Introduction to Computational Physics HS23

Dr. Andreas Adelmann
Paul Scherrer Institute (PSI), Villigen
Swiss Federal Institute of Technology (ETH), Zurich

Chapter 1

General Information

References Much of the material (ideas, definitions, concepts, examples, figures, etc.) in these notes is taken (in some cases verbatim) for teaching purposes, from several references, and in particular from the incomplete references listed in the text.

Disclaimer The present notes are only informally distributed and are intended ONLY as a study aid for ETHZ students registered for the course Introduction to Computational Physics (HS 2023).

1.1 Useful Addresses and Information

- The content of this class including the exercise material is available online: https://moodle-app2.let.ethz.ch/course/view.php?id=20999.
- The script is constantly evolving and will be online, chapter by chapter before each lecture.
- The script and the slides should be synchronized as much as possible (except for this lecture ☺).

1.2 Who is the Target Audience of This Lecture?

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

- Mathematics and Computer Science (Bachelor and Master course)
- Physics (Bachelor and Master course)
- Material Science Master
- Civil Engineering Master
- Integrated Building Systems Master

- Neural Systems and Computation Master
- Computational Science and Engineering (Bachelor and Master course)

1.3 Some Words About Me ...

Dr. Adelmann is a senior scientist and head of the laboratory of scientific computing and modelling at PSI. My field of expertise is computational and statistical physics, in particular dynamical systems (particle accelerators).
My present research topics includes (parallel) numerical methods for relativistic n-body problems involving Maxwell's equations, plasma physics, quantum free-electron-lasers, uncertanty quantification, statistical and machine learning for surrogate model construction and inverse problems.
You can reache me at

andreaad@ethz.ch

or on Fridays (13-14) in HPK G 28, ETH Hönggerberg, Zürich, rsvp. My PSI group web page is located at http://amas.web.psi.ch/.

1.4 Some Words About the Pervious Authors of the Script

The script was started in 2007 by Dr. H.M. Singer who wrote large parts of the chapter on random number generators, a part of the chapter on percolation, and large parts of the chapters on solving equations. In 2009/2010, L. Müller and M.A. Buchmann continued the script and filled in the gaps, expanded said chapters and added chapters on Monte Carlo methods, fractal dimensions and the Ising model. Thereafter, Prof. Dr. H. J. Herrmann and Dr. L. Böttcher extended the lecture notes. In 2021, new lecture notes were written by M.F. Holst and Dr. A. Adelmann. However, large parts of the content of random numbers, percolation, fractals and Monte Carlo methods were reused from earlier versions.
If you have suggestions or corrections, please drop me a message. Thank you!

1.5 Outline of the Course

- Intro Julia (what is special) Random Number Generators
- Percolation
- Fractals
- Cellular Automata and a Simple Gas Model
- Monte Carlo Methods I & II
- Finite Differences & Fluid Dynamics

- Integration Methods
- Solution Methods for the Maxwell Equations I & II
- Particle In Cell Method
- N-Body Problems
- Collisions in N-Body Problems
- Uncertainty Quantification
- Surrogates of Physical Systems

The second part of the course is more work in progress!

1.6 Prerequisites for this Class

- Requirements in mathematics:
  - You should know the basics of statistical analysis (averaging, distributions, etc.).
  - Furthermore, basic knowledge of linear algebra and calculus, especially ODE and PDE solving, 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.

1.7 What is Computational Physics all about?

- Computational physics is the study and implementation of numerical algorithms to solve problems in physics by means of computers.
- Computational physics in particular solves equations numerically.
- Finding a solution numerically is useful, as there are very few systems for which an analytical solution is known.
- The evaluation and visualization of large data sets, which can come from numerical simulations or experimental data (for example complex phase-space)

Computational physics plays an important role in all fields of science, prominent ones are:

- 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 ("Lattice QCD")
- Astrophysics: Many-body simulations of stars and galaxies
- Accelerator Physics (see also PAM-1 and PAM-2)
- Geophysics and Solid Mechanics: Earthquake simulations, fracture, rupture, crack propagation etc.
- Agent Models (interdisciplinary): Complex networks in biology, economy, social sciences and many others

1.8 Suggested Literature

This lecture is self contained but only a basic introduction. In case you are interested in more details and a deeper discussion the following literature is a good starting point.

- H. Gould, J. Tobochnik and W. 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:

- International Conference on Computational Physics 2020, Rome, https://waset.org/computational-physics-conference-in-december-2020-in-rome

For the exams, none of these additional source of information is necessary.

1.9 Programming Languages

There have been many discussions and fights about the "perfect language" for numerical simulations. The simple answer is: It depends on the problem. Here, we will give a short overview to help you choose the right tool.

1.9.1 Symbolic Algebra Programs

Mathematica and Maple have become very powerful tools and allow symbolic manipulation at a high level of abstraction. They are useful not only for exactly solvable problems but also provide powerful numerical tools for many simple programs. Choose Mathematica or Maple when you either want an exact solution or the problem is not too complex.

1.9.2 Interpreted Languages

Interpreted languages range from simple shell scripts and Perl programs, most useful for data handling and simple data analysis to fully object-oriented programming languages such as Python. We will regularly use such tools in the exercises.

1.9.3 Compiled Procedural Languages

Compiled procedural languages are substantially faster than the interpreted languages discussed above, but usually need to be programmed at a lower level of abstraction (e.g. manipulating numbers instead of matrices).

1.9.4 FORTRAN (FORmula TRANslator)

Fortran was the first scientific programming language. The simplicity of FORTRAN 77 and earlier versions allows aggressive optimization and unsurpassed performance. The disadvantage is that complex data structures such as trees, lists or text strings, are hard to represent and manipulate in FORTRAN.
Newer versions of FORTRAN (FORTRAN 90/95, FORTRAN 2000) converge towards object oriented programming (discussed below) but at the cost of decreased performance. Unless you have to modify an existing FORTRAN program use one of the languages discussed below.

1.9.5 Other procedural languages: C, Pascal, Modula, etc.

Other procedural languages simplify the programming of complex data structures but cannot be optimized as aggressively as FORTRAN 77. This can lead to performance drops by up to a factor of two! Of all the languages in this category C is the best choice today.

1.9.6 Object Oriented Languages

The class concept in object oriented languages allows programming at a higher level of abstraction. Not only do the programs get simpler and easier to read, they also become easier to debug. This is usually paid for by an "abstraction penalty", sometimes slowing programs down by more than a factor of ten if you are not careful.

1.9.7 Java

Java is very popular in web applications since a compiled Java program will run on any machine, though not at the optimal speed. Java is most useful in small graphics applets for simple physics problems.

1.9.8 C++

Two language features make C++ one of the best languages for scientific simulations: Operator overloading and generic programming. Operator overloading allows to define mathematical operations such as multiplication and addition not only for numbers but also for objects such as matrices, vectors or group elements. Generic programming, using template constructs in C++, allow to program at a high level of abstraction, without incurring the abstraction penalty of object oriented programming.

1.10 Which programming language should I learn?

We recommend Julia for ICP, mainly for these three reasons:

- **it combines the best features of Python and C/C++**
- **avoids the 2 language problem**
- **parallel computing is included.**
- **it is new ... ☺**

There are more technical reasons *Why Julia?* written on the slides.

Chapter 2

The Programming Language Julia

Quoting from the abstract of the paper of Bezanson et al. (2014) [1, 2], the programming language Julia questions notions generally held as "laws of nature" by practitioners of numerical computing namely:

1. High-level dynamic programs have to be slow.
2. One must prototype in one language and then rewrite in another language for speed or deployment.
3. There are parts of a system for the programmer, and other parts best left untouched as they are built by the experts.

The Julia programming language and its design can be regarded as a dance between specialization and abstraction. Specialization allows for custom treatment. Multiple dispatch, a technique from computer science, picks the right algorithm for the right circumstance. Abstraction, what good computation is really about, recognizes what remains the same after differences are stripped away. Abstractions in mathematics are captured as code through another technique from computer science, generic programming. Julia shows that one can have machine performance without sacrificing human convenience.
In the following, we aim at giving you a short introduction into the most important aspects of the Julia programming language. You can find more details in Refs. [3, 4, 5].

2.1 Julia in a Nutshell

What is Julia? Julia is a *functional programming language* released in 2012. Its creators wanted to combine the readability and simplicity of Python with the speed of statically-typed, compiled languages like C.

Should I Learn Julia? Julia is a relatively new language, Julia is still under development. This means there are more bugs and fewer native packages than you would expect from a more mature language. Established languages like Python and Java also have much larger communities. On the other hand, Julia's speed, ease of use, and suitability for big-data applications (through its high-level support for parallelism and cloud computing) have helped

it to grow quickly and it continues to attract new users steadily. Julia developers are already working at companies including Google, NASA, and Intel, and major projects like RStudio have announced plans to add support for Julia.

Compiling Julia is a compiled language, which is one of the reasons why it performs faster than interpreted languages. However, unlike traditional compiled languages, Julia is not strictly statically typed. It uses JIT (Just In Time) compilation to infer the type of each individual variable in your code. The result is a dynamically-typed language that can be run from the command line like Python, but that can achieve comparable speeds to compiled languages like C or Fortran.

Parallelism It is possible to run code in parallel in Python in order to take advantage of all of the CPU cores on your system. This requires importing modules and involves some quirks that can make concurrency difficult to work with. In contrast, Julia has top-level support for parallelism and a simple, intuitive syntax for declaring that a function should be run concurrently:

```
nheads = @parallel (+) for i = 1:10000000
    rand(Bool)
end
```

Type Checking Python is a dynamically-typed language, meaning that you declare a variable without specifying its type; the Python interpreter determines the type from the value provided (e.g. `m = 5` will be interpreted as an integer). Variables in Julia can be declared in this way as well; however, it is possible to specify types, or a range of possible types, for a variable. Specifying the expected types for a function helps the compiler optimize for better performance, and can also prevent errors resulting from unexpected or incorrect input.

Multiple Dispatch *Multiple dispatch* refers to declaring different versions of the same function to better handle input of different types. For example, you might write two different reverse functions, one that accepts an array as an argument and one that accepts a string. The Julia interpreter will check the type of the argument whenever reverse is called, and dispatch it to the version matching that type.

Array Indexing One small but significant difference between Julia and Python (along with most other modern programming languages) is that arrays in Julia are 1-indexed, meaning that you access the first element of an array with `foo[1]` rather than `foo[0]`. This choice was made to make Julia more intuitive for users of Mathematica and other technical computing tools, but can be a source of frustration (and errors) for users used to 0-indexed languages.

2.2 Technicalities

Julia's core idea is: **Multiple Dispatch + Type Stability → Speed + Readability**. Julia uses Just-In-Time (JIT) compilation (i.e. every statement is run using compiled functions which are either compiled right before they are used, or cached compilations from before) but this is not the main reason for speed. This leads to questions about what Julia gives over JIT'd implementations of Python/R etc. Julia is fast because of its design decisions. The core design decision, **type-stability through specialization via multiple-dispatch** is what allows Julia to be very easy for a compiler to make into efficient code, but also allow the code to be very concise and "look like a scripting language". This will lead to some very clear performance gains.
Type stability is the idea that there is only one possible type which can be outputted from a method. For example, the reasonable type to output from `* (::Float64,::Float64)` is a `Float64`. No matter what you give it, it will return a `Float64`. This right here is multiple-dispatch: the `*` operator calls a different method depending on the types that it sees. When it sees floats, it will return floats. Julia provides code introspection macros (`@code_llvm 2*5`) so that way you can see what your code actually compiles to. Thus, Julia is not just a scripting language, it is a scripting language which lets you deal with assembly! Julia, like many languages, compiles to LLVM (LLVM is a type of portable assembly language).
If you have type stability inside of a function (meaning, any function call within the function is also type-stable), then the compiler can know the types of the variables at every step. Therefore, it can compile the function with the full amount of optimizations since at this point the code is essentially the same as C/Fortran code. Multiple-dispatch works into this story because it means that `*` can be a type-stable function: it just means different things for different inputs. But if the compiler can know the types of `a` and `b` before calling `*`, then it knows which `*` method to use, and therefore it knows the output type of `c = a*b`. Thus, it can propogate the type information all the way down, knowing all of the types along the way, allowing for full optimiziations.

2.3 Main Differences to other Languages

The complete list of differences can be found on the official Julia documentation page. Here, we just repeat the most prominent ones among Python and C for convenience.

2.3.1 Python

- Julia's `for`, `if`, `while`, etc. blocks are terminated by the `end` keyword. Indentation level is not significant as it is in Python. Unlike Python, Julia has no `pass` keyword.

- Strings are denoted by double quotation marks (`"text"`) in Julia (with three double quotation marks for multi-line strings), whereas in Python they can be denoted either by single (`'text'`) or double quotation marks (`"text"`). Single quotation marks are used for characters in Julia (`'c'`).

- String concatenation is done with `*` in Julia, not `+` like in Python. Analogously, string repetition is done with `^` not `*`. Implicit string concatenation of string literals like in Python (e.g. `'ab' 'cd' == 'abcd'`) does not exist in Julia.

- Python Lists—flexible but slow—correspond to the Julia `Vector{Any}` type or more generally `Vector{T}` where `T` is some non-concrete element type. "Fast" arrays like Numpy arrays that store elements in-place (i.e., `dtype` is `np.float64`, `[('f1', np.uint64), ('f2', np.int32)]`, etc.) can be represented by `Array{T}` where `T` is a concrete, immutable element type. This includes built-in types like `Float64`, `Int32`, `Int64` but also more complex types like `Tuple{UInt64,Float64}` and user-defined types as well.

- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.

- Julia's slice indexing includes the last element, unlike in Python. `a[2:3]` in Julia is `a[1:3]` in Python.

- Julia does not support negative indices. In particular, the last element of a list or array is indexed with `end` in Julia, not `-1` as in Python.

- Julia requires `end` for indexing until the last element. `x[1:]` in Python is equivalent to `x[2:end]` in Julia.

- Julia's range indexing has the format of `x[start:step:stop]`, whereas Python's format is `x[start:(stop+1):step]`. Hence, `x[0:10:2]` in Python is equivalent to `x[1:2:10]` in Julia. Similarly, `x[::-1]` in Python, which refers to the reversed array, is equivalent to `x[end:-1:1]` in Julia.

- In Julia, indexing a matrix with arrays like `X[[1,2], [1,3]]` refers to a sub-matrix that contains the intersections of the first and second rows with the first and third columns. In Python, `X[[1,2], [1,3]]` refers to a vector that contains the values of cell `[1,1]` and `[2,3]` in the matrix. `X[[1,2], [1,3]]` in Julia is equivalent with `X[np.ix_([0,1],[0,2])]` in Python. `X[[0,1], [0,2]]` in Python is equivalent with `X[[CartesianIndex(1,1), CartesianIndex(2,3)]]` in Julia.

- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.

- Julia arrays are column major (Fortran ordered) whereas NumPy arrays are row major (C-ordered) by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to NumPy.

- Julia's updating operators (e.g. `+=`, `-=`, ...) are not in-place whereas NumPy's are. This means `A = [1, 1]; B = A; B += [3, 3]` doesn't change values in `A`, it rather rebinds the name `B` to the result of the right-hand side `B = B + 3`, which is a new array. For in-place operation, use `B .+= 3`, explicit loops, or `InplaceOps.jl`.

- Julia evaluates default values of function arguments every time the method is invoked, unlike in Python where the default values are evaluated only once when the function is defined. For example, the function `f(x=rand()) = x` returns a new random number every time it is invoked without argument. On the other hand, the function `g(x=[1,2]) = push!(x,3)` returns `[1,2,3]` every time it is called as `g()`.
- In Julia, keyword arguments must be passed using keywords, unlike Python in which it is usually possible to pass them positionally. Attempting to pass a keyword argument positionally alters the method signature leading to a `MethodError` or calling of the wrong method.
- In Julia `%` is the remainder operator, whereas in Python it is the modulus.
- In Julia, the commonly used `Int` type corresponds to the machine integer type (`Int32` or `Int64`), unlike in Python, where `int` is an arbitrary length integer. This means in Julia the `Int` type will overflow, such that `2^64 == 0`. If you need larger values use another appropriate type, such as `Int128`, `BigInt` or a floating point type like `Float64`.
- The imaginary unit `sqrt(-1)` is represented in Julia as `im`, not `j` as in Python.
- In Julia, the exponentiation operator is `^`, not `**` as in Python.
- Julia uses `nothing` of type `Nothing` to represent a null value, whereas Python uses `None` of type `NoneType`.
- In Julia, the standard operators over a matrix type are matrix operations, whereas, in Python, the standard operators are element-wise operations. When both `A` and `B` are matrices, `A * B` in Julia performs matrix multiplication, not element-wise multiplication as in Python. `A * B` in Julia is equivalent with `A @ B` in Python, whereas `A * B` in Python is equivalent with `A .* B` in Julia.
- The adjoint operator `'` in Julia returns an adjoint of a vector (a lazy representation of row vector), whereas the transpose operator `.T` over a vector in Python returns the original vector (non-op).
- In Julia, a function may contain multiple concrete implementations (called *Methods*), selected via multiple dispatch, whereas functions in Python have a single implementation (no polymorphism).
- There are no classes in Julia. Instead they are structures (mutable or immutable), containing data but no methods.
- Calling a method of a class in Python (`a = MyClass(x), x.func(y)`) corresponds to a function call in Julia, e.g. `a = MyStruct(x), func(x::MyStruct, y)`. In general, multiple dispatch is more flexible and powerful than the Python class system.
- Julia structures may have exactly one abstract supertype, whereas Python classes can inherit from one or more (abstract or concrete) superclasses.

- The logical Julia program structure (Packages and Modules) is independent of the file structure (`include` for additional files), whereas the Python code structure is defined by directories (Packages) and files (Modules).
- The ternary operator `x > 0 ? 1 : -1` in Julia corresponds to conditional expression in Python `1 if x > 0 else -1`.
- In Julia the `@` symbol refers to a macro, whereas in Python it refers to a decorator.
- Exception handling in Julia is done using `try`—`catch`—`finally`, instead of `try`—`except`—`finally`. In contrast to Python, it is not recommended to use exception handling as part of the normal workflow in Julia due to performance reasons.
- In Julia loops are fast, there is no need to write "vectorized" code for performance reasons.
- Be careful with non-constant global variables in Julia, especially in tight loops. Since you can write close-to-metal code in Julia (unlike Python), the effect of globals can be drastic.
- In Python, the majority of values can be used in logical contexts (e.g. `if "a":` means the following block is executed, and `if "":` means it is not). In Julia, you need explicit conversion to `Bool` (e.g. `if "a"` throws an exception). If you want to test for a non-empty string in Julia, you would explicitly write `if !isempty("")`.
- In Julia, a new local scope is introduced by most code blocks, including loops and `try`—`catch`—`finally`. Note that comprehensions (list, generator, etc.) introduce a new local scope both in Python and Julia, whereas `if` blocks do not introduce a new local scope in both languages.

2.3.2 C/C++

- Julia arrays are indexed with square brackets, and can have more than one dimension `A[i,j]`. This syntax is not just syntactic sugar for a reference to a pointer or address as in C/C++. See the manual entry about array construction.
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia arrays are not copied when assigned to another variable. After `A = B`, changing elements of `B` will modify `A` as well. Updating operators like `+=` do not operate in-place, they are equivalent to `A = A + B` which rebinds the left-hand side to the result of the right-hand side expression.
- Julia arrays are column major (Fortran ordered) whereas C/C++ arrays are row major ordered by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to C/C++.
- Julia values are not copied when assigned or passed to a function. If a function modifies an array, the changes will be visible in the caller.

- In Julia, whitespace is significant, unlike C/C++, so care must be taken when adding/removing whitespace from a Julia program.
- In Julia, literal numbers without a decimal point (such as `42`) create signed integers, of type `Int`, but literals too large to fit in the machine word size will automatically be promoted to a larger size type, such as `Int64` (if `Int` is `Int32`), `Int128`, or the arbitrarily large `BigInt` type. There are no numeric literal suffixes, such as `L`, `LL`, `U`, `UL`, `ULL` to indicate unsigned and/or signed vs. unsigned. Floating point literals are rounded (and not promoted to the `BigFloat` type) if they can not be exactly represented. Floating point literals are closer in behavior to C/C++. Octal (prefixed with `0o`) and binary (prefixed with `0b`) literals are also treated as unsigned (or `BigInt` for more than 128 bits).
- In Julia, the division operator `/` returns a floating point number when both operands are of integer type. To perform integer division, use `div`.
- Indexing an Array with floating point types is generally an error in Julia. The Julia equivalent of the C expression `a[i / 2]` is `a[div(i,2) + 1]`, where `i` is of integer type.
- String literals can be delimited with either `"` or `"""`, `"""` delimited literals can contain `"` characters without quoting it like `"""`. String literals can have values of other variables or expressions interpolated into them, indicated by `$variablename` or `$(expression)`, which evaluates the variable name or the expression in the context of the function.
- `//` indicates a `Rational` number, and not a single-line comment (which is `#` in Julia).
- `#=` indicates the start of a multiline comment, and `=#` ends it.
- Functions in Julia return values from their last expression(s) or the return keyword. Multiple values can be returned from functions and assigned as tuples, e.g. `(a, b) = myfunction()` or `a, b = myfunction()`, instead of having to pass pointers to values as one would have to do in C/C++ (i.e. `a = myfunction(&b)`.
- Julia does not require the use of semicolons to end statements. The results of expressions are not automatically printed (except at the interactive prompt, i.e. the REPL), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output. In the REPL, `;` can be used to suppress output. `;` also has a different meaning within `[ ]`, something to watch out for. `;` can be used to separate expressions on a single line, but are not strictly necessary in many cases, and are more an aid to readability.
- In Julia, the operator `xor` performs the bitwise XOR operation, i.e. `^` in C/C++. Also, the bitwise operators do not have the same precedence as C/++, so parenthesis may be required.

- Julia's `^` is exponentiation (pow), not bitwise XOR as in C/C++ (use `xor`, in Julia).
- Julia has two right-shift operators, `>>` and `>>>`. Where `>>>` performs an arithmetic shift, `>>` always performs a logical shift, unlike C/C++, where the meaning of `>>` depends on the type of the value being shifted.
- Julia's `->` creates an anonymous function, it does not access a member via a pointer.
- Julia does not require parentheses when writing if statements or for/while loops: use `for i in [1, 2, 3]` instead of `for (int i=1; i <= 3; i++)` and `if i == 1` instead of `if (i == 1)`.
- Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because if statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while`/`for`, and functions. In lieu of the one-line `if ( cond )` statement, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`, because of the operator precedence.
- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia macros operate on parsed expressions, rather than the text of the program, which allows them to perform sophisticated transformations of Julia code. Macro names start with the `@` character, and have both a function-like syntax, `@mymacro(arg1, arg2, arg3)`, and a statement-like syntax, `@mymacro arg1 arg2 arg3`. The forms are interchangeable; the function-like form is particularly useful if the macro appears within another expression, and is often clearest. The statement-like form is often used to annotate blocks, as in the distributed for construct: `@distributed for i in 1:n; #= body =#; end`. Where the end of the macro construct may be unclear, use the function-like form.
- Julia has an enumeration type, expressed using the macro `@enum(name, value1, value2, ...)`. For example: `@enum(Fruit, banana=1, apple, pear)`.
- By convention, functions that modify their arguments have a `!` at the end of the name, for example `push!`.
- In C++, by default, you have static dispatch, i.e. you need to annotate a function as virtual, in order to have dynamic dispatch. On the other hand, in Julia every method is "virtual" (although it's more general than that since methods are dispatched on every argument type, not only `this`, using the most-specific-declaration rule).

Chapter 3

Random Numbers

Random numbers (RN) are an important tool for scientific simulations. As we shall see in this class, they are used in many different applications, including the following:

- 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 walks).
- Test the stability of a system with respect to perturbations.
- Random sampling.

Additional literature about random numbers can be found in Ref. [6, 7, 8].

3.0.1 Notation and Definitions

$\mathbb{N} = 0, 1, 2, \ldots, \mathbb{N}^+ = 1, 2, 3, \ldots$ $\mathbb{Z} = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ is a subset of the set of all rational numbers $\mathbb{Q}$, which in turn is a subset of the real numbers $\mathbb{R}$. Like the natural numbers $\mathbb{N}$, $\mathbb{Z}$ are countably infinite. The boolean domain is a set consisting of exactly two elements whose interpretations include false and true and denoted by $\mathbb{B} = \{0, 1\}$.
Throughout the manuscript we will adopt the notation that closed square brackets $[\,]$ in intervals are equivalent to $\leq$ and $\geq$ and open brackets $]\,[$ correspond to $<$ and $>$ respectively. Thus the interval $[0, 1]$ corresponds to $0 \leq x \leq 1$, $x \in \mathbb{R}$ and $]0, 1]$ means $0 < x \leq 1$, $x \in \mathbb{R}$.

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

Two integers $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ are said to be co-prime if the only positive integer (factor) that divides both of them is 1 i.e. $\gcd(a, b) = 1$.

3.1 Uniform Random Numbers

3.1.1 Definition

Random numbers are a sequence $x_0, x_1, x_2, \ldots$ of numbers x_i (without loss of generality we assume $x_i \in [0, 1)$) in random or uncorrelated order. In particular, the probability that a given number occurs next in the sequence is always the same and not depending on the previous numbers, i.e. the conditional probability satisfies $p(x_{N+1} = z|x_0, \ldots, x_N) = p(x_{N+1} = z)$. Or in other words, we cannot predict the next number in the sequence.
Physical systems can produce random events, for example in electronic circuits ("electronic flicker noise") or in systems where quantum effects play an important role (radioactive decay, photon emission processes, etc.). However, even physical random numbers suffer from the fact that they are usually correlated.
The algorithmic creation of random numbers violates the assumption that the numbers are non-deterministic since a computer is completely deterministic. One therefore 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. The goal is that these numbers follow a well-defined distribution and have long periods. Furthermore, it should be possible to calculate them quickly and in a reproducible way. Reproducible means, that given a particular random number generator (RNG) [1] and the starting point of the sequence x_0—also called the *seed*—one can reconstruct the full sequence.
Generally speaking, we can divide all RNGs into one of two classes: the *multiplicative* and the *additive* ones. Typically, the multiplicative ones are simpler and faster to program and execute but do not produce very good (=uncorrelated) sequences. On the other hand, the additive ones produce much better random sequences but are more difficult to implement and take longer to run. In the following two subsections we will see an example for each of them.

3.1.2 Congruential RNG

The simplest RNG is the *congruential* RNG proposed by Lehmer in 1948. It is a multiplicative RNG and based on the properties of the modulo-operator (`%` in Julia). Let us choose $x_0, c, p \in \mathbb{N}$. Then, we can generate (pseudo-)random numbers with the formula

$$x_i = (cx_{i-1}) \mod p \tag{3.1}$$

in the range $\{0, \ldots, p-1\}$ by definition of the modulo-operation. In order to obtain random numbers in $[0, 1)$, we need to divide each number by p in the end.
Of course, the choice of x_0, c, p plays an important role on the quality of the random numbers. For example, if we choose $x_0 = 0$, we get the sequence $0, 0, 0, \ldots$ or if we choose $c = 1$, we get the sequence $x_0, x_0, x_0, \ldots$ (assuming that $x_0 < p$). Furthermore, we can note that every sequence must repeat after at least $p - 1$ iterations since p is finite and all integers x_i are smaller than p.

[1] When speaking of an RNG, we always mean pseudo-random number generator from now on.

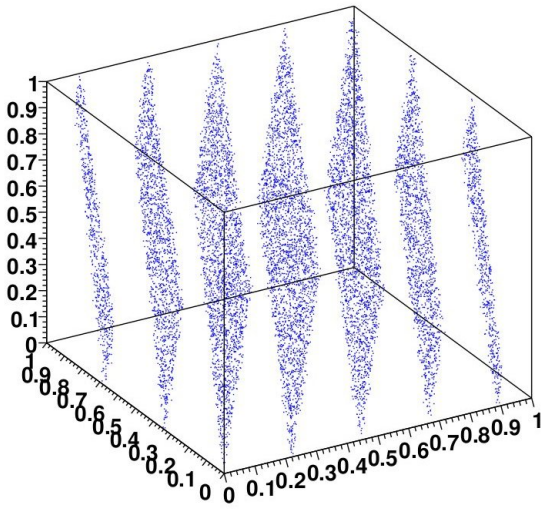


Figure 3.1: Consecutive random numbers with clearly visible hyperplanes ("RANDU" algorithm with $c = 65'539$, $p = 2^{31}$, $x_0 = 1$).

The *maximal period* $p - 1$ can be achieved if p is a Mersenne prime number, i.e. $p = 2^n - 1$, $n \in \mathbb{N}$ and p is prime, and c satisfies the condition $c^{p-1} \mod p \equiv 1$. This result was proven in 1910 by R.D. Carmichael.
In 1988, Park and Miller presented the following numbers (here in Julia code)

```
const p = 2147483647 # Mersenne prime 2^31-1
const c = 16807
x = 42               # seed
for i in 1:p-2
    x = (x*c)%p
    println(x)
end
```

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 they will form some patterns (mostly lines) depending on the chosen parameters. This pattern formation is a manifestation of the correlation between the numbers in the generated sequence. We will discuss this in more detail in Ch. 3.1.4.
Of course, it is also possible to do this kind of visualization for three consecutive numbers (x_i, x_{i+1}, x_{i+2}) (see e.g. Fig. 3.1). There is even a theorem which quantifies the patterns observed and the number of observed hyperplanes [9]. Furthermore, it is possible to show that for congruential RNGs the distance between these hyperplanes must be larger than

$$\sqrt{\frac{p}{n}}, \tag{3.2}$$

where n is the dimension of the corresponding hypercube.

c	d	
250	103	Kirkpatrick, Stoll (1981)
4'187	1'689	Heringa et al. (1992)
132'049	54'454	
6'972'592	3'037'958	Brent et al. (2003)

Table 3.1: Pairs (c, d) satisfying the Zierler Trinomial Condition.

3.1.3 Lagged Fibonacci RNG

A more complicated version of a RNG is the *Lagged Fibonacci* RNG proposed by Tausworth in 1965. It is an example for an additive RNG. Lagged Fibonacci type generators permit extremely large periods and even allow for advantageous predictions about correlations.
Consider a sequence of b (seed) bits $x_1, \ldots, x_b$ and $\mathcal{J} \subset \{1, \ldots, b\}$. Then, the next bit x_{b+1} in the sequence is calculated as

$$x_{b+1} = \left(\sum_{j \in \mathcal{J}} x_{b+1-j} \right) \mod 2. \tag{3.3}$$

For example, we consider a *two-element* Lagged Fibonacci RNG, that means that the subset $\mathcal{J}$ contains only two-elements. Let us call these numbers c and d such that $1 \leq d \leq c \leq b$. Then, we can compute a sequence of random numbers according to

$$x_{b+1} = (x_{b+1-c} + x_{b+1-d}) \mod 2. \tag{3.4}$$

Similar to the congruential RNG, it can be shown that there is a maximal period of length $2^c - 1$ and that it is obtained if the so-called *Zierler Trinomial Condition* is met, i.e. the polynomial

$$1 + x^c + x^d, \tag{3.5}$$

x being a binary number, is primitive or in other words not factorizable in subpolynomials. Table 3.1 shows examples of pairs (c, d) that satisfy the Zierler condition.
There are two methods to obtain decimal random numbers, e.g. 32 bit unsigned integers:

1. We slice the sequence in 32 bit long parts and use each as one single decimal random number. However, this method is relatively slow as for each random number one needs to generate 32 new bits. Furthermore, it has been shown that random numbers produced in this way show strong correlations.

2. We run 32 Fibonacci RNGs in parallel (which can be done very efficiently). Here, the main problem is the initialization, as the 32 initial sequences (each b bits) do not only need to be uncorrelated each one by itself but also among each other. The quality of the initial sequences has a major impact on the quality of the produced random numbers.

3.1.4 Testing

Regardless of which RNG we use, there will always be correlation between the numbers. In order to quantify the randomness of a particular generator, there is an impressive variety

of different tests to do so. Note that there does not exist any test that can with certainty determine whether a sequence of numbers is random or not. The argument is always a statistical one. For every test, we need to define a certain *confidence level* which allows us to categorize a sequence as trustworthy or not. With each test that we pass with our sequence, our confidence of having a good (enough) sequence of random numbers increases.
In the following, we give a couple of examples of tests:

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

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

- Average value: The arithmetic mean of all the numbers in the sequence $\{x_i\}$ should correspond to the analytical mean value. Let us assume here that $x_i \in [0, 1)$. The arithmetic mean should then be

$$\lim_{N\to\infty} \underbrace{\frac{1}{N}\sum_{i=1}^{N} x_i}_{\equiv s_N} \stackrel{!}{=} \frac{1}{2}. \tag{3.6}$$

  The more numbers included, the better 1/2 will be approximated.

- Fluctuation of the mean value (χ^2-test): The distribution of the s_N should behave like a Gaussian distribution around the mean value 1/2 according to the central limit theorem.

- Spectral analysis (= Fourier analysis): If we assume that the $\{x_i\}$ are the discrete 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), the randomness is good, otherwise peaks will show up (resonances).

- Correlation tests: These tests involve the analysis of correlations such as

$$\langle x_i x_{i+d} \rangle - \langle x_i^2 \rangle , \tag{3.7}$$

  for different $d \in \{1, \dots\}$.

Of course this list is not complete. There are many other tests that can be used to check the randomness of pseudo-random sequences.
Additionally, we would like to mention the very famous Marsaglia's "Diehard" tests [2]:

[2] These tests were developed over many years and published for the first time on CD-ROM by Marsaglia in 1995. They are not better or worse than the ones presented above. They have become famous though thanks to their rather creative naming.

- Birthday spacings: If random points are chosen in a large interval, the spacing between the points should be asymptotically Poisson distributed. The name stems from the birthday paradox [3].
- Overlapping permutations: When analyzing five consecutive random numbers, 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 $\{0, 1\}$. The rank of this matrix is then determined and the ranks are counted.
- Monkey test: Sequences of some number of bits are taken as words and 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 [4].
- Parking lot test: Randomly place unit circles in a 100×100 square. If a 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 $8'000$ randomly placed points in a $10'000 \times 10'000$ square. The square of this distance should be exponentially distributed with a certain mean.
- Random spheres test: Put $4'000$ randomly chosen points in a cube of edge $1'000$. 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: 2^{31} is multiplied by random floats in $[0, 1)$ until 1 is reached. After $100'000$ 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: $200'000$ games of craps (a dice game) are played. The number of wins and the number of throws per game should follow a certain distribution.

[3] 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 intuition would suggest probabilities much lower than 50%.

[4] 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.

3.2 Quasi-Random Numbers

In this section, we follow Ref. [10]. Uniform random numbers (even perfect ones) do not occupy space uniformly, i.e. there will always be certain cluster formations. For some applications this behavior is not advantageous. In Monte Carlo methods (more details in Ch. 7), we are trying to approximate an integral (sum) by randomly sampling the integrand (summand) according to a certain probability distribution instead of evaluating it over the whole domain. We will show that the expected error of such a Monte Carlo *sampling* is fairly independent of the problem dimension [5] and is of the order

$$\Delta_{\mathrm{MC}} \sim \frac{1}{\sqrt{N}}, \tag{3.8}$$

where N is the number of sample points. Random point sets generated by Monte Carlo sampling show often clusters of points and tend to take wasteful samples because of gaps in the sample space. This observation led to proposing error reduction methods by means of determinate point sets, such as *low-discrepancy sequences*. Low-discrepancy sequences try to utilize more uniformly distributed points. Application of low-discrepancy sequences to generation of sample points for Monte Carlo sampling leads to what is known as *quasi-Monte Carlo approaches*. The error bounds in quasi-Monte Carlo approaches are of the order of

$$\Delta_{\mathrm{QMC}} \sim \frac{(\log N)^d}{N}, \tag{3.9}$$

where d is the problem dimension and N is again the number of samples generated. When the number of samples is large enough, quasi-Monte Carlo methods are theoretically superior to conventional Monte Carlo sampling. The low-discrepancy sequences improve convergence and give rise to deterministic error bounds, as we will see later in more detail.

3.2.1 Discrepancy

Discrepancy is a measure of the non-uniformity of a sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ of points placed in a unary hypercube $[0,1]^d$. The most widely studied distance measure is the so-called D^**-discrepancy* [11]

$$D_N^*(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) = \sup_{\boldsymbol{v}} \left| \frac{1}{N} \sum_{i=1}^{N} \prod_{j=1}^{d} 1_{0 \le x_i^j \le v^j} - \prod_{j=1}^{d} v^j \right|. \tag{3.10}$$

Given a certain vector $\boldsymbol{v} \in [0,1]^d$, this vector spans a subbox in the unary hypercube. Then, the first term $\sum_{i=1}^{N} \prod_{j=1}^{d} 1_{0 \le x_i^j \le v^j}$ is a measure for the number of points of the sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ within this subbox. The second term $\prod_{j=1}^{d} v^j$ is the volume of the subbox. Therefore, the D^*-discrepancy is low when the spacing between the points is uniform, and high when it is not.
Based on this measure, we call our sequence a *low-discrepancy sequence* if for any $N > 1$

$$D_N^*(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) \le c(d) \frac{(\log N)^d}{N}, \tag{3.11}$$

[5] Here, we are talking about the dimension of the integration domain.

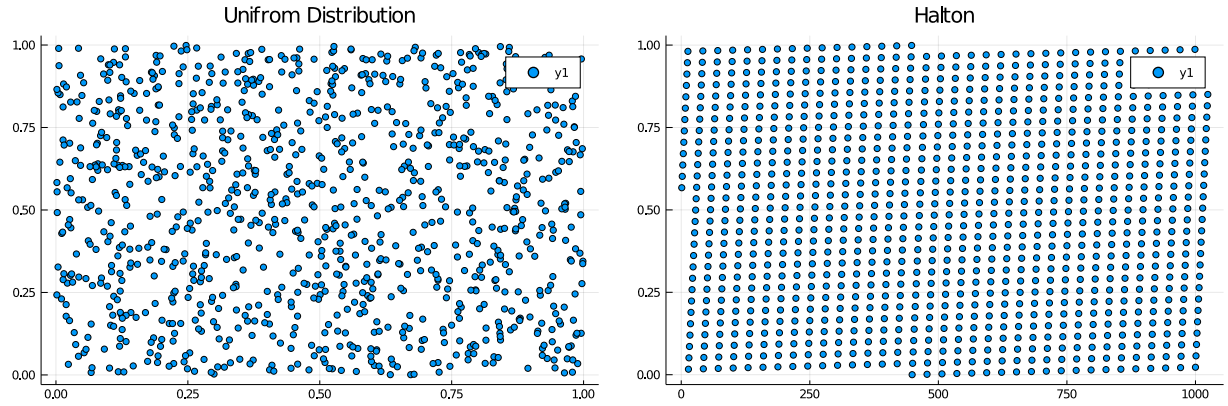


Figure 3.2: Uniform random and Halton sequence.

where the constant $c(d)$ depends only on the problem dimension d. The idea behind the low-discrepancy sequences is to let the fraction of the points within any subbox spanned by $\boldsymbol{v}$ be as close as possible to its volume. As a consequence, the low-discrepancy sequences will spread over $[0,1]^d$ as uniformly as possible, reducing gaps and clustering of points.
Figure 3.2 shows two-dimensional projections of a uniform random sequence and of a low-discrepancy, quasi-random sequence to demonstrate the fundamental difference between the two classes of sequences.

