^{-\frac{y_1^2+y_2^2}{\sigma}}\right)}_{\equiv z_2} \tag{1.27}$$

By separating these two terms and associating them to z_1 and z_2 respectively it is possible to invert the functions such that

$$y_1^2 + y_2^2 = -\sigma \ln(1 - z_2) \tag{1.28}$$

$$\frac{y_1}{y_2} = \tan(2\pi z_1) = \frac{\sin(2\pi z_1)}{\cos(2\pi z_1)} \tag{1.29}$$

Solving these two coupled equations finally yields

$$y_1 = \sqrt{-\sigma \ln(1 - z_2)} \sin(2\pi z_1) \tag{1.30}$$

$$y_2 = \sqrt{-\sigma \ln(1 - z_2)} \cos(2\pi z_1) \tag{1.31}$$

Thus, using two uniformly distributed random numbers z_1 and z_2 one obtains through the Box-Muller transform two normally distributed random numbers y_1 and y_2.

1.5.2 The rejection method

As shown in subsection 1.5.1 there are two conditions that have two be satisfied in order to apply the transformation method: integrability and invertability. If either of these conditions is not satisfied, there exists no analytical method to obtain random numbers in this distribution. In particular, this is the case for numbers that have been obtained from experimental data or other sources, where no analytical description is available. One has to resort then to a numerical method to obtain arbitrarily distributed random numbers. This method is called the rejection method.

Let again $P(y)$ be the distribution of which we would like to obtain random numbers. A necessary condition for the rejection method to work is that $P(y)$ is well-behaved i.e. $P(y)$ is finite over the domain of interest $P(y) < A$ for $y \in [0, B]$, with $A, B \in \mathbb{R}$ and $A, B < \infty$. Then we define an upper bound to be the box with edge length B and A.

We now produce two pseudo-random variables z_1 and z_2 with $z_1, z_2 \in [0, 1)$. Then the point with the coordinates $(Bz_1, Az_2)^T$ surely lies in the defined box. If the point lies above the curve $P(y)$, i.e. $Az_2 > P(Bz_1)$ then the point is rejected, hence the name of the method. Otherwise $y = Bz_1$ is retained as a random number, which is distributed according to $P(y)$.

The method works in principle quite well, however certain issues have to be taken into consideration when using it.

- It is desirable to have a good guess for the upper bound. Obviously the better the guess, the less points are rejected. In the above description of the algorithm we have assumed a rectangular box. This is however not a necessary conditions. The bound can be any distribution for which random numbers are easily generated.

- While the method is sound, in practice it is often faster to invert $P(y)$ numerically as mentioned already in subsection 1.5.1.

- A method, which makes the rejection method faster but also more complicated is to use N boxes to cover $P(y)$ and define then individual box edge sizes A_i and $b_i = B_{i+1} - B_i$ for $1 \leq i \leq N$. Thus the approximation of $P(y)$ is much better (idea of Riemann-integral).

Literature

- Numerical Recipes

- D. E. Knuth: "The Art of Programming Vol. 2: Seminumerical Algorithms" (Addison-Wesley, Reading MA, 1997): Chapter 3.3.1

- J.E. Gentle, "Random number generation and Monte Carlo Methods", (Springer, Berlin, 2003).

Chapter 2

Percolation

Percolation in material science and chemistry describes the movement or filtering of fluids through porous media. The name stems originally from Latin and is now still used as a common word in Italian[1].

A very simple and basic model of such a process was first introduced by Broadbent and Hammersley in (Proc. Cambridge Phil. Soc. Vol. **53**, p.629 (1957)).

While the model was originally designed to model in an abstract way the fluid motion through e.g. a container filled with glass beads, it was found that the model has many other applications, and some quite interesting universal features of so called critical phenomena[2]. Among these applications are

- General porous media: for example used in the oil industry and as a model for the pollution of soils.
- Sol-Gel transitions (will be discussed in the next section).
- Mixtures of conductors and insulators: find the point when a conductor suddenly does not conduct the current any longer.
- Spreading of fires for example in forest.
- Spreading of epidemics or computer virii.
- Crash of stock markets (D. Sornette, Professor at ETH).
- Landslide election victories (S. Galam).
- Recognition of antigens by T-cells (Perelson).

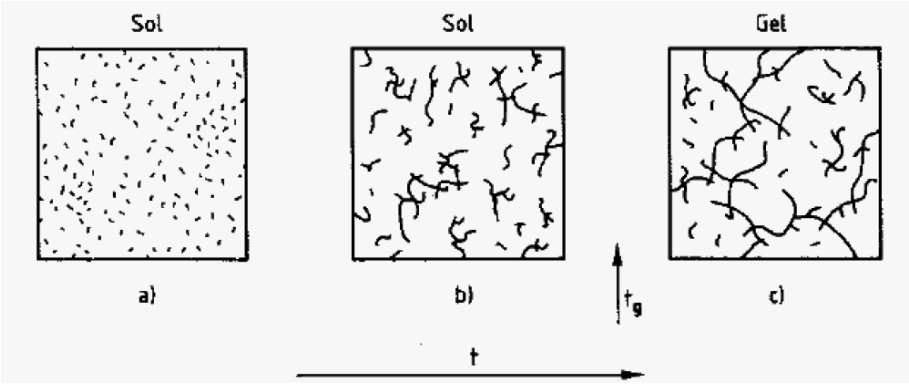

Figure 2.1: Sol-Gel transition: a) monomers are dispersed in the liquid (sol). b) When the liquid is cooled down the monomers start polymerizing and grow. c) Once a huge macromolecule has formed, which spans through the whole container, the Gel transition occurs.

2.1 The Sol-Gel Transition

The formation of Gelatin is quite astonishing from the chemical and physical point of view. The gelatin is a fluid in the beginning containing many small monomers (emulsion), which is called sol. When you place this liquid in the fridge, after a certain time is suddenly becomes a "solid" gel. What happens is this: upon cooling the monomers start polymerizing and the polymers start to grow. At a given time there is one molecule which has grown so big, that it spans from one end of the container to the other end (a true macromolecule). That is when the percolation transition occurs. This is represented schematically in Fig. 2.1. It is possible to perform experiments on this behavior by measuring the shear modulus and the viscosity as a function of the time. One observes that after a characteristic "Gel time" T_G suddenly the shear modulus increases from 0 to a finite value. At the same time measuring the viscosity one finds that it increases and essentially becomes infinite at T_G. Experimental data is shown in Fig. 2.2.

2.2 The Percolation model

The abstract model of this process is extremely simple. Let us assume that we create a square lattice with edge length $L = 16$ such that every cell can either be occupied or empty. The initial configuration is that all fields are empty. Now we fill each cell with the probability p, that is, for each cell we create a random number $z \in [0, 1)$ and compare it to p. If $z < p$ then the cell will be marked as

[1] *ita*: percolare:1 Passare attraverso. ~ filtrare. 2 Far filtrare. (1 pass through ~ filter, 2 make sth filter).

[2] Critical phenomena is the collective name associated with the physics of critical points.

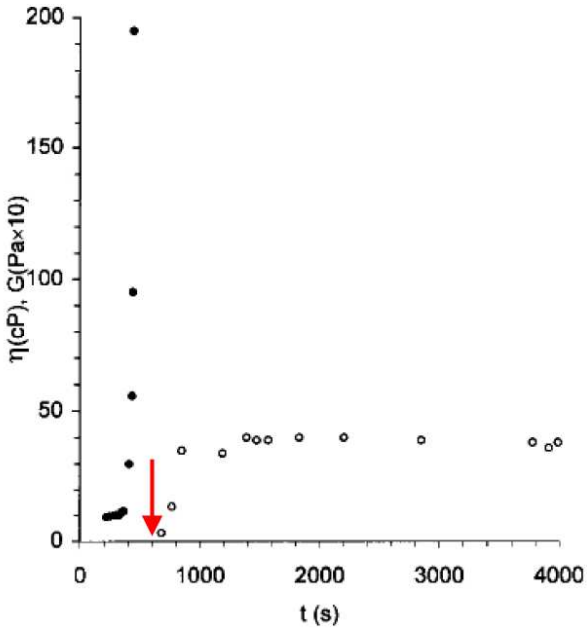

Figure 2.2: Viscosity (filled circles) and shear modulus (open circles) as a function of time. At the gelation time T_G the viscosity diverges to infinity and the shear modulus, previously 0, increases to a finite value.

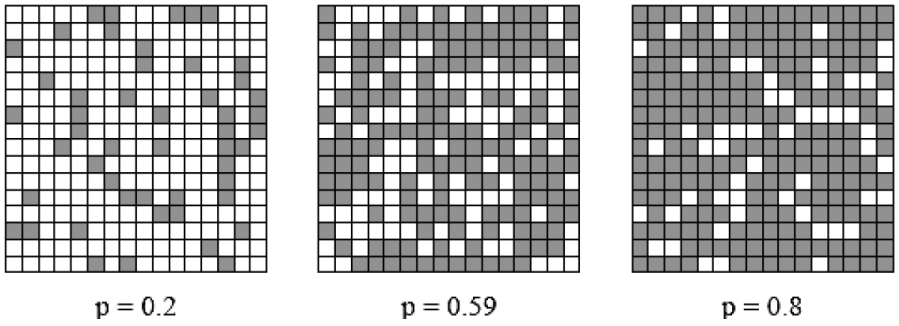

Figure 2.3: Percolation on a square lattice with $L = 16$ for different values of p. For $p = 0.2$ most cells are empty. For $p = 0.8$ most cells are occupied. There is a critical probability p where for the first time fully connected cells span from one end to the other $p = 0.592...$

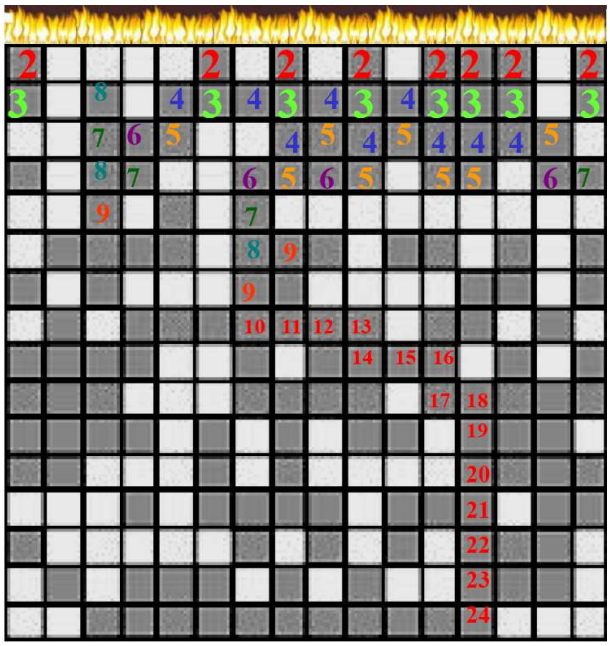

Figure 2.4: The burning method: at the top line a fire is started. At every iteration the burning trees start burning their occupied neighbor cells. The algorithm ends if all trees are burnt or the burning method has reached the bottom line. In that case the label indicates the shortest path through the percolating cluster from the top to the bottom line.

occupied otherwise the cell remains empty. This can be done for different values of p and qualitatively different results are obtained (Fig 2.3). As can be seen for small values of p most of the cells are empty, whereas for big values of p most cells are occupied. There exists, however, a critical probability $p = p_c = 0.592..$ when for the first time a fully connected amount of cells spans from one end of the box to the other end (this is also called a percolating cluster). The critical probability is sometimes called the percolation threshold.

The burning method

While above observation of a critical probability was rather general and qualitative, it would be desirable to have a tool at hand, which provides exact information of a configuration such as in Fig. 2.3, whether of not the full cells span over the whole box. The method, which provides this information is called burning method. It not only provides a yes/no information but also calculates the minimal path length to go e.g. from the top to the bottom. The name of the method stems from its direct application. Let's assume that the full cells actually represent trees and a fire is started at the top line of the box. Now every tree in the top row burns. If there are any neighboring trees that don't burn, then in the next iteration step

they will start to burn as well. This iterative process will continue as long as there are neighboring trees and/or the fire has reached the bottom line. This number of iterations defines the minimum shortest path of the percolating cluster.

The algorithm is as follows:

1. Label all occupied cells in the top line with the marker t=2.
2. Iteration step t+1:
   (a) Go through all the cells and find the cells, which have label t.
   (b) For each of the found t-label cells do
      i. Check if any direct neighbor (North, East, South, West) is occupied and not burning (label is 1).
      ii. Set the found neighbors to label t+1.
3. Repeat step 2 (with t=t+1) until either there are no neighbors to burn any more or the bottom line has been reached - in the latter case the latest label-1 defines the shortest path.

A graphical representation of the burning algorithm is given in Fig. 2.4.

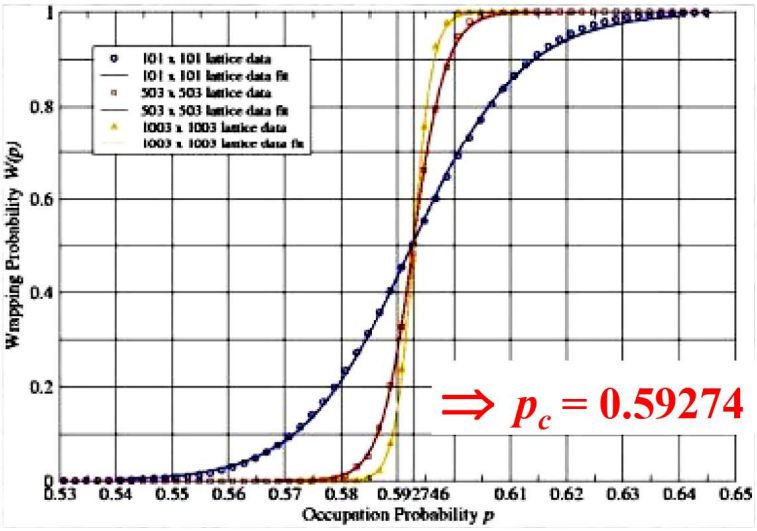

Figure 2.5: Percolation threshold probability as a function of the occupation probability for different lattice sizes.

The percolation threshold

While performing simulations with the percolation model one observes a percolation threshold with increasing occupation probability p, where for the first time a spanning cluster is produced. In order to pinpoint this transition different simulations can be performed. Obviously there are not that many parameters to choose, essentially only one, which can be analysed as a function of p, namely the lattice size. Assuming a square lattice one can now calculate the average value or probability to produce a spanning cluster for a given p. Since it is a random process the transition is not completly sharp. Results of many such simulations are given in Fig. 2.5. It can easily be seen, that for increasing lattice sizes the transition becomes sharper. Extrapolating to infinite cluster sizes one would find a step function behavior. From the series of simulations with increasing lattice sizes it is also possible to determine the exact position of the threshold, namely $p_c = 0.59274....$

The percolation threshold is a characteristic value for a given type of lattice. For different lattices different percolation threshold are found. Fig. 2.6 shows values for some lattice types.

Percolation can be performed not only on lattice sites, but also on the corresponding bonds. Obviously this changes the number of neighbors in a given lattice and therefore also the percolation threshold.

By looking at the values for different lattices in Fig. 2.6 it becomes clear that different geometries have significantly different thresholds. For example the honeycomb lattice has the highest 2D threshold whereas the trinagular lattice, which actually can be calculated analytically (denoted by a *) has the lowest one.

lattice	site	bond
cubic (body-centered)	0.246	0.1803
cubic (face-centered)	0.198	0.119
cubic (simple)	0.3116	0.2488
diamond	0.43	0.388
honeycomb	0.6962	0.65271*
4-hypercubic	0.197	0.1601
5-hypercubic	0.141	0.1182
6-hypercubic	0.107	0.0942
7-hypercubic	0.089	0.0787
square	0.592746	0.50000*
triangular	0.50000*	0.34729*

Figure 2.6: Percolation threshold for different lattices for site percolation and bond percolation. The numbers with the star (*) can be calculated analytically.

Intuitively it is clear that lattice geometries with many neighbors favor a lower threshold (triangular with 6 neighbors as opposed to the honeycomb with only 3 neighbors) since there are many more ways to form a spanning cluster with many neighbors.

Please note that if not otherwise stated we will only use 'sites' for the percolation. This implies in general also the behavior for 'bonds'.

The order parameter

We have determined the percolation threshold in the previous section and stated that for infinitely big system the transition indeed occurs exactly at p_c. Let us consider now probabilities $p > p_c$, where we will always find a spanning cluster. A natural question arises, how many of the sites belong to the biggest cluster. Or more exactly we can define the fraction of sites, which belong to the biggest cluster $P(p)$ as a function of the occupation probability p. We call this measure the order parameter. This is shown in Fig. 2.7. Obviously for $p < p_c$ the order parameter is 0, since there are no spanning clusters. For $p \geq p_c$ the order parameter is increasing. Analysing the functional behavior of the order parameter one finds, that $P(p)$ behaves like a power law in the region close to the percolation threshold

$$P(p) \propto (p_c - p)^{\beta} \tag{2.1}$$

Such a behavior is called "universal criticality". This concept will be discussed in a much more details in later chapters. The concepts of universality of critical phenomena is a framework or theory which describes many different systems, which

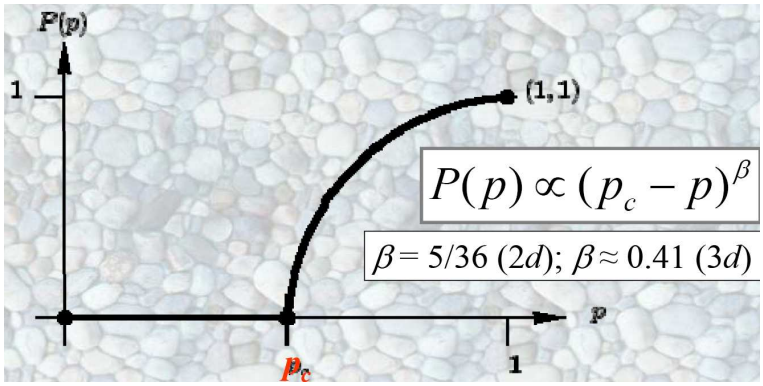

Figure 2.7: Order parameter $P(p)$ as a function of the occupation probability p. The system shows a "critical" behavior.

at first site have very little in common, such as phase transitions, magnetization or percolation.

The cluster size distribution

After having investigated the percolation threshold and the fraction of sites in the biggest cluster the natural extension would be to know how the different clusters are distributed. A tool similar to the Burning algorithm would be necessary to achieve such a task and to identify all the different cluster. Indeed there are several algorithms, which are able to do this. The most popular and for this purpose most efficient one is the Hoshen-Kopelman algorithm, which was developed in 1976. Let us denote the lattice as a matrix N_{ij}, which can have values of 0 if the site is not occupied and 1 if the site is occupied and let k be the running variable. Additionally an array M_k is introduced, which counts the amount of sites belonging to a given cluster numbered by k. We start the algorithm by setting $k = 2$ (since 0 and 1 are taken already) and searching for the first occupied site in N_{ij}. Then, of course, we add this site to the array $M_{k=2} = 1$.

Now we start looping over all lattice sites N_{ij} and try to detect whether an occupied site belongs to an already known cluster or a new one. The criterion is rather simple: if a site is occupied and the top and left neighbors are empty then we have found a new cluster and we set $k \leftarrow k+1$ and $N_{ij} = k$. Then we continue the loop.

If one of the sites (top or left) has the value k_0 then we add to the array $M_{k_0} \leftarrow M_{k_0}+1$. On the other hand if both are occupied with k_1 and k_2 respectively - meaning that they are connected - then we choose one of them (e.g. k_1) and set $N_{ij} \leftarrow k_1$ and add to the array $M_{k_1} \leftarrow M_{k_1} + M_{k_2} + 1$. Also we have to set $M_{k_2} \leftarrow -k_1$. Thus we mark the M_{k_2} not to be a counter for the amount of sites by making it negative, since now all the points of cluster k_2 are added to the cluster k_1 and we say that in the future, if ever a site belonging to k_2 is found

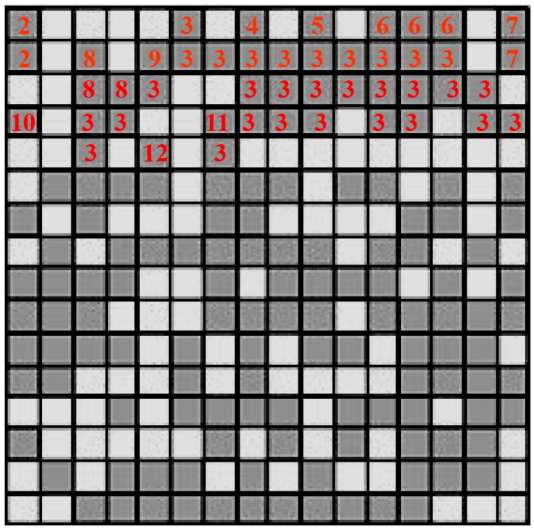

Figure 2.8: The Hoshen-Kopelman algorithm applied to a percolation cluster. The red numbers denote the running cluster variable k, however without the negative links as described in the text.

then we would want to add it to the cluster k_1, since they are connected. Thus $M_{k_2} \leftarrow -k_1$says that this is not a valid cluster any more (negative) and refers to the actual valid cluster of k_1 (link).

Now, if ever a site with negative M_k is encountered, then by following the links recursively, the actual cluster is found. Thus we set $k \leftarrow -M_k$ as long as M_k is negative. The recursive condition is bound to stop at the cluster, which really counts the amount of sites, since this number evidently is positive.

When all the sites N_{ij} have been visited the algorithm ends up with a number k_{max} of clusters. Now the only thing left to do is construct a histogram n of the different cluster sizes. This is done by looping through all the clusters $k \leftarrow 2..k_{max}$ while excluding the M_k, which are negative.

Written as an algorithm the Hoshen-Kopelmann algorithm looks like

1. $k \leftarrow 2$, $M_k \leftarrow 1$
2. for all i, j of N_{ij}
   (a) if top and left are empty $k \leftarrow k + 1$, $N_{ij} \leftarrow k$, $M_k \leftarrow 1$
   (b) if one is occupied with k_0 then $N_{ij} \leftarrow k_0$, $M_{k_0} \leftarrow M_{k_0} + 1$
   (c) if both are occupied with $k_1$and k_2 then choose one, e.g. k_1 and $N_{ij} \leftarrow k_1$, $M_{k_1} \leftarrow M_{k_1} + M_{k_2} + 1$, $M_{k_2} \leftarrow -k_1$
   (d) if any k has negative M_k: while $M_k < 0$ do $k \leftarrow -M_k$
3. for $k \leftarrow 2..k_{max}$ do

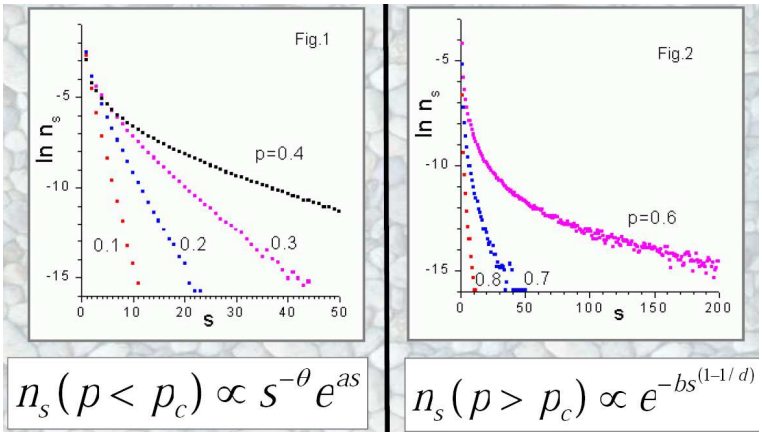

Figure 2.9: Cluster size distribution for $p < p_c$ (left) and $p > p_c$ (right).