3.2.2 Halton Sequence

One example for a low-discrepancy sequence is the *Halton sequence*. Let $p_1, \ldots, p_d$ be a sequence of consecutive prime numbers. The d-dimensional Halton sequence is then defined by

$$\boldsymbol{x}_i = (\varphi_{p_1}(i), \ldots, \varphi_{p_d}(i)), \tag{3.12}$$

where $\varphi_p(i)$ is the radical inverse function

$$\varphi_p(i) = \sum_{n=0}^{l(i)} c_{i,n} p^{-n-1}, \tag{3.13}$$

$c_{i,n}$ the coefficients appearing in the expansion of the number i in the base of p

$$i = \sum_{n=0}^{l(i)} c_{i,n} p^{n} \tag{3.14}$$

and $l(i) = \lceil \log_p(i) \rceil$ the length of this expansion. In words, to obtain the j-th component of the i-th number in the Halton sequence, we first compute the expansion of the number i in the base p_j, invert the order of the coefficients and shift them to the right side of the decimal point.
In Julia, Halton sequences can be generated by making use of the package `HaltonSequences.jl`. A `d = 3`-dimensional sequence of length `N = 500` can be created like this

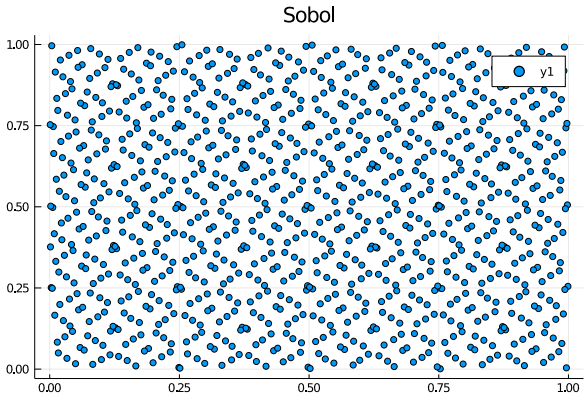


Figure 3.3: Sobol sequence.

```
using HaltonSequences # Halton
using Primes          # nextprime

N = 500
d = 3
H = zeros(N, d)       # container for the sequence
p = 29
offset = 1000
for i in 1:d
    H[:, i] = Halton(nextprime(p, i), start=offset, length=N)
end
```

where we choose to throw away the first `offset = 1000` numbers in the sequence. Finally, we would like to note that many more quasi random number sequences exist. Figure 3.3 shows, for example, the Sobol sequence.

3.2.3 Quasi-Monte Carlo Methods

Let us assume, we want to estimate the integral

$$I = \int_{[0,1]^d} f(\boldsymbol{x}) d^d x. \tag{3.15}$$

With (conventional) Monte Carlo sampling, we first generate a uniform random sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ of independent points in $[0,1]^d$. This allows us to approximate the integral as

$$I \approx \frac{1}{N} \sum_{i=1}^{N} f(\boldsymbol{x}_i). \tag{3.16}$$

The error bound is probabilistic with order $\mathcal{O}(N^{-1/2})$. More information on this will be given in Ch. 7.

Now, in quasi-Monte Carlo methods, we use a low-discrepancy sequence $x_1, \ldots, x_N$ to estimate I. The integration accuracy for quasi-Monte Carlo methods relates to the D^*-discrepancy by the Koksma-Hlawka inequality [12]

$$\left| I - \frac{1}{N} \sum_{i=1}^{N} f(\boldsymbol{x}_i) \right| \leq V(f) D^*(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) \leq V(f) c(d) \frac{(\log(N))^d}{N}, \tag{3.17}$$

where $V(f) < \infty$ is the variation of f in the sense of Hardy and Krause. Important to see is that with an increase in N quasi-Monte Carlo methods may offer better convergence rates than achievable with conventional Monte Carlo sampling. Another advantage of quasi Monte Carlo methods is that we obtain deterministic error bounds $\mathcal{O}(\frac{(\log N)^d}{N})$.

3.3 Non-Uniform Random Numbers

So far, we have only considered the generation of uniformly distributed random numbers. For this, we have seen two examples, the congruential RNG and the lagged Fibonacci RNG. However, if the goal is to produce random numbers which are distributed according to a certain—in particular—non-uniform probability distribution, e.g. Gaussian, the algorithms presented until now are not very well suited. There are essentially two different ways to perform this transformation,

- the transformation method and
- the rejection method.

Both methods are explained in the following subsections.

3.3.1 Transformation Method

For a certain class of probability distributions, it is possible to create these random numbers from a uniformly distributed random sequence by applying a certain mathematical transformation. The transformation method works particularly nicely for the most common distributions (e.g. exponential distribution, normal distribution, etc.). While the transformation is rather straightforward, it is not always feasible - this depends on the analytical description of the corresponding probability distribution.
Let us assume that we can draw random numbers according to the uniform probability distribution $p_{\mathrm{u}}(x)$ [6] (within the interval $[0, 1)$), where

$$p_{\mathrm{u}}(x) = \begin{cases} 1, & x \in [0, 1) \\ 0, & \text{else} \end{cases} . \tag{3.18}$$

With this probability density function, the probability to get a random number between 0 and x is given by

$$x = \int_0^x p_{\mathrm{u}}(y) dy. \tag{3.19}$$

[6] Formally speaking, p_{u} is a probability density function. We generally use both terms interchangeably.

Now, by means of an integral transformation, we can write

$$x = \int_0^x p_{\mathrm{u}}(y)dy \stackrel{!}{=} \int_0^{x'} p_{\mathrm{nu}}(z)dz \equiv I(x'), \tag{3.20}$$

where p_{nu} is the desired non-uniform probability distribution. Thus, if

1. the corresponding integral transformation does exist,
2. the integral $I(x')$ can be solved analytically in a closed form, and
3. the analytical inverse of the equation $x = I(x')$ exists,

we can draw uniform random numbers $x_1, \ldots, x_N$ and get a sequence of non-uniform random numbers according to p_{nu} via $x'_1 = I^{-1}(x_1), \ldots, x'_N = I^{-1}(x_N)$ [7].
In the following, we are going to demonstrate this method for two commonly used distributions: the exponential distribution and the normal distribution. Already in the case of the Gaussian distribution, we are going to see that quite a bit of work is required to derive the appropriate transformation.

Exponential Distribution

The *exponential distribution* is defined as

$$p_{\mathrm{exp}}(z) = ke^{-kz}, \tag{3.21}$$

where we can simply check that it is a properly normalized distribution function on $[0, \infty)$ by calculating

$$\int_0^\infty p_{\mathrm{exp}}(z)dz = \left[-e^{-kz}\right]_0^\infty = 1. \tag{3.22}$$

By applying Eq. (3.20), we find

$$x = \int_0^{x'} p_{\mathrm{exp}}(z)dz = \left[-e^{-kz}\right]_0^{x'} = 1 - e^{-kx'} \tag{3.23}$$

and by inverting it arrive at the final transformation

$$x' = -\frac{1}{k}\log(1-x). \tag{3.24}$$

Normal Distribution

The *normal (=Gaussian) distribution* is defined on $(-\infty, +\infty)$ as

$$p_{\mathrm{n}}(z) = \frac{1}{\sqrt{\pi c}} e^{-\frac{z^2}{c}}, \tag{3.25}$$

[7] Of course, these conditions can be overcome to a certain extent by precalculating/tabulating and inverting $I(x')$ numerically, as long as the integral is well-behaved (i.e. in particular non-singular).

where $c = 2\sigma^2$ contains the standard deviation σ. Since the integral [Eq. (3.20)]

$$x = \int_0^{x'} p_{\mathrm{n}}(z)dz \tag{3.26}$$

cannot be solved analytically—except for the limit $x' \to \infty$, where the result is $\sqrt{\pi}/2$—we follow the procedure outlined by Box and Muller [13].
Let us assume we take two (uncorrelated) uniform random variables x and y. Of course, we can apply the area equality of Eq. (3.20) again but this time we write it as a product of the two random variables

$$xy = \int_0^{x'} \int_0^{y'} \frac{1}{\pi c} e^{-\frac{z_1^2+z_2^2}{c}} dz_1 dz_2. \tag{3.27}$$

Changing to polar coordinates [8] with $r^2 = z_1^2 + z_2^2$ and $dz_1 dz_2 = r dr d\varphi$, this leads to

$$\begin{aligned} xy &= \frac{1}{\pi c} \int_0^{r'} \int_0^{\varphi'} e^{-\frac{r^2}{c}} r dr d\varphi && (3.28)\\ &= \frac{\varphi'}{\pi c} \int_0^{r'} e^{-\frac{r^2}{c}} r dr && (3.29)\\ &= \frac{\varphi'}{\pi c} \left[-\frac{c}{2} e^{-\frac{r^2}{c}} \right]_0^{r'} && (3.30)\\ &= \frac{\varphi'}{2\pi} \left(1 - e^{-\frac{r'^2}{c}} \right). && (3.31) \end{aligned}$$

We can resubstitue the polar coordinates

$$\begin{aligned} r'^2 &= x'^2 + y'^2 && (3.32)\\ \varphi' &= \arctan\left(\frac{y'}{x'}\right) && (3.33) \end{aligned}$$

and separate Eq. (3.31) into

$$\begin{aligned} x &= \frac{1}{2\pi} \arctan\left(\frac{y'}{x'}\right) && (3.34)\\ y &= 1 - e^{-\frac{x'^2+y'^2}{c}}. && (3.35) \end{aligned}$$

Inverting these equations leads to

$$\begin{aligned} \frac{y'}{x'} &= \tan(2\pi x) && (3.36)\\ x'^2 + y'^2 &= -c \log(1-y) && (3.37) \end{aligned}$$

which is solved by

$$\begin{aligned} x' &= \sqrt{-c \log(1-y)} \cos(2\pi x) && (3.38)\\ y' &= \sqrt{-c \log(1-y)} \sin(2\pi x). && (3.39) \end{aligned}$$

Using the Box-Muller transformation, we obtain two normally distributed random numbers x' and y' from the two uniformly distributed random numbers x and y.

[8] Here, we use the convention $z_1 = r\cos(\varphi)$, $z_2 = r\sin(\varphi)$.

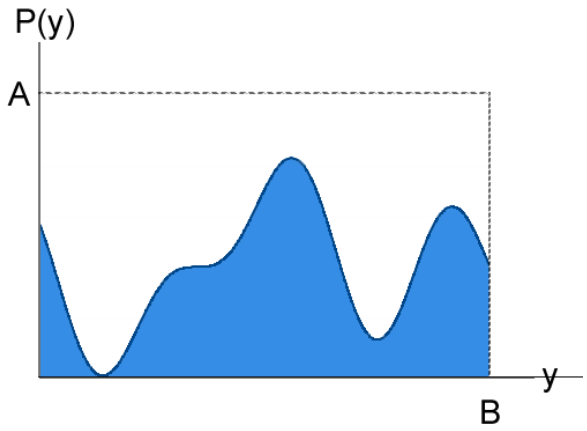


Figure 3.4: Visualization of the rejection method. Here, with $X \equiv B$ and $Y \equiv A$.

3.3.2 Rejection Method

As discussed earlier, certain conditions need to be fulfilled in order to be able to apply the transformation method. If either of these conditions is not met, there does not exist an analytical method to obtain random numbers in this probability distribution. It is important to note that this is particularly relevant for experimentally obtained data (or other sources), where no analytical description is available. In that case, one has to resort to a numerical method which is called the *rejection method*.
Let p be the probability distribution according to which we would like to obtain a sequence of random numbers. Let us assume furthermore that p is well-behaved, i.e. here, in particular, that it is finite over the domain of interest. More formally, we define a box of size $X \times Y$ such that $p(x) < Y, \forall x \in [0, X)$, see Fig. 3.4. This allows us to draw two uniformly distributed random numbers x and y in $[0, 1)$ and in case that the point (Xx, Yy)—now uniformly distributed in $[0, X) \times [0, Y)$—is below the curve of p, i.e. $p(Xx) < Yy$, then we accept the point. Otherwise, we reject it; hence the name rejection method. The accepted numbers will be distributed according to the desired probability distribution p.
In principle, the method works 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 Y. Obviously, the better the guess, the less points need to be rejected. In the above description of the algorithm we have assumed a rectangular box. This is however not a necessary condition. The bound can be any distribution for which random numbers are easily generated. To mention one variant: We use N boxes to cover p and define the individual boxes with side lengths X_i and Y_i for $1 \leq i \leq N$ (related to the idea of the Riemann integral).

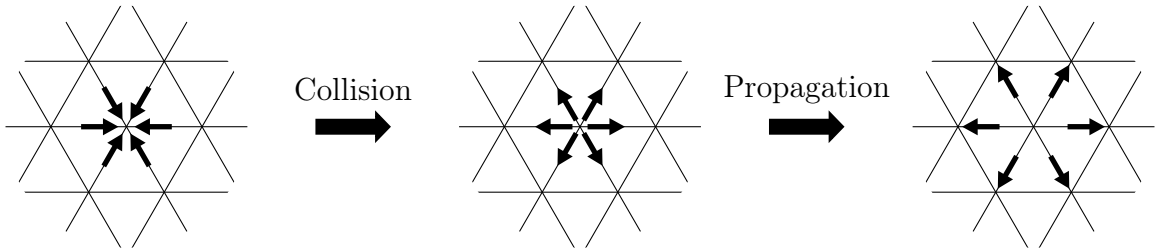
- In practice it is often faster to use a numerical variant of the transformation method as briefly mentioned above.

Chapter 4

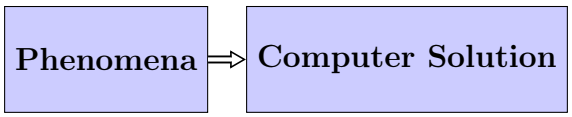
Cellular Automata

The theory of cellular automata provides a natural framework for modelling physical systems which are composed of (short-ranged) interacting components, so-called complex systems. These systems are generally difficult to study and often exhibit interesting spatio-temporal patterns and collective behaviors. Cellular automata offer a possibility to study such systems by being themselves simple, fully discrete complex systems. The reason for their success is the close relation between their framework and a mesoscopic abstraction of the corresponding natural phenomenon.
To illustrate this, we consider the example of modelling the particle dynamics in a fluid. Within an Eulerian formulation of fluid mechanics, we consider the fluid as a continuum, subject to the conservation laws for mass, momentum and energy as well as to the state equations connecting the macroscopic variables that define the thermodynamic states of the fluid, namely pressure, density and temperature. In such a situation, the fluid can be mathematically described by a set of partial differential equation, the so-called Navier-Stokes equations. Since in general analytical solutions—except in some rare cases—are not available, the dynamics of such a system needs to be studied numerically.
From the perspective of the Navier-Stokes formulation, the straightforward way would be a discretization of the set of partial differential equations:

However, we could instead also use a cellular automaton model—here on a triangular lattice—to simulate the same system:



This would allow us to arrive at a solution scheme of the following form:



4.1 Definition

A *cellular automaton* is a tuple $(\mathcal{L}, \boldsymbol{\psi}, \mathcal{N}, \mathcal{R})$, where

- $\mathcal{L}$ is a lattice of sites $\boldsymbol{i}$ in d-dimensional space,
- $\boldsymbol{\psi}(\boldsymbol{i}, t) = \begin{pmatrix} \psi_1(\boldsymbol{i}, t) & \dots & \psi_m(\boldsymbol{i}, t) \end{pmatrix} \in \{0, 1\}^m$ is the state of site $\boldsymbol{i}$ at time t,
- $\mathcal{N}_{\boldsymbol{i}}$ is a finite neighborhood (i.e. a set of sites) around site $\boldsymbol{i}$—including $\boldsymbol{i}$ itself—and
- $\mathcal{R} = \{R_1, \dots, R_m\}$ is the set of rules which specify the time evolution of the states $\{\boldsymbol{\psi}(\boldsymbol{i}, t)\}_{\boldsymbol{i}}$ in the following way

$$\psi_k(\boldsymbol{i}, t+1) = R_k\left(\{\boldsymbol{\psi}(\boldsymbol{j}, t)\}_{\boldsymbol{j} \in \mathcal{N}_{\boldsymbol{i}}}\right). \tag{4.1}$$

According to this definition, there are 2^m possible states per lattice site and therefore

$$(2^m)^{\left[(2^m)^{|\mathcal{N}|}\right]}, \tag{4.2}$$

where $|\mathcal{N}|$ is the number of sites in the neighborhood $\mathcal{N}$, possible sets of rules in total. Furthermore, we note that the rules are identical for all sites and also applied simultaneously to all sites leading to a synchronous dynamics.
In principle, the definition of a cellular automaton could be generalized to spatial or even temporal inhomogeneities. For example, this is very common when boundary cells are implemented and systematically fixed at a certain value. Similarly, it is easy to alternate between two rules by having an additional bit which is 0 at even time steps and 1 at odd time steps. According to our definition, a cellular automaton is fully *deterministic*. Given a certain initial configuration, the time evolution will always be identical. However, it may be convenient for some applications to have a certain degree of randomness in the rule. For instance, it may be desirable that a rule selects one outcome among several possible states with a probability p. Cellular automata whose updating rules are driven by some external probabilities are called *probabilistic*.
In the following we will only consider deterministic cellular automatons on a square lattice. In such situations there are two commonly appearing neighborhoods, the *von Neumann* and the *Moore* neighborhood, see Fig. 4.1 for a $d = 2$-dimensional lattice.
In practice, when simulating a given cellular automaton, one cannot deal with an infinite lattice. The system must be finite and have boundaries. Clearly, a site belonging to the lattice boundary does not have the same neighbourhood as other internal sites. In order to define the behavior of these sites, a different evolution rule can be considered, which "sees" the appropriate neighborhood. This means that the information of being a boundary cell or not can be coded on-site using a particular value of ψ_k for a chosen k. Following this approach, it is possible to define several types of boundaries, all with a different behavior.

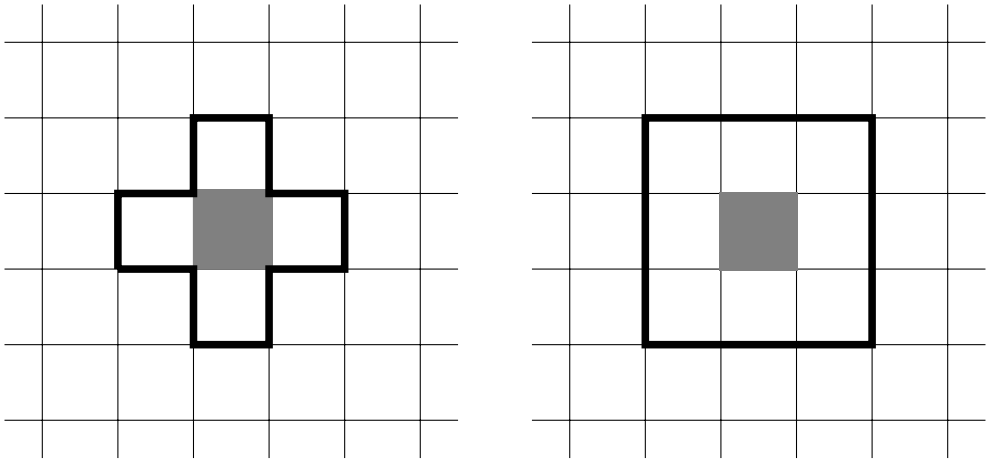


Figure 4.1: (Left) 2D von Neumann neighborhood with $|\mathcal{N}| = 5$ and (right) 2D Moore neighborhood with $|\mathcal{N}| = 9$. The shaded region indicates the central cell which is updated according to the state of the cells located within the domain marked with the bold line.

Instead of having a different rule at the limits of the system, another possibility is to extend the neighborhood for the sites at the boundary. For instance, a very common solution is to assume *periodic* (=cyclic) boundary conditions, that is one supposes that the lattice is embedded in a torus-like topology. In the case of a two-dimensional lattice, this means that the left and right sides are connected, and so are the upper and lower sides. Other possible types of boundary conditions are illustrated in Fig. 4.2 in the case of a one-dimensional lattice. We assume that the lattice is augmented by a set of virtual cells beyond its limits. A *fixed* boundary is defined so that the neighborhood is completed with cells having a pre-assigned value. An *adiabatic* (=zero gradient) boundary condition is obtained by duplicating the value of the site to the extra virtual cells. A *reflecting* boundary amounts to copying the value of the other neighbor in the virtual cell.

4.2 One-Dimensional Cellular Automata

The simplest cellular automata are the $d = 1$-dimensional ones for which each site has only two possible states and the rule involves only the nearest-neighboring sites. A systematic study of these rules was undertaken by S. Wolfram in 1983.
Here, we consider the cellular automata on a one-dimensional lattice of sites with states $\psi(i) \in \{0, 1\}$ and neighborhood $\mathcal{N}_i = \{i - 1, i, i + 1\}$. According to Eq. (4.2), we have $2^{2^3} = 256$ possible rules, each of them defining a separate cellular automaton typically labeled according to the so-called *Wolfram code*. Every rule is a well-defined function that maps the combination $(\psi_{i-1}(t), \psi_i(t), \psi_{i+1}(t)) \in \{0, 1\}^3$ to the new state $\psi_i(t + 1) \in \{0, 1\}$. Therefore, every rule can be specified by 8 outcome-bits $\{\alpha_k\}_{k\in\{0,\dots,7\}}$ that form together the rule name R_c given by

$$c = \sum_{k=0}^{7} \alpha_k 2^k. \tag{4.3}$$

The easiest way is to illustrate this by some simple examples. Consider, for example, rule R_{21}.

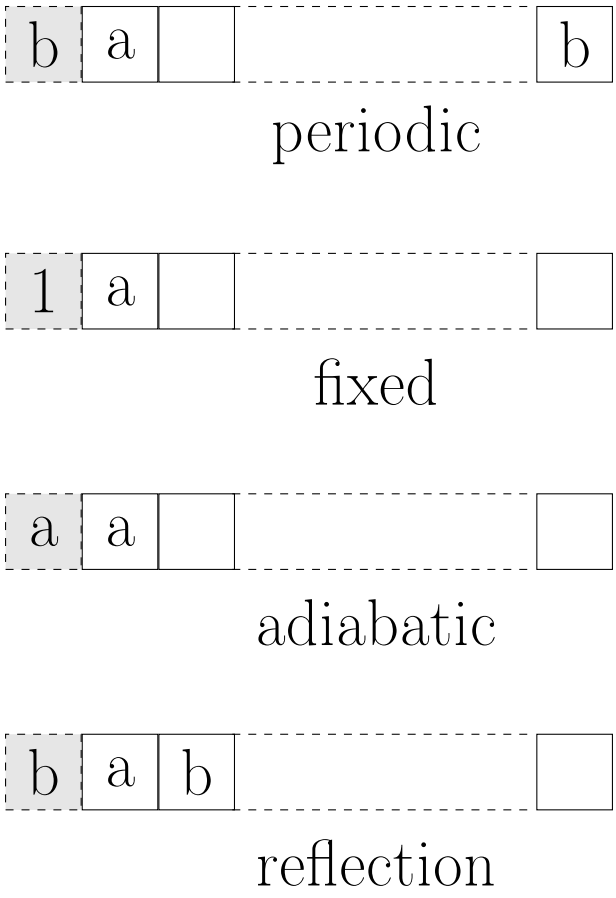


Figure 4.2: Various types of boundary conditions obtained by extending the neighborhood. The shaded block represents a virtual cell which is added at the extremity (here, the left one) of the lattice to complete the neighborhood.

Translating $(21)_{10}$ to binary results in $(00010101)_2$, where each bit gives information about the rule mapping. The first bit 1 corresponding to $(2^0)_{10} = (000)_2$ states that $(\psi_{i-1}(t) = 0, \psi_i(t) = 0, \psi_{i+1}(t) = 0)$ is mapped to $\psi_i(t+1) = 1$, the second bit 0 corresponding to $(2^1)_{10} = (001)_2$ states that $(\psi_{i-1}(t) = 0, \psi_i(t) = 0, \psi_{i+1}(t) = 1)$ is mapped to $\psi_i(t+1) = 0$, etc. until we end up with the completely specified rule

$$
\begin{aligned}
000 &\to 1 \\
001 &\to 0 \\
010 &\to 1 \\
011 &\to 0 \\
100 &\to 1 \\
101 &\to 0 \\
110 &\to 0 \\
111 &\to 0.
\end{aligned}
$$

Conversely, consider another rule defined by

$$\begin{aligned}
000 &\to 1\\
001 &\to 1\\
010 &\to 0\\
011 &\to 0\\
100 &\to 0\\
101 &\to 0\\
110 &\to 0\\
111 &\to 1.
\end{aligned}$$

Interpreting the outcomes as the 8 bits of a single number yields $(10000011)_2 = (131)_{10}$, labelling rule R_{131}. More examples are given in the following table:

entries:	111	110	101	100	011	010	001	000
R_4:	0	0	0	0	0	1	0	0
R_8:	0	0	0	0	1	0	0	0
R_{20}:	0	0	0	1	0	1	0	0
R_{28}:	0	0	0	1	1	1	0	0
R_{90}:	0	1	0	1	1	0	1	0

4.2.1 Classification of the 1D Wolfram Automata

The 256 cellular automata were divided by Wolfram into four groups according to their evolution in time:

- Class 1: Almost all initial patterns evolve quickly into a stable, homogeneous state. Any randomness in the initial pattern disappears. The set of exceptional initial configurations which behave differently is of measure zero when the number of cells N goes to infinity. An example is given by rule 40, see Fig. 4.3 (a). From the point of view of dynamical systems, these automata evolve towards a simple limit point in the phase space.

- Class 2: Almost all initial patterns evolve quickly into stable or oscillating structures. Some of the randomness in the initial pattern may be filtered out, but some remains. Local changes to the initial pattern tend to remain local. The simple structures generated are either stable or periodic with small periods. An example is given by the rule 56, see Fig. 4.3 (b). Here again, some particular initial states (set of measure zero) can lead to unbounded growth. The evolution of these automata is analogous to the evolution of some continuous dynamical systems to limit cycles.

- Class 3: Nearly all initial patterns evolve in a pseudo-random, aperiodic or chaotic manner. Any stable structures that appear are quickly destroyed by the surrounding noise. Local changes to the initial pattern tend to spread indefinitely. An example is given by the rule 18, see Fig. 4.3 (c). Small changes in the initial conditions almost always lead to increasingly large changes in the later stages. The evolution of these

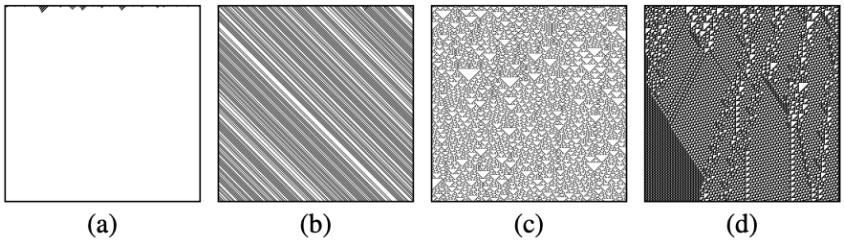


Figure 4.3: Example of the four Wolfram rules with a random initial configuration. Horizontal lines correspond to consecutive iterations. The initial state is the uppermost line. (a) Rule 40 belonging to class 1 reaches very quickly a fixed point (stable configuration). (b) Rule 56 of class 2 reaches a pattern composed of stripes which move from left to right. (c) Rule 18 is in class 3 and exhibits a self-similar pattern. (d) Rule 110 is an example of a class 4 cellular automaton. Its behavior is not predictable and as a consequence, we observe a rupture in the pattern, on the left part.

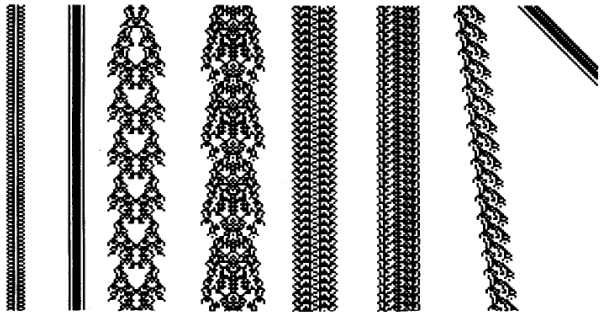


Figure 4.4: Examples of persistent structures.

automata is analogous to the evolution of some continuous dynamical systems to strange attractors.

- Class 4: Nearly all initial patterns evolve into structures that interact in complex and interesting ways. Eventually a class 4 may become a class 2 but the time necessary to reach that point is very large. An example is given by rule 110, see Fig. 4.3 (d). The behavior of such cellular automata can generally only be determined by explicit simulation of their time evolution.

Some results can be deduced analytically using algebraic techniques but most of the conclusions follow from numerical iterations of the rules. One can start from a simple initial state (i.e. only one cell in state 1) or with a random initial state.
More examples, for potentially appearing persistent structures are found in Fig. 4.4.
Although very simple in construction, we see that the toy rules considered by Wolfram are capable of very complex behavior. The validity of this classification is not restricted to the simple rules described above but is somehow generic for more complicated rules (including more states per site or different neighborhoods). Several cases have been studied in the literature and the different rules can be classified in one of the four above classes. Note

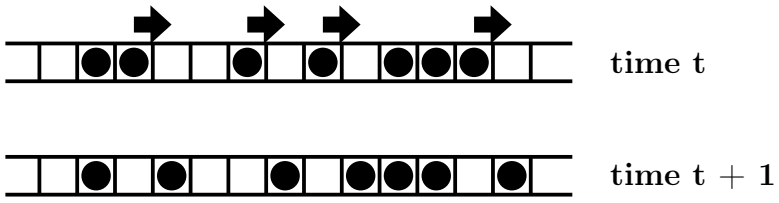


Figure 4.5: One time step for rule 184.

however, that the above "phenomenological" classification suffers drawbacks, the most serious of which is its non-decidability. For the Wolfram rules defined above, it is undecidable whether all the finite configurations of a given cellular automaton eventually become quiescent and consequently undecidable to which class a given automaton really belongs. Interestingly, many of the class 4 cellular automata have the property of computational universality and initial configurations can specify arbitrary algorithmic procedures.

4.2.2 Simple Traffic Model

Cellular automata models for road traffic have received a great deal of interest. For single lane car motion such cellular automata are quite simple. We can represent the road as a line of cells, each of them being occupied or not by a vehicle. All cars travel in the same direction (say to the right). Their positions are updated synchronously. During the motion, each car can be at rest or jump to the nearest neighbouring site along the direction of motion. The rule is simply that a car moves only if its destination cell is empty. This means that the drivers do not know whether the car in front will move or is blocked by another car. Therefore, the state ψ_i of each cell is entirely determined by the occupancy of the cell itself and that of its two nearest neighbors ψ_{i-1} and ψ_{i+1}. The rule describing the correct motion is R_{184}

$$(\psi_{i-1}, \psi_i, \psi_{i+1})_t \to (\psi_i)_{t+1} \tag{4.4}$$

	111	110	101	100	011	010	001	000
c = 184	1	0	1	1	1	0	0	0

and is illustrated in Fig. 4.5.
You might wonder whether such simple dynamics can capture any interesting features of real car motion? An the answer is yes, for example traffic congestion. Suppose we have a low car density ρ in the system, for instance something like

$$\ldots 0010000010010000010 \ldots \tag{4.5}$$

This is a free traffic regime in which all the cars are able to move. The average velocity $\langle v \rangle$ defined as the number of moving cars divided by the total number of cars in the system is then

$$\langle v_{\text{free}} \rangle = 1. \tag{4.6}$$

On the other hand, in a high density configuration such as

$$\ldots 110101110101101110 \ldots \tag{4.7}$$

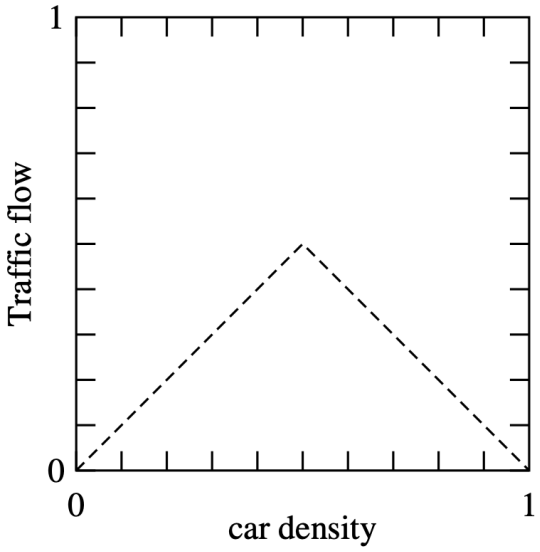


Figure 4.6: Traffic flow diagram for the simple traffic rule 184.

only 6 cars out of 12 will move in the particular example. This is a partially jammed regime. More generally, in the high density regime the number of moving cars is proportional to the number of free cells $(1-\rho)$ and therefore

$$\langle v_{\text{dense}} \rangle = \frac{1-\rho}{\rho}. \tag{4.8}$$

From the above relations we can compute the so-called fundamental flow diagram, i.e. the relation between the flow of cars $\rho \langle v \rangle$ as a function of the car density ρ: For $\rho < 1/2$, we use the free regime expression and $\rho \langle v \rangle = \rho$. For densities $\rho > 1/2$, we use the expression for the traffic jam and $\rho \langle v \rangle = 1 - \rho$. The resulting diagram is shown in Fig. 4.6. As in real traffic, we observe that the car flow reaches a maximum value before decreasing.

4.3 Examples in Two Dimensions

4.3.1 Fredkin's Self Replicator

One of the simplest examples of a two-dimensional cellular automaton giving surprisingly rich behavior despite its simplicity is Fredkin's Self Replicator introduced in the 1970s.
We consider a square lattice, where each site can either take the value 0 or 1. The rule is simply given by

$$\psi(\boldsymbol{i}, t+1) = \left(\sum_{\boldsymbol{j}}^{nn} \psi(\boldsymbol{j}, t) \right) \mod 2, \tag{4.9}$$

where the sum is over all nearest neighbors of $\boldsymbol{i}$ (von Neumann neighborhood).
When this rule is applied iteratively, very nice geometric patterns are observed, as shown in Fig. 4.7. Here, complexity results from some spatial organization which builds up as the rule

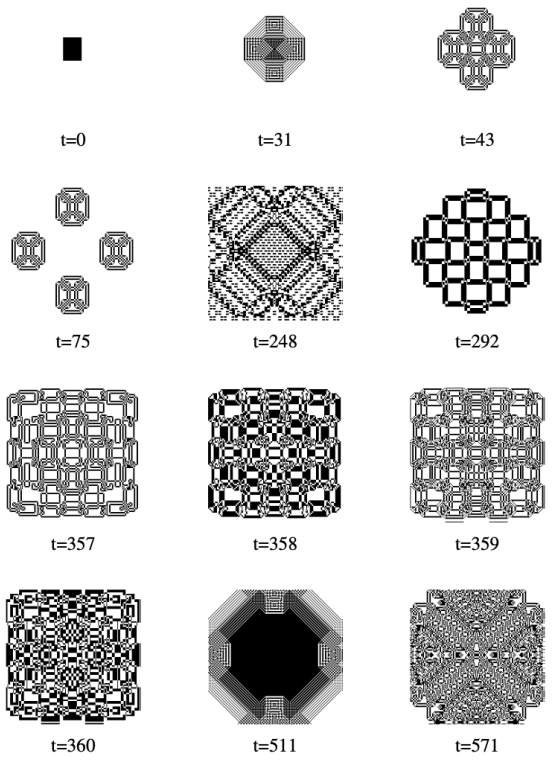


Figure 4.7: Snapshot of the parity rule on a 196×196 periodic lattice. The upper left image correspond to an initial configuration made up of a rectangle of 27×34 cells in state 1.

is iterated. The various contributions of successive iterations combine together in a specific way. The spatial patterns that are observed reflect how the terms are combined algebraically.

4.3.2 Conway's Game of Life

Also in the 1970s, the mathematician J. Conway proposed the now famous Game of Life. The motivation was to find a simple rule leading to complex behaviors in a system of fictitious one-cell organisms evolving in a fully discrete universe. Originally, it is defined on a two-dimensional square lattice in which each spatial cell can be either occupied by a living organism or empty. Here, the surrounding cells correspond to a Moore neighborhood. It turns out that the Game of Life automaton has an unexpectedly rich behavior as complex structures emerge out of a primitive "soup" and evolve or die. The rule is given by

$$\psi(\boldsymbol{i}, t+1) = \begin{cases} 0 & n = 0, 1 \text{ (isolation)} \\ \psi(\boldsymbol{i}, t) & n = 2 \\ 1 & n = 3 \text{ (birth)} \\ 0 & n = 4, \ldots \text{ (overpopulation)}, \end{cases} \tag{4.10}$$

where

$$n = \sum_{\boldsymbol{j}}^{nn} \psi(\boldsymbol{j}, t). \tag{4.11}$$

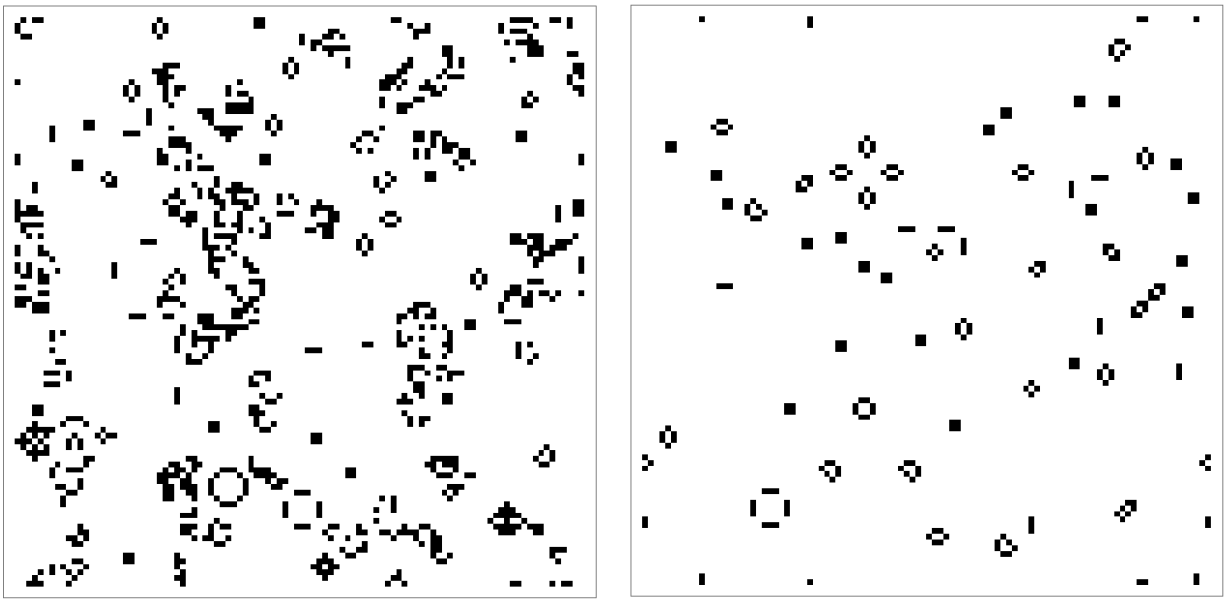


Figure 4.8: Steps n and ∞ of the Game of Life.

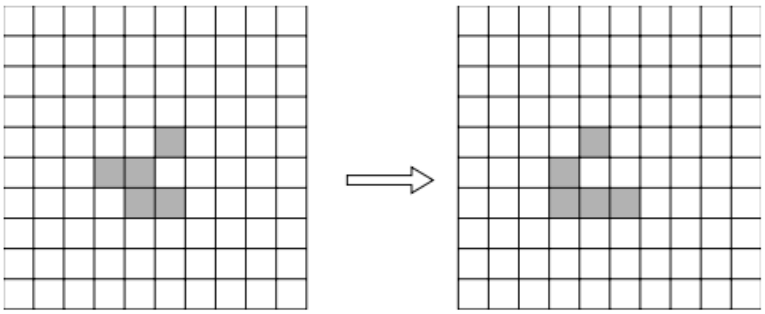


Figure 4.9: The detailed structure of a glider over two consecutive iterations. A glider is an assembly of cells that has a higher functionality than its constituent, namely the capability to move in space by changing its internal structure in a periodic way.

Snapshots of the (time) evolution can be found in Fig. 4.8. One might ask: What exactly does "evolution" mean? For example, the property to move across the lattice. Such an object, called glider, can be seen as a higher level organism because it is composed of several simple elementary cells. Its detailed structure is shown in Fig. 4.9. Thus, by assembling in a clever way cells that are unable to move, it is possible to produce, at a larger scale, a new capability. This is a signature of complex systems. Of course, more complex objects can be built, such a as for instance glider guns which are arrangements of cell producing gliders rhythmically. An illustration of such a glider gun can be found on http://upload.wikimedia.org/wikipedia/commons/e/e5/Gospers_glider_gun.gif (Wikipedia, Game of Life). The animation keeps producing "shots" from a "gun".
Finally, it can be noted that the Game of Life is a cellular automaton capable of universal computations: It is always possible to find an initial configuration of the cellular space re-

producing the behavior of any electronic gate and, thus, to mimic any computation process. Although this observation has little practical interest, it is very important from a theoretical point of view since it assesses the ability of cellular automata to be a non restrictive computational technique. For example, the Game of Life has been used to compute prime numbers.