(a) if $M_k > 0$ then $n(M_k) \leftarrow n(M_k) + 1$

A visualization of this algorithm is given in Fig. 2.8. The algorithm is very efficient since it scales linearly with the amount of lattice sites.

Once we have identified all the clusters in the system and calculated the histogram we are now able to analyse the results. In Fig. 2.9 plots are shown for different p. As can be seen the functional dependence is different for $p < p_c$ and $p > p_c$. After careful analysis it is possible to find following distributions:

$$n_p(s) \propto \begin{cases} s^{-\theta} e^{as} & p < p_c \\ e^{-bs^{1-1/d}} & p > p_c \end{cases} \tag{2.2}$$

For $p < p_c$ $n_p(s)$ is a power law multiplied by an exponential. On the other hand for $p > p_c$ the distribution behaves like an exponential decay with an argument that is stretched with a power $s^{1-1/d}$. Since the descriptions differ there has to be a transition from one distribution to the other. As can be seen in Fig. 2.10 this transition occurs at $p = p_c$ where a power law distribution

$$n_{p_c}(s) = s^{-\tau} \tag{2.3}$$

is found.

The exponent τ is calculated to be 187/91 in 2D and 2.18.. in 3D. One can also calculate the bounds for τ: $2 \leq \tau \leq 5/2$.

It is now also possible to observe how the distributions for different p can be rescaled to one single universal distribution. By plotting the ratio $n_p(s)/n_{p_c}(s)$ versus $(p - p_c)s^{\sigma}$ as in Fig. 2.11 one obtains the universal curve, which can be described as

$$n_p(s) = s^{-\tau} \Re_{\pm}[(p - p_c)s^{\sigma}] \tag{2.4}$$

with $\Re_{\pm}$ scaling functions, which are defined for $+ : p > p_c$ and $- : p < p_c$.

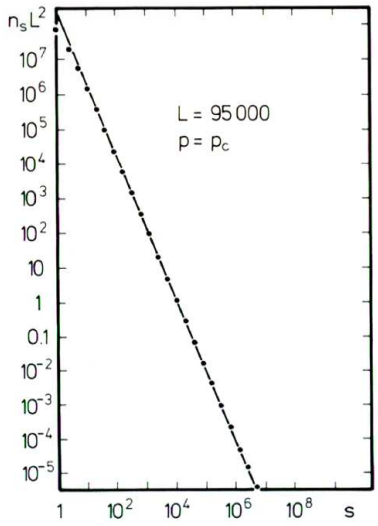

Figure 2.10: Cluster size distribution at $p = p_c$. The distribution is a power law.

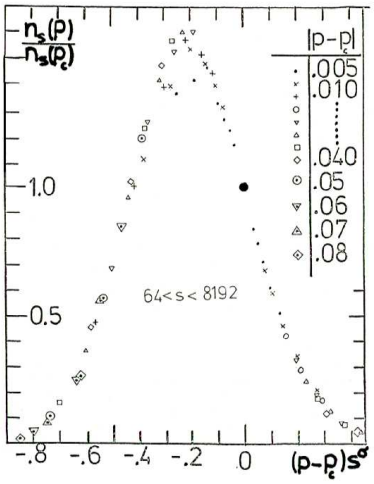

Figure 2.11: Scaling behavior of the percolation cluster size distribution

Table 2. Percolation exponents for $d = 2, 3, 4, 5, 6 - \varepsilon$ and in the Bethe lattice together with the page number defining the exponent. Rational numbers give (presumably) exact results, whereas those with a decimal fraction are numerical estimates.

Exponent	$d=2$	$d=3$	$d=4$	$d=5$	$d=6-\varepsilon$	Bethe	Page
α	$-2/3$	$-0{\cdot}62$	$-0{\cdot}72$	$-0{\cdot}86$	$-1+\varepsilon/7$	-1	39
β	5/36	0·41	0·64	0·84	$1-\varepsilon/7$	1	37
γ	43/18	1·80	1·44	1·18	$1+\varepsilon/7$	1	37
ν	4/3	0·88	0·68	0·57	$\frac{1}{2}+5\varepsilon/84$	1/2	60
σ	36/91	0·45	0·48	0·49	$\frac{1}{2}+O(\varepsilon^2)$	1/2	35
τ	187/91	2·18	2·31	2·41	$\frac{5}{2}-3\varepsilon/14$	5/2	33
$D(p=p_c)$	91/48	2·53	3·06	3·54	$4-10\varepsilon/21$	4	10
$D(p<p_c)$	1·56	2	12/5	2·8	–	4	62
$D(p>p_c)$	2	3	4	5	–	4	62
$\xi(p<p_c)$	1	1	1	1	–	1	56
$\xi(p>p_c)$	1/2	2/3	3/4	4/5	–	1	56
$\theta(p<p_c)$	1	3/2	1·9	2·2	–	5/2	54
$\theta(p>p_c)$	5/4	$-1/9$	1/8	$-449/450$	–	5/2	54
$f_{\max}$	5·0	1·6	1·4	1·1		1	42
μ	1·30	2·0	2·4	2·7	$3-5\varepsilon/21$	3	91
s	1·30	0·73	0·4	$0{\cdot}1_5$		0	93
D_B	1·6	$1{\cdot}7_4$	1·9	2·0	$2+\varepsilon/21$	2	95
$D_{\min}(p=p_c)$	1·13	1·34	1·5	1·8	$2-\varepsilon/6$	2	97
$D_{\min}(p<p_c)$	1·17	1·36	1·5		–	2	98
$D_{\max}(p=p_c)$	1·4	1·6	1·7	1·9	$2-\varepsilon/42$	2	97

For the exponents at p_c, the Bethe lattice values are exact at $d \geqslant 6$. A dash means that 6 is not the upper critical dimension for the ε-expansion.

Figure 2.12: Critical exponents for 2 to 6 dimensions.

Again a power law is found when calculating the second moment[3] of the distribution $n(s)$:

$$\chi = < \sum_s{}' s^2 n(s) > \tag{2.5}$$

where the ' denotes that the largest cluster has been excluded. Then one finds

$$\chi \propto C_{\pm} |p - p_c|^{-\gamma} \tag{2.6}$$

with $\gamma = 43/18 \approx 2.39$ in 2D and $\gamma = 1.8$ in 3D.

Interestingly it can be shown that all the scaling exponents found so far are related by

$$\gamma = \frac{3 - \tau}{\sigma} \tag{2.7}$$

These exponents (and some others, which were not discussed here) can be, of course, extracted for different lattice types and different dimensions. Fig. 2.12 gives an overview of different exponents measured for a regular (square) lattice up to 6 dimensions.

The shortest path

In this chapter we have presented ways to calculate whether or not there exists a spanning cluster by means of the Burning-algorithm and we have determined

[3]The n-th moment of a distribution $P(x)$ is defined as

$$\mu_n = \int x^n P(x) dx$$

therefore the 0-th moment of a normalized distribution is simply $\mu_0 = 1$ and the 1-st moment is $\mu_1 = \int x P(x) dx = E(x)$ the expectation value. Accordingly μ_2 is the standard deviation of the distribution around the expectation value.

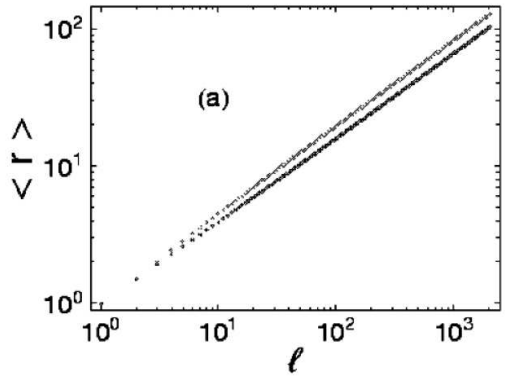

Figure 2.13: Shortest path $t_s =< r >$ as a function of the system size for 4-dimensional site (upper) and bond (lower) percolation.

and analysed the cluster size distribution with the Hoshen-Kopelman-algorithm. One unanswered question, however, is how the shortest path of a spanning cluster behaves as a function of the system size. Performing simulations yields to

$$t^s \propto L^{d_{min}} \tag{2.8}$$

thus, also a power law. The exponent d_{min} is dependent on the dimension

	2D	3D	4D
d_{min}	1.13	1.33	1.61

Of course it is possible to determine d_{min} for site percolation and bond percolation and for different lattice types. As an example Fig. 2.13 shows the site and bond percolation for a 4-dimensional system (square lattice), which was calculated by Ziff in 2001.

Literature

- D. Stauffer: „Introduction to Percolation Theory“ (Taylor and Francis, 1985).
- D. Stauffer and A. Aharony: „Introduction to Percolation Theory, Revised Second Edition“ (Taylor and Francis, 1992).
- M. Sahimi: „Applications of Percolation Theory“ (Taylor and Francis, 1994).
- G. Grimmett: „Percolation“ (Springer, 1989).
- B. Bollobas and O.Riordan: „Percolation“ (Cambridge Univ. Press, 2006).

Chapter 3

The fractal dimension

The fractal dimension is a concept, which has been introduced in the field of fractal geometry (coined by B. Mandelbrot). The underlying idea is to find a measure to describe how well a given (fractal) object fills a certain space.

The Sierpinski-Triangle and experiments

The Sierpinski-triangle[1] is a mathematical object, which is constructed by an iterative application of a simple operation. The first step is to take a triangle as in Fig 3.1. This triangle is then subdivided into 4 sub-triangles. The center triangle is discarded leaving a hole. Now in the next iteration each of the three remaining triangles is again subdivided and the respective central triangle removed. The object becomes a fractal for the limit of infinite iterations.

Now in order to calculate how well the Sierpinski triangle fills the 2D Euclidean space we define the dimension as the number of self-similar objects $N(l)$ divided by the length scale l at which the object is considered. Thus we find

$$D = \frac{\log N(l)}{\log(1/l)}. \tag{3.1}$$

[1]Invented by the Polish mathematician Wacław Franciszek Sierpiński (1882-1969)

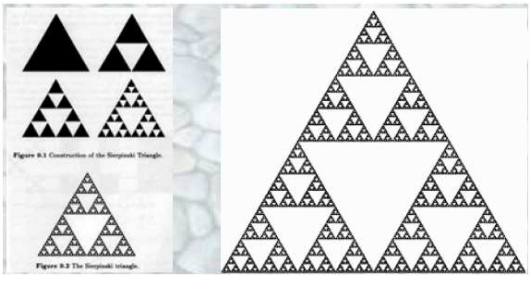

Figure 3.1: The Sierpinski triangle - a self-similar mathematical object, which is created iteratively.

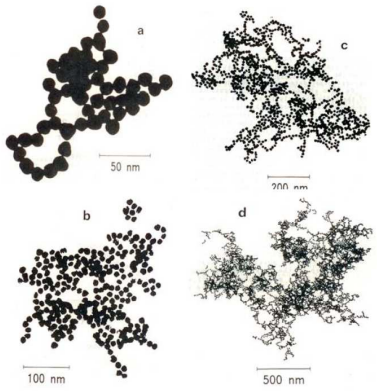

Figure 3.2: The structure of gold colloids at different scales.