4.3.3 Langton's Ant

As shown with the last example, cellular automata exemplify the fact that a collective behavior can emerge out of the sum of many, simply interacting, components. Even if the basic and local interactions are perfectly known, it is possible that the global behavior obeys new laws that are not obviously extrapolated from the individual properties. The example of Langton's ant (C. Langton and G. Turk) further illustrate this aspect.
The ant moves on a square lattice whose sites are either white or gray. The rule is defined in the following way (see Fig. 4.10)

1. When the ant enters a white cell, it turns 90 degrees to the left and paints the cell in gray.

2. When the ant enters a gray cell, it turns 90 degrees to the right and paints the cell in white.

It turns out that the motion of this ant exhibits a very complex behavior. Suppose the ant starts in a completely white space. After a series of about 500 steps, where it essentially keeps returning to its initial position, it enters a chaotic phase during which its motion is unpredictable. Then, after about 10′000 steps of this very irregular motion, the ant suddenly performs a very regular motion which brings it far away from where it started, see Fig. 4.11. The path the ant creates to escape the chaotic initial region has been called a highway. Although this highway is oriented at 45 degrees with respect to the lattice direction, it is traveled by the ant in a way similar to a sewing machine: The pattern is a sequence of exactly 104 steps which are repeated indefinitely.
Langton's ant is another good example of a cellular automaton whose rule is very simple and yet generates a complex behavior which seems beyond our understanding. Somehow, this fact is typical of the cellular automata approach: Although we do know everything about the fundamental laws governing a system (because we set up the rules ourselves!), we are often unable to explain its macroscopic behavior. There is anyway a global property of the ant motion: The ant visits an unbounded region of space, whatever the initial space texture is (configuration of gray and white cells). The proof of this statement (by Bunimovitch and Troubetzkoy) is solely based on the symmetry properties of the rule. Although we are not able to predict the detailed motion of the ant analytically (the only way is to perform the simulation), we have learned something about the global behavior of the system: The ant goes to infinity. This observation illustrates the fact that, often, global features are related to symmetries instead of details.

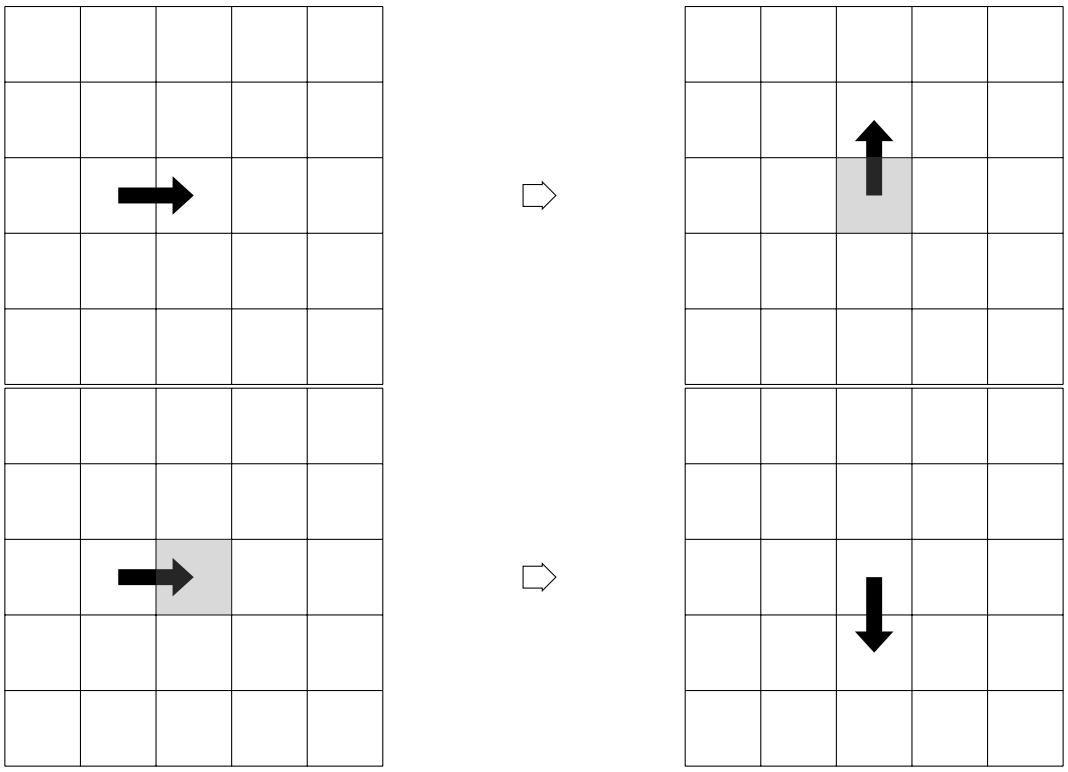


Figure 4.10: Illustration of the rules for Langton's ant.

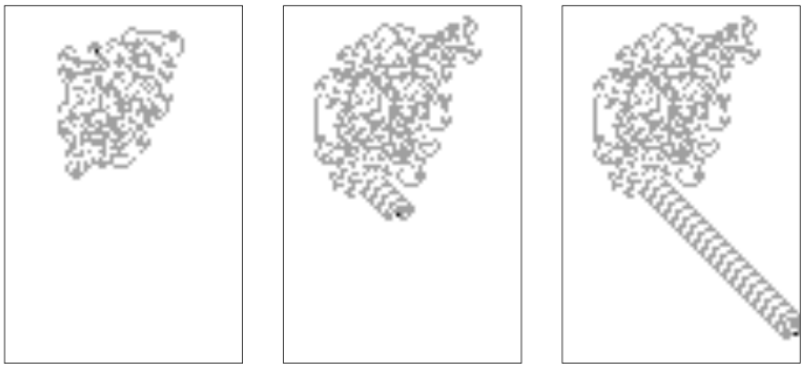


Figure 4.11: Langton's ant rule. The motion of a single ant starts with a chaotic phase of about 10′000 time steps, followed by the formation of a highway. The figure shows the state of each lattice cell (gray or white) and the ant position (marked by the black dot). In the initial condition all cells are white and the ant is located in the middle of the image.

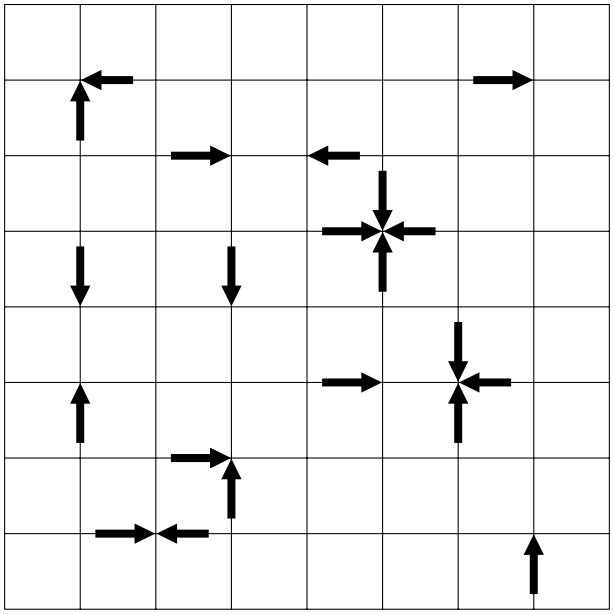


Figure 4.12: A particular configuration of HPP particles.

4.3.4 HPP Gas Model

The HPP model, named after Hardy, Pomeau and De Pazis, is a simple example of an important class of cellular automata models, the *lattice gas automata* (LGA). The goal of this particular cellular automaton is to simulate colliding point particles on a two-dimensional square lattice and is able to illustrates many important features of more complicated LGA in a simple way. Particles can move along the main directions of the lattice, as shown in Fig. 4.12. In this model the number of particles entering a given site with a given direction of motion is limited to one. Thus, with at most one particle per site and direction, four bits of information at each site are enough to describe the system during its evolution. For instance, if at time t site $\boldsymbol{i}$ has the following state $\psi(\boldsymbol{i}, t) = (1011)$, it means that three particles are entering the site along direction 1, 3 and 4, respectively.
The rule is split in two steps:

1. Collision,

2. Propagation (=streaming).

The collision phase specifies how the particles entering the same site will interact and change their trajectories. During the propagation phase, the particles actually move to the nearest neighbouring site they are traveling too. This decomposition into two phases is a quite convenient way to partition the space so that the collision rule is purely local. Figure 4.13 illustrates the HPP rules.
According to our Boolean representation of the particles at each site, the collision rules for the two-particle head on collisions are expressed as

$$(1010) \rightarrow (0101) \tag{4.12}$$

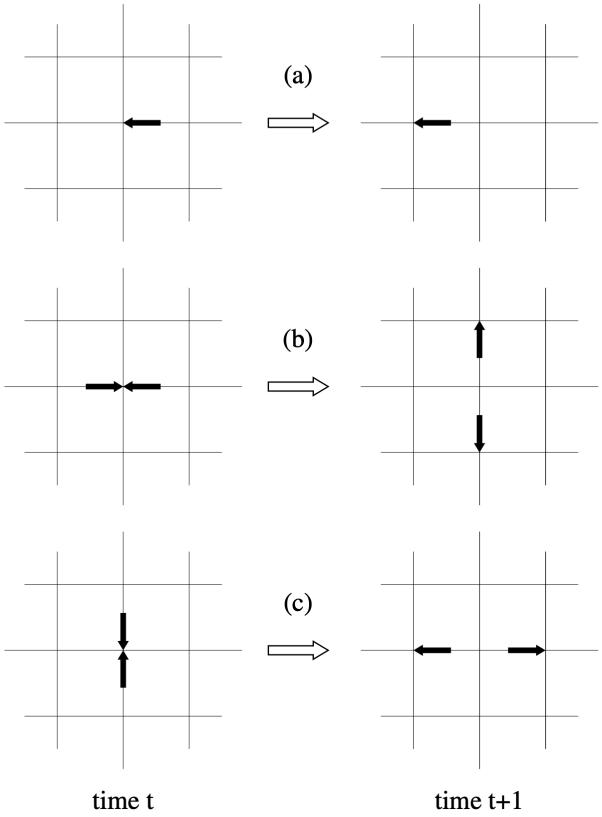


Figure 4.13: The HPP rule: (a) A single particle has a ballistic motion until it experiences a collision; (b), (c) The two non-trivial collisions of the HPP model: Two particles experiencing a head on collision are deflected in the perpendicular direction. In the other situations, the motion is ballistic, that is, the particles are transparent to each other when they cross the same site.

and

$$(0101) \rightarrow (1010). \tag{4.13}$$

All other configurations are unchanged by the collision process. After the collision, the propagation phase moves information to the nearest neighbours: The first bit of the state variable is shifted to the eastern cell, the second bit to the northern and so on. This gives the new state of the system at time $t + 1$. Note that both collision and propagation are applied simultaneously to all lattice sites.

The aim of the HPP cellular automaton is to reproduce some aspects of the real interactions between particles, namely that momentum and particle number are conserved during a collision. From Fig. 4.13, one can easily check that these properties are fulfilled: A pair of zero momentum particles along a given direction is transformed into another pair of zero momentum particles along the perpendicular axis.

Formally, the HPP rule can be expressed as

$$n_k(\boldsymbol{i}, t) = \begin{cases} 1 & \text{if a particle is entering site } \boldsymbol{i} \text{ at time } t \text{ along direction } k, \\ 0 & \text{otherwise,} \end{cases} \tag{4.14}$$

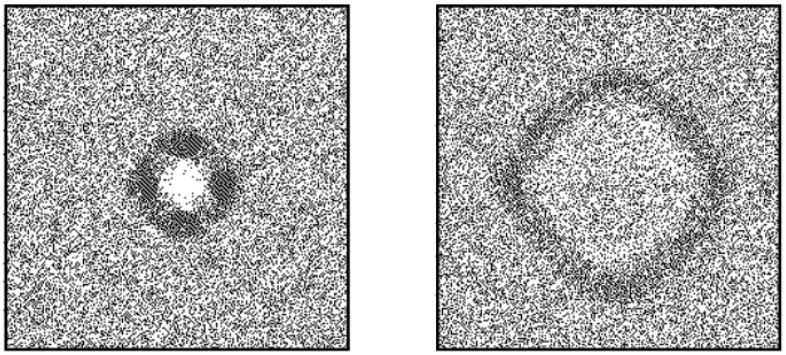


Figure 4.14: Time evolution of a HPP gas. (Left) The initial state is a homogeneous gas with a higher spatial density of particles in the middle region (dark area). (Right) After several time steps the initial perturbation propagates as a wave across the system. As one can be observed, the isotropy is lost in this propagation.

where $n_k(\boldsymbol{i}, t)$ is the so-called *occupation numbers*. From the definition of the occupation numbers it is clear that, for the HPP automaton, the state of lattice site $\boldsymbol{i}$ is simply given by

$$\boldsymbol{\psi}(\boldsymbol{i}, t) = \begin{pmatrix} n_1(\boldsymbol{i}, t) & n_2(\boldsymbol{i}, t) & n_3(\boldsymbol{i}, t) & n_4(\boldsymbol{i}, t) \end{pmatrix}. \tag{4.15}$$

While the HPP model conserves the number of particles during the collision and propagation and furthermore conserves momentum, it lacks rotational invariance, which makes the model highly anisotropic. This means for example, that the vortices produced by the HPP model are square-shaped.
The behavior of the HPP model is illustrated in Fig. 4.14. From this simulation it is clear that some spatially anisotropic behavior builds up during the time evolution of the rule.
Note that similar models can be used to simulate fluid systems. However, a square lattice is actually too poor to represent such a system correctly. Instead, one can use a hexagonal grid model (first introduced in 1986 by U. Frisch, B. Hasslacher and Y. Pomeau known as the FHP model). The FHP model, in essence similar to the HPP one, also includes three-particle collision rules. This is, however, out of the scope of this discussion.

Chapter 5

Percolation

Originally, *percolation* [1] describes the filtration of a fluid through a porous medium. The first and very basic model of such a process was introduced in 1957 by Broadbent and Hammersley [14], where a liquid poured over a container filled with glass beads was considered. Similar processes occur often in chemistry and material sciences.
A very prominent example is the sol-gel transition, where the formation of gelatine is described. Initially, gelatine is a fluid containing many small monomers (emulsion), which is referred to as sol. If we place the sol in a fridge, it becomes a "solid" gel. The process taking place is schematically illustrated in Fig. 5.1: Upon cooling, the monomers start polymerizing and the polymers start growing. At some point in time, one molecule has become sufficiently big to span from one side of the container to the other, which is when the so-called percolation transition occurs. The polymerization as well as the growth is experimentally accessible; one can for instance measure the shear modulus or the viscosity as a function of time. After a characteristic time, the gel time t_G, the shear modulus [2] suddenly increases from zero to a finite value. Similarly, the viscosity increases and becomes singular/infinite at t_G; this is reflected in experimental findings such as those in Fig. 5.2.
Very interesting and important is that it was noticed that percolation phenomena appear in many more applications, for example:

- Spreading of forest fires, epidemics, etc.,
- Crash of stock markets (D. Sornette, e.g. [16]),
- Landslide victories in elections (S. Galam, e.g. [17]),
- Recognition of antigens by T-cells (A. Perelson, e.g. [18]).

Common to all of them is that the models all share some universal features. In this context, we speak of so-called *critical phenomena* that occur in and classify systems that undergo *phase transitions*.
More extensive literature discussing the material covered in this chapter can be found in Refs. [15, 19, 20, 21].

[1] Latin: "to filter" or "pass through"

[2] The shear modulus is defined as shear stress (=force/area) divided by shear strain (=displacement/length).

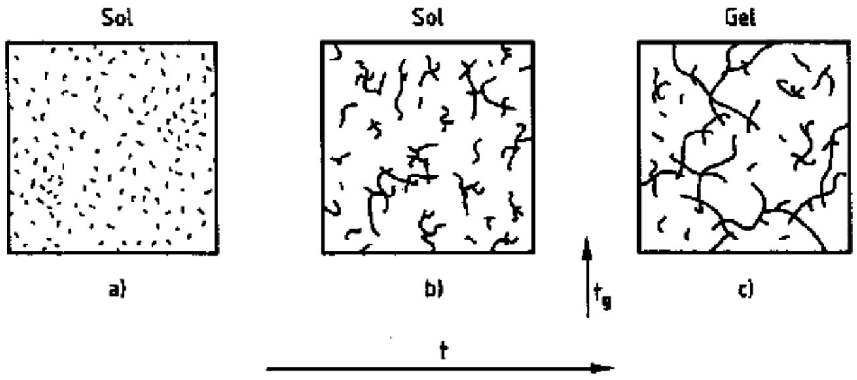


Figure 5.1: Sol-Gel transition: a) Monomers are dispersed in the liquid (sol). b) When the liquid is cooled down, the monomers start polymerizing and grow. c) Once a huge macromolecule (which spans through the whole container) has formed the gel transition occurs. [15]

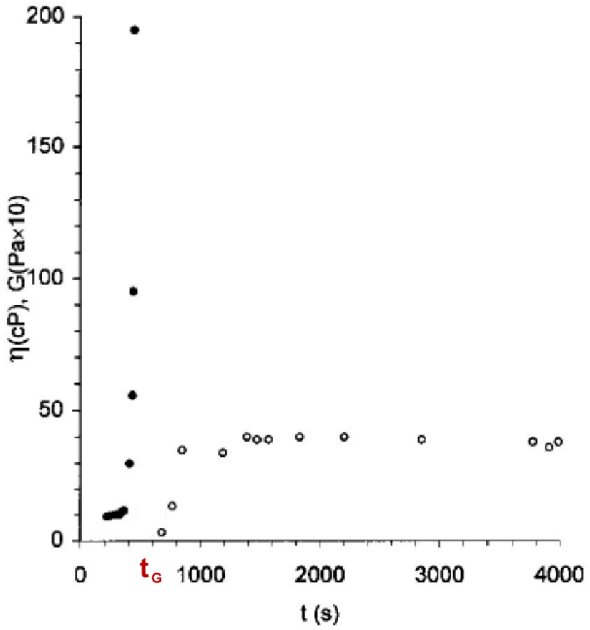


Figure 5.2: Viscosity (filled circles) and shear modulus (open circles) as a function of time. At the gel time t_G the viscosity diverges to infinity and the shear modulus, previously 0, increases to a finite value. [15]

5.1 Infinite System

Let us consider an infinite lattice $\mathcal{L}$ of sites with index $\boldsymbol{i}$ that can either be empty (value 0) or occupied (value 1). In this section, we do not restrict our discussion to a particular lattice shape (e.g. square lattice, triangular lattice, etc.) or dimension d. Furthermore, for each lattice site, we define the neighborhood $\mathcal{N}_{\boldsymbol{i}}$ consisting of $\boldsymbol{i}$'s nearest neighbors. For example, in a 2D square lattice, the neighborhood of site (i_1, i_2) is formed by the neighbors in the

north $(i_1, i_2 + 1)$, east $(i_1 + 1, i_2)$, south $(i_1, i_2 - 1)$ and west $(i_1 - 1, i_2)$, i.e.

$$\mathcal{N}_{(i_1,i_2)} = \{(i_1, i_2 + 1), (i_1 + 1, i_2), (i_1, i_2 - 1), (i_1 - 1, i_2)\}. \tag{5.1}$$

For now, we assume a fully empty lattice. Starting from this, we go through each lattice site and occupy it with *occupation probability* p, each independently of all other sites. Given such a system, the main goal of percolation theory is to study the formation of so-called *clusters*. A cluster $\mathcal{C}$ is the set of occupied sites that are connected to their neighboring sites, or in other words, it consists of sites that form an island. Generally, such a system has many clusters, possibly even infinitely many. If we continuously increase the occupation probability, starting from $p = 0$, up to $p = 1$, we can observe that the clusters get bigger and bigger. Eventually, at some point a cluster of infinite size appears. In this case we say that the system is *percolated*.
Throughout this chapter, we will only discuss the above described case of *site percolation*. Nevertheless, we note that it is also possible to study *bond percolation*, in which case we are interested in the clustering of bonds between lattice sites. However, bond percolation is less general as every bond model can be reformulated as a site model. The opposite is not always true.

5.1.1 Phase Transition

As soon as the occupation probability p reaches a certain value, a phase transition from the non-percolated system (e.g. sol phase) to the percolated system (e.g. gel phase) containing an infinitely-sized cluster happens. We call this value *percolation threshold* p_c, or more generally *critical point*. Formally, phase transitions are described by a so-called *order parameter*. An order parameter is, loosely speaking, a quantity which is equal to 0 in one phase but finite in the other phase. Additionally and importantly, the order parameter characterizes the ordered phase [3]. Broadly speaking, we can group order parameters in two different categories: Situations where the order parameter jumps at the critical point and situations where the order parameter varies continuously at the critical point. In the former case, we speak of a *first-order* phase transition, in the latter of a *second-order* phase transition.
The phase transition occuring in the percolation problem is a second-order phase transition. The corresponding order parameter $P(p)$ is called *percolation strength* and is defined as the fraction of sites belonging to the infinite cluster, see Fig. 5.3 (left). Trivially, we can identify the two situations $P(p < p_c) = 0$ (because no infinite cluster exists) and $P(p = 1) = 1$ (because every site is occupied and therefore connected to all other sites). Above but close to the critical point p_c, i.e. $p \gtrsim p_c$, the order parameter behaves as a power law

$$P(p \gtrsim p_c) \sim |p - p_c|^{\beta}. \tag{5.2}$$

Indeed, this is a universal behavior common to all second-order phase transitions. However, the so-called *critical exponent* β strongly depends on the problem that we consider (here in particular the percolation model and the dimension).

[3] In percolation theory we typically speak about the *non-percolated* and the *percolated* phase. In the general theory of phase transitions one comes often across the terms *disordered* and *ordered* phase. They mean, however, exactly the same thing.

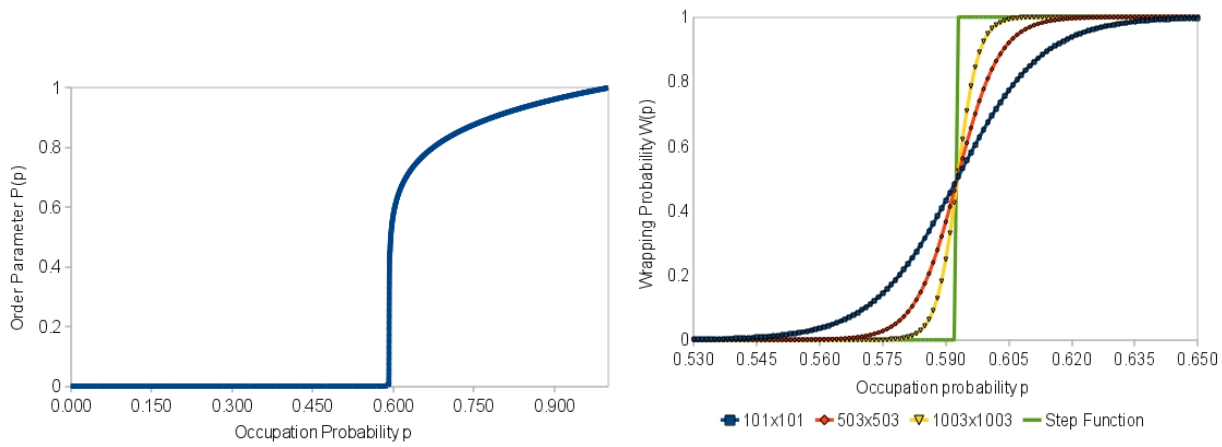


Figure 5.3: Percolation strength (=order parameter) and wrapping probability for the infinite 2D square lattice.

The order parameter is also the quantity that defines the *phase diagram* which determines the regions in parameter space that belong to a certain phase. In our case, the phase diagram is one-dimensional (depending only on the occupation probability p) and bisected in two regions: The trivial (=non-percolated) phase for $0 \leq p < p_c$ and the percolated phase for $p_c < p \leq 1$.
Besides the order parameter, we can introduce another important quantity, the so-called *wrapping probability* $W(p)$. It is the probability that the system is percolated and is given by

$$W(p) = \begin{cases} 0, & 0 \leq p < p_c \\ 1, & p_c < p \leq 1 \end{cases} . \tag{5.3}$$

Figure 5.3 (right) shows the wrapping probability (in the infinite size case, only the green curve is important for us).

5.1.2 Cluster-Size Distribution

Besides the appearance of an infinite cluster, we might be interested in how many clusters we have and what their sizes are. This information is all contained in the so-called *cluster-size distribution* which is defined as

$$n_s(p) = \lim_{L \to \infty} \frac{N_s(p, L)}{L^d}, \tag{5.4}$$

where $N_s(p, L)$ is the number of clusters of size s for given occupation probability p and system's side length L [4] . It is found that it behaves fundamentally different in the different

[4]Here, we assume for simplicity that the total number of lattice sites is given by L^d, where d is the system's dimension. More generally, we could of course have a different side length for each direction.

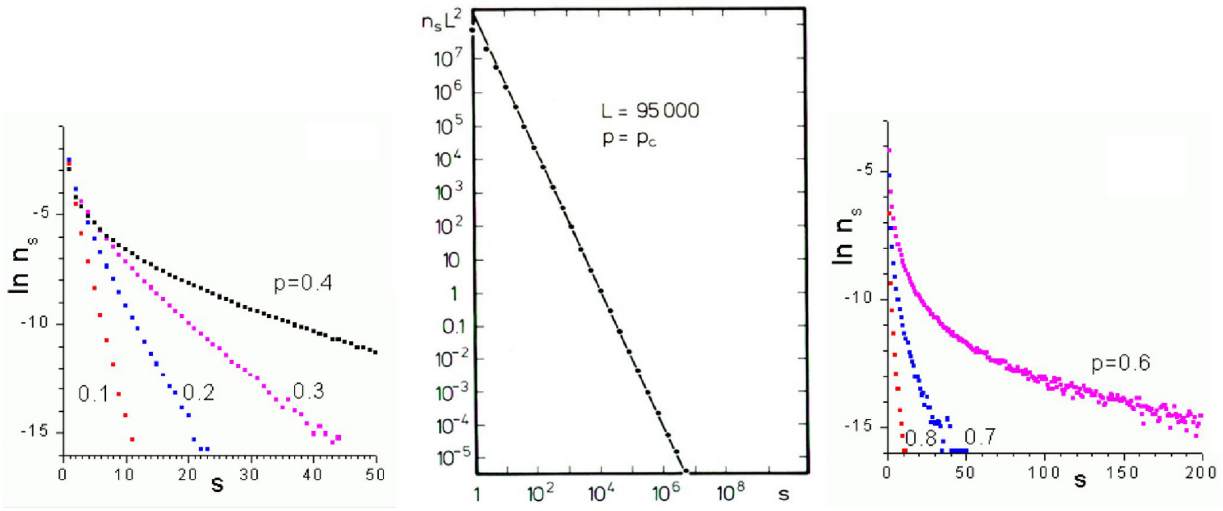


Figure 5.4: Cluster-size distribution for the three different regions $p < p_c$, $p = p_c$ and $p > p_c$ for a finite size square lattice.

parameter regions

$$n_s(p) \sim \begin{cases} s^{-\theta} e^{as} & p < p_c, \\ s^{-\tau} & p = p_c, \\ e^{-bs^{1-\frac{1}{d}}} & p > p_c, \end{cases} \tag{5.5}$$

where the parameters θ, a, τ, b depend on the concrete case and d is again the system's dimension. For a finite size square lattice a visualization of the cluster-size distribution is shown in Fig. 5.4.
It turns out that important for us is the second moment [5] of the cluster-size distribution

$$\chi(p) = \left\langle s^2 \right\rangle'(p) = \sum_s{}' s^2 n_s(p), \tag{5.7}$$

where the $'$ indicates that the sum is without the infinite cluster (otherwise $\chi(p > p_c) = \infty$). Close to the phase transition the second moment behaves as

$$\chi(p \approx p_c) \sim |p - p_c|^{-\gamma}, \tag{5.8}$$

where γ is again a critical exponent characterizing the phase transition.

[5] The k-th moment of some discrete probability distribution p_n is given by

$$\langle n^k \rangle = \sum_n n^k p_n. \tag{5.6}$$

Especially, the first and second moment of a probability distribution are important as they give information about the mean and the variance of the distribution, respectively.

5.1.3 Critical exponents

Second-order phase transitions are fully characterized by their critical exponents. Above, we already saw the exponent β characterisitic for the system's order parameter as well as γ for the second moment of the cluster-size distribution. These are sufficient for our purposes but it is worth noting that there a more of them associated to other important quantities, e.g. α, δ, ν, η, etc.
Critical exponents do not only appear for the phase transition of the percolation model as we will see for example in Ch. 7 for the case of a magnetic phase transition. Furthermore, even for the percolation phase transition the coefficients vary depending on the lattice shape and the dimension. For example, for a 2D square lattice, it is found that $\beta = 5/36$ and $\gamma = 43/18$.
Important for us is the fact that once we have found the critical exponents, the behavior of the system around the critical point is fully specified. To achieve this we do not even need all the critical exponents as not all are independent. In the theory of phase transitions, there are so-called *scaling laws* that relate certain exponents with each other. The most famous ones go under the name of Rushbrooke, Widom, Fisher and Josephson. At this point, the exact relations are not any significance for us, only the fact that is enough to find two exponents to be able to derive the remaining ones!
In the following section, we will learn how to determine the critical exponents numerically.

5.2 Finite System

In numerical simulations, we are only able to simulate finite system sizes. Nonetheless, we would like to find out when the percolation transition occurs (i.e. when we find a spanning cluster) and how the clusters in the system look like, from which we can determine the critical exponents.
Here, for simplicity, we restrict our discussion to the case of a 2D square lattice of size $L_1 L_2$. To initialize the system, we create a 2-dimensional array of integers filled with zeros and choose our occupation probability p. For each lattice site, we draw a random number x in $[0, 1)$. If $x < p$, we mark the site as occupied (1), otherwise we leave it empty (0). Figure 5.5 shows an example for a 5×5 lattice.
The following code snippet shows how a 2D square lattice of size 5×5 can be initialized:

```
L1 = 5
L2 = 5
p = 0.6                          # occupation probability
lattice = zeros(Int64, L1, L2)
for j in 1:L2, i in 1:L1         # Julia is column major
    if rand() < p
        lattice[i, j] = 1
    end
end
```

In the following subsections, closely following Ref. [22], we are going to discuss two different algorithms that give us many insights into the percolation problem and we will observe

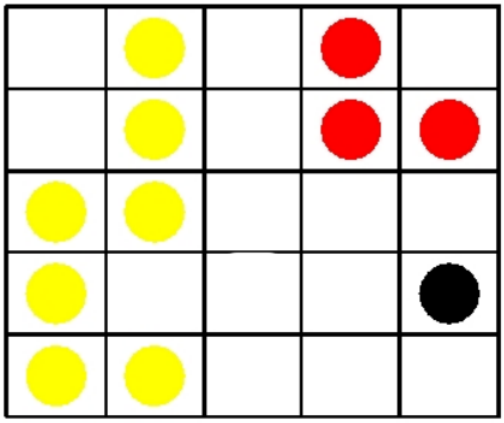


Figure 5.5: 5×5 square lattice after the initialization. For visualization purposes the different clusters are colored. We have one (percolating) cluster of size $s = 7$, one cluster of size $s = 3$ and one clusters of size $s = 1$

important features of phase transitions in finite-sized systems. First, we will have a look at the Burning algorithm which will essentially determine whether the system is percolated or not. Second, we will discuss the Hoshen-Kopelman algorithm which will give us the cluster-size distribution.

5.2.1 Burning Algorithm

In order to calculate the critical probability associated with a lattice, we need a method to detect if a spanning cluster exists in a given system. The Burning algorithm provides us with exactly this information; not only does it provide us with a boolean feedback (yes/no), but it also calculates the minimal path length (the minimal distance between opposite sides, following only occupied sites). The name of the method stems from its implementation. Imagine a grid with occupied and unoccupied sites; an occupied site represents a tree while an unoccupied site is simply empty space. If we start a fire at the very top of our grid, all trees in the first row will start to light up as they are in the immediate vicinity of the fire. Obviously, this forest fire will spread and neighbors of the trees in the first row will soon catch on fire as well. Thus, in the second iterative step all trees immediately adjacent to burning trees get torched. Clearly, the iterative method only comes to an end if the fire has reached the bottom or if the fire runs out of fuel (i.e. no unburnt trees neighboring a burning site) and consequently dies out. If the inferno has reached the bottom, we can determine the length of the shortest path from the iteration counter; that is, the number of iterations defines the shortest-path length of the percolating cluster.
The Burning algorithm is an algorithm for determining whether the system is percolated or not and furthermore gives us the shortest path length from one side of the system to the other. The name of the method stems from its implementation; let us assume that we have a grid with occupied and unoccupied sites. An occupied site represents a tree while an unoccupied site stands for empty space. If we start a fire at the very top of our grid, all trees in the first row will start to light up as they are in the immediate vicinity of the fire.

Obviously, not only the first row of trees will fall victim to this forest fire, as the neighbors of the trees in the first row will soon catch on fire as well. The second iteration step is thus that all trees that are neighbors of already burning trees are being torched.
Clearly, the iterative method only comes to an end when the fire finds no new victims (i.e. unburnt occupied sites neighboring a burning site) to devour and consequently dies out or if the fire has reached the bottom. If the inferno has reached the bottom, we can read off the length of the shortest path from the iteration counter since the number of iterations defines the shortest path length of the percolating cluster. Furthermore, we can get the longest fire duration as the highest number in the whole array.
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 which have label t.
   (b) For each t-labeled cell do
      i. Check if any nearest neighbor (on the square lattice: North, East, South, West) is occupied and not burning (label is 1).
      ii. Assign any such neighbors the label $t + 1$.
3. Repeat step 2 (with $t \to t + 1$) until either there are no neighbors to burn anymore or the bottom line has been reached. In the latter case the latest label minus 1 defines the minimal path length.

A graphical representation of the burning algorithm is given in Fig. 5.6. For the 2D square lattice, we can find the nearest neighbors with the following Julia function:

```
"""
Find the nearest neighbors of site ('i', 'j') in 'lattice'.
'i' and 'j' need to be valid indices for 'lattice'.
"""
function nearestneighbors(lattice::AbstractMatrix{<:Integer},
                          i::Integer, j::Integer)
    L1, L2 = size(lattice)
    if 1 < i < L1
        if 1 < j < L2  return CartesianIndex(i, j-1),
                             CartesianIndex(i-1, j),
                             CartesianIndex(i+1, j),
                             CartesianIndex(i, j+1)
        elseif j == 1  return CartesianIndex(i-1, j),
                             CartesianIndex(i+1, j),
                             CartesianIndex(i, j+1)
        elseif j == L2 return CartesianIndex(i, j-1),
                             CartesianIndex(i-1, j),
                             CartesianIndex(i+1, j)
```

Figure 5.6: The Burning Method: A fire is started in the first line. At every iteration, the fire from a burning tree lights up occupied neighboring cells. The algorithm ends if all (neighboring) trees are burnt or if the burning method has reached the bottom line.

```
        end
    end
    if i == 1
        if 1 < j < L2  return CartesianIndex(i, j-1),
                              CartesianIndex(i+1, j),
                              CartesianIndex(i, j+1)
        elseif j == 1  return CartesianIndex(i+1, j),
                              CartesianIndex(i, j+1)
        elseif j == L2 return CartesianIndex(i, j-1),
                              CartesianIndex(i+1, j)
        end
    end
    if i == L1
        if 1 < j < L2  return CartesianIndex(i, j-1),
                              CartesianIndex(i-1, j),
                              CartesianIndex(i, j+1)
        elseif j == 1  return CartesianIndex(i-1, j),
                              CartesianIndex(i, j+1)
        elseif j == L2 return CartesianIndex(i, j-1),
                              CartesianIndex(i-1, j)
        end
    end
end
```

Given the Burning algorithm, we can compute both the order parameter and the wrapping probability as functions of the occupation probability p and the number of lattice sites L. From them we can determine the percolation threshold to be approximately $p_c \approx 0.59274$ for the square lattice. Importantly, we observe that we find percolating clusters even when $p < p_c$ as long as $L < \infty$. In the case of $L < \infty$, the curves for the order parameter and the wrapping probability (see Fig. 5.3 (right)) are smeared out compared to the infinite size case.

5.2.2 Hoshen-Kopelman Algorithm

After having investigated the percolation threshold and the fraction of sites in the spanning cluster, the natural extension would be to identify all the clusers. An efficient algorithm is necessary to perform such a task. In fact, there are several such algorithms; the most popular (and for this purpose most efficient) algorithm is the Hoshen-Kopelman algorithm, which was developed in 1976.
We represent the percolation configuration as a matrix N_{ij} that can have values of 0 (site is unoccupied) and 1 (site is occupied). Furthermore, let k be a running index labeling the clusters in N_{ij}. We also introduce an array M_k to keep track of the mass of cluster k (the number of sites belonging to the cluster k). We start the algorithm by setting $k = 2$ (since 0 and 1 are already taken) and search for the first occupied site in N_{ij} beginning in the upper left corner of the lattice. We then add this site to the array $M_{k=2} = 1$ and set the entry in N_{ij} to k (so it is branded as pertaining to the cluster k).
We then go over all lattice sites N_{ij}, column by column (since Julia is column-major), and try to detect whether an occupied site belongs to an already known cluster or a new one. We comb through the lattice from top-left to bottom-right; the criteria are rather simple:

- If a site is occupied and the top and left neighbors are empty, we have found a new cluster and we set k to $k + 1$, $N_{ij} = k$ and $M_k = 1$.
- If one of the sites (top or left) has the value k_0 (i.e. is part of a cluster), we increase the corresponding value in the array M_{k_0} by one (setting M_{k_0} to $M_{k_0} + 1$). We label the new site accordingly. That is, we set $N_{ij} = k_0$.
- If both neighboring sites are occupied with k_1 and k_2, respectively (assuming $k_1 \neq k_2$)- meaning that they are already part of a cluster - we choose one of them (e.g. k_1). We set the matrix entry to the chosen value, $N_{ij} = k_1$, and increase the array value not only by one but also by the whole number of sites already in the second cluster (here k_2), $M_{k_1} \to M_{k_1} + M_{k_2} + 1$. Of course we have to mark the second array M_{k_2} in some way so that we know that its cluster size has been transferred over to M_{k_1} which we do by setting it to $-k_1$. We have thus branded M_{k_2} in such a way that we immediately recognize that it does not serve as a counter anymore (as a cluster cannot consist of a negative number of sites). Furthermore, should we encounter an occupied site neighboring a k_2 site, we can have a look at M_{k_2} to see that we are actually dealing with cluster k_1 (revealing the "true" cluster number).

The last point is crucial, as we usually deal with a number of sites that are marked in such a way that we have to first recursively detect which cluster they pertain to before carrying

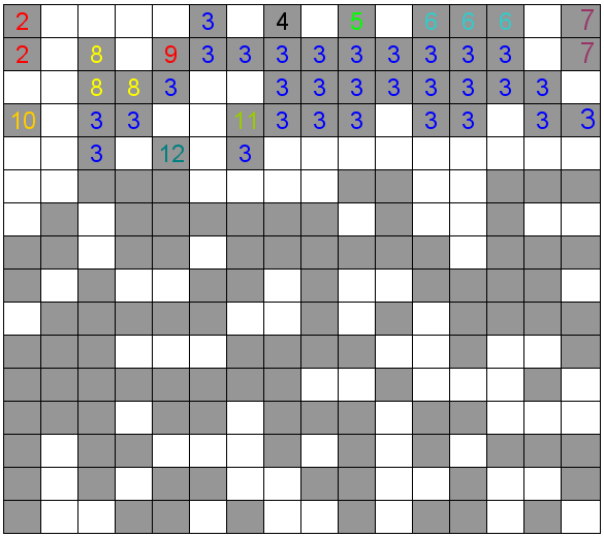


Figure 5.7: The Hoshen-Kopelman algorithm applied to a percolation cluster. The numbers denote the running cluster variable k, however, without the negative links as described in the text.

out the algorithm any further. The recursive detection stops as soon as we have found a k_0 with $M_{k_0} \geq 0$ (i.e. a "true" cluster number).
Once all the sites N_{ij} have been visited, the algorithm ends up with a number l of clusters, where l is smaller than the largest obtained cluster index k_{max}. The only thing left to do is to construct a histogram of the different cluster sizes. This is done by looping through all the clusters $k \in \{2, \ldots, k_{max}\}$ while skipping negative M_k.
A visualization of the algorithm is given in Fig. 5.7. The algorithm is very efficient because it visits every site only once and it scales linearly with the number of sites.
Let us write down the Hoshen-Kopelman algorithm:

1. $k = 2$, $M_k = 1$.
2. For all i, j of N_{ij}
   (a) If top and left are empty (or non-existent) $k \to k + 1$, $N_{ij} = k$, $M_k = 1$.
   (b) If one is occupied with k_0 then $N_{ij} = k_0$, $M_{k_0} \to M_{k_0} + 1$.
   (c) If both are occupied with k_1 and k_2 (and $k_1 \neq k_2$) then choose one, e.g. k_1, and $N_{ij} \to k_1$, $M_{k_1} \to M_{k_1} + M_{k_2} + 1$, $M_{k_2} = -k_1$.
   (d) If both are occupied with k_1, $N_{ij} = k_1$, $M_{k_1} \to M_{k_1} + 1$.
   (e) If any of the ks considered has a negative mass M_k, find the original cluster they reference to and use its cluster number and weight instead by using while$(M_k < 0)k = -M_k$.
3. For $k \in \{2, \ldots, k_{max}\}$, if $M_k > 0$ then $N(M_k) \to n(M_k) + 1$.
4. For all cluster sizes s, determine the cluster-size distribution n_s.

Once we have run the algorithm for a given lattice (or collection of lattices and taken the average) we can evaluate the results. We find different behaviors of the relative cluster size n_s (where s denotes the size of the clusters) depending on the occupation probability p. These results are illustrated in Fig. 5.4. From these data we can find the exponents θ, s, τ, b, d. The second moment χ of the cluster-size distribution n_s is a very strong indicator of p_c as we can see a very clear divergence around p_c. Of course, we see a connection to the Ising model (see Ch. **??**), where the magnetic susceptibility diverges near the critical temperature.

5.2.3 Finite-Size Scaling

For the infinite system, we saw that the order parameter scales at $p \gtrsim p_c$ as

$$P(p) \sim |p - p_c|^{\beta}, \tag{5.9}$$

while the second moment of the cluster-size distribution as

$$\chi(p) \sim |p - p_c|^{-\gamma}, \tag{5.10}$$

where β and γ are the corresponding critical exponents. As discussed earlier, this behavior is universal. When we simulate finite-sized systems, we observe that the change in P and χ is not as abrupt as in the infinite system. Instead, we have to modify the expressions as

$$P(p, L) = L^{-\frac{\beta}{\nu}} \mathcal{F}_P \left[|p - p_c| L^{\frac{1}{\nu}} \right] \tag{5.11}$$

$$\chi(p, L) = L^{+\frac{\gamma}{\nu}} \mathcal{F}_\chi \left[|p - p_c| L^{\frac{1}{\nu}} \right] \tag{5.12}$$

such that they include also the dependence on the system size [6] . The functions $\mathcal{F}_P$ and $\mathcal{F}_\chi$ are so-called *scaling functions*. While we are not interested in their exact functional form, it is important to notice that they imply a collapse (*data collapse*) of all our finite size simulation data points onto a single curve once we have the exact critical exponents, see Fig. 5.8. Inversely, this means that we can estimate the critical exponents that characterize the phase transition by performing simulations on finite-sized systems and enforce the data collapse. This procedure is called *finite-size-scaling analysis*. As noted earlier, we only need to find two critical exponents from which we can in principle derive all others using the scaling relations. Here, as an example, we focus on the second moment of the cluster-size distribution. Note, however, that this procedure also works for other quantities, e.g. the order parameter.
The steps are as follows:

1. Estimate the critical point p_c, e.g. with the wrapping probability $W(p)$ obtained from the Burning algorithm.

[6] For completeness, we mention that ν is another critical exponent belonging to the so-called *coherence length*. At this point, it is not explicitly needed and to avoid confusion, we omit its formal introduction here. The mere fact that it is one of the many critical exponents is sufficient. However, more details about the coherence length can be found in Ch. 6.2.

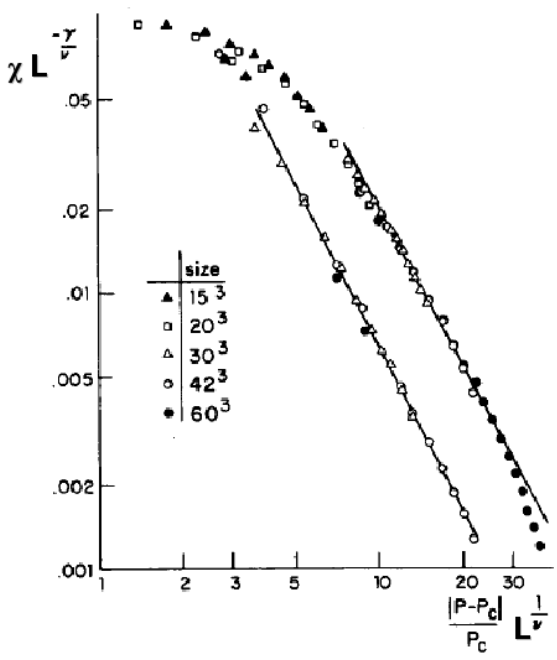


Figure 5.8: Data collapse for the second moment of the cluster-size distribtion.

2. At $p = p_c$, compute χ with the Hoshen-Kopelman algorithm for different system sizes L. Since $\chi(p_c, L) \sim L^{\frac{\gamma}{\nu}}$, we can plot $\log(\chi)$ vs. $\log(L)$ to get an estimate for the fraction γ/ν.

3. Plot $\chi(p, L)L^{-\frac{\gamma}{\nu}}$ vs. $|p - p_c|L^{\frac{1}{\nu}}$ for different L. Tune γ/ν and ν until all data points collapse on the two curves (one for $p < p_c$, one for $p > p_c$).

Finally, we would like to highlight that the accuracy of γ and ν depends strongly on the accuracy of the critical point p_c. Furthermore, we need accurate simulation values for $\chi(p, L)$. Potential errors can be reduced by repeating the simulation for the same system size multiple times and averaging over them. The larger the system size, the less samples are needed in the averaging process to obtain accurate results.

Chapter 6

Fractals

Most of the material in this chapter can be found in Ref. [22].
Fractals are objects that exhibit *self-similarity*. That is, certain patterns occur repeatedly at different spatial scales. B.B. Mandelbrot coined the term *fractal*. He also introduced the concept of the *fractal dimension* (formal introduction see Ch. 6.1) to characterize fractal objects. Before taking a closer look at this concept, it may be useful to consider some examples of self-similarity. In a nutshell, we could define an object to be self-similar if it is built up of smaller copies of itself. Such objects occur both in mathematics and nature. Let us discuss a few examples.

Sierpinski Triangle The Sierpinski triangle is a mathematical object that is constructed by an iterative application of a simple operation. It was first described by the Polish mathematician W.F. Sierpiński. To construct a Sierpinski triangle, we subdivide a triangle into four sub-triangles and discard the center triangle (see Fig. 6.1). In the next step of the iteration, we subdivide each of the three remaining triangles and remove each central triangle. This obviously produces an object that is exactly self-similar in the limit of infinitely many iterations. When zooming in, we see the exact same image at all scales. The object becomes mathematically a fractal in the limit of an infinite number of iterations.

Nature Naturally occurring self-similar objects are usually only approximately self-similar. What we mean by that is that we cannot zoom into natural patterns infinitely often and expect to find the same pattern at all scales.
To better illustrate this, we consider a tree. A tree has different branches, and the whole tree looks similar to a branch connected to the tree trunk. The branch itself resembles the smaller branches attached to it and so on. Evidently, this breaks down after a few iterations, when the leaves of the tree are reached. This example is based on a hierarchical tree structure. However, natural self-similar patterns are not necessarily hierarchical objects as illustrated by the following example.
Gold colloids were shown to arrange in fractals of fractal dimension 1.70 (the meaning of this will soon be explained) by D. Weitz in 1984. Colloidal gold is a suspension of sub-micrometer-sized gold particles in a fluid (e.g. water). These gold colloids aggregate to fractal structures as we show in Fig. 6.2. As in the case of trees, gold colloids are also only approximately self-similar. If we zoom too deeply into the colloidal structure, we will end up seeing individual

Figure 6.1: The Sierpinski triangle is a self-similar mathematical object that is created iteratively.

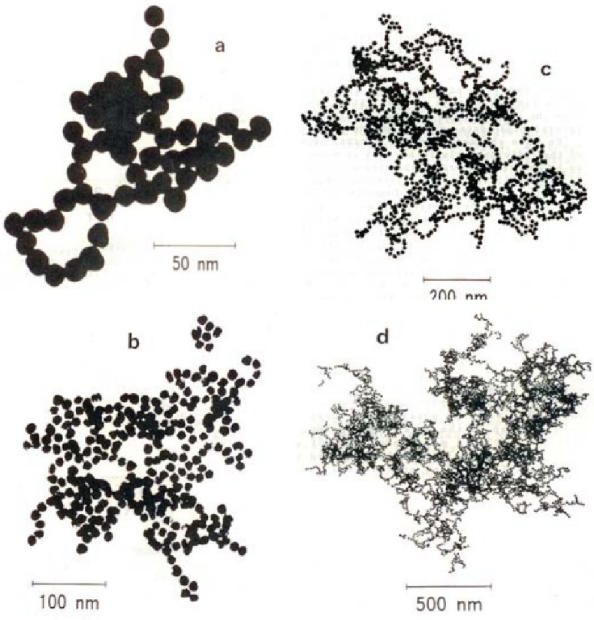


Figure 6.2: Gold colloids at different scales.

gold particles and not a colloidal structure (see Fig. 6.2 (top left panel)). Moreover, the finite size of the aggregate colloids does not allow us to zoom out of the structure infinitely far. The diameter of gold particles and the finite size of aggregate colloids therefore define lower and upper cutoffs, respectively.

6.1 Fractal Dimension

6.1.1 Mathematical Definition

Keeping these examples in mind, we will now give the precise mathematical definition of the fractal dimension. To determine the fractal dimension of an object, we can use the following (theoretical) procedure. Consider all possible coverings [1] of an object with d-dimensional spheres of radius $r_i \leq \epsilon$, where ϵ is the resolution. Let $N_\epsilon(\mathcal{C})$ be the number of spheres used

[1] A covering of an object $X \subset \mathbb{R}^d$ is a collection of sets whose union contains X as a subset.

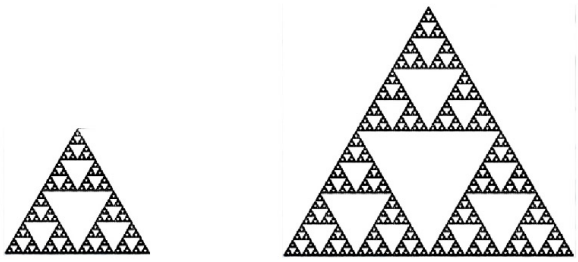


Figure 6.3: Sierpinski triangle.

in the covering $\mathcal{C}$. Then, the volume of the covering is

$$V_\epsilon(\mathcal{C}) \sim \sum_{i=1}^{N_\epsilon(\mathcal{C})} r_i^d, \tag{6.1}$$

where d is the dimension of the space into which the object is embedded. We define V_ϵ^* as the volume of the covering that, among all coverings with the smallest numer of spheres, has minimal volume

$$V_\epsilon^* = \min_{V_\epsilon(\mathcal{C})} \left[\min_{N_\epsilon(\mathcal{C})} \left[V_\epsilon(\mathcal{C}) \right] \right]. \tag{6.2}$$

The fractal dimension of the object is then defined as

$$d_f = \lim_{\epsilon \to 0} \frac{\log\left(\frac{V_\epsilon^*}{\epsilon^d}\right)}{\log\left(\frac{L}{\epsilon}\right)}, \tag{6.3}$$

where L is the linear system dimension.
For most objects, we may simplify the definition of the fractal dimension in Eq. (6.3) in the following way: When the length of the object is stretched by a factor of a, its volume (or mass) grows by a factor of a^{d_f}. We obtain this interpretation by rewriting Eq. (6.3) (in the limit $\epsilon \to 0$) as

$$\frac{V_\epsilon^*}{\epsilon^d} = \left(\frac{L}{\epsilon}\right)^{d_f}. \tag{6.4}$$

If we now consider a rescaling of L by a according to $L \to aL$, we find that $V_\epsilon^* \to a^{d_f} V_\epsilon^*$. Thus, the volume V_ϵ^* scales as claimed. As a simple example, we again consider the Sierpinski triangle. Stretching its sides by a factor of two ($L \to 2L$) evidently increases its area by a factor of three ($V_\epsilon^* \to 3V_\epsilon^*$), see Fig. 6.3. Using this observation and the aforementioned proportionality, we find that the Sierpinski triangle has fractal dimension

$$2^{d_f} = 3 \iff d_f = \frac{\log(3)}{\log(2)} \approx 1.585. \tag{6.5}$$

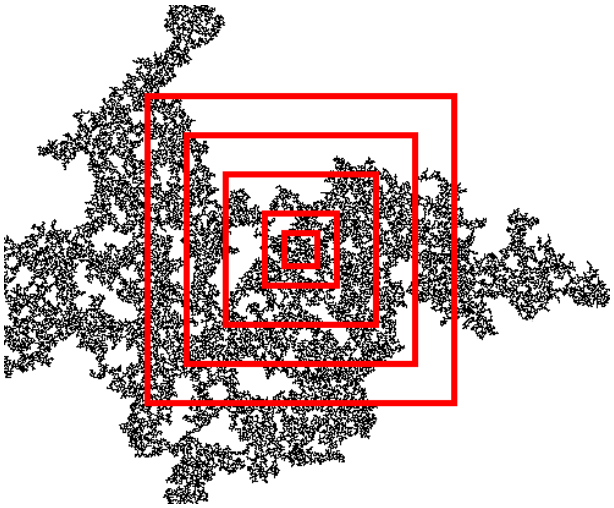


Figure 6.4: Illustration of the sandbox method for a percolation cluster on a square lattice: One measures the bumber of filled pixels in increasingly large concentrically placed boxes.

6.1.2 Sandbox Method

The sandbox method is a technique that can be easily implemented to numerically determine the fractal dimension of an approximately self-similar object. It is conceptually easy because the underlying idea is close to the mathematical definition. We show an illustration of this method in Fig. 6.4. First, we choose a site belonging to the fractal which is located in its center region [2] . Next, we place around this site a small box of size $R = 1$ on the fractal and count the number of occupied sites (or pixels) $N(R)$ in the box. We then successively increase the box size in small steps, $R = 1, 3, 5, \ldots$, until any of the boundaries of the whole picture is reached, always storing $N(R)$. We finally plot $N(R)$ as a function of R in a log-log plot where the fractal dimension is the corresponding slope (see Fig. 6.5).

6.1.3 Box-Counting Method

The box-counting method is another method to numerically determine the fractal dimension of an object. In the box-counting method, we superimpose a grid with grid spacing $\epsilon = 1, 2, \ldots, L$ on the fractal object. We define the number of boxes in the grid that are not empty (contain a part of the fractal) as $N(\epsilon)$. We do this for a large range of ϵ and plot $N(\epsilon)$ as a function of $1/\epsilon$ in a log-log plot. We show a typical result in Fig. 6.6. We recognize a region where the slope is constant in the plot. In this region, the object is self-similar and the slope equals the fractal dimension. Outside this self-similar regime, the finite resolution and finite size of the object destroy the self-similarity. To convince ourselves that this indeed reproduces the fractal dimension that we defined before, recall that ϵ defines a characteristic length scale, while $N(\epsilon)$ is proportional to the volume of the fractal object. Numerically, the box counting method has rather pronounced finite-size effects, because at the border of an object very sparsely occupied cells can have an important weight. Furthermore, the

[2] This guarantees that for the smallest box size $R = 1$, we have $N(R) = 1$. Otherwise this data point does not behave well in the log-log plot.

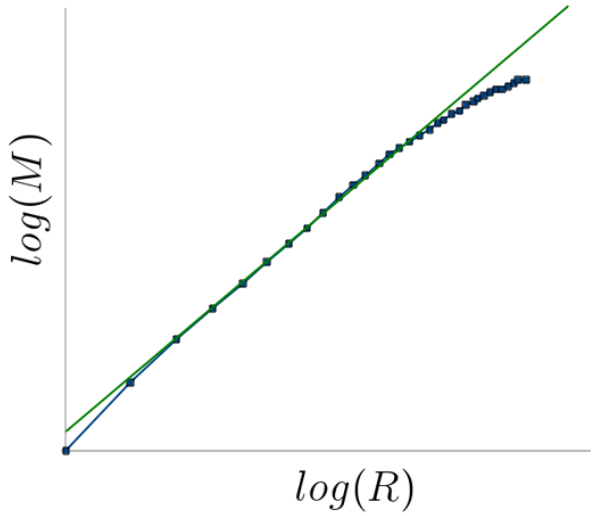


Figure 6.5: We apply the sandbox algorithm to one single percolation cluster at p_c on a square lattice and plot the number of nonempty sites $N(R)$ (here, $M \equiv N(R)$) against the linear box size R in log-log scale. The slope of the linear fit (green) corresponds to the fractal dimension. To obtain a clearer linear dependence, one has to average each data point over several clusters.

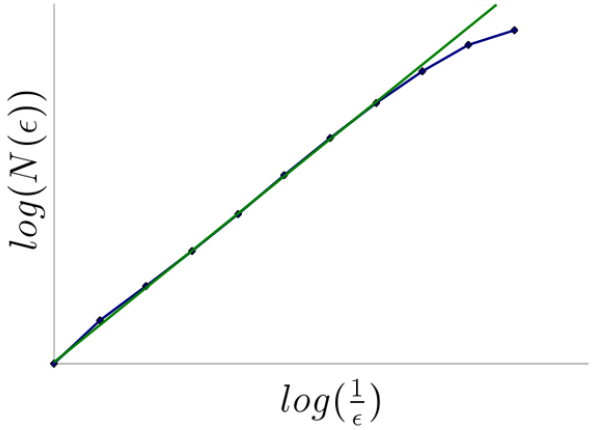


Figure 6.6: We apply the box-counting algorithm to a percolation cluster on the square lattice at p_c and plot the number of nonempty boxes $N(\epsilon)$ against the inverse grid spacing $1/\epsilon$ in a log-log scale. For $\epsilon = 1$, we reach the grid spacing of the underlying lattice (lower cutoff). For $\epsilon = L$, one box covers the whole lattice of linear size L (upper cutoff). The slope of the linear fit (green) corresponds to the measured fractal dimension.

convergence is typically slower than for the sandbox method.

6.2 Fractals and Percolation

In the previous section you may have already wondered why we are using a percolating cluster at p_c as an example for a fractal for which we illustrated both the sandbox as well as the box-counting method. The reason for the percolating cluster at p_c being a fractal lies again at the heart of the theory of phase transitions. In the infinite system we saw that we can clearly distinguish the disordered phase (non-percolating) ($p < p_c$) from the ordered phase (percolating) ($p > p_c$). However, this is not exactly possible at $p = p_c$. The reason for this is that the fundamental length scale of the problem, the *coherence length* ξ, diverges at this point and therefore making it impossible to decide in which phase we are.
The coherence length is the fundamental length scale appearing in the *correlation function*

$$C(\boldsymbol{r}, \boldsymbol{r}') = \langle \rho(\boldsymbol{r})\rho(\boldsymbol{r}')\rangle = C(|\boldsymbol{r} - \boldsymbol{r}'|), \tag{6.6}$$

where ρ is the density of occupied sites at a point. In simple words, the correlation function measures how the density at a point $\boldsymbol{r}$ changes when the density at $\boldsymbol{r}'$ is modified. Here, we assume the limit of vanishing lattice spacing (continuum limit model). In the finite lattice model, we would need to approximate the correlation function. This can be done for example via

$$C(r) \approx \frac{1}{\frac{2\pi^{d/2} r^{d-1}}{\Gamma(\frac{d}{2})}} \frac{N(r + \Delta r) - N(r)}{\Delta r}, \tag{6.7}$$

where we have chosen some center "0", $r = |\boldsymbol{r} - \boldsymbol{r}'|$, the prefactor is the surface of a d-dimensional sphere with radius r, including the Euler Γ-function $\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} dx$ and the numerator counts the number of occupied sites in the spheres with radius $r + \Delta r$ and r, respectively.
At $p \neq p_c$, it can be found that

$$C(|\boldsymbol{r} - \boldsymbol{r}'|) \sim e^{-\frac{|\boldsymbol{r}-\boldsymbol{r}'|}{\xi}} + \underbrace{const}_{=0,\ p<p_c}, \tag{6.8}$$

where ξ is the coherence length. For completeness, we note that at $p = p_c$ the correlation function behaves as a power law.
Close to the phase transition, the coherence length behaves as

$$\xi(p \approx p_c) \sim |p - p_c|^{-\nu}, \tag{6.9}$$

where ν is a critical exponent and $\nu = 4/3$ for the example of the square lattice. Again, as we have seen for the second moment χ of the cluster-size distribution, in the finite system ξ cannot be larger than L [3] .
Clearly, in Eq. (6.9), we can observe that ξ diverges as $p \to p_c$. In this case, we can zoom in and out of the system and it will look always the same - we cannot distinguish in which phase we are. Therefore, the percolating cluster at $p = p_c$ exhibits self-similarity and is a

[3] In practice, in our numerical simulations, we may find values for the correlation length that can be higher than L. This behavior is unphysical and is typically simply cut away. In earlier lectures, you may have encountered a similar procedure in the Maxwell construction of the gas-liquid transition of a van der Waals gas.

fractal object. Its fractal dimension can be determined with the numerical methods described earlier. For this, we first need to set up the system by choosing a grid with side length L and set the occupation probability to $p \gtrsim p_c$. To find the percolating cluster, we can use the burning method that we already encountered in Ch. 5. However, instead of setting the whole first row to fire, we choose only one occupied site. If the fire reaches the other side of the system, the system percolates and all the burned sites belong to the percolating cluster. Note that in order to find all sites of this cluster, we have to let the fire burn until it dies out. If the fire dies out without reaching the other side, the chosen starting site did not belong to the percolating cluster and we have to repeat the same procedure with a different choice for the starting point. If the fire does not reach the other side starting from any occupied site in the first row, the sample needs to be thrown away and a new one has to be created for which we repeat the above described procedure. To compute the fractal dimension of the obtained spanning cluster, we assign a "1" to all sites that belong to the spanning cluster and a "0" to all remaining sites. From here on, only sites that belong to the percolating cluster are considered to be occupied.
Interestingly, we can connect the fractal dimension to the critical exponents of the phase transition. Recall Eq. (5.11), at $p = p_c$ the order parameter behaves as

$$P(p_c, L) \sim L^{-\frac{\beta}{\nu}}. \tag{6.10}$$

At the same time, the number of occupied sites belonging to the percolating cluster, V_ϵ^*, behaves as

$$V_\epsilon^* \sim L^{d_f}. \tag{6.11}$$

By definition of the order parameter, we can combine both relations as

$$V_\epsilon^* \sim PL^d \sim L^{-\frac{\beta}{\nu}} L^d \overset{!}{\sim} L^{d_f} \tag{6.12}$$

from which we find

$$d_f = d - \frac{\beta}{\nu}. \tag{6.13}$$

Chapter 7

Monte Carlo Methods

Most sections in this chapter follow Ref. [22].

7.1 Introduction

Let us assume that we want to compute an integral [1]

$$I = \int_V f(\boldsymbol{x}) d^d x, \tag{7.1}$$

where V is some integration domain in $\mathbb{R}^d$, its volume given by $|V| = \int_V d^d x$. Clearly, it is not possible, in general, to solve this task analytically and even numerically—for conventional methods—it might not be feasible in practice due to the large computational complexity depending on the dimension d and the chosen discretization scheme.
The basic idea of Monte Carlo methods is to approximate the integral by only choosing certain $\boldsymbol{x}_i \in V, i \in \{1, \ldots, N\}$ such that

$$I \approx I_{\mathrm{MC}} = \frac{|V|}{N} \sum_{i=1}^{N} f(\boldsymbol{x}_i). \tag{7.2}$$

The history of Monte Carlo methods goes back to the foundations of probability theory and is largely shaped by scientists like Pascal, Fermat, Buffon, Laplace, Brown, Kelvin, etc. Famous are the stochastic computations of π, e.g. by Buffon in the 18th century. In Buffon's needle experiment, many needles of length l_n are thrown on a grid of straight lines separated by a distance l_d. Due to the fact that the probability of a needle falling on a line is given by $p = \frac{2l_n}{\pi l_d}$, one can repeatedly (N times) throw a needle and count the number N_{hit} of needles falling on a line to compute the probability $p = \frac{N_{\mathrm{hit}}}{N}$ which allows then to get an approximation of π. Impressively, in 1901, Lazzarini carried out Buffon's needle experiment with 34′080 throws which allowed him to get a pretty good approximation

$$\pi \approx \frac{355}{133} \approx 3.14159292. \tag{7.3}$$

[1] Similarly, we could be interested in computing a sum.

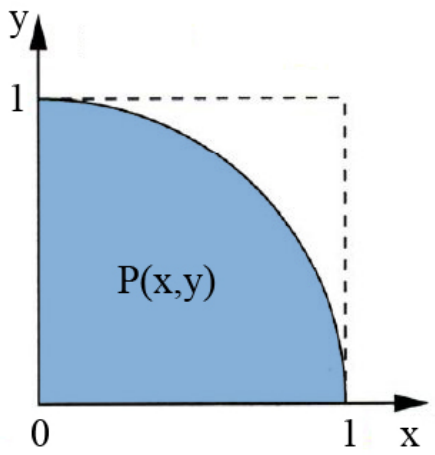


Figure 7.1: Illustration of the areas considered in the computation of π.

Alternatively, one could arrive at this result by computing the integral

$$\frac{\pi}{4} = \int_0^1 \sqrt{1-x^2}dx \tag{7.4}$$

which can be approximated by randomly choosing N points in the unit square and counting the number N_{inside} of points inside the quarter unit circle such that

$$\frac{\pi}{4} \approx \frac{N_{\text{inside}}}{N}, \tag{7.5}$$

see Fig. 7.1.
Monte Carlo methods have many different possible applications, including the following:

- Physics: They are used in many areas in physics, some applications include statistical physics (e.g. Monte Carlo molecular modeling) or in Quantum Chromodynamics. In the Large Hadron Collider (LHC) at CERN, Monte Carlo methods were used to simulate signals of Higgs particles and for designing detectors.
- Design: Monte Carlo methods have also penetrated areas that might—at first sight—seem surprising. They help in solving coupled integro-differential equations of radiation fields and energy transport which is essential for global illumination in photorealistic images of 3D models.
- Economics: Monte Carlo models have also found their place in economics where they have been used e.g. for financial derivatives.

Due to their broad applicability, a lot of different schemes were developed of which we only aim at discussing some basic but nevertheless important ones. Finally, we note that there exist many excellent textbooks on the topic, for example Refs. [23, 24, 25, 26].

7.2 Basics

The two key ingredients that need to be discussed in order to understand Eq. (7.2) are the choice of the points $\boldsymbol{x}_i$, called *samples*, and to determine the error between I_{MC} and I.

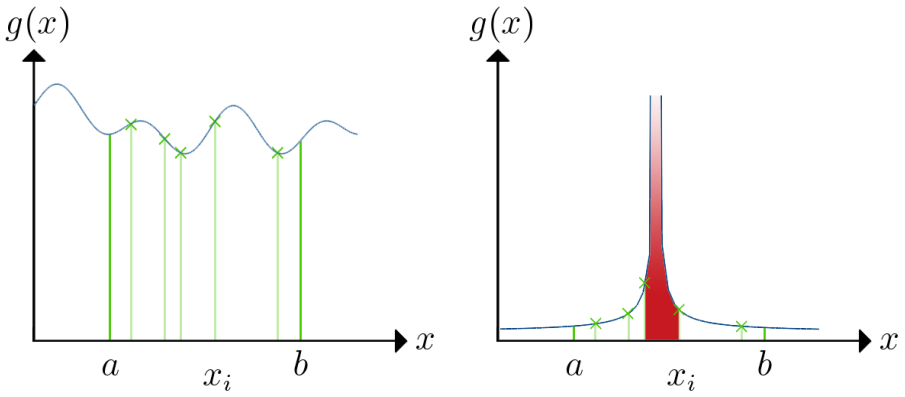


Figure 7.2: (Left) Simple sampling for a smooth function. (Right) Simple sampling for a function with singularity.

7.2.1 Sampling

In Monte Carlo methods, the samples $\boldsymbol{x}_i$ are chosen probabilistically. We mainly differentiate between two different cases:

1. Uniformly chosen $\boldsymbol{x}_i$: In this case, we speak of *simple sampling* (Ch. 7.3).
2. Non-uniformly chosen $\boldsymbol{x}_i$: In this case, we speak of *importance sampling*, e.g. Quasi-Monte Carlo methods (Ch. 7.5) or Markov Chain Monte Carlo methods (Ch. 7.6).

While simple sampling is the easier method, it is prone to more integration errors since not all regions in space are equally important for computing the integral, see e.g. Fig. 7.2.

7.2.2 Error

Conventional Methods

We consider the one-dimensional integral

$$\int_a^b f(x)dx \tag{7.6}$$

and analyze the integration error using the so-called *trapezoidal rule* that you might already have seen in a different class. The trapezoidal rule splits the integration domain $[a, b)$ into N even intervals of size $\Delta x = \frac{b-a}{N}$ and approximates the function f on each interval as a linear function, see Fig. 7.3.
Defining the positions $x_i = a+i\Delta x, i \in \{0, \dots, N\}$, this means that the function f is piecewise approximated as

$$f(x) \approx \tilde{f}(x) = \frac{f(x_i) - f(x_{i-1})}{\Delta x}(x - x_i) + f(x_i), \quad x \in [x_{i-1}, x_i) \tag{7.7}$$

such that we can solve

$$\int_{x_{i-1}}^{x_i} \tilde{f}(x)dx = \frac{f(x_{i-1}) + f(x_i)}{2}\Delta x \tag{7.8}$$

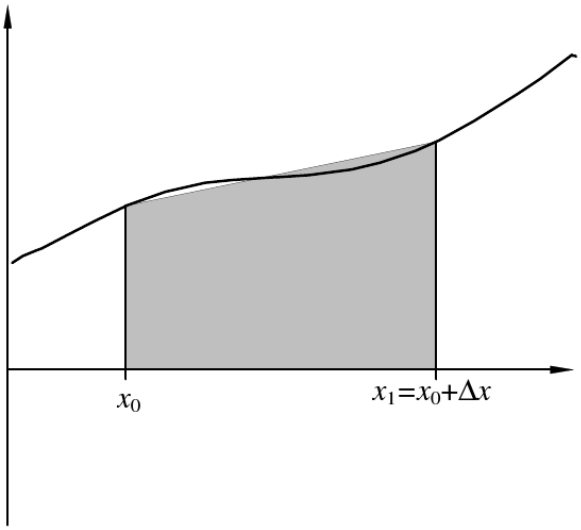


Figure 7.3: Graphical interpretation of the trapezium rule.

analytically. This allows us to decompose the original integral as

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} \tilde{f}(x)dx. \tag{7.9}$$

The error in the approximation Eq. (7.7) is $\mathcal{O}(\Delta x)$. This results in an integration error in Eq. (7.8) of $\mathcal{O}(\Delta x^3)$ (local error) and finally in Eq. (7.9) of $\mathcal{O}(N\Delta x^3) = \mathcal{O}(\Delta x^2)$ (global error) or equivalently $\mathcal{O}(N^{-2})$.
Repeating the same procedure in d dimensions is straightforward. Assuming that our grid has $\tilde{N}$ intervals along each direction, each carrying a global error $\mathcal{O}(\tilde{N}^{-2})$, the total integration error is given by $\mathcal{O}(N^{-\frac{2}{d}})$, where $N = \tilde{N}^d$.

Monte Carlo Methods

Again we consider the one-dimensional integral in Eq. (7.6). In Monte Carlo methods, we approximate this integral (see Eq. (7.2)) by

$$I = \int_a^b f(x)dx \approx I_{\mathrm{MC}} = \frac{b-a}{N} \sum_{i=1}^{N} f(x_i) = (b-a)\langle f \rangle, \tag{7.10}$$

$\langle f \rangle$ being the sample mean of the integrand and $x_i \in [a, b)$. The variance of f is given by

$$\mathbb{V}(f) = \frac{1}{N-1} \sum_{i=1}^{N} \left(\langle f \rangle - f(x_i)\right)^2, \tag{7.11}$$

where the denominator $N-1$ is chosen in order to obtain the unbiased estimate of the sample variance. Using the central limit theorem, the variance of the approximated integral I_{MC} can be linked to the variance of the integrand f

$$\mathbb{V}(I_{\mathrm{MC}}) = \frac{(b-a)^2 \mathbb{V}(f)}{N}. \tag{7.12}$$

Thus, the Monte Carlo integration error scales like

$$\delta I_{\mathrm{MC}} = \sqrt{\mathbb{V}(I_{\mathrm{MC}})} \sim N^{-\frac{1}{2}}. \tag{7.13}$$

Interestingly, in d dimensions, we only need to replace the factor $(b-a)$ by the volume $|V|$ of the integration domain and find the important result that the error scaling of $\mathcal{O}(N^{-\frac{1}{2}})$ remains: The error of a Monte Carlo integration is independent of the dimension of the integral! When we remember that for conventional methods the integration error for a d-dimensional integral scales like $\mathcal{O}(N^{-\frac{2}{d}})$, we can conclude that for $d > 4$, Monte Carlo methods become more efficient as they require to evaluate less points in the integration domain to obtain the same accuracy.

7.3 Simple Sampling Monte Carlo

Throughout the following sections, we are going to explore different classes of Monte Carlo methods mostly differing in the construction of the sampling process. Here, we start with the simplest, most straight-forward one: Simple Sampling Monte Carlo. As already mentioned, Monte Carlo methods using simple sampling choose uniformly distributed points in the integration domain. In 1D this becomes particularly simple and Eq. 7.2 reduces to

$$\int_a^b f(x)dx \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i), \tag{7.14}$$

where we draw x_i uniformly in $[a, b)$. Note that this can simply be achieved with a linear transformation. Suppose that x is a uniform random number in $[0, 1)$, then

$$x' = (b-a)x + a \tag{7.15}$$

is a uniform random number in $[a, b)$.
A more complicated example is the following one: Consider N hard spheres of radius R in a 3D box of volume V, see Fig. 7.4. Each sphere is characterized by its center position $\boldsymbol{r}_i = (x_i, y_i, z_i),\ 1 \leq i \leq N$. The distance between two points is induced by the Euclidean norm

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}. \tag{7.16}$$

Since we assume that the spheres are hard, they cannot overlap and thus we require

$$d_{ij} > 2R. \tag{7.17}$$

Now, given our box filled with N spheres, we might be interested in finding the average distance between the centers of the spheres for all possible configurations. Mathematically, this average is described by the following integral

$$D = \frac{\int \frac{2}{N(N-1)} \sum_{i<j} d_{ij} d^3r_1 \dots d^3r_N}{\int d^3r_1 \dots d^3r_N}, \tag{7.18}$$

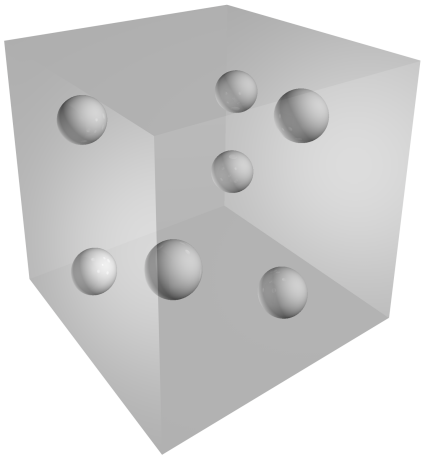


Figure 7.4: A cube filled with spheres.

where the integration boundaries need to take into account that the spheres cannot overlap and the factor $\frac{2}{N(N-1)}$ is a combinatorical normalization factor for the sum [2]. It is clear that in practice we cannot solve this integral analytically. Consequently, we can use a Monte Carlo approach to obtain an approximation. The idea to obtain a sample the integration domain is relatively simple:

- Choose a position within the box.
- Check if a sphere centered at the chosen position overlaps with the boundaries of the box or any other sphere that is already inside the box. If it does, reject it and choose a new position. If it does not, place the sphere inside the box.
- Repeat the process until all N spheres are placed. Once the particular N-sphere configuration is obtained, compute the expectation value $\langle d_{ij} \rangle_{i,j}$.

Repeating the process M times and averaging over all obtained expectation values $\langle d_{ij} \rangle_{i,j}$ will result in an approximation of D with error $\mathcal{O}(M^{-\frac{1}{2}})$. It is important to note that our result will of course depend on the number of spheres positioned: Placing many spheres in the volume will significantly slow down the algorithm as the number of rejections increases. It might even be possible that not every sphere can be placed at all!

[2] Consider a particular configuration of spheres in the box. For this configuration, the average distance between the spheres is simply the sum of the distances between each pair divided by the number of possible pairs that we can build. The normalization factor is of combinatorial origin. In fact, it stems from randomly drawing 2 spheres from a finite (N) population without replacement. Therefore, it is given by $N(N-1)$ (For the first sphere, we have N possible choices and for the second sphere $N-1$ choices.) This means that the average distance for this configuration is given by

$$\langle d_{ij} \rangle_{i,j \in \{1,\dots,N\}} = \frac{1}{N(N-1)} \sum_{i \neq j} d_{ij} = \frac{2}{N(N-1)} \sum_{i<j} d_{ij} \tag{7.19}$$

and since $d_{ij} = d_{ji}$.

7.3.1 Control Variates

As we can understand from Fig. 7.2, simple sampling works best when the function f, that we want to integrate over, is flat. But we might ask ourselves: How can we effectively integrate non-flat functions? One possibility is via *control variates*. The idea is quite simple. Let us write our integral as

$$I = \int_a^b f(x)dx = \int_a^b f(x) - g(x)dx + \int_a^b g(x)dx. \tag{7.20}$$

and find g such that

- $f - g$ is flat and
- $\int_a^b g(x)dx$ is known (or can at least be calculated easily).

If we can find such a function g, then we can compute

$$\int_a^b f(x) - g(x)dx \tag{7.21}$$

using simple sampling Monte Carlo.
Let us analyze the method of control variates a little bit. The variance of the function $f - g$ is given by

$$\text{var}(f - g) = \text{var}(f) + \text{var}(g) - 2\text{cov}(f, g), \tag{7.22}$$

$\text{cov}(f, g)$ being the covariance between f and g. Therefore, in order to be useful, we require that $\text{var}(f - g) < \text{var}(f)$ or, in other words, $f - g$ is flatter than f. This is achieved as long as $\text{var}(g) < 2\text{cov}(f, g)$, i.e. f and g need to be positively correlated.

7.4 Importance Sampling Monte Carlo

Consider again a non-flat function f, see Fig. 7.5. An alternative to the method of control variates is to use importance sampling. Instead of chosing our samples uniformly in the integration domain, we are aiming at picking more samples close to regions where the integral has higher weight, e.g. close to a singularity.
The idea of importance sampling is similar to the one that we have seen for the control variates. We want to find a transformation to make the integrand flatter. However, this time we choose a positive function g, normalized over the integration domain [3] , such that

$$\frac{f(x)}{g(x)} \tag{7.24}$$

[3]

$$g(x) > 0, \forall x \in [a, b) \quad \int_a^b g(x)dx = 1 \tag{7.23}$$

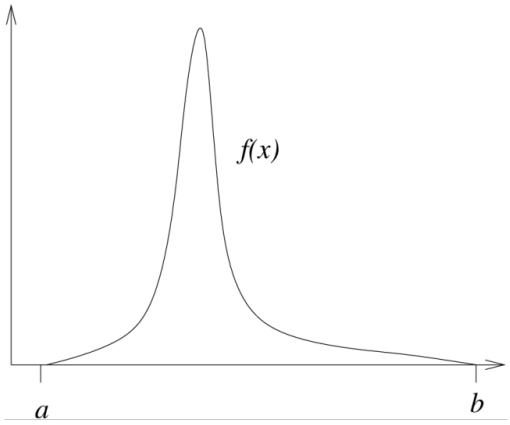


Figure 7.5: Illustration of importance sampling.

is flat. This allows us to rewrite our integral in the following way

$$I = \int_a^b f(x)dx = \int_a^b \frac{f(x)}{g(x)} g(x)dx = \int_a^b \frac{f(x)}{g(x)} dG(x) = \int_{G(a)}^{G(b)} \frac{f(G^{-1}(y))}{g(G^{-1}(y))} dy, \tag{7.25}$$

where

$$G(x) = \int_a^x g(x')dx' \tag{7.26}$$

and in the last step we changed variables. Again we can compute this integral, with flatter integrand, with simple sampling

$$I_{\mathrm{MC}} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(G^{-1}(y_i))}{g(G^{-1}(y_i))} \tag{7.27}$$

or, if we cannot determine G^{-1}, we can sample x_i according to the distribution g and compute

$$I_{\mathrm{MC}} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{g(x_i)}. \tag{7.28}$$

Note in both cases the normalization numerator does not appear because $G(b) - G(a) = 1 - 0 = 1$.