As an example we consider the two dimensional unit square and divide each axis in two intervals: we have thus 4 subsquares of a length scale of $l = 1/2$ and we find $D = 2$. Similarly if we set $l = 1/10$ we find $N(l) = 100$ and so $D = 2$. So far, this is not really surprising. For a fractal object however this is not enough, since an object becomes only fractal for an infinite amount of iterations, therefore taking the limit

$$D_f = \lim_{l \to 0} \frac{\log N(l)}{\log(1/l)} \tag{3.2}$$

For the Sierpinski triangle we find that each iteration yields to 3 new subtriangles and the length scale is halfed. Hence, combining this as a function of the number of iterations k we find

$$D_f = \lim_{k \to \infty} \frac{\log 3^k}{\log 2^k} = \frac{\log 3}{\log 2} = 1.585... \tag{3.3}$$

Thus inversly one can write that the volume of the object scales as

$$V = L^{D_f} \tag{3.4}$$

with L the length of the object. This is indeed mathematically similar to the scaling behavior in the percolation system of the previous chapter.

The idea of calculating how well the space if filled is not just a mathematical fancy, but there are a number of applications in physics. For example D. Weitz has investigated the space filling behavior of gold colloids in 1984 (see Fig. 3.2). Calculating the fractal dimension of these clusters he found a value of $D_f = 1.7$.

In the following sections we will discuss methods how to determine the fractal dimension for experimental of simulated data when there is no simple way of analytic determination as with the Sierpinski triangle.

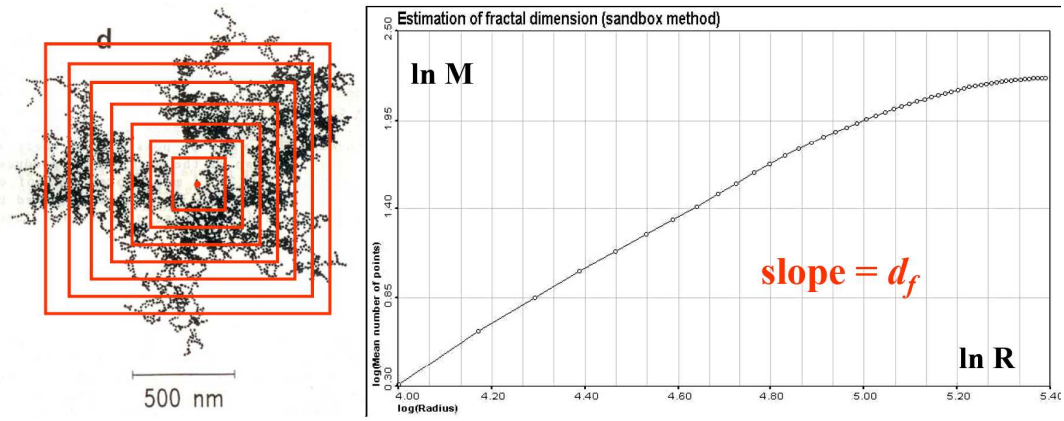

Figure 3.3: The sandbox method of determining the fractal dimension (left) and the results of the analysis (right).

The sandbox method

The sandbox approach was first presented by Forrest and Witten 1979. The key idea is to measure from a chosen point in the structure how many other points lie in a box of a given size. By performing measurements for increasing box sizes one finds a distribution as in Fig. 3.3. It can be shown that the slope of the measured data in a log-log plot roughly corresponds to the fractal dimension D_f. Obviously it is not possible to make the sand box too small, since no other points would be entrapped. Similarly for very big box sizes the whole structure would be entrapped. Thus one expects to find lower and higher cutoffs. In order to determine the slope one has to carefully choose, which regime to take in order not to be in these cutoff regions. An experimental or simulated object should be called (partially) fractal *only* when it shows more than a given amount of decades of linear behavior (usually more than 3).

Chapter 6

Solving equations

6.1 One-dimensional case

The problem of finding a root of an equation can be written as

$$f(x) = 0 \tag{6.1}$$

which is equivalent to the optimization problem of finding an extremum of $F(x)$ given by

$$\frac{d}{dx}F(x) = 0 \tag{6.2}$$

This function f can be a scalar function but it can be also a vector function of a vector: $\vec{f}(\vec{x})$. Let's first look at the one-dimensional case of a scalar function.

6.1.1 Newton method

The Newton method needs a start value x_0 and uses the first two terms of the Taylor expansion to linearize the function $f(x)$ around this start value:

$$f(x_0) + (x - x_0)f'(x_0) = 0 \tag{6.3}$$

which leads directly to the formulation of the iteration:

$$\begin{aligned}
(6.3) \Rightarrow x - x_0 &= \frac{f(x_0)}{f'(x_0)} \\
\Rightarrow x &= x_0 - \frac{f(x_0)}{f'(x_0)}, \quad \text{now consider } x \text{ as the next value } x_{n+1} \\
x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \qquad (6.4)
\end{aligned}$$

The Newton method converges extremly fast but the start value has to be relatively close to the exact value, otherwise it won't converge.

6.1.2 Secant method

If we don't know the derivative analytically, we have to determine it numerically (more about this in section 8.2):

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Insert this into (6.4) and you get the secant method:

$$x_{n+1} = x_n - (x_n - x_{n-1})\frac{f(x_n)}{f(x_n) - f(x_{n-1})} \tag{6.5}$$

In this method you need two different start values x_0 and x_1. The start values don't have to be as close to the solution as in the Newton method but the Secant method doesn't converge as fast as the Newton method.

6.1.3 Bisection method

Another simple method to solve an equation for its root is the Bisection method. It doesn't need the derivative too.

[1.]

1. Take two starting values x_0 and x_1 with $f(x_0) < 0$ and $f(x_1) > 0$.
2. Calculate the mid-point of the two values: $x_m = (x_0 + x_1)/2$.
3. If
$$sign(f(x_m)) = sign(f(x_0))$$
then replace x_0 by x_m, otherwise replace x_1 by x_m.
4. Repeat steps 2-3.

6.1.4 Regula falsi

The Regula falsi is a modification of the Bisection method:

[1.]

1. Take two starting values x_0 and x_1 with $f(x_0) < 0$ and $f(x_1) > 0$.
2. Approximate f by a straight line between $f(x_0)$ and $f(x_1)$ and calculate the root of this line as
$$x_m = \frac{f(x_0)x_1 - f(x_1)x_0}{f(x_0) - f(x_1)}$$
3. If
$$sign(f(x_m)) = sign(f(x_0))$$
then replace x_0 by x_m, otherwise replace x_1 by x_m.
4. Repeat steps 2-3.

6.2 N-dimensional case

As a generalization of the previously discussed methods, we can solve a system of N coupled equations:

$$\vec{f}(\vec{x}) = 0$$

The corresponding optimization problem is then

$$\vec{\nabla} F(\vec{x}) = 0$$

6.2.1 Newton method

The N-dimensional Newton method replaces the derivative used in the scalar case by a Jacobi matrix:

$$J_{i,j}(\vec{x}) = \frac{\partial f_i(\vec{x})}{\partial x_j} \tag{6.6}$$

In the two-dimensional case, this matrix looks like that:

$$J = \begin{pmatrix} \frac{\partial f_1(\vec{x})}{\partial x_1} & \frac{\partial f_1(\vec{x})}{\partial x_2} \\ \frac{\partial f_2(\vec{x})}{\partial x_1} & \frac{\partial f_2(\vec{x})}{\partial x_2} \end{pmatrix}$$

This matrix has to be non-singular and well-conditioned because it has to be inverted numerically. The N-dimensional Newton equation is defined as

$$\vec{x}_{n+1} = \vec{x}_n - J^{-1}\vec{f}(\vec{x}_n) \tag{6.7}$$

The N-dimensional Newton method is a linearization too. We can show this by solving a system of linear equations, which should be solved exactly in one step by this method:
Consider the system of linear equations

$$B\vec{x} = \vec{c} \tag{6.8}$$

whose exact solution is

$$\vec{x} = B^{-1}\vec{c} \tag{6.9}$$

The system (6.8) can also be written as

$$\vec{f}(\vec{x}) = B\vec{x} - c = 0 \quad \Rightarrow \quad J = B \quad \text{(first derivative)} \tag{6.10}$$

Now we apply the Newton method by inserting (6.10) into (6.7):

$$\vec{x}_{n+1} = \vec{x}_n - B^{-1}(B\vec{x}_n - \vec{c}) = B^{-1}\vec{c} \tag{6.11}$$

6.2.2 Secant method

The N-dimesional secant method is characterized by the numerically calculated Jacobi matrix:

$$J_{i,j}(\vec{x}) = \frac{f_i(\vec{x} + h_j\vec{e}_j) - f_i(\vec{x})}{h_j} \tag{6.12}$$

with $\vec{e}_j$ the unit vector in the direction j. Insert this J in (6.7) and iterate.

6.2.3 Other techniques

We can solve a N-dimensional system of equations with the relaxation method:

$$\vec{f}(\vec{x}) = 0 \rightarrow x_i = g_i(x_j, j \neq i), \quad i = 1, \ldots, N \tag{6.13}$$

Start with $x_i(0)$ and iterate: $x_i(t+1) = g_i(x_j(t))$. You can read more about this method in section **??**.

Chapter 7

Ordinary differential equations

First order ordinary differential equations (ODE) and initial value problems can be written as

$$\frac{dy}{dt} = f(y,t) \quad \text{with} \quad y(t_0) = y_0 \tag{7.1}$$

If we want to solve an ODE numerically, we have to discretisize time so that we can advance in timesteps of size Δt.

7.1 Euler method

The Euler method is a simple numerical method to solve a first order ODE:
[1.]

1. Take the initial value $y(t_0) = y_0$
2. Calculate $\frac{dy}{dt}$.
3. Advance linearly for Δt with the derivative at the initial value as the slope.
4. Take the reached point as initial value and repeat steps 2-3.

If we take finite differences to find the derivative, the method looks as follows:

$$y_{n+1} = y_n + \Delta t f(y_n, t_n) + \mathcal{O}(\Delta^2 t) \tag{7.2}$$

As you can see, this method is locally of order 2 because the error at one timestep ist $\mathcal{O}(\Delta^2 t)$. Globally, this method is not of order 2, because one needs $\frac{T}{\Delta t}$ timesteps to traverse the whole time interval T:

$$\frac{T}{\Delta t}\mathcal{O}(\Delta^n t) = \mathcal{O}(\Delta^{n-1} t) \tag{7.3}$$

Therefore, the Euler method is globally of order 1.
Second order ODEs can be transformed into two coupled ODEs of first order. Let's take Newton's equation as an example:

$$m\frac{d^2x}{dt^2} = F(x) \Rightarrow \begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = \frac{F(x)}{m} \end{cases} \tag{7.4}$$

This idea can be extended to N coupled first order ODEs:

$$\frac{dy_i}{dt} = f_i(y_1, \ldots, y_N, t), \quad i = 1, \ldots, N \tag{7.5}$$

The Euler algorithm can then be written as

$$y_i(t_{n+1}) = y_i(t_n) + \Delta t f_i[y_1(t_n), \ldots, y_N(t_n), t_n] + \mathcal{O}(\Delta^2 t) \tag{7.6}$$

$t_n = t_0 + n\Delta t$, $y_1 \ldots y_N$ are the N functions described by the N ODEs.
The Euler method is not very accurate and is therefore expensive, because it needs very small timesteps to achieve acceptable results.