7.5 Quasi-Monte Carlo

We have already seen in Ch. 3.2 the basic idea behind Quasi-Monte Carlo methods. Instead of approximating an integral

$$I \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) \tag{7.29}$$

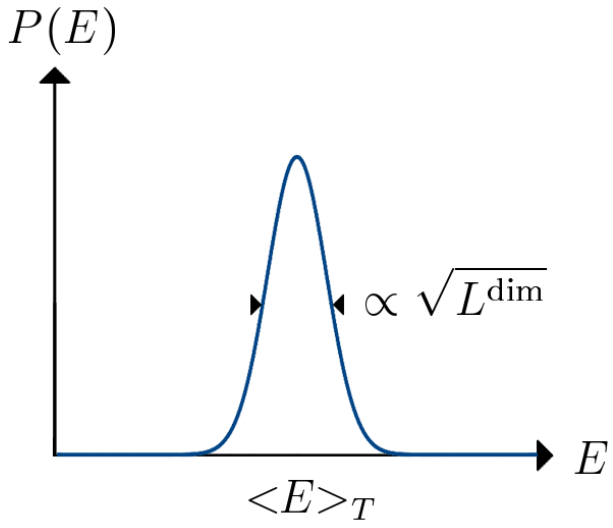


Figure 7.6: Energy distribution.

with uniformly distributed numbers, we use a low discrepancy sequence over the integration domain. We have seen that this gives us an upper bound for the integration error scaling as

$$\mathcal{O}\left(\frac{(\log(N))^d}{N}\right) \tag{7.30}$$

as long as $V(f) < \infty$. In practice, it has been observed that the convergence rate of Quasi-Monte Carlo methods is even faster than its theoretical bound [27]. However, this advantage is not universal and a few obvious drawbacks are immediately visible:

- For $\mathcal{O}\left(\frac{(\log(N))^d}{N}\right)$ to be superior over $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$, we require that $N > 2^d$.
- In practice, many functions do not satisfy $V(f) < \infty$.

Some of these deficiencies can be overcome by using so-called *randomized* Quasi-Monte Carlo methods but these are out of the scope of this introduction.

7.6 Markov Chain Monte Carlo

In most cases, sampling from the phase space of a physical system based on uniform random numbers is very inefficient because the underlying distribution often exhibits huge peaks and regions where it is virtually zero. As an example, we may consider the kinetic energy of an ideal gas. The distribution of the mean energy will exhibit a sharp peak, which becomes sharper with increasing system size (see Fig. 7.6).
There exist many different methods which avoid unnecessary sampling of regions where the system is unlikely to be found (importance sampling). A common way to efficiently choose appropriate samples out of a large pool of possible configurations is to explore the phase space using a *Markov chain*.

7.6.1 Markov Chain

According to Wikipedia, a Markov chain is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. More formally, it is a sequence $X_1, X_2, \ldots$ of random variables which satisfy the following property

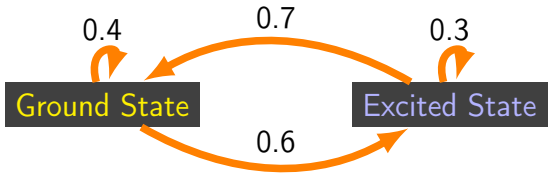
$$P(X_n = x_n | X_{n-1} = x_{n-1}, \ldots X_1 = x_1) = P(X_n = x_n | X_{n-1} = x_{n-1}). \tag{7.31}$$

As a first specific example we consider a Markov chain on the discrete state space $\{G, E\}$ (ground state, excited state). A Markov chain on this space is determined by the initial distribution

$$p_0 = P(X_1 = G), \quad p_1 = P(X_1 = E) \tag{7.32}$$

and the one-step transition probabilities

$$p_{00} = P(X_2 = G | X_1 = G), \quad p_{10} = P(X_2 = G | X_1 = E), \quad p_{01} = 1 - p_{00}, \quad p_{11} = 1 - p_{10}. \tag{7.33}$$



More generally, we can introduce the virtual time τ [4] and call $T(X \to Y)$ the *transition probability* from a configuration X (at time τ_i) to a new configuration Y (at time τ_{i+1}). We impose the following properties on the transition probability:

1. Normalization: $\sum_Y T(X \to Y) = 1$,
2. Reversibility: $T(X \to Y) = T(Y \to X)$,
3. Ergodicity: $T(X \to Y) > 0$.

In order to implement importance sampling, not every new configuration Y will be accepted. So we will handle this with an *acceptance probability* $A(X \to Y)$. That is, we propose a new configuration and use an acceptance probability such that we only sample the important regions in phase space. We use $A(X \to Y)$ to denote the probability of acceptance of a new configuration Y starting from a configuration X. In practice, we are interested in the overall probability of a configuration actually making it through these two steps; this probability is the product of the transition probability $T(X \to Y)$ and the acceptance probability $A(X \to Y)$ and is called the *Markov chain probability*:

$$W(X \to Y) = T(X \to Y) A(X \to Y). \tag{7.34}$$

If we are interested in the evolution of the probability $p(X, \tau)$ that the system is in state X at time τ, we can derive the evolution equation in the following way:

[4] The virtual time only represents the steps of a stochastic process and should not be confused with physical time.

- A configuration X is reached by coming from Y (this will contribute positively to $p(X,\tau)$) and
- a configuration X is left by going to some other configuration Y (this will decrease the probability $p(X,\tau)$).

The first of these two processes is proportional to the probability for a system to be in Y, $p(Y)$, while the second one needs to be proportional to the probability for a system to be in X, $p(X)$. When we combine all of this, we obtain the so-called master equation:

$$\frac{dp(X,\tau)}{d\tau} = \underbrace{\sum_{Y,Y\neq X} p(Y)W(Y\to X)}_{\text{in-flow}} - \underbrace{\sum_{Y,Y\neq X} p(X)W(X\to Y)}_{\text{out-flow}}. \tag{7.35}$$

If we additionally require that W satisfies the following properties:

1. Normalization: $\sum_Y W(X\to Y) = 1$,
2. Homogeneity: $\sum_Y p(Y)W(Y\to X) = p(X)$,
3. Ergodicity: $W(X\to Y) > 0, \forall X, Y$,

then the Markov chain converges towards a unique stationary distribution p_{st} which does not change in time

$$\frac{dp_{\text{st}}(X,\tau)}{d\tau} = 0. \tag{7.36}$$

It then follows that

$$\sum_{Y,Y\neq X} p_{\text{st}}(Y)W(Y\to X) = \sum_{Y,Y\neq X} p_{\text{st}}(X)W(X\to Y). \tag{7.37}$$

A sufficient—but not necessary—condition for this equation to hold is the so-called *detailed balance condition*

$$p_{\text{st}}(Y)W(Y\to X) = p_{\text{st}}(X)W(X\to Y). \tag{7.38}$$

It determines which Markov chain probabilities W we can choose such that our Markov chain samples configurations according to our desired probability distribution p_{st}. In the following, we are going to discuss two possible choices for W.

7.6.2 $M(RT)^2$ Algorithm

The Metropolis algorithm—or more precisely $M(RT)^2$ algorithm, named after its inventors—proposes the following choice for the Markov chain probability W:

- T uniform and
- A such that

$$A_M(X\to Y) = \min\left[1, \frac{p_{\text{st}}(Y)}{p_{\text{st}}(X)}\right]. \tag{7.39}$$

We can easily show that the Metropolis Markov chain probability satisfies the detailed balance equation. Since T is uniform, $T(X \to Y) = T(Y \to X)$ such that Eq. (7.38) can be restated as

$$\frac{p_{\text{st}}(Y)}{p_{\text{st}}(X)} = \frac{A(X \to Y)}{A(Y \to X)}. \tag{7.40}$$

Without loss of generality, we assume that $A_M(X \to Y) = p_{\text{st}}(Y)/p_{\text{st}}(X) < 1$. Since $A_M(Y \to X) = \min\left[1, 1/A_M(X \to Y)\right]$, this means that $A_M(Y \to X) = 1$ which concludes the proof.

7.6.3 Glauber Algorithm (Not Exam-Relevant)

The Glauber algorithm proposes the following choice for the Markov chain probability W:

- T uniform and
- A such that

$$A_G(X \to Y) = \frac{1}{\frac{p_{\text{st}}(X)}{p_{\text{st}}(Y)} + 1}. \tag{7.41}$$

Again, we can easily show that the Glauber Markov chain probability satisfies the detailed balance equation. Since T is uniform, $T(X \to Y) = T(Y \to X)$ such that Eq. (7.38) can be restated as

$$\frac{p_{\text{st}}(Y)}{p_{\text{st}}(X)} = \frac{A(X \to Y)}{A(Y \to X)}. \tag{7.42}$$

Now, using Eq. (7.41), we can calculate

$$\frac{A_G(X \to Y)}{A_G(Y \to X)} = \frac{\frac{1}{\frac{p_{\text{st}}(X)}{p_{\text{st}}(Y)}+1}}{\frac{1}{\frac{p_{\text{st}}(Y)}{p_{\text{st}}(X)}+1}} = \frac{1}{\frac{p_{\text{st}}(X)}{p_{\text{st}}(Y)} + 1} \frac{\frac{p_{\text{st}}(Y)}{p_{\text{st}}(X)} + 1}{1} = \frac{p_{\text{st}}(Y)}{p_{\text{st}}(X) + p_{\text{st}}(Y)} \frac{p_{\text{st}}(Y) + p_{\text{st}}(X)}{p_{\text{st}}(X)} = \frac{p_{\text{st}}(Y)}{p_{\text{st}}(X)}. \tag{7.43}$$

7.6.4 Canonical Ensemble

The goal of statistical physics is to understand the macroscopic properties of an equilibrium system by using statistical arguments on the microscopic parameters that it depends on.
We consider a system of N particles. The phase space of this system is a $6N$-dimensional space spanned by the canonical coordinates $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_N$ and the corresponding conjugate momenta $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N$. The state of the system at time t is a single point in phase space. The time evolution of the system—a trajectory in phase space—is determined by the corresponding Hamiltonian $H(\{\boldsymbol{q}_i(t)\}_i, \{\boldsymbol{p}_i(t)\}_i, t)$ and the equations of motion are a set of $6N$ coupled first-order partial differential equations

$$\dot{p}_i = -\frac{\partial H}{\partial q_i}, \quad \dot{q}_i = \frac{\partial H}{\partial p_i}, \quad \forall i \in \{1, \ldots, 3N\}. \tag{7.44}$$

In general, for a large number N of particles—even with numerical methods—we cannot solve these equations directly. [5] Consequently, we also cannot compute the *time average* of a macroscopic observable Q given by

$$\langle \bar{Q} \rangle = \lim_{T\to\infty} \frac{1}{T} \int_0^T Q(\{\boldsymbol{q}_i(t)\}_i, \{\boldsymbol{p}_i(t)\}_i, t)dt \tag{7.45}$$

as it requires the knowledge of the system's trajectory in phase space.
Instead we introduce a statistical formulation of the problem. Let $\rho(\{\boldsymbol{q}_i(t)\}_i, \{\boldsymbol{p}_i(t)\}_i, t)$ be the probability density that the system is at a certain point in phase space. It satisfies Liouville's theorem

$$\partial_t \rho = \{H, \rho\}, \tag{7.46}$$

where $\{\cdot,\cdot\}$ is the Poisson bracket and in thermal equilibrium we have $\partial_t \rho = 0$. Assuming that the system explores the whole energetically possible phase space over time—or in other words, it reaches all possible microstates—(the so-called *ergodic hypothesis*), we can replace the time average in Eq. (7.45) by a so-called *ensemble average* [6]

$$\langle \bar{Q} \rangle \stackrel{!}{=} \langle Q \rangle = \frac{\int Q\rho d^{3N}q d^{3N}p}{\int \rho d^{3N}q d^{3N}p}. \tag{7.48}$$

An *ensemble* is the set of a large number of copies of our original system. So, the ergodic hypothesis allows us to compute the average of a macroscopic observable by averaging over the copies of our system distributed according to ρ instead of computing the full dynamics and performing the time average.
The most often encountered ensembles in statistical physics are

- Microcanonical ensemble: (E, V, N) constant,
- Canonical ensemble: (T, V, N) constant,
- Grand-canonical ensemble: (T, V, μ) constant.

Here, we focus on the canonical ensemble. In an canonical ensemble the system is placed in a heat bath that allows for an exchange of energy between the system and the bath such that the temperature T is kept constant. The partition function of the canonical ensemble is given by

$$Z_N = \sum_x e^{-\beta H(x)}, \tag{7.49}$$

[5] The area of *Molecular Dynamics* deals with exact particle dynamics and describes methods that—under certain assumptions—can exactly solve the system's dynamics. However, the limiting factor is still the number N of particles.

[6] We often experience the situation, where the configuration space is discrete. In this case the ensemble average is given by

$$\langle Q \rangle = \frac{\sum_x Q(x)\rho(x)}{\sum_x \rho(x)}. \tag{7.47}$$

where $\beta = 1/k_B T$ is the inverse temperature and x are (discrete) configurations in phase space. From the partition function we can derive the free energy—the corresponding thermodynamic potential—

$$F = -\frac{1}{\beta}\log(Z_N) \tag{7.50}$$

and all other related thermodynamic quantities. For the canonical ensemble the phase space density is given by

$$\rho(x) = \frac{1}{Z_N} e^{-\beta H(x)} \tag{7.51}$$

and is normalized $\sum_x \rho(x) = 1$ such that the ensemble average in Eq. (7.47) reduces to

$$\langle Q \rangle = \sum_x Q(x)\rho(x). \tag{7.52}$$

This equation looks insofar familiar as it is the typical kind of application for our Monte Carlo methods. We will see for the example of the two-dimensional Ising model how to apply the Markov Chain Monte Carlo method.

7.6.5 Ising Model

Basic Physics

The Ising model is one of the simplest models that exhibits a (second-order) phase transition. It is used to describe the transition from a *paramagnetic* state to a *ferromagnetic* state upon temperature reduction.

We consider a two-dimensional square lattice, where each lattice site i is occupied by a *classical spin* S_i, see Fig. 7.7. A classical spin can only take the values ± 1, in particular we do not allow for quantum superpositions. The Hamiltonian of the Ising model is given by

$$H(\{S_i\}) = -J\sum_{\langle i,j\rangle} S_i S_j - H\sum_i S_i \tag{7.53}$$

and describes the interaction of magnetic dipole moments [7] [8] [9].

Before we dive into the numerical simulation using Markov Chain Monte Carlo, we try to get a little bit of an intuition and an overview of known properties.

[7] In fact, magnetism in materials does typically not arise from the magnetic interaction between spin moments (small effect) but rather from the exchange interaction generated by Coulomb repulsion and the Pauli exclusion principle (see e.g. the analogous example of bonding/antibonding states in the molecule H_2^+ that you might have considered in the course Quantum Mechanics 2). This effect is typically orders of magnitude larger and the main contribution to our phenomenological parameter J.

[8] The Ising Hamiltonian can be derived from a Hubbard model

$$H = -t\sum_{i,s}\left(c^\dagger_{i+1,s}c_{i,s} + h.c.\right) - U\sum_i n_{i,\uparrow}n_{i,\downarrow} \tag{7.54}$$

at half filling and with strong Coulomb repulsion $U << 0$. In this situation, electrons prefer not to sit on the same lattice site. Since we are at half filling, each lattice site will be occupied by exactly one spinful electron.

[9] The Ising model Eq. (7.53) can be generalized in various ways:

- $J\sum_{\langle i,j\rangle} S_i S_j \to \sum_{i,j} J_{ij} S_i S_j$. This means that we give up the assumption of only considering nearest neighbors and the assumptions of homogeneity (translational independece) and isotropy (directional

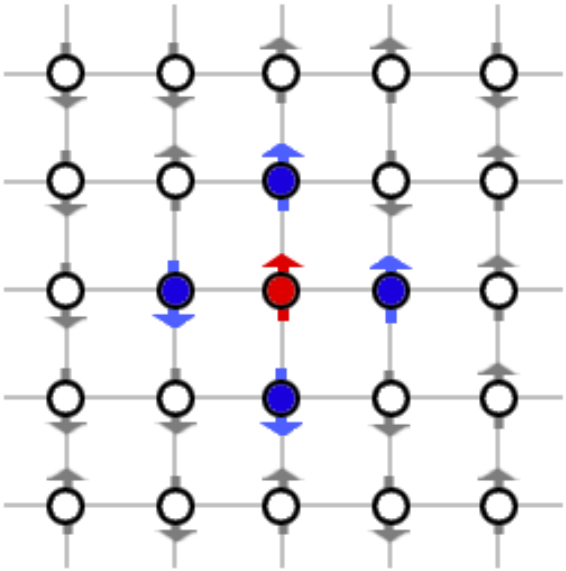


Figure 7.7: Ising spins on a square lattice.

$T = 0$

- $J = 0$ implies that the spins align with the external magnetic field H.
- $H = 0, J > 0$ implies that the spins align with each other (ferromagnet).
- $H = 0, J < 0$ implies that the spins anti-align (anti-ferromagnet).

Each of the above situations describes the *ground state* of the system, i.e. the state of lowest energy (at zero temperature).

$T > 0$ At finite temperatures we have thermal fluctuations. These fluctuations are competing with the (magnetic) ordering. While at low temperatures the fluctuations are small and the system will be close to the ground state, at some point there will occur a 2nd order phase transition to a paramagnetic phase (=disordered).

Critical Temperature T_c The critical temperature is dimension dependent:

- 1D: $T_c = 0$ (analytical solution by Ising, no phase transition at all),

independence).

- Another lattice structure and another dimension d.
- $S_i \to \boldsymbol{S}_i$ with $|\boldsymbol{S}_i| = 1$ and $H \to \boldsymbol{H}$. This model is called Heisenberg model.
- Higher spins.
- ...

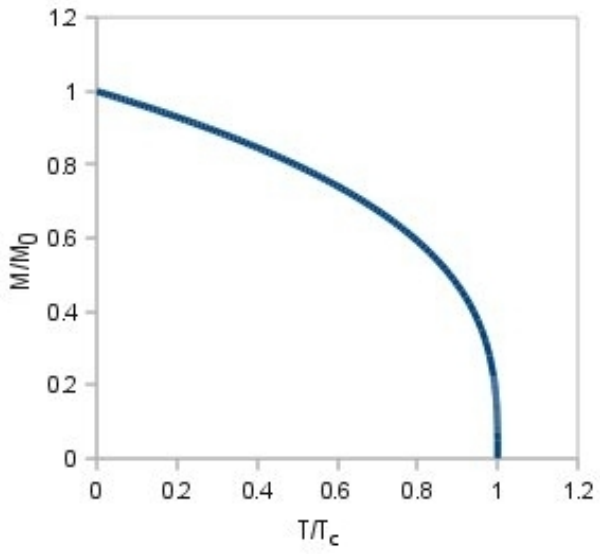


Figure 7.8: Magnetization in units of J/k_B for a 3D Ising model.

- 2D: $T_c = \frac{2}{\log\left(1+\sqrt{2}\right)} \left[\frac{J}{k_B}\right] \approx 2.\ldots$ (analytical solution by Onsager),
- 3D: $T_c \approx 4.\ldots \left[\frac{J}{k_B}\right]$ (numerical solution),
- $\geq$ 4D: T_c is analytically known from mean-field theory (for $d > 4$ the neglected fluctuations are not relevant!).

Order Parameter and Phase Diagram The order parameter of the Ising model is the *spontaneous magnetization*

$$M_s(T) = \lim_{H \gtrless 0} \langle M(T,H) \rangle = \lim_{H \gtrless 0} \left\langle \frac{1}{L^2} \sum_{i=1}^{L^2} S_i \right\rangle, \tag{7.55}$$

where L is the number of spins and is depicted in Fig. 7.8. For $T \lesssim T_c$

$$M(T \lesssim T_c) \sim |T - T_c|^{\beta}, \tag{7.56}$$

where β is a critical exponent. Here, it is very important to note the analogy to the percolation problem in Ch. 5.
The phase diagram is two-dimensional and spanned by the temperature T and the external magnetic field H. For $H = 0$, there is a 2nd order phase transition [10] at $T = T_c$ (one point in the phase diagram), where M_s switches from 0 to a finite value. For $T < T_c$ (a line in the phase diagram) there is a 1st order phase transition at $H = 0$, where M_s switches sign, $M_s \to -M_s$.

[10] In particular, we observe a so-called *spontaneous symmetry breaking*. This means that we break the $\mathbb{Z}_2$ symmetry ($S_i \to -S_i$).

Note that for the Ising model—like for the percolation problem in Ch.5—we can derive other characteristic quantities besides the order parameter and introduce the full zoo of critical exponents. Here, however, restrict our discussions to the order parameter with the focus on illustrating the numerical Markov chain sampling process.

Markov Chain Monte Carlo

Let us consider the field-free case

$$H = -J \sum_{\langle i,j \rangle} S_i S_j. \tag{7.57}$$

Our goal is to obtain the average normalized spontaneous magnetization M_s as a function of temperature T

$$M_s(T) = \left\langle \frac{1}{L^2} \left| \sum_{i=1}^{L^2} S_i \right| \right\rangle, \tag{7.58}$$

where the expectation value is to be understood in the sense of Eq. (7.52). Therefore, we need to sample phase space efficiently using a Markov chain. By requiring that

$$p_{\text{st}} \overset{!}{=} p_{\text{eq}}, \tag{7.59}$$

where

$$p_{\text{eq}}(X) = \frac{1}{Z_{L^2}} e^{-\beta H(X)}, \tag{7.60}$$

X being a particular spin configuration, we can use for example the Metropolis or the Glauber algorithm—both satisfying the detailed balance equation Eq. (7.38)—to obtain our configurations $X_1(T), \ldots, X_N(T)$. For each of these spin configurations, we can compute the corresponding magnetization and approximate the overall spontaneous magnetization as

$$M_s(T) \approx \frac{1}{N} \sum_{i=1}^{N} M(X_i(T)). \tag{7.61}$$

Concretely, for the Metropolis algorithm, the Markov chain is built using single-spin-flip updates:

- Choose a site i uniformly at random.
- Compute $\Delta E = E(Y) - E(X) = 2JS_i \sum_{j,\langle i,j \rangle} S_j$.
- If $\Delta E < 0$, flip the spin.
- If $\Delta E > 0$, draw a uniform random number r. If $r < e^{-\beta \Delta E}$, flip the spin, else not.

7.7 Multi-Level Monte Carlo

This is based on several articles and presentations of Michael B. Giles, the inventor of Multi-Level Monte Carlo. In stochastic models we often have

$$\underset{\text{random input}}{\omega} \longrightarrow \underset{\text{intermediate value}}{S} \longrightarrow \underset{\text{scalar output}}{P}$$

The Monte Carlo estimate for $\mathbb{E}[P]$ is an average of N samples $P(\omega^i)$

$$Y = \frac{1}{N}\sum_{i=1}^{N} P(\omega^i).$$

This is unbiased $\mathbb{E}[Y] = \mathbb{E}[P]$ and the CLT proves that as $N \to \infty$ the error becomes Normally distribute with variance $\frac{1}{N}\mathbb{V}\text{ar}[P] < \infty$ and hence on needs $N = \mathbb{O}(\epsilon^{-2})$ samples to achieve ϵ RMS accuracy. In this particular case we are able to do the calculations exactly however, this is the exception.
In many cases we have

$$\underset{\text{random input}}{\omega} \longrightarrow \underset{\text{intermediate value}}{\hat{S}} \longrightarrow \underset{\text{scalar output}}{\hat{P}}$$

where $\hat{S}, \hat{P}$ are approximations (from discretisaton) to S, P in which case the MC estimate

$$\hat{Y} = \frac{1}{N}\sum_{i=1}^{N} \hat{P}(\omega^i).$$

is biased, and the MSE is

$$\mathbb{E}[(\hat{Y} - \mathbb{E}(P)^2] = \frac{1}{N}\mathbb{V}\text{ar}[\hat{P}] + (\mathbb{E}(\hat{P}) - \mathbb{E}(P))^2, i.e. the discretization error in expectation.$$

Greater accuracy requires larger N and smaller weak error $\mathbb{E}(\hat{P}) - \mathbb{E}(P)$.

SDE Example for Motivation

Lets look at a simple stochastic differential equation (SDE) for path simulation, appearing in finance physics etc. We look at the simplest scalar form

$$dS_t = a(S_t, t)dt + b(S_t, t)dE_t$$

with a we denote the deterministic drift term, b is the diffusion term and W is the increment of a Brownian motion – Normally distributed with variance dt.
This is usually approximated by the Euler-Maruyama (standard Euler forward time integration) method

$$\hat{S}_{t_{n+1}} = \hat{S}_{t_n} + a(\hat{S}_{t_n} \cdot t_n)h + b(\hat{S}_{t_n} \cdot t_n)\Delta W_n$$

with uniform time step h and increments ΔW_n (change of Brownian path) with variance h.
In simple applications, the output of interest is as function of the final value

$$\hat{P} \equiv f(\hat{S}_T).$$

Solving standard geometric Brownian Motion for 2 different timesteps:
Two kinds of discretization errors are involved here:

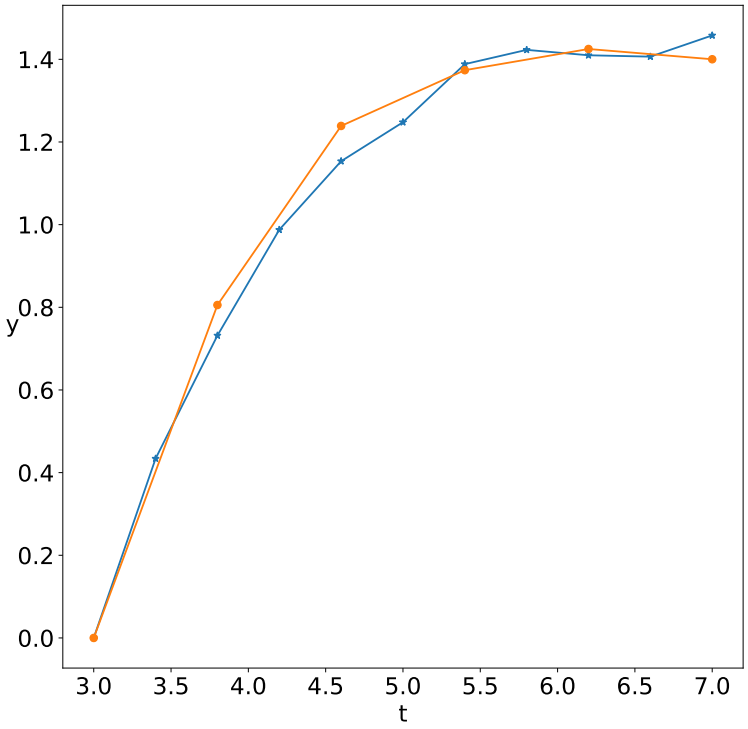


Figure 7.9: Standard geometric Brownian Motion with two different time steps

1. weak error:
$$\mathbb{E}[(\hat{P})] - \mathbb{E}[(P)] = \mathbb{O}(h)$$

2. strong error (in expectaion):
$$\left(\mathbb{E}\left[\sup_{[0,T]} (\hat{S}_t - S_t)^2 \right] \right)^{1/2} = \mathbb{O}(h^{1/2})$$

Better schemes exists for example Milstein discretization for which both weak and strong error are of $\mathbb{O}(h)$.
The MSE is
$$\frac{1}{N}\mathbb{V}\text{ar}[\hat{P}] + (\mathbb{E}(\hat{P}) - \mathbb{E}(P))^2 = aN^{-1} + bh^2$$
If we want this to be ϵ^2, then we need
$$N = \mathbb{O}(\epsilon^{-2}), \quad h = \mathbb{O}(\epsilon)$$
hence, the total computational cost is
$$\mathbb{O}(\epsilon^{-3})$$

this is the number of path × the number of time steps per path.
To improve we can work on two fronts:

- reduce N - variance reduction or Quasi-Monte Carlo methods
- reduce the cost of each path - MLMC

7.7.1 Control variates and two-level MLMC

One of the classic approaches to Monte Carlo variance reduction is through the use of a control variate. Suppose we wish to estimate $\mathbb{E}[f]$, and there is a control variate g which is well correlated to f and has a known expectation $\mathbb{E}[g]$. In that case, we can use the following unbiased estimator for $\mathbb{E}[f]$:

$$\frac{1}{N}\sum_{n=1}^{N}\left\{f^{(n)}-\lambda\left(g^{(n)}-\mathbb{E}[g]\right)\right\}. \tag{7.62}$$

The optimal value for λ is $\rho\sqrt{\mathbb{V}[f]\,/\,\mathbb{V}[g]}$, where ρ is the correlation between f and g, and the variance of the control variate estimator is reduced by factor $1-\rho^2$ compared to the standard estimator.
A two-level version of MLMC (multilevel Monte Carlo) is very similar. If we want to estimate $\mathbb{E}[P_1]$ but it is much cheaper to simulate $P_0 \approx P_1$, then since

$$\mathbb{E}[P_1] = \mathbb{E}[P_0] + \mathbb{E}[P_1 - P_0] \tag{7.63}$$

we can use the unbiased two-level estimator

$$N_0^{-1}\sum_{n=1}^{N_0} P_0^{(n)} \;+\; N_1^{-1}\sum_{n=1}^{N_1}\left(P_1^{(n)}-P_0^{(n)}\right). \tag{7.64}$$

Here $P_1^{(n)}$–$P_0^{(n)}$ represents the difference between P_1 and P_0 for the same underlying stochastic sample, so that $P_1^{(n)}$–$P_0^{(n)}$ is small and has a small variance; the precise construction depends on the application and various examples will be shown later. The two key differences from the control variate approach are that the value of $\mathbb{E}[P_0]$ is not known, so has to be estimated, and we use $\lambda = 1$.
If we define C_0 and C_1 to be the cost of computing a single sample of P_0 and $P_1 - P_0$, respectively, then the total cost is $N_0\,C_0$+$N_1\,C_1$, and if V_0 and V_1 are the variance of P_0 and P_1–P_0, then the overall variance is $N_0^{-1}V_0$+$N_1^{-1}V_1$, assuming that $\displaystyle\sum_{n=1}^{N_0} P_0^{(n)}$ and $\displaystyle\sum_{n=1}^{N_1}\left(P_1^{(n)}-P_0^{(n)}\right)$ use independent samples.
Hence, treating the integers N_0, N_1 as real variables and performing a constrained minimisation using a Lagrange multiplier, the variance is minimised for a fixed cost by choosing $N_1/\,N_0 = \sqrt{V_1/C_1}\,/\sqrt{V_0/C_0}$.

7.7.2 Multilevel Monte Carlo

The full multilevel generalisation is quite natural: given a sequence $P_0, P_1, \ldots,$ which approximates P_L with increasing accuracy, but also increasing cost, we have the simple identity

$$\mathbb{E}[P_L] = \mathbb{E}[P_0] + \sum_{\ell=1}^{L} \mathbb{E}[P_\ell - P_{\ell-1}], \tag{7.65}$$

and therefore we can use the following unbiased estimator for $\mathbb{E}[P_L]$,

$$N_0^{-1} \sum_{n=1}^{N_0} P_0^{(0,n)} \; + \; \sum_{\ell=1}^{L} \left\{ N_\ell^{-1} \sum_{n=1}^{N_\ell} \left(P_\ell^{(\ell,n)} - P_{\ell-1}^{(\ell,n)} \right) \right\} \tag{7.66}$$

with the inclusion of the level ℓ in the superscript (ℓ, n) indicating that the samples used at each level of correction are independent.
If we define C_0, V_0 to be the cost and variance of one sample of P_0, and C_ℓ, V_ℓ to be the cost and variance of one sample of $P_\ell - P_{\ell-1}$, then the overall cost and variance of the multilevel estimator is

$$\sum_{\ell=0}^{L} N_\ell \; C_\ell \tag{7.67}$$

and

$$\sum_{\ell=0}^{L} N_\ell^{-1} \; V_\ell, \tag{7.68}$$

respectively.
For a fixed cost, the variance is minimised by choosing

$$N_\ell = \lambda \sqrt{V_\ell \, / \, C_\ell} \tag{7.69}$$

for some value of the Lagrange multiplier λ. In particular, to achieve an overall variance of ε^2 requires that

$$\lambda = \varepsilon^{-2} \; \sum_{\ell=0}^{L} \sqrt{V_\ell \; C_\ell}. \tag{7.70}$$

The total computational cost is then

$$C = \varepsilon^{-2} \left(\sum_{\ell=0}^{L} \sqrt{V_\ell \; C_\ell} \right)^2 . \tag{7.71}$$

It is important to note whether the product $V_\ell \, C_\ell$ increases or decreases with ℓ, i.e. whether or not the cost increases with level faster than the variance decreases. If it increases with level, so that the dominant contribution to the cost comes from $V_L \, C_L$ then we have $C \approx \varepsilon^{-2} V_L \, C_L$, whereas if it decreases and the dominant contribution comes from $V_0 \, C_0$ then $C \approx \varepsilon^{-2} V_0 \, C_0$. This contrasts to the standard MC cost of approximately $\varepsilon^{-2} V_0 \, C_L$, assuming that the cost of computing P_L is similar to the cost of computing $P_L - P_{L-1}$, and that $\mathbb{V}[P_L] \approx \mathbb{V}[P_0]$. This shows that in the first case the MLMC cost is reduced by factor V_L / V_0, corresponding to the ratio of the variances $\mathbb{V}[P_L - P_{L-1}]$ and $\mathbb{V}[P_L]$, whereas in the second case it is reduced by factor C_0 / C_L, the ratio of the costs of computing P_0 and $P_L - P_{L-1}$. If the product $V_\ell \; C_\ell$ does not vary with level, then the total cost is $\varepsilon^{-2} L^2 \, V_0 \, C_0 = \varepsilon^{-2} L^2 \, V_L \, C_L$.

MLMC Theorem (Not Exam-Relevant)

Above, we considered the case of a general multilevel method in which the output P_L on the finest level corresponds to the quantity of interest. However, in many infinite-dimensional applications, such as in SDEs and SPDEs, the output P_ℓ on level ℓ is an approximation to a random variable P. In this case, the mean square error (MSE) has the usual decomposition into the total variance of the multilevel estimator, plus the square of the bias $(\mathbb{E}[P_L - P])^2$. To achieve an MSE which is less than ε^2, it is sufficient to ensure that each of these terms is less than $\frac{1}{2}\varepsilon^2$. This leads to the following theorem:
Let P denote a random variable, and let P_ℓ denote the corresponding level ℓ numerical approximation. If there exist independent estimators Y_ℓ based on N_ℓ Monte Carlo samples, and positive constants $\alpha, \beta, \gamma, c_1, c_2, c_3$ such that $\alpha \geq \frac{1}{2}\min(\beta, \gamma)$ and

i) $|\mathbb{E}[P_\ell - P]| \leq c_1 2^{-\alpha \ell}$

ii) $\mathbb{E}[Y_\ell] = \begin{cases} \mathbb{E}[P_0], & \ell = 0 \\ \mathbb{E}[P_\ell - P_{\ell-1}], & \ell > 0 \end{cases}$

iii) $\mathbb{V}[Y_\ell] \leq c_2 N_\ell^{-1} 2^{-\beta \ell}$

iv) $\mathbb{E}[C_\ell] \leq c_3 N_\ell 2^{\gamma \ell}$, where C_ℓ is the computational complexity of Y_ℓ

then there exists a positive constant c_4 such that for any $\varepsilon < e^{-1}$ there are values L and N_ℓ for which the multilevel estimator

$$Y = \sum_{\ell=0}^{L} Y_\ell,$$

has a mean-square-error with bound

$$MSE \equiv \mathbb{E}\left[(Y - \mathbb{E}[P])^2\right] < \varepsilon^2$$

with a computational complexity C with bound

$$\mathbb{E}[C] \leq \begin{cases} c_4 \varepsilon^{-2}, & \beta > \gamma, \\ c_4 \varepsilon^{-2} (\log \varepsilon)^2, & \beta = \gamma, \\ c_4 \varepsilon^{-2-(\gamma-\beta)/\alpha}, & \beta < \gamma. \end{cases}$$

The statement of the theorem is a slight generalisation of the original theorem in [28]. It corresponds to the theorem and proof in [**?**], except for the minor change to expected costs to allow for applications such as jump-diffusion modelling in which the simulation cost of individual samples is itself random.
The theorem is based on the idea of a geometric progression in the levels of approximation, leading to the exponential decay in the weak error in condition *i)*, and the variance in condition *iii)*, as well as the exponential increase in the expected cost in condition *iv)*. This geometric progression was based on experience with multigrid methods in the iterative solution of large systems of linear equations, but it is worth noting that it is not necessarily the optimal choice in all circumstances.

The result of the theorem merits some discussion. In the case $\beta > \gamma$, the dominant computational cost is on the coarsest levels where $C_\ell = O(1)$ and $O(\varepsilon^{-2})$ samples are required to achieve the desired accuracy. This is the standard result for a Monte Carlo approach using i.i.d. samples; to do better would require an alternative approach such as the use of Latin hypercube sampling or quasi-Monte Carlo methods. In the case $\beta < \gamma$, the dominant computational cost is on the finest levels. Because of condition *i)*, $2^{-\alpha L} = O(\varepsilon)$, and hence $C_L = O(\varepsilon^{-\gamma/\alpha})$. If $\beta = 2\alpha$, which is usually the largest possible value for a given α, for reasons explained below, then the total cost is $O(C_L)$ corresponding to $O(1)$ samples on the finest level, again the best that can be achieved. The dividing case $\beta = \gamma$ is the one for which both the computational effort, and the contributions to the overall variance, are spread approximately evenly across all of the levels; the $(\log \varepsilon)^2$ term corresponds to the L^2 factor in the corresponding discussion in Sec. 7.7.2.

The natural choice for the multilevel estimator is

$$Y_\ell = N_\ell^{-1} \sum_i P_\ell(\omega_i) - P_{\ell-1}(\omega_i), \tag{7.72}$$

where $P_\ell(\omega_i)$ is the approximation to $P(\omega_i)$ on level ℓ, and $P_{\ell-1}(\omega_i)$ is the corresponding approximation on level $\ell-1$ for the same underlying stochastic sample ω_i. Note that $\mathbb{V}[P_\ell - P_{\ell-1}]$ is usually similar in magnitude to $\mathbb{E}[(P_\ell - P_{\ell-1})^2]$ which is greater than $(\mathbb{E}[P_\ell - P_{\ell-1}])^2$; this implies that $\beta \le 2\alpha$ and hence the condition in the theorem that $\alpha \ge \frac{1}{2}\min(\beta, \gamma)$ is satisfied.

However, the multilevel theorem allows for the use of other estimators, provided they satisfy the restriction of condition *ii)* which ensures that $\mathbb{E}[Y] = \mathbb{E}[P_L]$. Two examples of this will be given later in the paper. In the first, slightly different numerical approximations are used for the coarse and fine paths in SDE simulations, giving

$$Y_\ell = N_\ell^{-1} \sum_i P_\ell^f(\omega_i) - P_{\ell-1}^c(\omega_i). \tag{7.73}$$

Provided $\mathbb{E}[P_\ell^f] = \mathbb{E}[P_\ell^c]$ so that the expectation on level ℓ is the same for the two approximations, then condition *ii)* is satisfied and no additional bias (other than the bias due to the approximation on the finest level) is introduced into the multilevel estimator. The second example defines an antithetic ω_i^a with the same distribution as ω_i, and then uses the multilevel estimator

$$Y_\ell = N_\ell^{-1} \sum_i \tfrac{1}{2}\left(P_\ell(\omega_i) + P_\ell(\omega_i^a)\right) - P_{\ell-1}(\omega_i). \tag{7.74}$$

Since $\mathbb{E}[P_\ell(\omega_i^a)] = \mathbb{E}[P_\ell(\omega_i)]$, then again condition *ii)* is satisfied. In each case, the objective in constructing a more complex estimator is to achieve a greatly reduced variance $\mathbb{V}[Y_\ell]$ so that fewer samples are required.

This section is based on several articles and presentations of Michael B. Giles and Stefan Heinrich, see for example Refs. [29, 30, 28, 31, 32, 33, 34].

For further information on multilevel Monte Carlo methods, see the webpage http://people.maths.ox.ac.uk/gilesm/mlmc_community.html which lists the research groups working in the area, and their main publications.

Chapter 8

Finite Difference Methods

8.0.1 Basic Concepts in Error Estimation

Approximation is a central concept in almost all the uses of mathematics. One must often be satisfied with approximate values of the quantities with which one works. Another type of approximation occurs when one ignore some quantities which are small compared to other quantities. Such approximations are often necessary to insure that the mathematical and numerical treatment of a problem does not become hopelessly complicated.
We can distinguish the following sources of error

A *Errors in given input data* Input data can be the results of measurements which have been influenced by statistical and systematical errors. Format conversions for example 12-bit analog digital to double precision.

B *Rounding errors during the computation* A rounding error occurs whenever an irrational number, for example π, is shortened ("rounded off") to a fixed number of digits, or when a decimal fraction is converted to the binary form used in the computer. The limitation of floating-point numbers in a computer leads at times to a loss of information that, depending on the context, may or may not be important. Two typical cases are

1. If the computer cannot handle numbers which have more than, say, s digits, then the exact product of two s-digit numbers (which contains $2s$ or $2s-1$ digits) cannot be used in subsequent calculations; the product must be rounded off
2. In a floating-point computation, if a relatively small term b is added to a, then some digits of b are "shifted out", and they will not have any effect on future quantities that depend on the value of $a+b$.

C *Truncation Errors* These are errors committed when a limiting process is truncated (broken off) before one has come to the limiting value. A truncation error occurs, for example, when an infinite series is broken off after a finite number of terms, or when a derivative is approximated with a difference quotient (although in this case the term discretization error is better). Another example is when a nonlinear function is approximated with a linear function, as in Newton's method. Observe the distinction between truncation error and rounding error

D *Simplifications in the Mathematical Model* In most of the applications of mathematics, one makes idealizations. In a mechanical problem one might assume that a string in a pendulum has zero mass. In many other types of problems it is advantageous to consider a given body to be homogeneously filled with matter, instead of being built of atoms. For a calculation in economics, one might assume that the rate of interest is constant over a given period of time. The effects of such sources of error are usually more difficult to estimate than the types named in A, B, and C.

E *Human & Machine Errors* In all numerical work, one must expect that clerical errors, errors in hand calculation, and misunderstandings will occur. One should even be aware that textbooks (!), tables, and formulas may contain errors. When one uses computers, one can expect errors in the program itself, typing errors in entering the data, operator errors, and pure machine errors [35]

Examples: Errors of type E do occur, sometimes with serious consequences. The first American Venus probe was lost due to a program fault caused by the inadvertent substitution of a statement in a Fortran program of the form `DO 3 I = 1.3` for one of the form `DO 3 I= 1,3`. Erroneously replacing the comma "," with a dot "." converts the intended loop statement into an assignment statement!
A hardware error that got much publicity surfaced in 1994, when it was found that the INTEL Pentium processor gave wrong results for division with floating-point numbers of certain patterns. This was discovered by A Edelman during research on prime numbers and later fixed.
From a different point of view, one may distinguish between controllable and uncontrollable (or unavoidable) error sources. Errors of type A and D are usually considered to be uncontrollable in the numerical treatment (although feedback to the constructor of the mathematical model may sometimes be useful). Errors of type C are usually controllable. For example, the number of iterations in the solution of an algebraic equation, or the step size in a simulation, can be chosen either directly or by setting a tolerance. The rounding error in the individual arithmetic operation (type B) is, in a computer, controllable only to a limited extent, mainly through the choice between single and double precision. A very important fact is, however, that it can often be controlled by appropriate rewriting of formulas or by other changes of the algorithm
An example we compute compounded interest. Consider depositing the amount c every day in an account with an interest rate i compounded daily. With the accumulated capital, the total at the end of the year equals

$$c[(1+x)^n - 1]/x, \text{with } x = \frac{1}{n} \ll 1,$$

and $n = 365$. Using this formula does not give accurate results. The reason is that a rounding error occurs in computing $(1+x) = 1+\bar{x}$ and low order bits of x are lost. For example, if $i = 0.06$, then $i/n = 0.0001643836$; in decimal arithmetic using six digits when this is added to one we get $fl(1+i/n) = 1.000164$, and thus four low order digits are lost.
The problem then is to accurately compute $(1+x)^n = \exp(n\log(1+x))$.

$$\log(1+x) = \begin{cases} x & \text{if } (1+x) = 1 \\ x\frac{\log(1+x)}{(1+x)-1} & \text{otherwise} \end{cases} \tag{8.1}$$

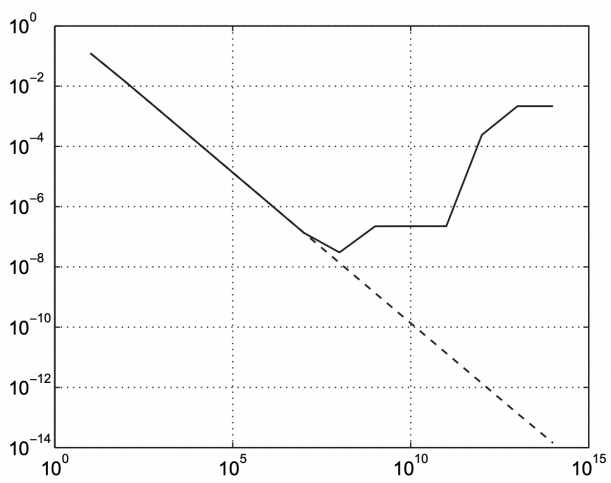


Figure 8.1: Computed values for $n = 10^p, p = 1 : 14$, of the sequences: solid line $|(1 + 1/n)^n - e|$; dashed line $|exp(nlog(1 + 1/n)) - e|$ using (8.1) [36]

.

can be shown to yield accurate results when $x \in [0, 3/4]$ and the computed value of $\log(1 + x)$ equals the exact result rounded (Goldberg, p 12).
To check this formula we recall that the base e of the natural logarithm can be defined by the limit

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n$$

In Figure 8.1 we show computed values, using double precision floating-point arithmetic, of the sequence $|(1 + 1/n)^n - e|$ for $n = 10^p, p = 1 : 14$. More precisely, the expression was computed as $|\exp(nlog(1 + 1/n)) - \exp(1)|$.
A fundamental insight from the above example can be expressed in the following way: *mathematically equivalent* formulas or algorithms are not in general *numerically equivalent*.

8.0.2 Very Basics of Error Propagation

In scientific computing the given input data are usually associated with uncertainties i.e. errors. The uncertainties/errors in the input will propagate and give rise to errors in the output. Here we will give a general feeling of how errors propagate. Error-propagation formulas are also of great interest in the planning and analysis of scientific experiments, a good expose can be found in [36].
Note that rounding errors from each step in a calculation are also propagated to give errors in the final result. For many algorithms a rounding error analysis can be given, which shows that the computed result always equals the exact (or slightly perturbed) result of a nearby problem, where the input data have been slightly perturbed.
We first consider two simple special cases of error propagation. For a sum of an arbitrary number of terms we get the following: in addition (and subtraction) a bound for the absolute

errors in the result is given by the sum of the bounds for the absolute errors of the operands:

$$y = \sum_{i=1}^{n} x_i, \quad |\Delta y| \leq \sum_{i=1}^{n} |\Delta x_i|$$

To obtain a corresponding result for the error propagation in multiplication and division, we start with the observations that for $y = \log x$ we have $\Delta(\log x) \approx \Delta(x)/x$. In words, the relative error in a quantity is approximately equal to the absolute error in its natural logarithm. This is related to the fact that displacements of the same length at different places on a logarithmic scale mean the same relative change of the value. From this we derive the following result:
In multiplication and division, an approximate bound for the relative error is obtained by adding the relative errors of the operands.

$$\left|\frac{\Delta y}{y}\right| \lessapprox \sum_{i=1}^{n} |m_i| \left|\frac{\Delta x_i}{x_i}\right|$$

From this result a general formula for error propagation can be derived. Let the real-valued function $f = f(x_1, x_2, \ldots, x_n)$ be differentiable in a neighborhood of the point $x = (x_1, x_2, \ldots, x_n)$ with errors $\Delta x_1, \Delta x_2, \ldots, \Delta x_n$. Then it holds that

$$\Delta f \approx \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \Delta x_i. \tag{8.2}$$

The for the maximal error in $f(x_1, x_2, \ldots, x_n)$ we obtain the approximate upper bound

$$|\Delta f| \lessapprox \sum_{i=1}^{n} \left|\frac{\partial f}{\partial x_i}\right| |\Delta x_i|. \tag{8.3}$$

where the partial derivatives are evaluated at x.
In order to get a strict bound for , $|\Delta f|$ one should use in (8.3) the maximum absolute values of the partial derivatives in a neighbourhood of the known point x. In most practical situations it suffices to calculate $|\partial f/\partial x_i|$ at x and then add a certain marginal amount (5 to 10 percent, say) for safety. Only if the Δx_i are large or if the derivatives have a large relative variation in the neighbourhood of x need the maximal values be used. (The latter situation occurs, for example, in a neighbourhood of an extremal point of $f(x)$.) The bound in (8.2) is the best possible, unless one knows some dependence between the errors of the terms. Sometimes it can, for various reasons, be a gross overestimate of the real error.

Example: Compute error bounds for $f = x_1^2 - x_2$, where $x_1 = 1.03 \pm 0.01$, $x_2 = 0.45 \pm 0.01$. We obtain

$$\left|\frac{\partial f}{\partial x_1}\right| = |2x_1| \leq 2.1, \quad \left|\frac{\partial f}{\partial x_2}\right| = |-1| = 1.$$

and find $|\Delta f| \leq 2.1 \cdot 0.01 + 1 \cdot 0.01 = 0.031$, or $f = 1.061 - 0.450 \pm 0.032 = 0.611 \pm 0.032$. The error bound has been raised by 0.001 because of the rounding in the calculation of x_1^2.

One is seldom asked to give mathematically guaranteed error bounds. More often it is satisfactory to give an estimate of the order of magnitude of the anticipated error. Mostly one assumes that the bound for $|\Delta f|$ estimates the maximal error, i.e., covers the worst possible cases.
In practice, the trouble with formula (8.3) is that it often gives bounds which are too coarse. More realistic estimates are often obtained for the general case, which can be derived using probability theory:
Assume that the errors $\Delta x_1, \Delta x_2, \ldots, \Delta x_n$ are distributed with mean zero and standard deviations $\epsilon_1, \epsilon_2, \ldots, \epsilon_{2n}$. $f(x_1, x_2, \ldots, x_n)$ is given by the formula

$$\epsilon \approx \left(\sum_{i=1}^{n} \left(\frac{\partial f}{\partial x_i} \right)^2 \epsilon_i^2 \right)^{1/2}$$

Analysis of error propagation is more than just a means for judging the reliability of calculated results. As remarked above, it has an equally important function as a means for the planning of a calculation or scientific experiment. It can help in the choice of algorithm, and in making certain decisions during a calculation. An example of such a decision is the choice of step length during a numerical integration. Increased accuracy often has to be bought at the price of more costly or complicated calculations. One can also shed some light on the degree to which it is advisable to obtain a new apparatus to improve the measurements of a given variable when the measurements of other variables are subject to error as well. It is useful to have a measure of how sensitive the output data are to small changes in the input data. In general, if `small` changes in the input data can result in `large` changes in the output data, we call the problem **ill-conditioned**; otherwise it is called **well- conditioned**. (The definition of large may differ from problem to problem depending on the accuracy of the data and the accuracy needed in the solution.)

8.1 Navier-Stokes Equation

More detailed texts about fluid mechanics and finite difference methods can be found in Ref. [37] and Ref. [38], respectively.
The dynamics of a fluid are determined by the famous *Navier-Stokes equation*

$$\rho \left[\partial_t \boldsymbol{v} + (\boldsymbol{v} \cdot \boldsymbol{\nabla}) \boldsymbol{v} \right] = \rho \boldsymbol{F} - \boldsymbol{\nabla} p + \eta \Delta \boldsymbol{v} + \left(\zeta + \frac{\eta}{3} \right) \boldsymbol{\nabla} (\boldsymbol{\nabla} \cdot \boldsymbol{v}), \tag{8.4}$$

where $\eta = \eta(p, T)$ and $\zeta = \zeta(p, T)$ are viscosity coefficients, $p = p(\boldsymbol{x}, t)$ is the pressure, $T = T(\boldsymbol{x}, t)$ the temperature, $\rho = \rho(\boldsymbol{x}, t)$ the density of the fluid, $\boldsymbol{v} = \boldsymbol{v}(\boldsymbol{x}, t)$ the velocity of the fluid and $\boldsymbol{F} = \boldsymbol{F}(\boldsymbol{x}, t)$ the sum of drag force (parallel to $\boldsymbol{v}$) and lift force (perpendicular to $\boldsymbol{v}$) originating from forces (like e.g. gravity) acting on the continuum. Additionally, the fluid satisfies the so-called *continuity equation*

$$\partial_t \rho + (\boldsymbol{\nabla} \rho) \cdot \boldsymbol{v} + \rho \boldsymbol{\nabla} \cdot \boldsymbol{v} = 0, \tag{8.5}$$

essentially stating that mass is conserved; a flow of particles of the fluid into one point is always reflected by an appropriate change of the density at the same point. Together with

the Navier-Stokes equation and additional thermodynamic state equations, these equations form a system of non-linear partial differential equations that is, in general, very difficult to solve!
For *incompressible* fluids, i.e. fluids that have a constant density $\rho \neq \rho(\boldsymbol{x}, t)$, Eq. (8.5) simplifies to $\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0$ such that Eq. (8.4) becomes

$$\partial_t \boldsymbol{v} + (\boldsymbol{v} \cdot \boldsymbol{\nabla})\boldsymbol{v} = \boldsymbol{F} - \frac{1}{\rho}\boldsymbol{\nabla} p + \frac{\eta}{\rho}\Delta \boldsymbol{v}, \tag{8.6}$$

the ratio η/ρ being known as *kinematic viscosity*.
In the following sections we will study the incompressible Navier-Stokes equation [Eq. 8.6] numerically under different assumptions.

8.2 Poisson Equation

We can rewrite Eq. (8.6) by taking the divergence on both sides and rearranging

$$\begin{aligned} \Delta p &= \rho \boldsymbol{\nabla} \cdot \boldsymbol{F} + \eta \boldsymbol{\nabla} \cdot (\Delta \boldsymbol{v}) - \rho \boldsymbol{\nabla} \cdot (\partial_t \boldsymbol{v}) - \rho \boldsymbol{\nabla} \cdot [(\boldsymbol{v} \cdot \boldsymbol{\nabla})\boldsymbol{v}] && (8.7)\\ &= \rho \boldsymbol{\nabla} \cdot \boldsymbol{F} + \eta \Delta \underbrace{\boldsymbol{\nabla} \cdot \boldsymbol{v}}_{=0} - \rho \partial_t \underbrace{\boldsymbol{\nabla} \cdot \boldsymbol{v}}_{=0} - \rho \boldsymbol{\nabla} \cdot [(\boldsymbol{v} \cdot \boldsymbol{\nabla})\boldsymbol{v}] && (8.8)\\ &= \rho \boldsymbol{\nabla} \cdot \boldsymbol{F} - \rho \boldsymbol{\nabla} \cdot [(\boldsymbol{v} \cdot \boldsymbol{\nabla})\boldsymbol{v}]\,. && (8.9) \end{aligned}$$

Under the assumption that we already know the velocity $\boldsymbol{v}$ of the fluid, we can solve for the pressure distribution p by solving the so-called *Poisson equation*. The Poisson equation is an *elliptic* [1] second-order linear partial differential equation of the form

$$\Delta f(\boldsymbol{x}) = g(\boldsymbol{x}), \tag{8.11}$$

where $g(\boldsymbol{x})$ is a source term. Let us assume that we want to find a solution of the Poisson equation within a certain domain $\boldsymbol{x} \in \Omega$. If we additionally impose boundary conditions, the *Helmholtz theorem* guarantees that the solution is unique. We assume boundary conditions of one of the following types [2] :

- Dirichlet: $f(\boldsymbol{x}) = h(\boldsymbol{x}), \forall \boldsymbol{x} \in \partial\Omega$,

[1] Let $f(x, t)$ be a function of two independent variables. A general second-order linear partial differential equation is of the form

$$A\partial_x^2 f + 2B\partial_{xt} f + C\partial_t^2 f + D\partial_x f + E\partial_t f + F = 0, \tag{8.10}$$

where the coefficients can in general be dependent on x and t as well. Then, we classify the equation as

- Hyperbolic if $B^2 - AC > 0$.
- Parabolic if $B^2 - AC = 0$.
- Elliptic if $B^2 - AC < 0$.

For more than two independent variables the classification depends on the eigenvalues of the coefficient matrix of the second-order derivative terms. Note that the generalization of these concepts to higher-order linear partial differential equations is only partly possible.

[2] There are also other possible boundary conditions, see e.g. mixed, Cauchy or Robin.

- Neumann: $\boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) = h(\boldsymbol{x}), \forall \boldsymbol{x} \in \partial\Omega$,

where the function $h(\boldsymbol{x})$ needs to be known in either case and $\boldsymbol{n}(\boldsymbol{x})$ is the inward unit normal of the surface.
Finally, note that the Poisson equation also arises in various other contexts besides fluid mechanics. For example, f could be the gravitational/electrostatic potential and g the mass/charge density distribution.

8.2.1 Finite Difference Approximation

Consider a connected domain $\Omega \subset \mathbb{R}^d$ and let $f : \Omega \to \mathbb{R}$ be a sufficiently well-behaved function. The partial derivative of f is defined as

$$\partial_{x_i} f(x_1, \ldots, x_d) = \lim_{x_i' \to x_i} \frac{f(x_1, \ldots, x_i', \ldots, x_d) - f(x_1, \ldots, x_i, \ldots, x_d)}{x_i' - x_i}. \tag{8.12}$$

Our goal is to find an (approximate) solution of the Poisson equation [Eq. (8.11)] on Ω. In finite difference methods, we discretize Ω—here, we assume for simplicity an equidistant grid spacing—which allows us to approximate the derivatives in the partial differential equation by finite differences. This will allow us to map the partial differential equation onto a system of linear equations.
We start with a $d = 1$-dimensional system. In this case, Ω is an interval with end points x_l (left) and x_r (right). We decompose the interval in L discrete bins and superimpose a grid containing the $L + 1$ points

$$x_0 \equiv x_l, x_1 \equiv x_l + \Delta x, \ldots, x_L \equiv x_r = x_l + L\Delta x. \tag{8.13}$$

Now, we can do a Taylor expansion of the following expression

$$f(x \pm \Delta x) = f(x) \pm \partial_x f(x)\Delta x + \frac{1}{2}\partial_x^2 f(x)\Delta x^2 + \mathcal{O}(\Delta x^3) \tag{8.14}$$

and notice that upon rearranging we can find that

$$\partial_x f(x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} + \mathcal{O}(\Delta x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{8.15}$$

$$\partial_x f(x) = \frac{f(x) - f(x - \Delta x)}{\Delta x} + \mathcal{O}(\Delta x) \approx \frac{f(x) - f(x - \Delta x)}{\Delta x}, \tag{8.16}$$

which are called *forward* and *backward* difference approximation, respectively. In the limit $\Delta x \to 0$, i.e. the grid transitions back to continuous space, we recover the mathematically exact derivative in Eq. (8.12). Note that we can improve the accuracy of the derivative approximation by considering the difference

$$f(x + \Delta x) - f(x - \Delta x) = 2\partial_x f(x)\Delta x + \mathcal{O}(\Delta x^3) \tag{8.17}$$

and rearranging it such that

$$\partial_x f(x) = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2) \approx \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} \tag{8.18}$$

which is called *centered* difference approximation and has an improved accuracy of $\mathcal{O}(\Delta x^2)$ instead of $\mathcal{O}(\Delta x)$ as in the forward or backward difference approximation.
Similarly, we can consider the sum

$$f(x+\Delta x)+f(x-\Delta x)=2f(x)+\partial_x^2 f(x)\Delta x^2+\mathcal{O}(\Delta x^4) \tag{8.19}$$

and get a centered difference approximation for the second derivative

$$\partial_x^2 f(x)=\frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2}+\mathcal{O}(\Delta x^2)\approx\frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2}. \tag{8.20}$$

Note that the above described procedure can be systematically exploited to construct both higher order derivatives as well as better accuracies by including grid points beyond nearest neighbors.
Introducing the short-hand notation $f(x_j)=f_j,\ j\in\{0,\dots,L\}$ and using Eq. (8.20), we can rewrite the 1D Poisson equation

$$\partial_x^2 f(x)=g(x) \tag{8.21}$$

as

$$f_2-2f_1+f_0=\Delta x^2 g_1 \tag{8.22}$$
$$f_3-2f_2+f_1=\Delta x^2 g_2 \tag{8.23}$$
$$\dots \tag{8.24}$$
$$f_L-2f_{L-1}+f_{L-2}=\Delta x^2 g_{L-1} \tag{8.25}$$

which is a linear system of equations. Suppose that we know f on the boundaries of $\partial\Omega$, i.e. f_0 and f_L, which corresponds to Dirichlet boundary conditions, we can bring the linear system in the following $(L-1)\times(L-1)$ matrix form

$$\begin{pmatrix} -2 & 1 & 0 & \dots & & & 0\\ 1 & -2 & 1 & 0 & \dots & & 0\\ 0 & 1 & -2 & 1 & 0 & \dots & 0\\ \vdots & & & \ddots & & & \vdots\\ 0 & \dots & 0 & 1 & -2 & 1 & 0\\ 0 & & \dots & 0 & 1 & -2 & 1\\ 0 & & & \dots & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} f_1\\ f_2\\ f_3\\ \vdots\\ f_{L-3}\\ f_{L-2}\\ f_{L-1}\end{pmatrix}=\Delta x^2\begin{pmatrix} g_1-f_0\\ g_2\\ g_3\\ \vdots\\ g_{L-3}\\ g_{L-2}\\ g_{L-1}-f_L\end{pmatrix}. \tag{8.26}$$

8.2.2 Numerical Solutions of Linear Systems of Equations

There are plenty of numerical methods to solve a linear system of equations

$$A\boldsymbol{x}=\boldsymbol{b}, \tag{8.27}$$

A being a $N\times N$ matrix. Ideally, we choose a method such that it makes use of the properties of A, e.g. symmetric, positive-definite, sparse, etc. Generally speaking, the complexity of solving a linear system is $\mathcal{O}(N^3)$ for dense and $\mathcal{O}(N^2)$ for sparse matrices containing only $\mathcal{O}(N)$ instead of $\mathcal{O}(N^2)$ elements.
Overall, we distinguish between *direct* and *iterative* methods. A quick overview is given below:

Direct Methods

- Matrix inversion: Compute A^{-1} to obtain the solution via $\boldsymbol{x} = A^{-1}\boldsymbol{b}$. Computationally expensive ($\mathcal{O}(N^3)$) and can have significant rounding errors. But useful if e.g. A is orthogonal, i.e. $A^{-1} = A^T$.
- Gaussian elimination: Transform the linear system of equations into an equivalent one, where the matrix is an upper triangular matrix. Typically, preferred over matrix inversion as it is in practice much faster (despite still being $\mathcal{O}(N^3)$ in general) and less prone to rounding errors. Problem: Needs to be done for every $\boldsymbol{b}$.
- LU decomposition: Based on Gaussian elimination. Splits the matrix $A = LU$, where L is a lower triangular and U is an upper triangular matrix. $\boldsymbol{x}$ is obtained by solving first $L\tilde{\boldsymbol{x}} = \boldsymbol{b}$ via forward substitution and $U\boldsymbol{x} = \tilde{\boldsymbol{x}}$ via backward substitution. Advantage over direct Gaussian elimination: Decomposition of A is independent of $\boldsymbol{b}$ and only needs to be performed once. Problem: Does not preserve sparsity of matrix.
- LL^T decomposition (=Cholesky decomposition): Similar to LU decomposition but requires that A is a symmetric positive-definite matrix.
- ...

Iterative Methods

- Relaxation methods (=splitting methods)
  - Jacobi: Splits the matrix $A = L + D + U$, where L is lower-triangular, D diagonal and U upper-triangular. $n+1$-th iteration $\boldsymbol{x}^{n+1}$ as an approximation of $\boldsymbol{x}$ is given by
$$\boldsymbol{x}^{n+1} = D^{-1}(\boldsymbol{b} - (L + U)\boldsymbol{x}^n). \tag{8.28}$$
  - Gauss-Seidel: Same matrix splitting as for the Jacobi relaxation method but the $n+1$-th iteration is instead given by
$$\boldsymbol{x}^{n+1} = (D + U)^{-1}(\boldsymbol{b} - L\boldsymbol{x}^n) \tag{8.29}$$
  - ...
- Gradient methods
  - Steepest descent: Define the residual $\boldsymbol{r}(\boldsymbol{x}) = \boldsymbol{b} - A\boldsymbol{x}$ and minimize the following paraboloid function $f(\boldsymbol{x}) = \boldsymbol{r}^T A^{-1}\boldsymbol{r} \geq 0$ by steepest descent.
  - Conjugate gradient: Similar to the steepest descent gradient method but f is transformed into a regular paraboloid before steepest descent is applied. Advantage over direct steepest descent: Does not suffer under "zigzag" motion in configuration space.
  - ...

- Other solvers
  - Fixed point iteration: Define the function $f(\boldsymbol{x}) = (A + I_N)\boldsymbol{x} - \boldsymbol{b}$ and obtain the $n+1$-th approximation of $\boldsymbol{x}$ via

    $$\boldsymbol{x}^{n+1} = f(\boldsymbol{x}^n). \tag{8.30}$$

    Problem: Convergence is only achieved for contractions, i.e. $|f(\boldsymbol{x}^n) - f(\boldsymbol{x}^{n+1})| < |\boldsymbol{x}^n - \boldsymbol{x}^{n+1}|$.
  - Newton-Raphson iteration: Define the function $f(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b}$ and apply the Newton-Raphson method. Note that this method is equivalent to inverting the matrix because matrix inversion is needed to compute the Jacobian.
  - Krylov methods
  - ...

Note that in many cases convergence can be drastically improved with *preconditioning*. In this case, we choose a matrix P which is easy to invert and which satisfies $P^{-1}A \approx I_N$. This allows us to transform the linear system into

$$P^{-1}A\boldsymbol{x} = P^{-1}\boldsymbol{b} \tag{8.31}$$

which we then solve instead of the original problem.
In Julia, the `LinearAlgebra` package provides solid functionality for solving dense linear systems of equations, see https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/. The syntax is simple. Once the matrix `A` and the vector `b` are implemented, the linear system is solved via `x = A\b`. Especially useful are the large number of matrix types and possible matrix factorizations from which you can choose a suitable one. Behind the scenes, Julia then automatically calls the appropriate LAPACK functions. Note that wrappers for both BLAS and LAPACK exist.
For sparse matrices you can use the `SparseArrays` package, see https://docs.julialang.org/en/v1/stdlib/SparseArrays/ and https://docs.julialang.org/en/v1/stdlib/SuiteSparse/ for sparse matrix routines.

8.2.3 Higher Dimensions

The same procedure can be applied to higher dimensions as well. To illustrate this, we consider the case of $d = 2$ dimensions. Here, we assume Ω to be of rectangular shape bounded by the corner points (x_l, y_t) (top left), (x_l, y_b) (bottom left), (x_r, y_t) (top right) and (x_r, y_b) (bottom right). The rectangular domain is decomposed into L_x discrete bins along the x-direction and L_y discrete bins along the y-direction such that the grid contains in total $(L_x + 1)(L_y + 1)$ points

$$(x_0, y_0) \equiv (x_l, y_t), \tag{8.32}$$
$$(x_0, y_1) \equiv (x_l, y_t - \Delta y), \tag{8.33}$$
$$\dots \tag{8.34}$$

$$(x_0, y_{L_y}) \equiv (x_l, y_b) = (x_l, y_t - L_y \Delta y), \tag{8.35}$$
$$\dots \tag{8.36}$$
$$(x_{L_x}, y_{L_y}) \equiv (x_r, y_b) = (x_l + L_x \Delta x, y_t - L_y \Delta y). \tag{8.37}$$

With the short-hand notation $f(x_j, y_k) = f_{j,k},\ j \in \{0, \dots, L_x\},\ k \in \{0, \dots, L_y\}$ and using again Eq. (8.20) (for both directions), we can rewrite the 2D Poisson equation

$$\partial_x^2 f(x, y) + \partial_y^2 f(x, y) = g(x, y) \tag{8.38}$$

as

$$\Delta y^2 (f_{j+1,k} + f_{j-1,k}) + \Delta x^2 (f_{j,k+1} + f_{j,k-1}) - 2(\Delta x^2 + \Delta y^2) f_{j,k} = \Delta x^2 \Delta y^2 g_{j,k} \tag{8.39}$$

for $j \in \{1, \dots, L_x - 1\},\ k \in \{1, \dots, L_y - 1\}$ which is again a linear system of equations. Introducing a single index $(j, k) \to j + (k-1)(L_x - 1)$ allows us to bring the system to a $(L_x - 1)(L_y - 1) \times (L_x - 1)(L_y - 1)$ matrix form which we can again solve for $\boldsymbol{f} = \begin{pmatrix} f_1 & \dots & f_{(L_x-1)(L_y-1)} \end{pmatrix}$. Note, however, that both the matrix A as well as the vector $\boldsymbol{b}$ have a different form than we have seen for the 1D Poisson equation.

8.3 Linear Advection Equation

Let us, for a moment, ignore the incompressible Navier-Stokes equation [Eq. (8.6)] but instead only focus on the mass continuity equation Eq. (8.5) which simplifies for an incompressible fluid to

$$\partial_t \rho + \boldsymbol{v} \cdot (\boldsymbol{\nabla} \rho) = 0. \tag{8.40}$$

This equation is also known as *Linear Advection Equation* and describes how the density concentration changes with the flow of the fluid. The linear advection equation is another example for a hyperbolic linear partial differential equation. Although, it has a unique analytic solution when the fluid density is known at time $t = 0$

$$\rho(\boldsymbol{x}, t) = \rho(\boldsymbol{x} - \boldsymbol{v}t, 0) \tag{8.41}$$

it is well-suited for us to analyze possible numerical schemes.

8.3.1 Discretization Schemes

As discussed before, depending on our choice for the approximation of the partial derivatives Eq. (8.12) appearing in Eq. (8.40), we obtain different solution schemes with different errors. Here, we will discuss in more details the implications of combining different finite difference approximations.
In 1D the linear advection equation Eq. (8.40) simplifies to

$$\partial_t \rho + v \partial_x \rho = 0, \tag{8.42}$$

where $\rho = \rho(x, t)$ is a function of two (continuous) variables. As before, we can discretize space equidistantly across the x and t directions separately with lattice spacing Δx and Δt,

respectively. And we introduce the shorthand notation $\rho_j^n = \rho(x_j, t_n)$. Using the forward, backward and centered difference approximations Eq. (8.15), Eq. (8.16) and Eq. (8.20), respectively, we can combine them for space and time to obtain various numerical schemes. Here, we will focus on the following ones:

- Forward in time, backward in space (FTBS):

$$\frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} + v\frac{\rho_j^n - \rho_{j-1}^n}{\Delta x} = 0 \quad \Longrightarrow \quad \rho_j^{n+1} = \rho_j^n - c\left(\rho_j^n - \rho_{j-1}^n\right), \tag{8.43}$$

  where $c = \frac{v\Delta t}{\Delta x}$ is the so-called *Courant number* (dimensionless) for the FTBS scheme. The scheme is 1st-order accurate in space and time.

- Forward in time, centered in space (FTCS):

$$\frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} + v\frac{\rho_{j+1}^n - \rho_{j-1}^n}{2\Delta x} = 0 \quad \Longrightarrow \quad \rho_j^{n+1} = \rho_j^n - c\left(\rho_{j+1}^n - \rho_{j-1}^n\right), \tag{8.44}$$

  where $c = \frac{v\Delta t}{2\Delta x}$ is the Courant number for the FTCS scheme. The scheme is 2nd-order accurate in space and 1st order accurate in time.

- Backward in time, centered in space (BTCS):

$$\frac{\rho_j^n - \rho_j^{n-1}}{\Delta t} + v\frac{\rho_{j+1}^n - \rho_{j-1}^n}{2\Delta x} = 0 \quad \Longrightarrow \quad \rho_j^{n+1} = \rho_j^n - c\left(\rho_{j+1}^{n+1} - \rho_{j-1}^{n+1}\right), \tag{8.45}$$

  where $c = \frac{v\Delta t}{2\Delta x}$ is the Courant number for the BTCS scheme. The scheme is 2nd-order accurate in space and 1st order accurate in time.

- Centered in time, centered in space (CTCS):

$$\frac{\rho_j^{n+1} - \rho_j^{n-1}}{2\Delta t} + v\frac{\rho_{j+1}^n - \rho_{j-1}^n}{2\Delta x} = 0 \quad \Longrightarrow \quad \rho_j^{n+1} = \rho_j^{n-1} - c\left(\rho_{j+1}^n - \rho_{j-1}^n\right), \tag{8.46}$$

  where $c = \frac{v\Delta t}{\Delta x}$ is the Courant number for the CTCS scheme. The scheme is 2nd-order accurate in space and time.

Note that there is a fundamental difference between the FTBS, FTCS, CTCS schemes on one hand and the BTCS scheme on the other hand. While the former schemes are so-called *explicit* schemes, the latter one is an *implicit* scheme. In explicit schemes, the next time step depends only on the previous time steps. However, for the implicit BTCS scheme the update rule is of the form

$$M\boldsymbol{\rho}^n = \boldsymbol{\rho}^{n-1}, \tag{8.47}$$

where

$$M = \begin{pmatrix} 1 & -c & 0 & \dots & & & 0 \\ c & 1 & -c & 0 & \dots & & 0 \\ 0 & c & 1 & -c & 0 & \dots & 0 \\ \vdots & & & \ddots & & & \vdots \\ 0 & \dots & 0 & c & 1 & -c & 0 \\ 0 & & \dots & 0 & c & 1 & -c \\ 0 & & & \dots & 0 & c & 1 \end{pmatrix} \quad M = \begin{pmatrix} 1 & -c & 0 & \dots & & 0 & c \\ c & 1 & -c & 0 & \dots & & 0 \\ 0 & c & 1 & -c & 0 & \dots & 0 \\ \vdots & & & \ddots & & & \vdots \\ 0 & \dots & 0 & c & 1 & -c & 0 \\ 0 & & \dots & 0 & c & 1 & -c \\ -c & 0 & & \dots & 0 & c & 1 \end{pmatrix}, \tag{8.48}$$

open boundary conditions periodic boundary conditions

i.e. the next time step n does not only depend on the previous time step $n-1$ but also on the next time step n and we have to solve a set of additional equations. Note that for linear partial differential equations, this set of equations is always linear.
In the case of the CTCS scheme, the value of ρ at time step $n+1$ depends on two previous time steps (n and $n-1$). This requires either additional initial values or an approximation of the second time step by means of another numerical scheme such as a FTCS scheme for the first step.
Importantly, all of the above mentioned schemes are at least 1st-order accurate in both space and time. This implies that as $\Delta x \to 0$ and $\Delta t \to 0$ all of the error terms appearing from the finite difference approximations vanish. Since this is the case, we call such schemes *consistent* with the underlying linear partial differential equation. However, this requirement is not enough in order for our discretized solution of the scheme to *converge* to the exact solution of the linear partial differential equation. Another requirement is the *stability* of the scheme. For a stable scheme the errors do not tend to infinity for any number of time steps. Generally, we distinguish between *unconditionally stable* schemes which are stable for any time step size and *conditionally stable* schemes which are only stable for sufficiently small time steps. Otherwise we call the scheme *unconditionally unstable*. The relation between consistency, stability and convergence is summarized in the *Lax Equivalence Theorem* which states that a consistent finite difference scheme is convergent if and only if it is stable.

8.3.2 Stability

As highlighted in the last subsection, in order to show that a finite difference scheme is convergent, we only need to show that it is at least 1st-order accurate (implying consistency) and that it is stable. In the following, we are going to see how to determine the stability of a finite difference scheme. There are two important reasons for why a thorough numerical analysis is helpful:

- Not every scheme can be tested for every situation by trial and error.
- The numerical analysis provides helpful insights besides the Boolean feedback stable/unstable that can help to improve the scheme.

CFL Criterion

We call the region in parameter space, spanned by space and time, which influences the solution at a certain point (x_0, t_0), the *domain of dependence*. For example, for the linear advection equation, we know the analytical solution to be

$$\rho(x,t) = \rho(x - vt, 0). \tag{8.49}$$

This means that for the linear advection equation the (analytical) domain of dependence is given by the linear function

$$x - vt = x_0 - vt_0 \tag{8.50}$$

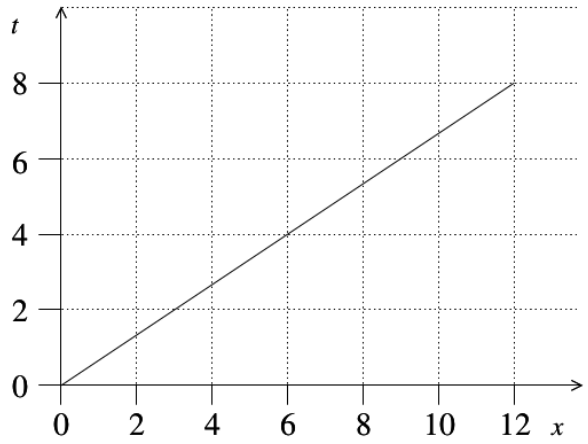


Figure 8.2: Domain of dependence (analytical) of the linear advection equation with $v = 1.5$ in units $[x/t]$.

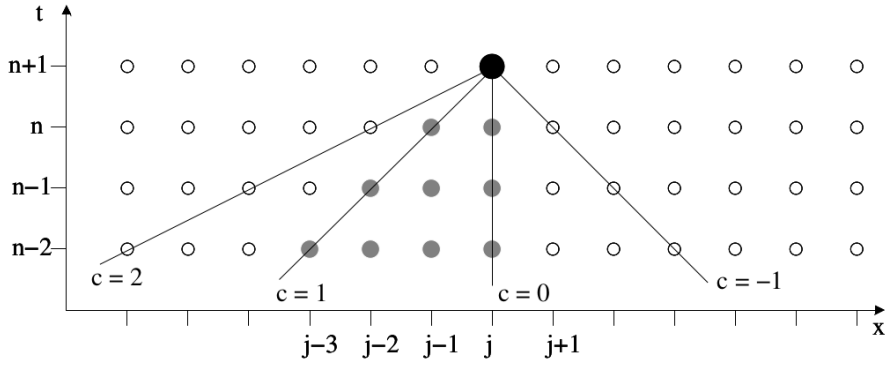


Figure 8.3: Domain of dependence (numerical) of the linear advection equation for the FTBS scheme enclosed by the lines $c = 1$ and $c = 0$.

which has slope $1/v$ in xt-space, see Fig. 8.2.
Similarly, we have a (numerical) domain of dependence for our finite difference scheme. For example, for the FTBS and the CTCS the domain of dependence is shown in Fig. 8.3 and Fig. 8.4.
The *CFL criterion*—named after Courant, Friedrichs and Lewy—states that the stability of a finite difference scheme implies that the analytical domain of dependence is included in the numerical domain of dependence. Note, however, that the converse is in general not true! The condition that the analytical domain of dependence is included in the numerical domain of dependence is only necessary, not sufficient. But it helps us, in our case, to narrow down the region of possible Courant numbers c for which the numerical scheme is stable. Again from Fig. 8.3 and Fig. 8.4, in the case of the FTBS scheme, we necessarily require $0 \leq c \leq 1$ and for the CTCS scheme $-1 \leq c \leq 1$.

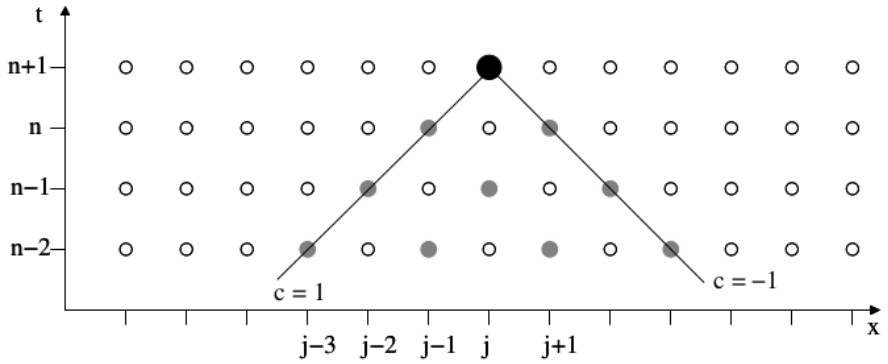


Figure 8.4: Domain of dependence (numerical) of the linear advection equation for the CTCS scheme enclosed by the lines $c = 1$ and $c = -1$.