7.2 Runge-Kutta methods

Instead of taking smaller and smaller timesteps to achieve an accurrate solution of the ODE, we could also improve our method in a way that we don't need so small timesteps for a result of equal accurracy. For this, we need a better estimate for the slope of our linear function.
The Runge-Kutta methods are a generalization of the Euler method which leads to the same accurracy using larger timesteps. Therefore the Runge-Kutta methods are numerically less expensive (but more complicated to implement).
Formally, the Runge-Kutta methods are derived using a Taylor expansion for $y(t + \Delta t)$:

$$y(t + \Delta t) = y(t) + \frac{\Delta t}{1!}\frac{dy}{dt} + \frac{\Delta t^2}{2!}\frac{d^2y}{dt^2} + \cdots + \frac{\Delta t^q}{q!}\frac{d^qy}{dt^q} + \mathcal{O}(\Delta t^{q+1}) \tag{7.7}$$

Now we present some examples of Runge-Kutta methods and discuss later the order of this methods.

7.2.1 2^{nd} order Runge-Kutta method

Let's start with the 2^{nd} order Runge-Kutta method:
[1.]

1. Perform an Euler step of the size $\frac{\Delta t}{2}$ starting at the initial value $y_i(t_0)$:

$$y_i\left(t + \frac{1}{2}\Delta t\right) = y_i(t) + \frac{1}{2}\Delta t f[y_i(t), t]$$

2. Calculate the derivative at the reached point.
3. Advance full timestep with the calculated drivative as slope.
4. Take the reached point as initial value and repeat steps 1 to 3.

This leads to the following iteration:

$$y_i(t+\Delta t) = y_i(t) + \Delta t f\left[y_i\left(t+\frac{1}{2}\Delta t\right), t+\frac{1}{2}\Delta t\right] + \mathcal{O}(\Delta^3 t) \tag{7.8}$$

7.2.2 4^{th} order Runge-Kutta method

There exist more optimized versions of the Runge-Kutta method than the 2^{nd} order one. A common used one is the so called RK4 method, which is a 4^{th} order Runge-Kutta method:
[1.]

1. Define 4 coefficients:

$$k_1 = \Delta t \cdot f(y_n, t_n)$$
$$k_2 = \Delta t \cdot f\left(y_n + \frac{k_1}{2}, t_n + \frac{1}{2}\Delta t\right)$$
$$k_3 = \Delta t \cdot f\left(y_n + \frac{k_2}{2}, t_n + \frac{1}{2}\Delta t\right)$$
$$k_4 = \Delta t \cdot f(y_n + k_3, t_n + \Delta t)$$

2. Calculate the next step using a weighted sum of the defined coefficients:

$$y_{n+1} = y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + \mathcal{O}(\Delta^5 t) \tag{7.9}$$

   As you can see, the weighting factors are chosen in such a way that the sum of them is 1: $\frac{1}{6} + \frac{1}{3} + \frac{1}{3} + \frac{1}{6} = 1$

This method is globally of order 4 because it is locally of order 5 (see (7.3)). The selection of the weighting factors is not unique (see section 7.2.4).

7.2.3 q-stage Runge-Kutta method

The Runge-Kutta methods are defined in a more general way than the previous described. They are defined by the so called stage, wich determines how many terms you have in the sum of the iteration formula:

$$y_{n+1} = y_n + \Delta t \cdot \sum_{i=1}^{q} \omega_i k_i \tag{7.10}$$

with

$$k_i = f\left(y_n + \Delta t \cdot \sum_{j=1}^{i-1} \beta_{ij} k_j, \quad t_n + \Delta t \alpha_i\right) \quad \text{and} \quad \alpha_1 = 0$$

As you see, this method defines multiple evaluation points for the function f and computes the more accurrate derivative using all this points.
The coefficients and the weighting factors in the sum can be described in a Butcher array: [Slide 9]
For example the RK4 method with $q = 4$: [Slide 10]
If stages get higher, the order of the method doesn't grow as much as the stage: [Slide 15]
=

7.2.4 Order of the q-stage Runge-Kutta method

To test if a Runge-Kutta method is of order p, we take the Taylor expansion and calculate the coefficients α_i, β_{ij} and ω_i for $i, j \in [1, p]$ such that the right hand side is zero in $\mathcal{O}(\Delta t^m)$ for all $m \leq p$:

$$y(t + \Delta t) - y(t) = \underbrace{\sum_{m=1}^{p} \frac{1}{m!} \Delta t^m \cdot \left[\frac{d^{m-1} f}{dt^{m-1}}\right]_{y(t),t}}_{=0} + \mathcal{O}(\Delta t^{p+1}) \tag{7.11}$$

If we can find the coefficients this way, the order of the Runge-Kutta method is at least p.

Example: $q = p = 1$

This should lead to the Euler method. There is only one weighting factor, which obviously must be 1:

$$\omega_1 f(y_n, t_n) = f(y_n, t_n) \Rightarrow \omega_1 = 1 \tag{7.12}$$

Example: $q = p = 2$

$$\omega_1 k_1 + \omega_2 k_2 = f_n + \frac{1}{2} \Delta t \left[\frac{df}{dt}\right]_n \tag{7.13}$$

with

$$\left[\frac{df}{dt}\right]_n = \left[\frac{\partial f}{\partial t}\right]_n + \left[\frac{\partial f}{\partial y}\right]_n$$

The subscript n means the evaluation at the point (y_n, t_n). Now insert the coefficients retrieved from (7.10):

$$k_1 = f(y_n, t_n) = f_n$$

$$\begin{aligned} k_2 &= f(y_n + \Delta t \beta_{21} k_1, \quad t_n + \Delta t \alpha_2) \\ &= f_n + \Delta t \beta_{21} \left[\frac{\partial f}{\partial y}\right]_n f_n + \Delta t \alpha_2 \left[\frac{\partial f}{\partial t}\right]_n + \mathcal{O}(\Delta t^2) \end{aligned}$$

Now we can define rules for the weighting factors to derive their values.

- The first rule holds for all Runge-Kutta methods: The sum of the weighting factors must be 1.
$$\omega_1 + \omega_2 = 1$$
- The second rule we get from the comparison of coefficients in (7.13): $\left[\frac{\partial f}{\partial t}\right]_n$ shows up there with a prefactor of $\frac{1}{2}$. In order to get the same factor on the left hand side, we require
$$\omega_2 \cdot \alpha_2 = \frac{1}{2}$$
- In a similar way we can show that
$$\omega_2 \cdot \beta_{21} = \frac{1}{2}$$

This are 3 rules for 4 parameters, so there are multiple selections of the parameters which fulfill all the rules.

7.2.5 Error estimation

The estimation of the error of a method is essential if one wants to achieve an accurrate result in the shortest time possible. And because that's what we want to do in most of the cases, we try to estimate the error.
But there's more to errors: Imagine you knew the error exactly (i.e. the difference between your computed result and the true result). Then you were able to solve the problem exactly after one step! Unfortunately, we can't calculate the real error, but we can try to estimate it and adapt our method with the estimated error. The Predictor-Corrector method discussed in this section will show you how to use this idea to improve the previously seen methods.

7.2.5.1 Improve methods using error estimation

As a first estimate, we define the difference between the value after two timesteps Δt (called y_2) and the value obtained by performing timestep of size $2\Delta t$ (called y_1), respectively:

$$\delta = y_1 - y_2 \tag{7.14}$$

If we have a method of order p, this definition leads to

$$y(t+\Delta t) = \begin{cases} y_1 + (2\Delta t)^{p+1}\Phi + \mathcal{O}(\Delta t^{p+2}) \\ y_2 + 2(\Delta t)^{p+1}\Phi + \mathcal{O}(\Delta t^{p+2}) \end{cases} \tag{7.15}$$

$$\Rightarrow \delta = (2^{p+1}-2)\Delta t^{p+1}\Phi + \mathcal{O}(\Delta t^{p+2}) \tag{7.16}$$

Where Φ denotes the evolution operator of the method. By inserting (7.16) into (7.15) we get

$$y(t+\Delta t) = y_2 + \frac{2\delta}{2^{p+1}-2} + \mathcal{O}(\Delta t^{p+2}) \tag{7.17}$$

which is a better method of getting the next point, because the error is one order higher than before.
Example: Improve RK4 by this estimation of the error:

$$y(t+\Delta t) = y_2 + \frac{\delta}{15} + \mathcal{O}(\Delta t^6)$$

7.2.5.2 Adaptive timestep Δt

We could also improve the performance, if we use error estimates to adapt our timestep and use large timesteps if the solution does not change much and small ones at the regions which change much.
First we define the biggest error that we want to accept, $\delta_{expected}$ and then we estimate the error $\delta_{measured}$. The timestep can then be defined as

$$\Delta t_{new} = \Delta t_{old}\left(\frac{\delta_{expected}}{\delta_{measured}}\right)^{\frac{1}{p+1}} \tag{7.18}$$

because $\delta \propto \Delta t^{p+1}$

7.2.6 Predictor-Corrector method

Starting from the error estimation, we can define a new method of improvement for the solution of an ODE: the Predictor-Corrector method. The idea is to perform euler with the mean value of the function value at time t and the function value at time $t+\Delta t$:

$$y(t+\Delta t) \approx y(t) + \Delta t\frac{f(y(t)) + f(y(t+\Delta t))}{2} \tag{7.19}$$

This is an implicit equation, which we can't solve straight forward. So we make a prediction of $y(t+\Delta t)$ using a Taylor expansion of the function:

$$y^p(t+\Delta t) = y(t) + \Delta t\frac{dy}{dt}(t) + \mathcal{O}(\Delta t^2)$$

Now we can compute $y(t+\Delta t)$ in (7.19) using $y^p(t+\Delta t)$:

$$y^c(t+\Delta t) = y(t) + \Delta t \frac{f(y(t)) + f(y^p(t+\Delta t))}{2} + \mathcal{O}(\Delta t^3) \tag{7.20}$$

The corrected value $y^c(t+\Delta t)$ can be inserted itself into the corrector as the predicted value for a better result. (This can be done several times)

7.2.6.1 Higher order Predictor-Corrector methods

The Predictor-Corrector method can be extended to higher orders if we take more terms of the Taylor expansion for the predictor, for example 4 terms for the 3^{rd} order one:

$$y^p(t+\Delta t) = y(t) + \frac{\Delta t}{1!}\frac{dy}{dt}(t) + \frac{\Delta t}{2!}\frac{d^2y}{dt^2}(t) + \frac{\Delta t}{3!}\frac{d^3y}{dt^3}(t) + \mathcal{O}(\Delta t^4) \tag{7.21}$$

The computation of the corrector is now also more complicated because we need to define one for the function and two for its drivatives.
Instead of (7.20), we will now use

$$\left(\frac{dy}{dt}\right)^c (t+\Delta t) = f(y^p(t+\Delta t)) \tag{7.22}$$

The error is then defined as

$$\delta = \left(\frac{dy}{dt}\right)^c (t+\Delta t) - \left(\frac{dy}{dt}\right)^p (t+\Delta t) \tag{7.23}$$

Now we have to adapt the function itself and its second and third derivative in order to get a complete corrector:

$$\begin{aligned} y^c(t+\Delta t) &= y^p + c_0\delta \\ \left(\frac{d^2y}{dt^2}\right)^c (t+\Delta t) &= \left(\frac{d^2y}{dt^2}\right)^p + c_2\delta \\ \left(\frac{d^3y}{dt^3}\right)^c (t+\Delta t) &= \left(\frac{d^3y}{dt^3}\right)^p + c_3\delta \end{aligned}$$

with the so called Gear coefficients

$$c_0 = \frac{3}{8}, \quad c_2 = \frac{3}{4}, \quad c_3 = \frac{1}{6}$$

There are also higher order predictor-corrector methods, whose Gear coefficients are listed below:
(tables from slide 24) .