Von Neumann Stability Analysis

In contrast to the simple CFL criterion, the von Neumann stability analysis can be used to derive sufficient conditions for the stability of a finite difference scheme [38, 39, 40, 41]. Let us consider the Fourier expansion

$$\rho_j^n = \sum_{k=-\infty}^{+\infty} \hat{\rho}_k^n e^{-ikj\Delta x}. \tag{8.51}$$

and assume that the Fourier components satisfy the relation

$$\hat{\rho}_k^{n+1} = A_k(\Delta x, \Delta t)\hat{\rho}_k^n, \tag{8.52}$$

where $A_k \in \mathbb{C}$ is an amplification factor. Then, the scheme is stable if $\forall k : |A_k|^2 \leq 1$ (damping for "$<$" and neutral for "$=$") and that it is unstable if $\exists k : |A_k|^2 > 1$ (amplifying).

FTBS scheme We plug the Fourier expansion Eq. (8.51) into the FTBS scheme Eq. (8.43) and obtain

$$\sum_k \hat{\rho}_k^{n+1} e^{-ikj\Delta x} = \sum_k \hat{\rho}_k^n e^{-ikj\Delta x} - c\left(\sum_k \hat{\rho}_k^n e^{-ikj\Delta x} - \sum_k \hat{\rho}_k^n e^{-ik(j-1)\Delta x}\right) \tag{8.53}$$

from which we identify

$$\hat{\rho}_k^{n+1} = \underbrace{\left(1 - c\left(1 - e^{ik\Delta x}\right)\right)}_{A_k(\Delta x, \Delta t)} \hat{\rho}_k^n. \tag{8.54}$$

Now we compute

$$|A_k|^2 = 1 - 2c(1-c)(1-\cos(k\Delta x)). \tag{8.55}$$

This means in order for $|A_k|^2 \leq 1$ that $2c(1-c)(1-\cos(k\Delta x)) \geq 0$ and since $(1-\cos(k\Delta x)) \geq 0$, we find that $c(1-c) \geq 0$ and consequently $0 \leq c \leq 1$. Hence, by definition of c, for $v < 0$ or time steps $\Delta t \geq \frac{\Delta x}{v}$ the FTBS scheme is unstable.

FTCS scheme We plug the Fourier expansion Eq. (8.51) into the FTBS scheme Eq. (8.44) and obtain

$$\sum_k \hat{\rho}_k^{n+1} e^{-ikj\Delta x} = \sum_k \hat{\rho}_k^n e^{-ikj\Delta x} - c\left(\sum_k \hat{\rho}_k^n e^{-ik(j+1)\Delta x} - \sum_k \hat{\rho}_k^n e^{-ik(j-1)\Delta x}\right) \tag{8.56}$$

from which we identify

$$\hat{\rho}_k^{n+1} = \underbrace{1 + 2ic\sin(k\Delta x)}_{A_k(\Delta x,\Delta t)}\hat{\rho}_k^n. \tag{8.57}$$

Now we compute

$$|A_k|^2 = 1 + 4c^2\sin(k\Delta x)^2. \tag{8.58}$$

This means in order for $|A_k|^2 \leq 1$ that $4c^2\sin(k\Delta x)^2 \leq 0$ and since $\sin(k\Delta x)^2 \geq 0$, we find that $c^2 \leq 0$ and consequently $c = 0$. Hence, the scheme is unconditionally unstable.

BTCS scheme We plug the Fourier expansion Eq. (8.51) into the FTBS scheme Eq. (8.45) and obtain

$$\sum_k \hat{\rho}_k^{n+1} e^{-ikj\Delta x} = \sum_k \hat{\rho}_k^n e^{-ikj\Delta x} - c\left(\sum_k \hat{\rho}_k^{n+1} e^{-ik(j+1)\Delta x} - \sum_k \hat{\rho}_k^{n+1} e^{-ik(j-1)\Delta x}\right) \tag{8.59}$$

from which we identify

$$\hat{\rho}_k^{n+1} = \underbrace{\frac{1}{1 - 2ic\sin(k\Delta x)}}_{A_k(\Delta x,\Delta t)}\hat{\rho}_k^n. \tag{8.60}$$

Now we compute

$$|A_k|^2 = \frac{1}{1 + 4c^2\sin(k\Delta x)^2}. \tag{8.61}$$

This means in order for $|A_k|^2 \leq 1$ that $4c^2\sin(k\Delta x)^2 \geq 0$ and since $\sin(k\Delta x)^2 \geq 0$, we find that $c^2 \geq 0$ and consequently $c \geq 0$. Hence, the scheme is unconditionally stable.

CTCS scheme We plug the Fourier expansion Eq. (8.51) into the FTBS scheme Eq. (8.46) and obtain

$$\sum_k \hat{\rho}_k^{n+1} e^{-ikj\Delta x} = \sum_k \hat{\rho}_k^{n-1} e^{-ikj\Delta x} - c\left(\sum_k \hat{\rho}_k^n e^{-ik(j+1)\Delta x} - \sum_k \hat{\rho}_k^n e^{-ik(j-1)\Delta x}\right) \tag{8.62}$$

and consequently

$$\hat{\rho}_k^{n+1} = \hat{\rho}_k^{n-1} + 2ic\sin(k\Delta x)\hat{\rho}_k^n. \tag{8.63}$$

Since $\hat{\rho}_k^{n+1} = A_k\hat{\rho}_k^n = A_k^2\hat{\rho}_k^{n-1}$, we can determine A_k from the equation $A_k^2 - 2ic\sin(k\Delta x)A_k - 1 = 0$ and find

$$A_k = ic\sin(k\Delta x) \pm \sqrt{1 - c^2\sin(k\Delta x)^2}. \tag{8.64}$$

For $c^2 > 1$,

$$|A_k|^2 = c^2\sin(k\Delta x)^2 + c^2\sin(k\Delta x)^2 - 1 \tag{8.65}$$

which implies $c^2 \sin(k\Delta x)^2 \leq 1$ in order to satisfy $|A_k|^2 \leq 1$ for all k. By choosing $k\Delta x = \frac{\pi}{2}$, we explicitly found a k which can never satisfy the previous condition. This leads to a contradiction with the original assumption and hence in this regime the scheme is unstable. For $c^2 \leq 1$,

$$|A_k|^2 = c^2 \sin(k\Delta x)^2 + 1 - c^2 \sin(k\Delta x)^2 = 1 \tag{8.66}$$

which means that in this case the solution is stable and not damping.
Interestingly, in both cases, regardless of the choice of c, A_k can take two possible values and therefore there will be two possible solutions for ρ in each time step. One solution is a realistic solution called *physical mode* and the other one is an unrealistic solution called *spurious computational mode*. In simulations using the CTCS scheme the result will be a mixture of both modes. Therefore, the spurious computational mode contaminates the desired physical solution.

Conserved Quantities

Consider a quantity $Q(x,t)$ which carries a certain property of our system, e.g. mass, energy, etc. Then, we say this quantity is conserved whenever

$$d_t Q(x,t) = 0, \tag{8.67}$$

i.e. it does not change over time. Since the dynamics of our system is determined by the corresponding equations of motion which we desire to solve, we might ask ourselves whether such a quantity Q is also conserved in the numerical scheme that we choose.

Mass Let

$$M(t) = \int_0^1 \rho(x,t)dx \tag{8.68}$$

be the total mass of our fluid and assume periodic boundary conditions over the spatial domain which we choose from 0 to 1, i.e. $\rho(0,t) = \rho(1,t),\ \forall t$. Then,

$$d_t M(t) = d_t \int_0^1 \rho(x,t)dx = \int_0^1 d_t\rho(x,t)dx = \int_0^1 -v d_x\rho(x,t)dx = -v\int_0^1 d_x\rho(x,t)dx \tag{8.69}$$

$$= -v(\rho(1,t) - \rho(0,t)) = 0, \tag{8.70}$$

where in the third step we use the linear advection equation Eq. (8.40) and in the final step the fact that we have assumed periodic boundary conditions. This means that the total mass is indeed conserved.
Let us check for the example of the FTBS scheme whether the mass is conserved as well. The mass at time step $n+1$ is given by

$$M^{n+1} = \Delta x \sum_{j=1}^{N} \rho_j^{n+1} = \Delta x \sum_{j=1}^{N} \left[\rho_j^n - c\left(\rho_j^n - \rho_{j-1}^n\right)\right] \tag{8.71}$$

$$= \underbrace{\Delta x \sum_{j=1}^{N} \rho_j^n}_{M^n} - c\Delta x \left[\sum_{j=1}^{N} \rho_j^n - \sum_{j=1}^{N} \rho_{j-1}^n\right] = M^n - c\Delta x \left[\sum_{j=1}^{N} \rho_j^n - \sum_{j=0}^{N-1} \rho_j^n\right] \tag{8.72}$$

$$= M^n - c\Delta x \left(\rho_N^n - \rho_0^n\right) = M^n, \tag{8.73}$$

where we used in the third step the update rule of the FTBS scheme and in the last step again the periodic boundary conditions. Hence, the FTBS scheme conserves the total mass as well.

Higher Moments of the Density Distribution The total mass M is the expectation value (first moment) of the density distribution ρ. Next, we consider the variance (second moment) of the density distribution

$$V(t) = \left\langle M(t)^2 \right\rangle - \left\langle M(t) \right\rangle^2 = \int_0^1 \rho(x,t)^2 dx - M(t)^2 = \int_0^1 \rho(x,t)^2 dx - M^2, \tag{8.74}$$

where in the last step we used the mass conservation proven before. Then,

$$d_t V(t) = d_t \int_0^1 \rho(x,t)^2 dx = \int_0^1 d_t \rho(x,t)^2 dx = \int_0^1 2\rho(x,t) d_t \rho(x,t) dx \tag{8.75}$$

$$= -2v \int_0^1 \rho(x,t) d_x \rho(x,t) dx = -v \int_0^1 d_x \rho(x,t)^2 dx = -v \left(\rho(1,t)^2 - \rho(0,t)^2\right) \tag{8.76}$$

$$= 0, \tag{8.77}$$

where in the fourth step we used again the linear advection equation Eq. (8.40) and in the final step the periodic boundary conditions.
For the FTBS scheme, we compute

$$V^{n+1} = \Delta x \sum_{j=1}^{N} \left(\rho_j^{n+1}\right)^2 - \left(M^{n+1}\right)^2 = \cdots = V^n - \underbrace{\text{additional terms}}_{>0}, \tag{8.78}$$

i.e. the variance will decrease with every time step in the FTBS scheme.

Dispersion Errors

Generally speaking, a *wave* is a (potentially periodic) perturbation of the equilbrium state of a system that is propagating through space and time. The functional form $f(x,t)$—here in one dimension—of the wave is determined by the corresponding differential equation. This could be—as the name suggests—the wave equation but also as in our case the linear advection equation [3].
Let us assume that this differential equation is a linear differential equation such that the superposition of solutions is a solution itself. Consider for a moment a simple plane wave of the form

$$f(x,t) = A \sin(kx - \omega t + \varphi), \tag{8.79}$$

where k is the wave number, ω the frequency and φ some additional phase offset. The velocity with which the wave propagates through space is the so-called *phase velocity*

$$v_p = \frac{\omega}{k}. \tag{8.80}$$

[3] Both have in common that they are linear partial differential equations and have solutions of the form $f(x - vt, 0)$. Note, however, that the wave equation has additional solutions of the form $f(x + vt, 0)$.

Since the underlying differential equation is assumed to be linear, arbitrary superpositions will still be valid solutions. If the superposed waves have different phase velocities, the resulting wave will in general have an overall modulation that can be described by an enveloping function that is propagating with the *group velocity*

$$v_G = \partial_k \omega(k). \tag{8.81}$$

For the propagating wave we have to consider whether the medium through which the wave propagates is *dispersive* or not. A dispersive medium is characterized by a refractive index n that is a function of the frequency of the wave, i.e. $n = n(\omega)$. In a uniform medium, a single wave propagates with the phase velocity

$$v_p(\omega) = \frac{c}{n(\omega)}. \tag{8.82}$$

Since ω is proportional to k, the corresponding *dispersion relation* is given by

$$\omega(k) = v_P(k)k, \tag{8.83}$$

characterized by a phase velocity that is explicitly depending on the wave vector. In such a situation, we have $v_G \neq v_P$. The observable effect will be that a wave packet—a wave that is localized in space and consisting of infinitely many sinusoidal waves of different wave numbers—will change its enveloping profile while it propagates; it disperses. This is not the case when $v_G = v_P$ (in non-dispersive media).
When analyzing the advection equation [Eq. (8.40)], we see that every wave is propagating with the constant velocity v. This means that the advection equation is describing a non-dispersive propagation of waves. However, this is not necessarily true for the numerical discretization schemes. We restrict our discussion here to the CTCS scheme.
For a wave satisfying the advection equation, a single Fourier mode—with wave vector k—that is multiplied by a factor $e^{-ikv\Delta t}$ propagates by the distance $v\Delta t$ or in other words

$$A = e^{-ikv\Delta t} \tag{8.84}$$

is the amplification factor of the linear advection equation. For the CTCS scheme, we saw that the amplification factor is given by

$$A_{\text{num}} = ic\sin(k\Delta x) \pm \sqrt{1 - c^2 \sin(k\Delta x)^2} \tag{8.85}$$

and in the stable region, where $c^2 \leq 1$, we saw that $|A_{\text{num}}|^2 = 1$, i.e. we can write

$$A_{\text{num}} = e^{-i\alpha_\pm}, \tag{8.86}$$

where $\alpha_\pm \in \mathbb{R}$. By identifying $\alpha_\pm = kv_{\text{num}}\Delta t$, the velocity—relative to the exact velocity v—with which a wave in the CTCS scheme is propagating is given by

$$\frac{v_{\text{num}}}{v} = \frac{\alpha_\pm}{kv\Delta t} = \frac{\alpha_\pm}{ck\Delta x}. \tag{8.87}$$

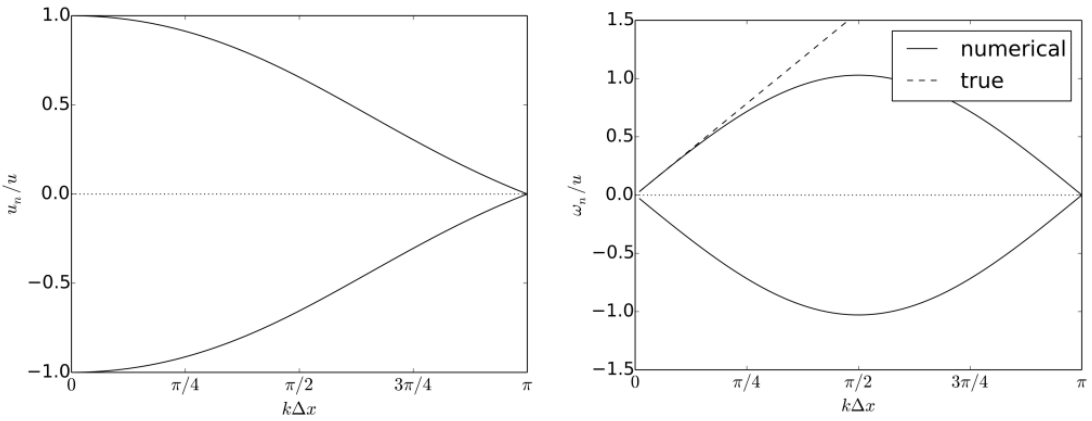


Figure 8.5: Dispersion for a CTCS scheme with $c = 0.4$. Physical solution with $v_{\text{num}} = u_n > 0$ and spurious computational mode with $v_{\text{num}} = u_n < 0$, $v = u$.

$\alpha_\pm$ is obtained from the relation

$$ic\sin(k\Delta x) \pm \sqrt{1 - c^2\sin(k\Delta x)^2} \stackrel{!}{=} e^{-i\alpha_\pm} = \cos(\alpha_\pm) - i\sin(\alpha_\pm) \tag{8.88}$$

$$\iff \sin(\alpha_\pm) = -c\sin(k\Delta x). \tag{8.89}$$

Figure 8.5 shows that the waves in the CTCS scheme propagate too slowly (compared to the exact linear advection equation) and show dispersive behavior, i.e. $v_{\text{num}} = v_{\text{num}}(k)$. For $k\Delta x \approx 0$, the waves propagate almost at the correct speed.

8.4 Shallow Water Equations

Starting point are the incompressible Navier-Stokes equation Eq. (8.6) together with the condition of incompressibility $\boldsymbol{\nabla}\cdot\boldsymbol{v} = 0$. In many situations, like e.g. coastal regions or rivers, we can assume that the depth of the fluid (measured along the z axis) is small compared to its horizontal extent (along the xy plane). In such a situation we speak about *shallow water*. If we assume for a moment that both the surface of the water as well as the ground are flat, we can choose our coordinates such that the surface of the water is at $z = 0$ and the ground at $z = -H_0$. Now, also including potential waves, we can decompose the total depth of the fluid

$$H(x, y, t) = H_0 + H'(x, y, t), \tag{8.90}$$

where H' is the elevation of the fluid relative to the average depth H_0. Furthermore, we assume that $H' << H_0$. Neglecting any viscosity effects ($\eta = 0$) but including the gravitational force $\boldsymbol{F} = (0, 0, g)$, the single components of the incompressible Navier-Stokes equation can be written as

$$\partial_t v_x + v_x\partial_x v_x + v_y\partial_y v_x + v_z\partial_z v_x = -\frac{1}{\rho}\partial_x p \tag{8.91}$$

$$\partial_t v_y + v_x\partial_x v_y + v_y\partial_y v_y + v_z\partial_z v_y = -\frac{1}{\rho}\partial_y p \tag{8.92}$$

$$\partial_t v_z + v_x \partial_x v_z + v_y \partial_y v_z + v_z \partial_z v_z = -\frac{1}{\rho}\partial_z p + g \tag{8.93}$$

$$\partial_x v_x + \partial_y v_y + \partial_z v_z = 0. \tag{8.94}$$

Incompressibility Equation Integrating Eq. (8.94) over depth yields

$$0 = \int_{-H_0}^{H'} \boldsymbol{\nabla} \cdot \boldsymbol{v} dz \tag{8.95}$$

$$= \int_{-H_0}^{H'} \partial_x v_x dz + \int_{-H_0}^{H'} \partial_y v_y dz + \int_{-H_0}^{H'} \partial_z v_z dz \tag{8.96}$$

$$= -v_x|_{z=H'} \partial_x H' + \partial_x \int_{-H_0}^{H'} v_x dz - v_y|_{z=H'} \partial_y H' + \partial_y \int_{-H_0}^{H'} v_y dz + v_z|_{z=H'} - v_z|_{z=-H_0} \tag{8.97}$$

since H' depends on the x and y coordinates. Due to the fact that there is no fluid above H' and below $-H_0$, we cannot have any normal flow to the two surfaces and get the additional boundary conditions

$$0 = v_z|_{z=-H_0} \tag{8.98}$$

$$0 = [-\partial_t H' + v_x \partial_x H' + v_y \partial_y H' - v_z]|_{z=H'}. \tag{8.99}$$

Plugging these into the previous equation results in

$$0 = \partial_x (H \langle v_x \rangle_z) + \partial_y (H \langle v_y \rangle_z) + \partial_t H', \tag{8.100}$$

where $\langle \cdot \rangle_z = \int_{-H_0}^{H'} \cdot dz$ is the depth average. In the case $H' << H_0$, we finally find

$$0 = H_0 (\partial_x \langle v_x \rangle_z + \partial_y \langle v_y \rangle_z) + \partial_t H'. \tag{8.101}$$

z-Momentum Equation Next, we consider Eq. (8.93). Since the water is shallow, we expect that the change in z-velocity is small. In particular, we assume that the complete left-hand side is much smaller than the right-hand side and neglect it completely. This results in

$$\partial_z p = \rho g, \tag{8.102}$$

i.e. that the pressure distribution is *hydrostatic*. Consequently, for the x and y directions we find

$$\partial_x p = \rho g \partial_x H' \tag{8.103}$$

$$\partial_y p = \rho g \partial_y H'. \tag{8.104}$$

x-Momentum Equation In the limit that $v_x \partial_x v_x$, $v_y \partial_y v_x$ and $v_z \partial_z v_x$ are small, we can integrate Eq. (8.91) over depth and find

$$\partial_t (H \langle v_x \rangle_z) = -\int_{-H_0}^{H'} \frac{1}{\rho} \partial_x p dz. \tag{8.105}$$

Since $\partial_x p = \rho g \partial_x H'$ [Eq. (8.103)],

$$\partial_t (H \langle v_x \rangle_z) = -gH\partial_x H'. \tag{8.106}$$

Again assuming that $H' << H_0$, we arrive at

$$\partial_t \langle v_x \rangle_z = -g\partial_x H'. \tag{8.107}$$

y-Momentum Equation Analogously as for the x-momentum equation, we arrive at

$$\partial_t \langle v_y \rangle_z = -g\partial_y H'. \tag{8.108}$$

8.4.1 Discretization Schemes

The *Shallow Water Equations* consist of Eq. (8.101), Eq. (8.107) and Eq. (8.108). For convenience we change the notation and introduce $u = \langle v_x \rangle_z$, $v = \langle v_y \rangle_z$ and $\eta = H'$ such that we can write

$$\partial_t \eta = -H_0(\partial_x u + \partial_y v) \tag{8.109}$$
$$\partial_t u = -g\partial_x \eta \tag{8.110}$$
$$\partial_t v = -g\partial_y \eta. \tag{8.111}$$

Restricting ourselves further to the one-dimensional case results in

$$\partial_t \eta = -H_0 \partial_x u \tag{8.112}$$
$$\partial_t u = -g\partial_x \eta. \tag{8.113}$$

Again we will see how to solve them numerically using finite difference schemes. As both equations depend on each other, it is quite natural to solve them using forward-backward time-stepping (here, forward for u and backward for η). Spatially, we consider two different situations

- A-Grid (unstaggered), centered in space:

$$\frac{\eta_j^n - \eta_j^{n-1}}{\Delta t} = -H_0 \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} \tag{8.114}$$
$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -g \frac{\eta_{j+1}^n - \eta_{j-1}^n}{2\Delta x} \tag{8.115}$$

- C-Grid (staggered), centered in space:

$$\frac{\eta_j^n - \eta_j^{n-1}}{\Delta t} = -H_0 \frac{u_{j+\frac{1}{2}}^n - u_{j-\frac{1}{2}}^n}{\Delta x} \tag{8.116}$$
$$\frac{u_{j+\frac{1}{2}}^{n+1} - u_{j+\frac{1}{2}}^n}{\Delta t} = -g \frac{\eta_{j+1}^n - \eta_j^n}{\Delta x} \tag{8.117}$$

  Here, we introduce two spatial grids shifted by 1/2 relative to each other: one for η (integer), one for u (integer + 1/2).

In both situations, we define the same Courant number $c = \sqrt{gH_0}\frac{\Delta t}{\Delta x}$ and doing a von Neumann stability analysis reveals that the scheme for the unstaggered grid is stable for $c \leq 2$ while for the staggered grid stability is only given for $c \leq 1$. However, in the latter case the dispersion is much better, see Fig. 8.6.

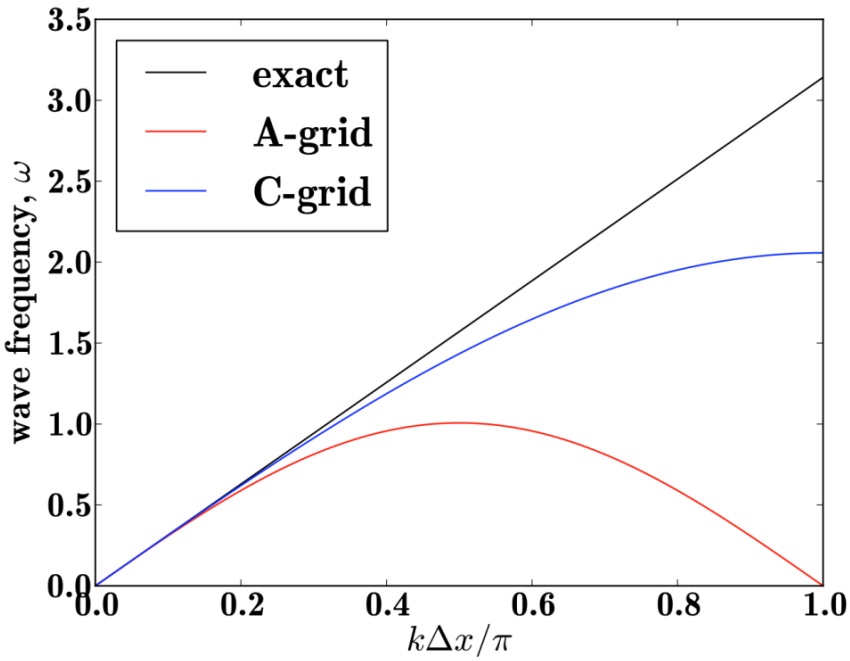


Figure 8.6: Dispersion for the A- and C-grids of the 1D shallow water equations.

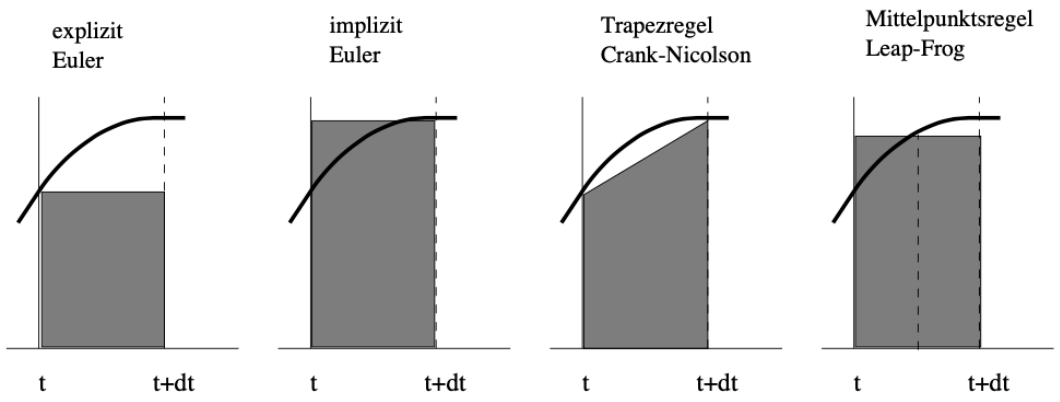


Figure 9.1: Illustration of different integration schemes.

Chapter 9

Time Integration and ODEs

9.1 Time Integration - Overview

9.1.1 Introduction

Consider the following initial value problem

$$d_t\phi = f(t, \phi(t)), \qquad \phi(t_0) = \phi^0. \tag{9.1}$$

The time dependency is solved by integration along the time direction, i.e. from time t_n to $t_{n+1} = t_n + \Delta t$. The exact integral of $d_t\phi$ is given by

$$\int_{t_n}^{t_{n+1}} d_t\phi = \phi^{n+1} - \phi^n. \tag{9.2}$$

Figure 9.1 shows four different possibilities of how an integral can be calculated by two time steps. These four procedures form a family of one step approximate numerical methods.

9.1.2 Numerical Errors

There are two major sources of errors associated with a numerical integration scheme for ODEs, namely, *truncation errors* and *round-off errors*. Truncation errors arise in Euler's

method because the curve of the solution is not generally a straight-line between the neighbouring grid-points. The error associated with this approximation can easily be assessed by Taylor expanding the solution around its argument.

Every time we take a step using Euler's method we incur a truncation error of $\mathcal{O}(\Delta t^2)$, where Δt is the step-length. Suppose that we use Euler's method to integrate our ODE over an interval of order unity. This requires $\mathcal{O}(\Delta t^{-1})$ steps. If each step incurs an error of $\mathcal{O}(\Delta t^2)$, and the errors are simply cumulative (a fairly conservative assumption), then the net truncation error is $\mathcal{O}(\Delta t)$. In other words, the error associated with integrating an ODE over a finite interval using Euler's method is directly proportional to the step-length. Thus, if we want to keep the relative error in the integration below, for example, 10^{-6} then we would need to take about 10^6 steps per unit interval. Incidentally, Euler's method is termed a first-order integration method because the truncation error associated with integrating over a finite interval scales like $\mathcal{O}(\Delta t)$.

Note that truncation errors would be present even if computers performed floating-point arithmetic operations to infinite accuracy. Unfortunately, computers do not perform such operations to infinite accuracy. In fact, a computer is only capable of storing a floating-point number to a fixed number of decimal places. For every type of computer, there is a characteristic number, η, which is defined as the smallest number which when added to a number of order unity gives rise to a new number. Every floating-point operation incurs a round-off error of $\mathcal{O}(\eta)$ which arises from the finite accuracy to which floating-point numbers are stored by the computer. Suppose that we use Euler's method to integrate our ODE over an interval of order unity. This entails $\mathcal{O}(\Delta t^{-1})$ integration steps, and, therefore, $\mathcal{O}(\Delta t^{-1})$ floating-point operations. If each floating-point operation incurs an error of $\mathcal{O}(\eta)$, and the errors are simply cumulative, then the net round-off error is

$$\mathcal{O}\left(\frac{\eta}{\Delta t}\right). \tag{9.3}$$

The total error, ϵ, associated with integrating our ODE over an interval of order unity is (approximately) the sum of the truncation and round-off errors. Thus, for Euler's method we get

$$\epsilon \sim \frac{\eta}{\Delta t} + \Delta t. \tag{9.4}$$

Clearly, at large step-lengths Δt the error is dominated by truncation errors, whereas round-off errors dominate at small step-lengths. The net error attains its minimal value, $\epsilon \sim \sqrt{\eta}$ when $\Delta t = \Delta t_0 = \sqrt{\eta}$. Therefore, there is clearly no point in making the step-length any smaller since this increases the number of floating point operations but does not lead to an increase in the overall accuracy. It is also clear that the ultimate accuracy of Euler's method (or any other numerical integration method) is determined by the accuracy, η, to which floating-point numbers are stored on the computer performing the calculation.

The value of η depends on how many bytes the computer hardware uses to store floating-point numbers. In the Julia-REPL you can check this with the `eps` function:

```
julia> eps(Float32)
1.1920929f-7
```

```
julia> eps(Float64)
2.220446049250313e-16
```

By default, Julia uses the type `Float64` for real floating-point numbers. Thus, the minimal optimal step length for Euler's method is $\Delta t_0 \sim 10^{-8}$, yielding a minimal relative integration error of $\epsilon_0 \sim 10^{-8}$. This level of accuracy is perfectly adequate for most scientific calculations. The corresponding η value for single precision (`Float32`) yields a minimal optimal step-length and a minimum relative error for Euler's method of $\Delta t_0 \sim 3 \times 10^{-4}$ and $\epsilon_0 \sim 3 \times 10^{-4}$, respectively. This level of accuracy is generally not adequate for many scientific calculations. However, there is a great interest in mixed precision arithmetic mainly to save memory and to make use of the faster low precision arithmetic available in modern CPUs und GPUs.

9.1.3 A Family of One Step Methods

The one step methods can be generalized as follows:

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} = \alpha f(t + \Delta t, \phi^{n+1}) + (1 - \alpha) f(t, \phi^n), \tag{9.5}$$

where $0 \leq \alpha \leq 1$ and we obtain

α	method	stability	order
0	explicit Euler	conditionally stable	1
1	implicit Euler	stable	1
1/2	Crank-Nicholson	stable	2

9.1.4 A Family of Multi Step Methods

By using multi step methods a higher approximation order can be obtained. This is achieved by using the information from more time steps. For example, the general two step method is given by

$$\frac{(1+\beta)\phi^{n+1} - (1+2\beta)\phi^n + \beta\phi^{n-1}}{\Delta t} = \gamma f(t+\Delta t, \phi^{n+1}) + (1-\gamma+\delta) f(t, \phi^n) - \delta(f(t-\Delta t, \phi^{n-1}) \tag{9.6}$$

and we obtain

γ	β	δ	method	order
0	0	0	explicit Euler	1
1	0	0	implicit Euler	1
1/2	0	0	Crank-Nicholson	2
1	1/2	0	Backward difference	2
0	0	1/2	2 step Adams-type	2
0	-1/2	0	Leapfrog	2
5/12	0	1/12	Adams-Moulton	3

9.1.5 Runge-Kutta Methods

There are two main reasons why Euler's method is not generally used in scientific computing. Firstly, the truncation error per step associated with this method is far larger than those associated with other, more advanced, methods (for a given value of Δt). Secondly, Euler's method is often prone to numerical instabilities.
The methods most commonly employed to integrate ODEs were first developed by the German mathematicians C.D.T. Runge and M.W. Kutta in the latter half of the 19th century. The basic reasoning behind these so-called *Runge-Kutta methods* is outlined in the following. The main reason that Euler's method has such a large truncation error per step is that in evolving the solution from ϕ^n to ϕ^{n+1} the method only evaluates derivatives at the beginning of the interval, i.e. at ϕ^n. In essence, Runge-Kutta methods introduce intermediate steps between the time step t_n and t_{n+1} and can be constructed with an arbitrary degree of accuracy. The method is, therefore, very asymmetric with respect to the beginning and the end of the interval. We can construct a more symmetric integration method by making an Euler-like trial step to the midpoint of the interval, and then using the values at the midpoint to make the real step across the interval. To be more exact,

$$k_1 = \Delta t f(t_n, \phi^n) \tag{9.7}$$

$$k_2 = \Delta t f(t_{n+1/2}, \phi^n + \frac{k_1}{2}) \tag{9.8}$$

$$\phi^{n+1} = \phi^n + k_2 + \mathcal{O}(\Delta t^3). \tag{9.9}$$

As indicated in the error term, this symmetrization cancels out the first-order error, making the method second-order. In fact, the above method is generally known as a second-order Runge-Kutta method. Therefore, Euler's method can be thought of as a first-order Runge-Kutta method. Of course, there is no need to stop at a second-order method. By using two trial steps per interval, it is possible to cancel out both the first and second-order error terms, and, thereby, construct a third-order Runge-Kutta method. Likewise, three trial steps per interval yield a fourth-order method, and so on.
The general form is

$$\phi^{n+1} = \phi^n + \Delta t \sum_{r=1}^{R} c_r f^r. \tag{9.10}$$

with

$$f^r = f(t + \Delta t a_r, \phi^n + \Delta t \sum_{s=1}^{r-1} b_{r,s} f^s), \quad r = 1, 2, \ldots R \tag{9.11}$$

and

$$a_r = \sum_{s=1}^{r-1} b_{r,s}, \quad r = 1, 2, \ldots R. \tag{9.12}$$

The most widely used RK-method is that of fourth order and can be written as

$$k_1 = \Delta t f(t_n, \phi^n) \tag{9.13}$$

$$k_2 = \Delta t f(t_{n+1/2}, \phi^n + \frac{k_1}{2}) \tag{9.14}$$

$$k_3 = \Delta t f(t_{n+1/2}, \phi^n + \frac{k_2}{2}) \tag{9.15}$$

$$k_4 = \Delta t f(t_{n+1}, \phi^n + k_3) \tag{9.16}$$

$$\phi^{n+1} = \phi^n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + \mathcal{O}(\Delta t^5). \tag{9.17}$$

9.2 Examples

Consider the following system of ODEs:

$$d_t x = v, \qquad d_t v = -kx \tag{9.18}$$

with initial values

$$x(0) = 0, \qquad v(0) = \sqrt{k}. \tag{9.19}$$

We know the analytic solution to be

$$x = \sin\Big(\sqrt{k}t\Big). \tag{9.20}$$

Let us compare the above solution with the one obtained numerically by using either Euler's method or a fourth-order Runge-Kutta method. Figure 9.2 shows the integration errors associated with these two methods (calculated by integrating the above system, with $k = 1$, from $t = 0 \ldots 10$, and then taking the difference between the numerical and analytic solutions) plotted against the step-length, Δt, in a log-log graph.

It can be seen that at large values of Δt, the error associated with Euler's method becomes much greater than unity (i.e. the magnitude of the numerical solution greatly exceeds that of the analytical solution), indicating the presence of a numerical instability. There are no similar signs of instability associated with the Runge-Kutta method. At intermediate Δt, the error associated with Euler's method decreases smoothly like Δt^{-1}. In this regime, the dominant error is truncation error, which is expected to scale like Δt^{-1} for a first-order method.

The error associated with the Runge-Kutta method similarly scales like Δt^{-4}—as expected for a fourth-order scheme—in the truncation error dominated regime. Note that, as Δt is decreased, the error associated with both methods eventually starts to rise in a jagged curve that scales roughly like Δt^1. This is a manifestation of round-off errors. The minimum error associated with both methods corresponds to the boundary between the truncation error dominated and the round-off error dominated regimes. Thus, for Euler's method the

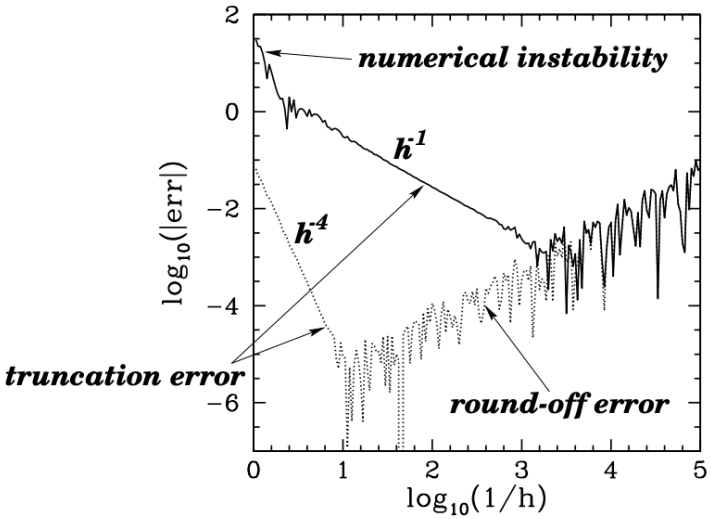


Figure 9.2: Single precision calculation. Global integration errors associated with Euler's method (solid) and a fourth-order Runge-Kutta method (dotted) plotted against the step-length Δt.

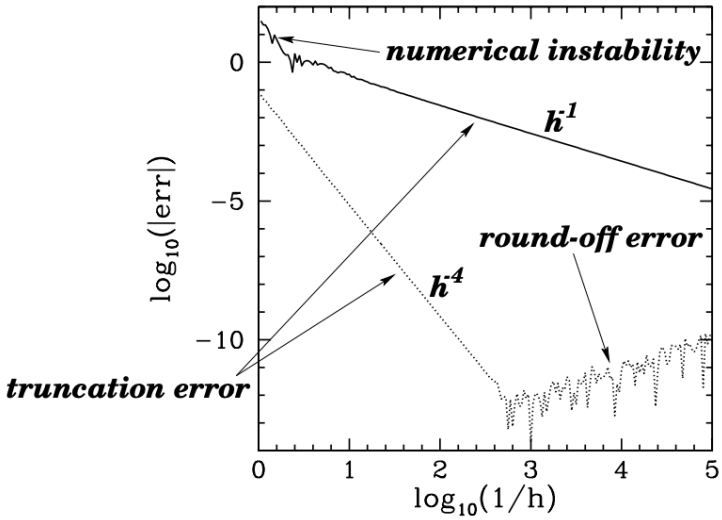


Figure 9.3: Double precision calculation. Global integration errors associated with Euler's method (solid) and a fourth-order Runge-Kutta method (dotted) plotted against the step-length Δt.

minimum error is about 10^{-3} at $\Delta t \sim 10^{-3}$, whereas for the Runge-Kutta method the minimum error is about 10^{-5} at $\Delta t \sim 10^{-1}$. Clearly, the performance of the Runge-Kutta method is vastly superior to that of Euler's method, since the former method is capable of attaining much greater accuracy than the latter using a far smaller number of steps (i.e. a far larger Δt).