Chapter 8

Partial differential equations

In physical problems we have often differential equations with more than one independent variable. These are called partial differential equations (PDEs).
In a general form, a PDE with two variables is defined as

$$\begin{aligned} a(x,t)\frac{\partial^2 u(x,t)}{\partial x^2} + b(x,t)\frac{\partial^2 u(x,t)}{\partial x \partial t} + c(x,t)\frac{\partial^2 u(x,t)}{\partial t^2} + \\ d(x,t)\frac{\partial u(x,t)}{\partial x} + e(x,t)\frac{\partial u(x,t)}{\partial t} + f(u,x,t) &= 0 \end{aligned}$$

There are three types of PDEs:
[1.]

1. A PDE is elliptic, if

$$a(x,t)c(x,t) - \frac{b(x,t)^2}{4} > 0$$

2. A PDE is parabolic, if

$$a(x,t)c(x,t) - \frac{b(x,t)^2}{4} = 0$$

3. A PDE is hyperbolic, if

$$a(x,t)c(x,t) - \frac{b(x,t)^2}{4} < 0$$

8.1 Examples for PDEs

8.1.1 Scalar boundary value problems (elliptic PDEs)

Elliptic PDEs are classified by their boundary conditions. The Dirichlet problem is defined by a fixed value on the boundary:

$$\Delta\Phi = \rho(\vec{x}), \quad \Phi(\Gamma) = \Phi_0 \tag{8.1}$$

with

$$\begin{aligned} \rho(\vec{x}) &: \text{Load vector} \\ \Phi_0 &: \text{Fixed value on the boundary } \Gamma \end{aligned}$$

The von Neumann problem has a fixed gradient on the boundary:

$$\Delta\Phi = 0, \quad \nabla_n\Phi(\Gamma) = \Psi_0 \tag{8.2}$$

with

$$\Psi_0 : \text{Fixed value of the gradient of } \Phi \text{ on the boundary } \Gamma$$

8.1.2 Vectorial boundary value problem

An example of a vectorial boundary value problem is the Lam equation of elasticity. It is also an elliptic boundary value problem.

$$\vec{\nabla}(\vec{\nabla}\vec{u}(\vec{x})) + (1-\nu)\Delta\vec{u}(\vec{x}) = 0 \tag{8.3}$$

with

$$\begin{aligned} \vec{u}(\vec{x}) &: \text{vector field defined on space} \\ \nu &: \text{Poisson's ratio} \end{aligned}$$

Poisson's ration ν describes how much the material gets thinner in two dimensions if you stretch it in the third one.

8.1.3 Wave equation

The wave equation describes a wave traveling with speed c:

$$\frac{\partial^2\Phi}{\partial t^2} = c^2\Delta\Phi \tag{8.4}$$

with the initial condition

$$\Phi(\vec{x}, t_0) = \tilde{\Phi}_0(\vec{x})$$

The wave equation is a hyperbolic equation.

8.1.4 Diffusion equation

The diffusion equation describes diffusions of all types, for example heat or particle diffusion:

$$\frac{\partial\Phi}{\partial t} = \kappa\Delta\Phi \tag{8.5}$$

with the boundary condition

$$\Phi(\Gamma, t), = \Phi_0(t)$$

The diffusion equation is a parabolic equation.

8.1.5 Navier-Stokes equation

One of the most used equations is the Navier-Stokes equation which describes fluid motion. The big disadvantage of this equation is that it is really hard to solve (especially for big systems). However, this is how it looks like:

$$\frac{\partial \vec{v}}{\partial t} = (\vec{v}\vec{\nabla})\vec{v} = -\frac{1}{\rho}\vec{\nabla}p + \mu\Delta\vec{v}, \quad \vec{\nabla}\vec{v} = 0 \tag{8.6}$$

The condition $\vec{\nabla}\vec{v} = 0$ denotes that a zero divergence field is described. A zero divergence field has no sources and no sinks and conserves therefore the material. The other symbols mean:

- $\vec{v} = \vec{v}(\vec{x}, t)$: vector field of velocity in space and time
- $p = p(\vec{x}, t)$: pressure
- ρ: density of the fluid
- μ: viscosity of the fluid

Additionally, we need the initial and boundary conditions:

$$\left.\begin{array}{rcl} \vec{v}(\vec{x}, t_0) & = & \vec{V}_0(\vec{x}) \\ p(\vec{x}, t_0) & = & P_0(\vec{x}) \end{array}\right\} \quad \text{defined on the whole domain at time } t_0$$

$$\left.\begin{array}{rcl} \vec{v}(\Gamma, t) & = & \vec{v}_0(t) \\ p(\Gamma, t) & = & p_0(t) \end{array}\right\} \quad \text{defined on the boundary at all times}$$

8.2 Discretization of the derivatives

In order to compute the solution of a PDE on a computer we have to discretize the derivatives because the computer can only handle a finite number of things.

8.2.1 First derivative in 1D

You know the first derivative in 1D using finite differences from your analysis course as the deviation of the derivative:

$$\begin{aligned} \frac{\partial \Phi}{\partial x} &= \frac{\Phi(x_{n+1}) - \Phi(x_n)}{\Delta x} + \mathcal{O}(\Delta x) \quad \text{(two-point formula)} \\ &= \frac{\Phi(x_n) - \Phi(x_{n-1})}{\Delta x} + \mathcal{O}(\Delta x) \quad \text{(two-point formula)} \\ &= \frac{\Phi(x_{n+1}) - \Phi(x_{n-1})}{2\Delta x} + \mathcal{O}(\Delta x^2) \quad \text{(three-point formula)} \end{aligned} \tag{8.7}$$

As you see, the accuracy is increased if you take more points. There exist also formulas which are using more than 3 points to increase the accuracy even more.

8.2.2 Second derivative in 1D

If we apply (??) twice on Φ, it should give us the second derivative:

$$\frac{\partial^2 \Phi}{\partial x^2} = \frac{\Phi(x_{n+1}) + \Phi(x_{n-1}) - 2\Phi(x_n)}{\Delta x^2} + \mathcal{O}(\Delta x^2) \tag{8.8}$$

A more accurrate approximation using the five-point formula for the second derivative is (without derivation)

$$\begin{aligned} \frac{\partial^2 \Phi}{\partial x^2} &= \frac{1}{12\Delta x^2}\left[-\Phi(x_{n-2}) + 16\Phi(x_{n-1})\right. \\ &- \left. 30\Phi(x_n) + 16\Phi(x_{n+1}) - \Phi(x_{n+2})\right] + \mathcal{O}(\Delta x^4) \end{aligned} \tag{8.9}$$

8.2.3 Derivatives in higher dimension

In many cases we need derivatives in two or more dimensions. For simplicity, we define $\Delta x = \Delta y = \Delta z$. So the discretization is the same in all directions.
The second derivative is defined as

$$\begin{aligned} \Delta\Phi\Delta x^2 &= \Phi(x_{n+1}, y_n) + \Phi(x_{n-1}, y_n) \\ &+ \Phi(x_n, y_{n+1}) + \Phi(x_n, y_{n-1}) - 4\Phi(x_n, y_n) \end{aligned} \tag{8.10}$$

You can see this as a stencil which is applied on the grid of discretization points and constructs the value of the derivative at the center point by adding up all the neighbors before the subtraction of 4 times the value of the center point.
The derivative in three dimensions (using the same idea as in two dimensions) is

$$\begin{aligned} \Delta\Phi\Delta x^2 &= \Phi(x_{n+1}, y_n, z_n) + \Phi(x_{n-1}, y_n, z_n) \\ &+ \Phi(x_n, y_{n+1}, z_n) + \Phi(x_n, y_{n-1}, z_n) + \Phi(x_n, y_n, z_{n+1}) \\ &+ \Phi(x_n, y_n, z_{n-1}) - 6\Phi(x_n, y_n, z_n) \end{aligned} \tag{8.11}$$

8.3 The Poisson equation

The Poisson equation is defined as

$$\Delta\Phi(\vec{x}) = \rho(\vec{x}) \tag{8.12}$$

At first we want to analyze the one-dimensional case with Dirichlet boundary conditions. In this case, the Poisson equation is

$$\frac{\partial^2 \Phi}{\partial x^2} = \rho(x) \tag{8.13}$$

In order to do this, we have to discretize the one-dimensional space by x_n with $n = 1, \ldots, N$. We will also abbreviate $\Phi(x_n)$ with Φ_n. If we insert the 1D-Poisson

equation (8.13) into the definition of the second derivative (8.8) and shift Δx^2 to the other side, we get the discretized equation:

$$\Phi_{n+1} + \Phi_{n-1} - 2\Phi_n = \Delta x^2 \rho(x_n) \tag{8.14}$$

The Dirichlet boundary conditions are incorporated with

$$\Phi_0 = c_0 \quad \text{and} \quad \Phi_N = c_1.$$

This discretized equation is in effect a set of $N-1$ coupled linear equations, which can be written in a matrix form (with $\rho(x) \equiv 0$):

$$\begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix} \cdot \begin{pmatrix} \Phi_1 \\ \\ \vdots \\ \\ \Phi_{N-1} \end{pmatrix} = - \begin{pmatrix} c_0 \\ 0 \\ \vdots \\ 0 \\ c_1 \end{pmatrix} \tag{8.15}$$

This is a system of the form $A\vec{\Phi} = \vec{b}$, which we have to solve for $\vec{\Phi}$. How you can do this you see in section 8.4.

8.4 Solving systems of linear equations

After the discretisation of a PDE we are left with a system of coupled linear equations:

$$A\vec{\Phi} = \vec{b} \tag{8.16}$$

Getting the solution $\vec{\Phi}$ of this system is the most time consuming part, so you would have to spend days getting the solution of a system of about 10^4 equations if you use Gauss elimination! But there exist multiple methods to solve systems of this size in less time at least approximatively. In this section, we will introduce some of them and tell you about their advantages and limitations.

8.4.1 Relaxation methods

Relaxation methods can be seen as a sort of smoothening operators on the matrices. They provide a realistic behavior in time if you solve for example the heat equation. You can solve even nonlinear problems with this methods if you consider A an operator as a generalization of a matrix. The drawback of this methods is the slow convergence.

8.4.1.1 Jacobi method

The Jacobi method is the simplest relaxation method for the solution of systems of linear equations. Let us decompose A in the lower triangle U, the diagonal elements D and the upper triangle O:

$$A = D + O + U \tag{8.17}$$

The Jacobi method is then defined as

$$\vec{\Phi}(t+1) = D^{-1}(\vec{b} - (O+U)\vec{\Phi}(t)) \tag{8.18}$$

The Jacobi method is not very fast and the exact solution is only reached for $t \to \infty$. Since we don't need the exact solution but a solution with some error ϵ, we can stop if the measured error δ' is smaller than the predefined accurracy ϵ:

$$\delta'(t+1) \equiv \frac{\|\vec{\Phi}(t+1) - \vec{\Phi}(t)\|}{\|\vec{\Phi}(t)\|} \leq \epsilon \tag{8.19}$$

δ' is not the real error, but we will show you that it is in most cases a good estimate for it. First, we have to define the matrix Λ, which is the evolution operator of the error of the Jacobi method. This operator transforms the error in the step t to the error in the step $t+1$:

$$\vec{\delta}(t+1) = -\Lambda\vec{\delta}(t)$$

It is defined as

$$\Lambda = D^{-1}(O+U) \tag{8.20}$$

and the operation is simply the matrix-vector product between Λ and the error vector $\vec{\delta}$. The largest eigenvalue of Λ gives us a measure of the error propagation from one timestep to the next one. It is called λ and is always between 0 and 1. For a large number of timesteps we can write now

$$\vec{\Phi}(t) \approx \vec{\Phi}^* + \vec{c}\lambda^t$$

λ is also the ratio between the actual step size and the previous one:

$$\frac{\|\vec{\Phi}(t+1) - \vec{\Phi}(t)\|}{\|\vec{\Phi}(t) - \vec{\Phi}(t-1)\|} \approx \frac{\lambda^{t+1} - \lambda^t}{\lambda^t - \lambda^{t-1}} = \lambda$$

(Why do we take the eigenvalue?)

8.4.1.2 Gauss-Seidel method

The matrix A is decomposed the same way as in the Jacobi method (see (8.17)) but the parts are put together in an other way:

$$\vec{\Phi}(t+1) = (D+O)^{-1}(\vec{b} - U\vec{\Phi}(t)) \tag{8.21}$$

The error is calculated the same way as in the Jacobi method but with another evolution operator:

$$\Lambda = (D+O)^{-1}U$$

8.4.1.3 Successive Overrelaxation (SOR) method

The SOR method tries to improve the convergence of the Gauss-Seidel relaxation method by introducing an overrelaxation parameter ω:

$$\vec{\Phi}(t+1) = (D+\omega O)^{-1}\left(\omega\vec{b} + [(1-\omega)D - \omega U]\vec{\Phi}(t)\right) \tag{8.22}$$

If we set $\omega = 1$, we get again the Gauss-Seidel method. ω is set typically to a value between 1 and 2 and has to be determined essentially by trial and error.

8.4.1.4 Nonlinear problems

As described above, you can solve nonlinear problems with a generalized relaxation. For example a network of resistors with a nonlinear $I - U$ relation called f and each resistor connected with four neighbors. Kirchhoff's law leads to the following equations:

$$f(U_{i+1,j} - U_{i,j}) + f(U_{i,j} - U_{i-1,j}) + f(U_{i,j+1} - U_{i,j}) + f(U_{i,j} - U_{i,j-1}) = 0$$

This means that the current going into the node (i, j) equals the outgoing current. If we reformulate this problem as

$$\begin{aligned} & f(U_{i+1,j}(t) - U_{i,j}(t+1)) + f(U_{i,j}(t+1) - U_{i-1,j}(t)) \\ + \quad & f(U_{i,j+1}(t) - U_{i,j}(t+1)) + f(U_{i,j}(t+1) - U_{i,j-1}(t)) = 0 \end{aligned}$$

we can solve it for $U_{i,j}(t+1)$ with relaxation.

8.4.2 Gradient methods

Gradient methods are powerful methods to solve systems of linear equations. They are using a functional which measures the error of a solution of the system of equations. If a system of equations has one unique solution, this functional is a paraboloid with the minimum at the exact solution. This functional is defined by the residuum $\vec{r}$, which can be seen as an estimate of the error $\vec{\delta}$:

$$\vec{r} = A\vec{\delta} = A(A^{-1}\vec{b} - \vec{\Phi}) = \vec{b} - A\vec{\Phi} \tag{8.23}$$

As you see, we don't have to invert the matrix A to get the residuum. That's the reason, why we use it instead of the error. If the residuum is small, the error is also going to be small, so we can minimize

$$\mathcal{J} = \vec{r}^T A^{-1}\vec{r} = \begin{cases} 0 & \text{if } \vec{\Phi} = \vec{\Phi}^* \\ > 0 & \text{otherwise} \end{cases} \tag{8.24}$$

where Φ^* is the exact solution. By insertion of (8.23) into (8.24), we get

$$\mathcal{J} = (\vec{b} - A\vec{\Phi})^T A^{-1}(\vec{b} - A\vec{\Phi}) = \vec{b}^T A^{-1}\vec{b} + \vec{\Phi}^T A\vec{\Phi} - 2\vec{b}\vec{\Phi} \tag{8.25}$$

Now we want to minimize the functional along straight lines. For this, we define

$$\vec{\Phi} = \vec{\Phi}_i + \alpha_i \vec{d_i}$$

with $\vec{\Phi}_i$ as the start value or the value of the last iteration, $\vec{d_i}$ the direction of the step and α_i the step length. If we insert this in (8.25), it looks like that:

$$\mathcal{J} = \vec{b}^T A^{-1} \vec{b} + \vec{\Phi}_i^T A \vec{\Phi}_i + 2\alpha_i \vec{d_i}^T A \vec{\Phi}_i + \alpha_i^2 \vec{d_i}^T A \vec{d_i} - 2\vec{b}^T \vec{\Phi}_i - 2\alpha_i \vec{b}^T \vec{d_i}$$

Now we have to determine how far we go in the direction $\vec{d_i}$ by the minimization of $\mathcal{J}$, which we get if we set the derivative with respect to α_i equal to zero:

$$\frac{\partial \mathcal{J}}{\partial \alpha_i} = 2\vec{d_i}^T (\alpha_i A \vec{d_i} - \vec{r_i}) = 0$$

This last expression leads to the formula for α_i:

$$\alpha_i = \frac{\vec{d_i}^T \vec{r_i}}{\vec{d_i}^T A \vec{d_i}} \tag{8.26}$$

The most difficult part of the Gradient methods is the computation of the direction of the step. So they are distiguishable by this part of the algorithm, which we will show you now.

8.4.2.1 Steepest descent

The most intuitive method of getting the direction is to choose the one with the largest (negative) gradient. This is the same as if you would take always the steepest possible direction if you want to get down from the mountain to the valley. In this case, the direction is set to the residual in the point we reached so far:

$$\vec{d_i} = \vec{r_i} \tag{8.27}$$

The steepest descent method has the disadvantage that it doesn't take the optimal direction if the functional is not a regular paraboloid. You can see that behaviour in figure ... [Bild von Folie 7] However, this is the algorithm:

[1.]

1. Start with $\vec{\Phi}_i$ and choose

   $$\vec{d_i} = \vec{r_i} = \vec{b} - A\vec{\Phi}_i$$

2. Calculate the matrix-vector product $A \cdot \vec{b}$ and store the result in $\vec{u_i}$ because we are going to use it twice:

   $$\vec{u_i} = A\vec{r_i}$$

   Calculate the length of the step:

   $$\alpha_i = \frac{\vec{r_i}^2}{\vec{r_i}\vec{u_i}}$$

3. Advance α_i in the direction of $\vec{d_i}$ (which is the same as $\vec{r_i}$) and compute the value of the function at this point:

$$\vec{\Phi}_{i+1} = \vec{\Phi}_i + \alpha_i \vec{r_i}$$

4. Update the residuum for the next step:

$$\vec{r}_{i+1} = \vec{r_i} + \alpha_i \vec{u}_i$$

5. Repeat steps 2 to 4 until the residuum is sufficiently small.

The vector $\vec{u}_i$ is the temporary stored product $A\vec{r_i}$ so that we need to compute it only once per step because this is the most time consuming part of the algorithm and scales with N^2, where N is the number of equations to be solved. If A is sparse, this matrix-vector product is of order N.

8.4.2.2 Conjugate gradient

This method takes the functional and deforms it such that it looks like a regular paraboloid and performs on it the steepest descent method. This can be achieved if we take the new direction conjugate to all previous ones using the Gram-Schmidt orthogonalization process:

$$\vec{d_i} = \vec{r_i} - \sum_{j=1}^{i-1} \frac{\vec{d_j}^T A \vec{r_i}}{\vec{d_j}^T A \vec{d_j}} \vec{d_j} \tag{8.28}$$

The algorithm is then
[1.]

1. Initialize with some vector $\vec{\Phi}_1$ and compute the first residual out of it. Then take the residual as the direction of your first step:

$$\vec{r}_1 = \vec{b} - A\vec{\Phi}_1, \quad \vec{d}_1 = \vec{r}_1$$

2. Compute temporary vector

$$c = (\vec{d_i}^T A \vec{d_i})^{-1}$$

Note: you don't have to invert the matrix A for this, as you first compute the scalar product which leads to a scalar. However, you need to compute a matrix-vector product, which is of complexity N^2 for dense matrices and N for sparse matrices.

3. Compute the length of the step:

$$\alpha_i = c\vec{d_i}$$

4. Do the step:
$$\vec{\Phi}_{i+1} = \vec{\Phi}_i + \alpha_i \vec{d}_i$$
If the residuum is sufficiently small, e.g. $\vec{r}_i^T \vec{r}_i < \epsilon$, you can **stop**.

5. Update the residuum for the error estimation and for the next step:
$$\vec{r}_{i+1} = \vec{b} - A\vec{\Phi}_{i+1}$$

6. Compute the direction of the next step:
$$\vec{d}_{i+1} = \vec{r}_{i+1} - (c\vec{r}_{i+1} A\vec{d}_i)\vec{d}_i$$

7. Repeat steps 2 to 6.

8.4.3 Preconditioning

If you use one of the previously seen methods to solve a system of linear equations, it may be the case that you notice poor performance because the method does converge very slowly. This is the case if the matrix is bad conditioned: The diagonal elements are almost equal to the sum of the other elements on their row. There is a solution to this problem: Preconditioning.
It's a simple idea: find a matrix P^{-1} which is cheap to compute and delivers a good approximation of the inverse of the matrix of your problem. (The idea is simple, not the process of finding such a matrix!). Then solve the preconditioned system obtained by a multiplication from the left:

$$\left(P^{-1}A\right)\vec{\Phi} = P^{-1}\vec{b} \tag{8.29}$$

8.4.3.1 Jacobi preconditioner

This is a simple preconditioner which has nothing to do with the Jacobi method seen before. It just takes all diagonal elements of the matrix A throwing away all other elements and invert the obtained diagonal matrix:

$$P_{ij} = A_{ij}\delta_{ij} = \begin{cases} A_{ii} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \Rightarrow P^{-1}_{ij} = \delta_{ij}\frac{1}{A_{ii}} \tag{8.30}$$

8.4.3.2 SOR preconditioner

Another example of a preconditioner is the SOR preconditioner:

$$P = \left(\frac{D}{\omega} + U\right)\frac{\omega}{2-\omega}D^{-1}\left(\frac{D}{\omega} + O\right) \tag{8.31}$$

8.4.4 Multigrid procedure

Consider multiple grids on the same matrix which select some matrix entries. Then solve the equation for the error on the coarsest grid.

[1.]

1. Apply the restriction operator $\mathcal{R}$:

$$\hat{\vec{r}}_n = \mathcal{R}\vec{r}_n$$

   This operator reduces the original system to a smaller system composing a new matrix out of the original one.

2. Solve the system obtained using one of the methods above or just by a Gauss elimination if the system is small enough.

3. As a last step, you have to reproduce the original matrix out of the smaller matrix using the extension operator $\mathcal{P}$. This is for example an interpolation operator which computes the elements lying between the ones of the solved system by a weighted sum.

Step 1 and 3 can be repeated several times in order to reduce a huge system to a system solvable by a Gauss elimination. In this case the error of the method is not in the solver but in the restriction and extension operators.
Now we will see an example of the restriction and extension operators. The restriction operator takes in each direction every second entry of the matrix (that means it takes a fourth of all entries) and calculates with its value and the values of the ones around it the value of the entry of the matrix on the coarser grid which has half the edge length of the original one:

$$\mathcal{R}\vec{r} \mapsto \begin{cases} \hat{r}_{i,j} & = & \frac{1}{4}r_{i,j} + \frac{1}{8}(r_{i+1,j} + r_{i-1,j} + r_{i,j+1} + r_{i,j+1}) \\ & + & \frac{1}{16}(r_{i+1,j+1} + r_{i-1,j+1} + r_{i-1,j+1} + r_{i-1,j-1}) \end{cases} \tag{8.32}$$

As you see, the elements left, right, above and below the selected element are taken twice as much as the diagonal ones because the diagonal ones have twice as much neighbors as the other.
The extension operator for this example looks as follows:

$$\mathcal{P}\hat{\vec{r}} \mapsto \begin{cases} r_{2i,2j} & = & \hat{r}_{i,j} \\ r_{2i+1,2j} & = & \frac{1}{2}(\hat{r}_{i,j} + \hat{r}_{i+1,j}) \\ r_{2i,2j+1} & = & \frac{1}{2}(\hat{r}_{i,j} + \hat{r}_{i,j+1}) \\ r_{2i+1,2j+1} & = & \frac{1}{4}(\hat{r}_{i,j} + \hat{r}_{i+1,j} + \hat{r}_{i,j+1} + \hat{r}_{i+1,j+1}) \end{cases} \tag{8.33}$$

This two operators are adjunct to each other:

$$\sum_{x,y} \mathcal{P}\hat{v}(\hat{x},\hat{y}) \cdot u(x,y) = h^2 \sum_{\hat{x},\hat{y}} \hat{v}(\hat{x},\hat{y}) \cdot \mathcal{R}u(x,y)$$

h denotes the scaling factor of the finer matrix to the coarser matrix $\frac{N}{\hat{N}}$, where N is the number of rows. In our example h is equal 2.

8.5 Finite Element methods (FEM)

The problem of the Finite Differences method is the regular mesh that can't be adapted to the problem. So it is not possible to take different mesh sizes at different regions. This is important if you want to achieve a better result without refining the whole mesh but only the regions with big gradients because the error depends on the ratio of the gradient to the mesh size.
This problem can be solved using Finite Element methods. They discretize the PDE by patching together continuous functions instead of the discretisation of the field on sites. With Finite Element methods you can solve PDEs for irregular geometries, inhomogeneous fields, such with moving boundaries and finally nonlinear PDEs. The mesh can even be adapted while you compute the solution in order to achieve the best possible convergence in each step.

8.5.1 Basis functions

The discretisation in Finite Element methods is done by the selection of basis functions. The exact solution of a PDE can always be written as a linear combination of infinitly many basis functions:

$$\Phi(x) = \sum_{i=1}^{\infty} a_i u_i(x) \tag{8.34}$$

But the computer can only handle a finite number of things, so we have to select some of them. This process is a projection from the vector space with infinitly many basis functions (where the exact solution lives) to a finite dimensional space (in which we can compute something):

$$\Phi_N(x) = \sum_{i=1}^{N} a_i u_i(x) \tag{8.35}$$

The basis functions $u_i(x)$ are known and defined for all elements, so we have only to compute the coefficients a_i.
In the one-dimensional case, the linear basis functions are hat functions which are 1 in the element on which they are defined and 0 everywhere else.

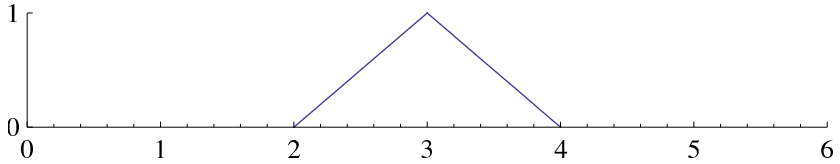

Figure 8.1: Hat basis functions for the one-dimensional case

A clear example of basis functions can be shown in the two-dimensional case. The domain is divided into triangles (which don't have to be of the same size). Each triangle can be transformed to the so called standard element, whose corners are located at $(0,0)$, $(0,1)$ and $(1,0)$. The basis functions are defined on this triangle and then transformed back to the original triangle for the computation. You see in the next section how you can do this in the concrete case.
Let's first define the basis functions on the standard element. In the linear case, each basis function is 1 in one corner and 0 in all others. There are three basis functions:

$$\begin{aligned} N_1 &= 1-\xi-\eta \\ N_2 &= \xi \\ N_3 &= \eta \end{aligned} \tag{8.36}$$

(The axes are called ξ and η for the standard element to distuingish it from the original triangle whose axes are called x and y) You can see a graphical representation of the linear basis functions in figure 8.5.1.

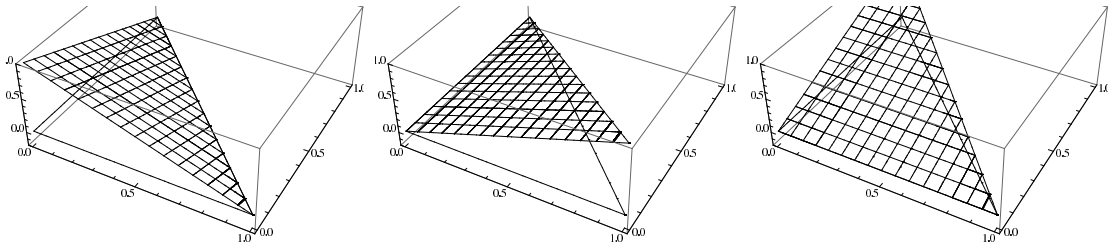

Figure 8.2: The three linear basis functions on the standard element

You can also use quadratic basis functions. In that case there are six points on each triangle. The rest is the same as in the linear case, each basis function is 1 in one of the points and 0 in all others. So you have 6 basis functions:

$$\begin{aligned} N_1 &= (1-\xi-\eta)(1-2\xi-2\eta) \\ N_2 &= \xi(2\xi-1) \\ N_3 &= \eta(2\eta-1) \\ N_4 &= 4\xi(1-\xi-\eta) \\ N_5 &= 4\xi\eta \\ N_6 &= 4\eta(1-\xi-\eta) \end{aligned} \tag{8.37}$$

A visualization of this functions can be found in figure 8.5.1.

You can divide the domain also into squares. Then you have 9 basis functions in the quadratic case: [Figures from slide 25]

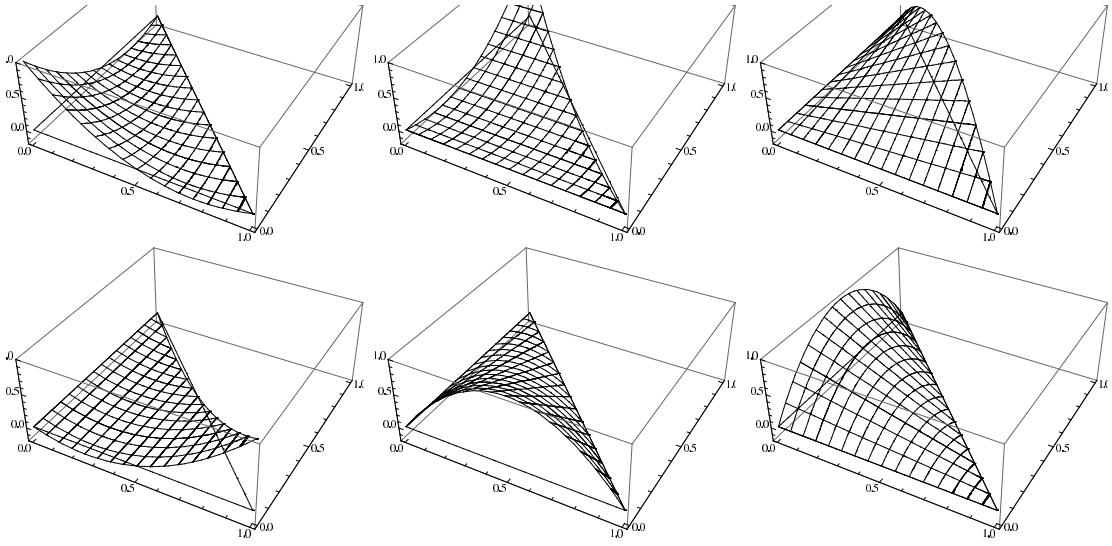

Figure 8.3: The six quadratic basis functions on the standard element

8.5.2 Transformation of the elements to the standard element

Any triangle in the two-dimensional space can be transformed to the previously discussed standard triangle with the following transformation:

$$\begin{aligned} x &= x_1 + (x_2 - x_1)\xi + (x_3 - x_1)\eta \\ y &= y_1 + (y_2 - y_1)\xi + (y_3 - y_1)\eta \end{aligned} \tag{8.38}$$

To transform the standard element back to the original one, you can use this transformation:

$$\begin{aligned} \eta &= \tfrac{1}{D}((y - y_1)(x_2 - x_1) - (x - x_1)(y_2 - y_1)) \\ \xi &= \tfrac{1}{D}((x - x_1)(y_3 - y_1) - (y - y_1)(x_3 - x_1)) \end{aligned} \tag{8.39}$$

with

$$D = (y_3 - y_1)(x_2 - x_1) - (x_3 - x_1)(y_2 - y_1)$$

8.5.3 Example: Poisson equation

In the following we will use the Poisson equation as an example to clarify things.

$$\Delta\Phi(x) = -4\pi\rho(x) \tag{8.40}$$

with Dirichlet boundary conditions

$$\Phi|_\Gamma(x) = 0 \quad (\Phi \text{ is 0 on the boundary})$$

8.5.4 Variational formulation

The variational formulation introduces a test function (or weight function) w to the problem:

$$-\int_\Omega \Delta\Phi(x)w(x)dx = 4\pi\int_\Omega \rho(x)w(x)dx$$

with Ω the domain on which we want to solve the problem. This has to be true for all $w(x) \in C_0^\infty$, which means that w has to be differentiable infinitly many times without loosing continuity.
Now we integrate by parts using Green's theorem and we get

$$-\int_\Omega \nabla\Phi(x)\nabla w(x)dx = 4\pi\int_\Omega \rho(x)w(x)dx \tag{8.41}$$

One can show that if we can find a solution Φ which holds true for all $w(x) \in C_0^\infty$, this is a unique solution for Φ and there's no other one that fullfills this condition.

8.5.5 Discretization of the variational formulation

Now we discretize the problem according to (8.35) and solve it in one dimension. We call the discretized functions $u_i(x)$ and $w_j(x)$ and build the equations to find the coefficients a_i. The coefficients are obtained by solving the system $A\vec{a} = \vec{b}$ with

$$A_{ij} = -\int_0^L u_i''(x)w_j(x)dx = \int_0^L u_i'(x)w_j'(x)dx$$

(you can derive this using integration by parts, see (8.41)) and

$$b_j = 4\pi\int_0^L \rho(x)w_j(x)dx$$

A is called stiffness matrix and $\vec{b}$ is called load vector. If we take the weight functions equal to the the basis functions ($w_j(x) = u_j(x)$) we get the Galerkin method. The basis functions $u_j(x)$ are for example hat functions centered around i (see figure 8.5.1). Let's take the element i which lies between x_{i-1} and x_i. The basis functions are then

$$u_i(x) = \begin{cases} \frac{x-x_{i-1}}{\Delta x} & \text{for } x \in [x_{i-1}, x_i] \\ \frac{x_{i+1}-x}{\Delta x} & \text{for } x \in [x_i, x_{i+1}] \\ 0 & \text{otherwise} \end{cases} \tag{8.42}$$

The stiffness matrix A is then defined as

$$A_{ij} = \int_0^L u_i'(x)u_j'(x)dx = \begin{cases} \frac{2}{\Delta x} & \text{for } i = j \\ \frac{-1}{\Delta x} & \text{for } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases} \tag{8.43}$$

The zero Dirichlet boundary conditions are fulfilled automatically because the basis functions $u_j(x)$ are zero at both ends of the interval. If the boundary conditions are not zero, e.g.

$$\Phi(0) = \Phi_0 \quad \text{and} \quad \Phi(L) = \Phi_1$$

we have use another decomposition for the function $\Phi_N(x)$:

$$\Phi_N(x) = \frac{1}{L}\left(\Phi_0(L-x) + \Phi_1 x \sum_{i=1}^{N} a_i u_i(x)\right) \tag{8.44}$$

In this way, we incorporate the boundary conditions directly in the discretization of the problem.