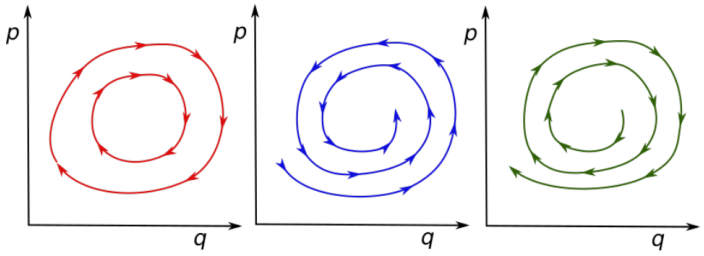


Figure 9.4: (Left) Phase space trajectories in a system where energy (=phase space volume) is conserved. (Middle) Phase space volume contraction as it would occur when energy is dissipated. (Right) Energy gain.

9.3 Conservation

Given a conservative Hamiltonian system, we recognize energy as a conserved quantity, i.e. it does not change with time. However, when we call our numerical integrator, it is not obvious whether energy is conserved to high order. In fact, it is unlikely. The integrators we have discussed so far are designed to match the equations of motion to a certain order and so to low order in step-size they will conserve energy. When constructing, for example, the Runge-Kutta methods, we have not required energy to be conserved at each step, so on long timescales there will be a drift in energy, see Fig. 9.4.

9.3.1 Volume Conservation in Phase Space

In a one-dimensional system with phase space density $\rho(\boldsymbol{r}, t)$, $\boldsymbol{r} = (q, p)$, the conservation of mass is described by the continuity equation

$$\partial_t \rho = -\boldsymbol{\nabla} \cdot (\rho \boldsymbol{v}), \tag{9.21}$$

where $\boldsymbol{v} = d_t \boldsymbol{r}$. If the system is incompressible, the density stays the same at all times, phase space volume is conserved and consequently

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0. \tag{9.22}$$

Explicitly, writing out the incompressibility condition yields

$$\partial_q (d_t q) + \partial_p (d_t p) = 0. \tag{9.23}$$

Now, consider a velocity-independent conservative force. In this case, we can write

$$\boldsymbol{F}(q) = -\partial_q \cdot U(q), \tag{9.24}$$

such that

$$\partial_p (d_t p) = \partial_p F(q) = 0. \tag{9.25}$$

Since $d_t q = p$ (for a unit mass), we also find $\partial_q(d_t q) = \partial_q p = 0$ and the incompressibility condition is satisfied, or equivalently, phase space volume is conserved at all times.
If, however, the force depends on velocity, we have in general $\partial_p(d_t p) \neq 0$ and phase space volume is not conserved. This is the situation for dissipating systems: phase space volume decreases.

9.3.2 Symplectic and Non-Symplectic First Order Integration for the Harmonic Oscillator

A simple example is that of a harmonic oscillator. With a spring constant $k = 1$ and a momentum per unit mass $p = d_t q$, the energy per unit mass is described by the Hamiltonian

$$H = \frac{1}{2}\left(q^2 + p^2\right). \tag{9.26}$$

The force per unit mass is given by $F(q) = -q$ and consequently the potential energy by $U(q) = q^2/2$. The equations of motion are

$$d_t q = \frac{\partial H}{\partial p} = p \tag{9.27}$$

$$d_t p = -\frac{\partial H}{\partial q} = -q \tag{9.28}$$

or equivalently

$$d_t^2 q = -q, \tag{9.29}$$

which describes an oscillation with angular frequency $\omega = 1$.
The exact evolution of the system can be described by

$$\begin{pmatrix} q(t) \\ p(t) \end{pmatrix} = \underbrace{\begin{pmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{pmatrix}}_{A} \begin{pmatrix} q(0) \\ p(0) \end{pmatrix}, \tag{9.30}$$

which is nothing else but a rotation ($\det(A) = 1$) in phase space!
If we expand for small Δt, we find

$$\begin{pmatrix} q^{n+1} \\ p^{n+1} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & \Delta t \\ -\Delta t & 1 \end{pmatrix}}_{\tilde{A}} \begin{pmatrix} q^n \\ p^n \end{pmatrix} \tag{9.31}$$

and $\det\left(\tilde{A}\right) = 1 + \Delta t^2 \neq 1$. This means that energy is not conserved since

$$H^{n+1} = \frac{1}{2}\left(q^{n+1^2} + p^{n+1^2}\right) = \frac{1}{2}\left(1 + \Delta t^2\right)\left(q^{n^2} + p^{n^2}\right). \tag{9.32}$$

Since $\Delta t > 0$, the energy will increase with every time step. Volume in phase space has not been conserved. After many steps the trajectory in phase space will spiral outwards and the phase space area increases.

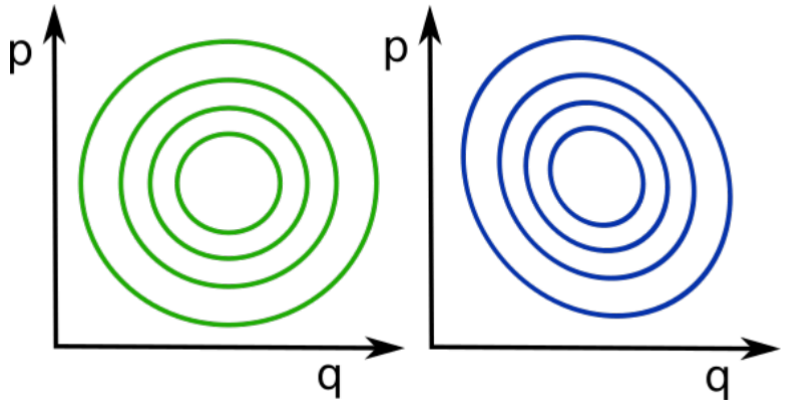


Figure 9.5: (Left) Level curves of the Hamiltonian H. (Right) Level curves of the Hamiltonian H'. The trajectories of H' do not continuously diverge from those of the real system because they are confined to closed level curves.

However, we can change the transformation to

$$\tilde{A}' = \begin{pmatrix} 1 & \Delta t \\ -\Delta t & 1 - \Delta t^2 \end{pmatrix} \tag{9.33}$$

such that again $\det\left(\tilde{A}'\right) = 1$. We call this transformation *symplectic*. It conserves phase space volume and therefore also an energy. However, not the original energy H but instead the energy

$$H' = \frac{1}{2}\left(q^2 + p^2\right) + \frac{\Delta t}{2} qp, \tag{9.34}$$

called the *modified Hamiltonian*. The advantage of such a symplectic method is the conservation, here of H', which means that the error $H - H'$ is bounded and will not grow forever, see Fig. 9.5.

Chapter 10

Maxwell Equations

Electromagnetism is one of the four fundamental forces of the universe. Apart from gravity, most daily phenomena can be explained by electromagnetism. The very existence of atoms requires the electric force (plus weird quantum effects) to hold electrons to the nucleus, and molecules are formed again due to electric forces between atoms. More macroscopically, electricity powers all our electrical appliances (by definition), while the magnetic force makes things stick on our fridge. Finally, it is the force that gives rise to *light*, and allows us to see things.

While it seems to be responsible for so many effects, the modern classical description of electromagnetism is rather simple. It is captured by merely *four* short and concise equations known as Maxwell's equations. In fact, in the majority of this course, we would simply be exploring different solutions to these equations.

Historically, electromagnetism has another significance — it led to Einstein's discovery of special relativity. At the end of the course, we will look at how Maxwell's equations naturally fit in nicely in the framework of relativity. Written relativistically, Maxwell's equation look *much* simpler and more elegant. We will discover that magnetism is *entirely* a relativistic effect, and makes sense only in the context of relativity.

As we move to the world of special relativity, not only do we get a better understanding of Maxwell's equation. As a takeaway, we also get to understand relativity itself better, as we provide a more formal treatment of special relativity, which becomes a powerful tool to develop theories consistent with relativity.

10.0.1 Charge and Current

The strength of the electromagnetic force experienced by a particle is determined by its *(electric) charge*. The SI unit of charge is the *Coulomb*. In this course, we assume that the charge can be any real number. However, at the fundamental level, charge is quantised. All particles carry charge $q = ne$ for some integer n, and the basic unit $e \approx 1.6 \times 10^{-19}\,\mathrm{C}$. For example, the electron has $n = -1$, proton has $n = +1$, neutron has $n = 0$.

Often, it will be more useful to talk about *charge density* $\rho(\mathbf{x}, t)$.

Definition (Charge density)**.** The *charge density* is the charge per unit volume. The total

charge in a region V is

$$Q(t) = \int_V \rho(\mathbf{x}, t)\, dV$$

When we study charged sheets or lines, the charge density would be charge per unit area or length instead, but this would be clear from context.

Definition (Current and current density)**.** For any surface S, the integral

$$I = \int_S \mathbf{J} \cdot d\mathbf{S}$$

counts the charge per unit time passing through S. I is the *current*, and $\mathbf{J}$ is the *current density*, "current per unit area".

Intuitively, if the charge distribution $\rho(\mathbf{x}, t)$ has velocity $\mathbf{v}(x, t)$, then (neglecting relativistic effects), we have

$$\mathbf{J} = \rho\mathbf{v}.$$

Charge is locally conserved, if it disappears here, it must have moved to somewhere nearby. Alternatively, charge density can only change due to continuous currents. This is captured by the *continuity equation*:

Law (Continuity equation)**.**

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{J} = 0.$$

We can write this into a more intuitive integral form via the divergence theorem.
The charge Q in some region V is defined to be

$$Q = \int_V \rho\, dV.$$

So

$$\frac{dQ}{dt} = \int_V \frac{\partial \rho}{\partial t}\, dV = -\int_V \nabla \cdot \mathbf{J}\, dV = -\int_S \mathbf{J} \cdot dS.$$

Hence the continuity equation states that the change in total charge in a volume is given by the total current passing through its boundary.
In particular, we can take $V = \mathbb{R}^3$, the whole of space. If there are no currents at infinity, then

$$\frac{dQ}{dt} = 0$$

So the continuity equation implies the conservation of charge.

10.0.2 Forces and Fields

In modern physics, we believe that all forces are mediated by *fields* (not to be confused with "fields" in algebra, or agriculture). A *field* is a dynamical quantity (i.e. a function) that assigns a value to every point in space and time. In electromagnetism, we have two fields:

- the electric field $\mathbf{E}(\mathbf{x}, t)$;
- the magnetic field $\mathbf{B}(\mathbf{x}, t)$.

Each of these fields is a vector, i.e. it assigns a *vector* to every point in space and time, instead of a single number.
The fields interact with particles in two ways. On the one hand, fields cause particles to move. On the other hand, particles create fields. The first aspect is governed by the Lorentz force law:

Law (Lorentz force law)**.**

$$\mathbf{F} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B})$$

while the second aspect is governed by *Maxwell's equations.*

Law (Maxwell's Equations)**.**

$$\begin{aligned} \nabla \cdot \mathbf{E} &= \frac{\rho}{\varepsilon_0} \\ \nabla \cdot \mathbf{B} &= 0 \\ \nabla \times \mathbf{E} + \frac{\partial \mathbf{B}}{\partial t} &= 0 \\ \nabla \times \mathbf{B} - \mu_0 \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t} &= \mu_0 \mathbf{J}, \end{aligned}$$

where we have two constants of nature:

- $\varepsilon_0 = 8.85 \times 10^{-12}\,\mathrm{m}^{-3}\,\mathrm{kg}^{-1}\,\mathrm{s}^2\,\mathrm{C}^2$ is the electric constant;
- $\mu_0 = 4\pi \times 10^{-7}\,\mathrm{m\,kg\,C}^{-2}$ is the magnetic constant.

These constants are the "permittivity of free space" and "permeability of free space".

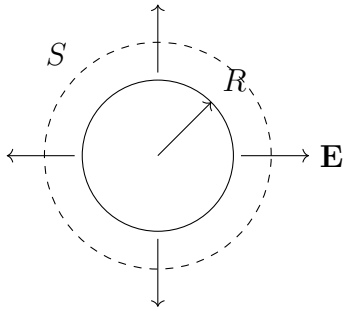
We've just completed the description of all of (classical, non-relativistic) electromagnetism. The remaining would be to study the consequences of and solutions to these equations.

10.1 Electrostatics

Electrostatics is the study of stationary charges in the absence of magnetic fields. We take $\rho = \rho(\mathbf{x})$, $\mathbf{J} = 0$ and $\mathbf{B} = 0$. We then look for time-independent solutions. In this case, the only relevant equations are

$$\begin{aligned} \nabla \cdot \mathbf{E} &= \frac{\rho}{\varepsilon_0} \\ \nabla \times \mathbf{E} &= 0, \end{aligned}$$

and the other two equations just give $0 = 0$.
In this chapter, our goal is to find $\mathbf{E}$ for any ρ. Of course, we first start with simple, symmetric cases, and then tackle the more general cases later.



10.1.1 Gauss' Law

Here we transform the first Maxwell's equation into an integral form, known as *Gauss' Law*. Consider a region $V \subseteq \boldsymbol{r}^3$ with boundary $S = \partial V$. Then integrating the first equation over the volume V gives

$$\int_V \nabla \cdot \mathbf{E}\, dV = \frac{1}{\varepsilon_0} \int_V \rho\, dV.$$

The divergence theorem gives $\int_V \nabla \cdot \mathbf{E}\, dV = \int_S \mathbf{E} \cdot d\mathbf{S}$, and by definition, $Q = \int_V \rho\, dV$. So we end up with

Law (Gauss' law)**.**

$$\int_S \mathbf{E} \cdot d\mathbf{S} = \frac{Q}{\varepsilon_0},$$

where Q is the total charge inside V.

Definition (Flux through surface)**.** The *flux* of $\mathbf{E}$ through the surface S is defined to be

$$\int_S \mathbf{E} \cdot d\mathbf{S}.$$

Gauss' law tells us that the flux depends only on the total charge contained inside the surface. In particular any external charge does not contribute to the total flux. While external charges *do* create fields that pass through the surface, the fields have to enter the volume through one side of the surface and leave through the other. Gauss' law tells us that these two cancel each other out exactly, and the total flux caused by external charges is zero.
From this, we can prove Coulomb's law, consider a spherically symmetric charge density $\rho(r)$ with $\rho(r) = 0$ for $r > R$, i.e. all the charge is contained in a ball of radius R.
By symmetry, the force is the same in all directions and point outward radially. So

$$\mathbf{E} = E(r)\hat{\mathbf{r}}.$$

This immediately ensures that $\nabla \times \mathbf{E} = 0$.
Put S to be a sphere of radius $r > R$. Then the total flux is

$$\begin{aligned}\int_S \mathbf{E} \cdot d\mathbf{S} &= \int_S E(r)\hat{\mathbf{r}} \cdot d\mathbf{S} \\ &= E(r) \int_S \hat{\mathbf{r}} \cdot d\mathbf{S}\end{aligned}$$

$$= E(r) \cdot 4\pi r^2$$

By Gauss' law, we know that this is equal to $\frac{Q}{\varepsilon_0}$. Therefore

$$E(r) = \frac{Q}{4\pi\varepsilon_0 r^2}$$

and

$$\mathbf{E}(r) = \frac{Q}{4\pi\varepsilon_0 r^2}\hat{\mathbf{r}}.$$

By the Lorentz force law, the force experienced by a second charge is

$$\mathbf{F}(\mathbf{r}) = \frac{Qq}{4\pi\varepsilon_0 r^2}\hat{\mathbf{r}},$$

which is Coulomb's law.
Strictly speaking, this only holds when the charges are not moving. However, for most practical purposes, we can still use this because the corrections required when they are moving are tiny.

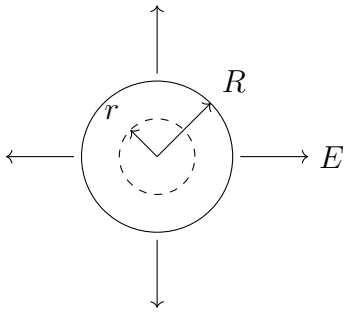
Example. Consider a uniform sphere with

$$\rho(r) = \begin{cases} \rho & r < R \\ 0 & r > R \end{cases}.$$

Outside, we know that

$$\mathbf{E}(r) = \frac{Q}{4\pi\varepsilon_0 r^2}\hat{\mathbf{r}}$$

Now suppose we are inside the sphere.



Then

$$\int_S \mathbf{E} \cdot d\mathbf{S} = E(r)4\pi r^2 = \frac{Q}{\varepsilon_0}\left(\frac{r^3}{R^3}\right)$$

So

$$\mathbf{E}(r) = \frac{Qr}{4\pi\varepsilon_0 R^3}\hat{\mathbf{r}},$$

and the field increases with radius.

10.1.2 Electrostatic potential

In the most general case, we will have to solve both $\nabla \cdot \mathbf{E} = \rho/\varepsilon_0$ and $\nabla \times \mathbf{E} = \mathbf{0}$. However, we already know that the general form of the solution to the second equation is $\mathbf{E} = -\nabla\phi$ for some scalar field ϕ.

Definition (Electrostatic potential)**.** If $\mathbf{E} = -\nabla\phi$, then ϕ is the *electrostatic potential*.

Substituting this into the first equation, we obtain

$$\nabla^2\phi = -\frac{\rho}{\varepsilon_0}.$$

This is the *Poisson equation*, which we have studied in other courses. If we are in the middle of nowhere and $\rho = 0$, then we get the *Laplace equation*.
There are a few interesting things to note about our result:

- ϕ is only defined up to a constant. We usually fix this by insisting $\phi(\mathbf{r}) \to 0$ as $r \to \infty$. This statement seems trivial, but this property of ϕ is actually very important and gives rise to a lot of interesting properties. However, we will not have the opportunity to explore this in this course.
- The Poisson equation is linear. So if we have two charges ρ_1 and ρ_2, then the potential is simply $\phi_1 + \phi_2$ and the field is $\mathbf{E}_1 + \mathbf{E}_2$. This is the *principle of superposition*. Among the four fundamental forces of nature, electromagnetism is the only force with this property.

Point charge

Consider a point particle with charge Q at the origin. Then

$$\rho(\mathbf{r}) = Q\delta^3(\mathbf{r}).$$

Here δ^3 is the generalization of the usual delta function for (3D) vectors.
The equation we have to solve is

$$\nabla^2\phi = -\frac{Q}{\varepsilon_0}\delta^3(\mathbf{r}).$$

Away from the origin $\mathbf{r} = \mathbf{0}$, $\delta^3(\mathbf{r}) = 0$, and we have the Laplace equation. From the IA Vector Calculus course, the general solution is

$$\phi = \frac{\alpha}{r} \quad \text{for some constant } \alpha.$$

The constant α is determined by the delta function. We integrate the equation over a sphere of radius r centered at the origin. The left hand side gives

$$\int_V \nabla^2\phi \, dV = \int_S \nabla\phi \cdot d\mathbf{S} = \int_S -\frac{\alpha}{r^2}\hat{\mathbf{r}} \cdot d\mathbf{S} = -4\pi\alpha$$

The right hand side gives

$$-\frac{Q}{\varepsilon_0}\int_V \delta^3(r) \, dV = -\frac{Q}{\varepsilon_0}.$$

So

$$\alpha = \frac{Q}{4\pi\varepsilon_0}$$

and

$$\mathbf{E} = -\nabla\phi = \frac{Q}{4\pi\varepsilon_0 r^2}\hat{\mathbf{r}}.$$

This is just what we get from Coulomb's law.

General charge distribution

To find ϕ for a general charge distribution ρ, we use the Green's function for the Laplacian. The Green's function is defined to be the solution to

$$\nabla^2 G(\mathbf{r}, \mathbf{r}') = \delta^3(\mathbf{r} - \mathbf{r}'),$$

In the section about point charges, We have shown that

$$G(\mathbf{r}, \mathbf{r}') = -\frac{1}{4\pi}\frac{1}{|\mathbf{r} - \mathbf{r}'|}.$$

We assume all charge is contained in some compact region V. Then

$$\begin{aligned}\phi(\mathbf{r}) &= -\frac{1}{\varepsilon_0}\int_V \rho(\mathbf{r}')G(\mathbf{r}, \mathbf{r}')\, d^3\mathbf{r}' \\ &= \frac{1}{4\pi\varepsilon_0}\int_V \frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|}\, d^3\mathbf{r}'\end{aligned}$$

Then

$$\begin{aligned}\mathbf{E}(\mathbf{r}) &= -\nabla\phi(\mathbf{r}) \\ &= -\frac{1}{4\pi\varepsilon_0}\int_V \rho(\mathbf{r}')\nabla\left(\frac{1}{|\mathbf{r} - \mathbf{r}'|}\right)\, d^3\mathbf{r}' \\ &= \frac{1}{4\pi\varepsilon_0}\int_V \rho(\mathbf{r}')\frac{(\mathbf{r} - \mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|^3}\, d^3\mathbf{r}'\end{aligned}$$

So if we plug in a very complicated ρ, we get a very complicated $\mathbf{E}$!
However, we can ask what ϕ and $\mathbf{E}$ look like very far from V, i.e. $|\mathbf{r}| \gg |\mathbf{r}'|$.
We again use the Taylor expansion.

$$\begin{aligned}\frac{1}{|\mathbf{r} - \mathbf{r}'|} &= \frac{1}{r} + \mathbf{r}' \cdot \nabla\left(\frac{1}{r}\right) + \cdots \\ &= \frac{1}{r} + \frac{\mathbf{r} \cdot \mathbf{r}'}{r^3} + \cdots.\end{aligned}$$

Then we get

$$\phi(\mathbf{r}) = \frac{1}{4\pi\varepsilon_0}\int_V \rho(\mathbf{r}')\left(\frac{1}{r} + \frac{\mathbf{r} \cdot \mathbf{r}'}{r^3} + \cdots\right)\, d^3\mathbf{r}'$$

$$= \frac{1}{4\pi\varepsilon_0}\left(\frac{Q}{r} + \frac{\mathbf{p}\cdot\hat{\mathbf{r}}}{r^2} + \cdots\right),$$

where

$$Q = \int_V \rho(\mathbf{r}')\, dV'$$
$$\mathbf{p} = \int_V \mathbf{r}'\rho(\mathbf{r}')\, dV'$$
$$\hat{\mathbf{r}} = \frac{\mathbf{r}}{\|\mathbf{r}\|}.$$

So if we have a huge lump of charge, we can consider it to be a point charge Q, plus some dipole correction terms.

10.1.3 Electrostatic energy

We want to calculate how much energy is stored in the electric field. Recall that a particle of charge q in a field $\mathbf{E} = -\nabla\phi$ has potential energy $U(\mathbf{r}) = q\phi(\mathbf{r})$.
$U(\mathbf{r})$ can be thought of as the work done in bringing the particle from infinity, as illustrated below:

$$\begin{aligned}\text{work done} &= -\int_\infty^{\mathbf{r}} \mathbf{F}\cdot d\mathbf{r}\\ &= -q\int_\infty^{\mathbf{r}} \mathbf{E}\cdot d\mathbf{r}\\ &= q\int_\infty^{\mathbf{r}} \nabla\phi\cdot d\mathbf{r}\\ &= q[\phi(\mathbf{r}) - \phi(\infty)]\\ &= U(\mathbf{r})\end{aligned}$$

where we set $\phi(\infty) = 0$.
Now consider N charges q_i at positions $\mathbf{r}_i$. The total potential energy stored is the work done to assemble these particles. Let's put them in one by one.

1. The first charge is free. The work done is $W_1 = 0$.

2. To place the second charge at position $\mathbf{r}_2$ takes work. The work is

$$W_2 = \frac{q_1 q_2}{4\pi\varepsilon_0}\frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}.$$

3. To place the third charge at position $\mathbf{r}_3$, we do

$$W_3 = \frac{q_3}{4\pi\varepsilon_0}\left(\frac{q_1}{|\mathbf{r}_1 - \mathbf{r}_3|} + \frac{q_2}{|\mathbf{r}_2 - \mathbf{r}_3|}\right)$$

4. etc.

The total work done is

$$U = \sum_{i=1}^{N} W_i = \frac{1}{4\pi\varepsilon_0} \sum_{i<j} \frac{q_i q_j}{|\mathbf{r}_i - \mathbf{r}_j|}.$$

Equivalently,

$$U = \frac{1}{4\pi\varepsilon_0} \frac{1}{2} \sum_{i\neq j} \frac{q_i q_j}{|\mathbf{r}_i - \mathbf{r}_j|}.$$

We can write this in an alternative form. The potential at point $\mathbf{r}_i$ due to all other particles is

$$\phi(\mathbf{r}_i) = \frac{1}{4\pi\varepsilon_0} \sum_{j\neq i} \frac{q_j}{|\mathbf{r}_i - \mathbf{r}_j|}.$$

So we can write

$$U = \frac{1}{2} \sum_{i=1}^{N} q_i \phi(\mathbf{r}_i).$$

There is an obvious generalization to continuous charge distributions:

$$U = \frac{1}{2} \int \rho(\mathbf{r}) \phi(\mathbf{r}) \, d^3\mathbf{r}.$$

Hence we obtain

$$\begin{aligned} U &= \frac{\varepsilon_0}{2} \int (\nabla \cdot \mathbf{E}) \phi \, d^3\mathbf{r} \\ &= \frac{\varepsilon_0}{2} \int [\nabla \cdot (\mathbf{E}\phi) - \mathbf{E} \cdot \nabla\phi] \, d^3\mathbf{r}. \end{aligned}$$

The first term is a total derivative and vanishes. In the second term, we use the definition $\mathbf{E} = -\nabla\phi$ and obtain

$$U = \frac{\varepsilon_0}{2} \int \mathbf{E} \cdot \mathbf{E} \, d^3\mathbf{r}.$$

This derivation of potential energy is not satisfactory. The final result shows that the potential energy depends only on the field itself, and not the charges. However, the result was derived using charges and electric potentials — there should be a way to derive this result directly with the field, and indeed there is. However, this derivation belongs to a different course.

Also, we have waved our hands a lot when generalizing to continuous distributions, which was not entirely correct. If we have a single point particle, the original discrete formula implies that there is no potential energy. However, since the associated field is non-zero, our continuous formula gives a non-zero potential.

This does *not* mean that the final result is wrong. It is correct, but it describes a more sophisticated (and preferred) conception of "potential energy". Again, we shall not go into the details in this course.

10.2 Electrodynamics

So far, we have only looked at fields that do not change with time. However, in real life, fields *do* change with time. We will now look at time-dependent $\mathbf{E}$ and $\mathbf{B}$ fields.

10.2.1 Induction

We'll explore the Maxwell equation

$$\nabla \times \mathbf{E} + \frac{\partial \mathbf{B}}{\partial t} = 0.$$

In short, if the magnetic field changes in time, i.e. $\frac{\partial \mathbf{B}}{\partial t} \neq 0$, this creates an $\mathbf{E}$ that accelerates charges, which creates a current in a wire. This process is called induction. Consider a wire, which is a closed curve C, with a surface S.
We integrate over the surface S to obtain

$$\int_S (\nabla \times \mathbf{E}) \cdot d\mathbf{S} = -\int_S \frac{\partial \mathbf{B}}{\partial t} \cdot d\mathbf{S}.$$

By Stokes' theorem and commutativity of integration and differentiation (assuming S and C do not change in time), we have

$$\int_C \mathbf{E} \cdot d\mathbf{r} = -\frac{d}{dt} \int_S \mathbf{B} \cdot d\mathbf{S}.$$

Definition (Electromotive force (emf))**.** The *electromotive force* (emf) is

$$\mathcal{E} = \int_C \mathbf{E} \cdot d\mathbf{r}.$$

Despite the name, this is not a force! We can think of it as the work done on a unit charge moving around the curve, or the "voltage" of the system.

For convenience we define the quantity

Definition (Magnetic flux)**.** The *magnetic flux* is

$$\Phi = \int_S \mathbf{B} \cdot d\mathbf{S}.$$

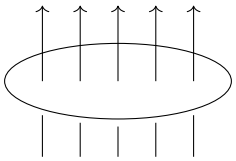
Then we have

Law (Faraday's law of induction)**.**

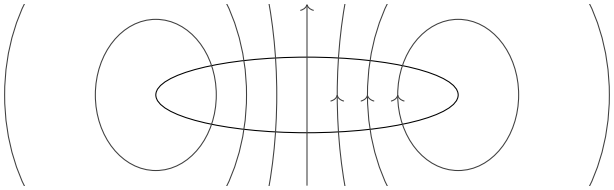
$$\mathcal{E} = -\frac{d\Phi}{dt}.$$

This says that when we change the magnetic flux through S, then a current is induced. In practice, there are many ways we can change the magnetic flux, such as by moving bar magnets or using an electromagnet and turning it on and off.
The minus sign has a significance. When we change a magnetic field, an emf is created. This induces a current around the wire. However, we also know that currents produce magnetic fields. The minus sign indicates that the induced magnetic field *opposes* the initial change in magnetic field. If it didn't and the induced magnetic field reinforces the change, we will get runaway behaviour and the world will explode. This is known as *Lenz's law*.

Example. Consider a circular wire with a magnetic field perpendicular to it.



If we decrease $\mathbf{B}$ such that $\dot{\Phi} < 0$, then $\mathcal{E} > 0$. So the current flows anticlockwise (viewed from above). The current generates its own $\mathbf{B}$. This acts to *increase* $\mathbf{B}$ inside, which counteracts the initial decrease.



This means you don't get runaway behaviour.

10.2.2 Magnetostatic energy

Suppose that a current I flows along a wire C. From magnetostatics, we know that this gives rise to a magnetic field $\mathbf{B}$, and hence a flux Φ given by

$$\Phi = \int_S \mathbf{B} \cdot d\mathbf{S},$$

where S is the surface bounded by C.

Definition (Inductance)**.** The *inductance* of a curve C, defined as

$$L = \frac{\Phi}{I},$$

is the amount of flux it generates per unit current passing through C. This is a property only of the curve C.

Inductance is something engineers care a *lot* about, as they need to create real electric circuits and make things happen. However, us mathematicians find these applications completely pointless and don't actually care about inductance. The only role it will play is in the proof we perform below.

Example (The solenoid)**.** Consider a solenoid of length ℓ and cross-sectional area A (with $\ell \gg \sqrt{A}$ so we can ignore end effects). We know that

$$B = \mu_0 I N,$$

where N is the number of turns of wire per unit length and I is the current. The flux through a single turn (pretending it is closed) is

$$\Phi_0 = \mu_0 I N A.$$

So the total flux is

$$\Phi = \Phi_0 N\ell = \mu_0 I N^2 V,$$

where V is the volume, $A\ell$. So

$$L = \mu_0 N^2 V.$$

We can use the idea of inductance to compute the energy stored in magnetic fields. The idea is to compute the work done in building up a current.
As we build the current, the change in current results in a change in magnetic field. This produces an induced emf that we need work to oppose. The emf is given by

$$\mathcal{E} = -\frac{d\Phi}{dt} = -L\frac{dI}{dt}.$$

This opposes the change in current by Lenz's law. In time δt, a charge $I\delta t$ flows around C. The work done is

$$\delta W = \mathcal{E} I \delta t = -LI\frac{dI}{dt}\delta t.$$

So

$$\frac{dW}{dt} = -LI\frac{dI}{dt} = -\frac{1}{2}L\frac{dI^2}{dt}.$$

So the work done to build up a current is

$$W = \frac{1}{2}LI^2 = \frac{1}{2}I\Phi.$$

Note that we dropped the minus sign because we switched from talking about the work done by the emf to the work done to oppose the emf.
This work done is identified with the energy stored in the system. Recall that the vector potential $\mathbf{A}$ is given by $\mathbf{B} = \nabla \times \mathbf{A}$. So

$$\begin{aligned} U &= \frac{1}{2}I\int_S \mathbf{B}\cdot d\mathbf{S} \\ &= \frac{1}{2}I\int_S (\nabla\times\mathbf{A})\cdot d\mathbf{S} \\ &= \frac{1}{2}I\oint_C \mathbf{A}\cdot d\mathbf{r} \\ &= \frac{1}{2}\int_{\boldsymbol{r}^3} \mathbf{J}\cdot\mathbf{A}\, dV \end{aligned}$$

Using Maxwell's equation $\nabla \times \mathbf{B} = \mu_0\mathbf{J}$, we obtain

$$\begin{aligned} &= \frac{1}{2\mu_0}\int (\nabla\times\mathbf{B})\cdot\mathbf{A}\, dV \\ &= \frac{1}{2\mu_0}\int [\nabla\cdot(\mathbf{B}\times\mathbf{A}) + \mathbf{B}\cdot(\nabla\times\mathbf{A})]\, dV \end{aligned}$$

Assuming that $\mathbf{B} \times \mathbf{A}$ vanishes sufficiently fast at infinity, the integral of the first term vanishes. So we are left with

$$= \frac{1}{2\mu_0} \int \mathbf{B} \cdot \mathbf{B} \, dV.$$

So

Proposition. The energy stored in a magnetic field is

$$U = \frac{1}{2\mu_0} \int \mathbf{B} \cdot \mathbf{B} \, dV.$$

In general, the energy stored in $\mathbf{E}$ and $\mathbf{B}$ is

$$U = \int \left(\frac{\varepsilon_0}{2} \mathbf{E} \cdot \mathbf{E} + \frac{1}{2\mu_0} \mathbf{B} \cdot \mathbf{B} \right) \, dV.$$

Note that while this is true, it does not follow directly from our results for pure magnetic and pure electric fields. It is entirely plausible that when both are present, they interact in weird ways that increases the energy stored. However, it turns out that this does not happen, and this formula is right.

10.2.3 Displacement currents

Recall that the Maxwell equations are

$$\begin{aligned} \nabla \cdot \mathbf{E} &= \frac{\rho}{\varepsilon_0} \\ \nabla \cdot \mathbf{B} &= 0 \\ \nabla \times \mathbf{E} &= -\frac{\partial \mathbf{B}}{\partial t} \\ \nabla \times \mathbf{B} &= \mu_0 \left(\mathbf{J} + \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t} \right) \end{aligned}$$

So far we have studied all the equations apart from the $\mu_0 \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t}$ term. Historically this term is called the *displacement current*.
The need of this term was discovered by purely mathematically, since people discovered that Maxwell's equations would be inconsistent with charge conservation without the term.
Without the term, the last equation is

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J}.$$

Take the divergence of the equation to obtain

$$\mu_0 \nabla \cdot \mathbf{J} = \nabla \cdot (\nabla \times \mathbf{B}) = 0.$$

But charge conservation says that

$$\dot{\rho} + \nabla \cdot \mathbf{J} = 0.$$

These can both hold iff $\dot{\rho} = 0$. But we clearly can change the charge density — pick up a charge and move it elsewhere! Contradiction.
With the new term, taking the divergence yields

$$\mu_0 \left(\nabla \cdot \mathbf{J} + \varepsilon_0 \nabla \cdot \frac{\partial \mathbf{E}}{\partial t} \right) = 0.$$

Since partial derivatives commute, we have

$$\varepsilon_0 \nabla \cdot \frac{\partial \mathbf{E}}{\partial t} = \varepsilon_0 \frac{\partial}{\partial t}(\nabla \cdot \mathbf{E}) = \dot{\rho}$$

by the first Maxwell's equation. So it gives

$$\nabla \cdot \mathbf{J} + \dot{\rho} = 0.$$

So with the new term, not only is Maxwell's equation consistent with charge conservation — it actually implies charge conservation.

10.2.4 Electromagnetic waves

We now look for solutions to Maxwell's equation in the case where $\rho = 0$ and $\mathbf{J} = 0$, i.e. in nothing/vacuum.
Differentiating the fourth equation with respect to time,

$$\begin{aligned} \mu_0 \varepsilon_0 \frac{\partial^2 \mathbf{E}}{\partial t^2} &= \frac{\partial}{\partial t}(\nabla \times \mathbf{B}) \\ &= \nabla \times \frac{\partial \mathbf{B}}{\partial t} \\ &= \nabla(\underbrace{\nabla \cdot \mathbf{E}}_{=\rho/\varepsilon_0 = 0}) + \nabla^2 \mathbf{E} \quad \text{by vector identities} \\ &= \nabla^2 \mathbf{E}. \end{aligned}$$

So each component of $\mathbf{E}$ obeys the wave equation

$$\frac{1}{c^2} \frac{\partial^2 \mathbf{E}}{\partial t^2} - \nabla^2 \mathbf{E} = 0.$$

We can do exactly the same thing to show that $\mathbf{B}$ obeys the same equation:

$$\frac{1}{c^2} \frac{\partial^2 \mathbf{B}}{\partial t^2} - \nabla^2 \mathbf{B} = 0,$$

where the speed of the wave is

$$c = \frac{1}{\sqrt{\mu_0 \varepsilon_0}}$$

Recall that

- $\varepsilon_0 = 8.85 \times 10^{-12}\,\mathrm{m}^{-3}\,\mathrm{kg}^{-1}\,\mathrm{s}^2\,\mathrm{C}^2$

- $\mu_0 = 4\pi \times 10^{-7}\,\mathrm{m\,kg\,C^{-2}}$

So

$$c = 3 \times 10^8\,\mathrm{m\,s^{-1}},$$

which is the speed of light!
We now look for plane wave solutions which propagate in the x direction, and are independent of y and z. So we can write our electric field as

$$\mathbf{E}(\mathbf{x}) = (E_x(x,t), E_y(x,t), E_z(x,t)).$$

Hence any derivatives wrt y and z are zero. Since we know that $\nabla \cdot \mathbf{E} = 0$, E_x must be constant. We take $E_x = 0$. Assume $E_z = 0$, ie the wave propagate in the x direction and oscillates in the y direction. Then we look for solutions of the form

$$\mathbf{E} = (0, E(x,t), 0),$$

with

$$\frac{1}{c^2}\frac{\partial^2 \mathbf{E}}{\partial t^2} - \frac{\partial^2 \mathbf{E}}{\partial x^2} = 0.$$

The general solution is

$$E(x,t) = f(x - ct) + g(x + ct).$$

The most important solutions are the *monochromatic* waves

$$E = E_0 \sin(kx - \omega t).$$

Definition (Amplitude, wave number and frequency)**.**

1. E_0 is the *amplitude*
2. k is the *wave number.*
3. ω is the *(angular) frequency.*

The wave number is related to the wavelength by

$$\lambda = \frac{2\pi}{k}.$$

Since the wave has to travel at speed c, we must have

$$\omega^2 = c^2 k^2$$

So the value of k determines the value of ω, vice versa.

To solve for $\mathbf{B}$,we use

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}.$$

So $\mathbf{B} = (0, 0, B)$ for some B. Hence the equation gives.

$$\frac{\partial B}{\partial t} = -\frac{\partial E}{\partial x}.$$

So

$$B = \frac{E_0}{c}\sin(kx - \omega t).$$

Note that this is uniquely determined by $\mathbf{E}$, and we do not get to choose our favorite amplitude, frequency etc for the magnetic component.
We see that $\mathbf{E}$ and $\mathbf{B}$ oscillate in phase, orthogonal to each other, and orthogonal to the direction of travel. These waves are what we usually consider to be "light".
Also note that Maxwell's equations are linear, so we can add up two solutions to get a new one. This is particularly important, since it allows light waves to pass through each other without interfering.
It is useful to use complex notation. The most general monochromatic takes the form

$$\mathbf{E} = \mathbf{E}_0 \exp(i(\mathbf{k}\cdot\mathbf{x} - \omega t)),$$

and

$$\mathbf{B} = \mathbf{B}_0 \exp(i(\mathbf{k}\cdot\mathbf{x} - \omega t)),$$

with $\omega = c^2|\mathbf{k}|^2$.

Definition (Wave vector)**.** $\mathbf{k}$ is the *wave vector*, which is real.

The "actual" solutions are just the real part of these expressions.
There are some restrictions to the values of $\mathbf{E}_0$ etc due to the Maxwell's equations:

$$\nabla\cdot\mathbf{E} = 0 \Rightarrow \mathbf{k}\cdot\mathbf{E}_0 = 0$$
$$\nabla\cdot\mathbf{B} = 0 \Rightarrow \mathbf{k}\cdot\mathbf{B}_0 = 0$$
$$\nabla\times\mathbf{E} = -\frac{\partial\mathbf{B}}{\partial t} \Rightarrow \mathbf{k}\times\mathbf{E}_0 = \omega\mathbf{B}_0$$

If $\mathbf{E}_0$ and $\mathbf{B}_0$ are real, then $\mathbf{k}, \mathbf{E}_0/c$ and $\mathbf{B}_0$ form a right-handed orthogonal triad of vectors.

Definition (Linearly polarized wave)**.** A solution with real $\mathbf{E}_0, \mathbf{B}_0, \mathbf{k}$ is said to be *linearly polarized*.

This says that the waves oscillate up and down in a fixed plane.
If $\mathbf{E}_0$ and $\mathbf{B}_0$ are complex, then the polarization is not in a fixed direction. If we write

$$\mathbf{E}_0 = \boldsymbol{\alpha} + i\boldsymbol{\beta}$$

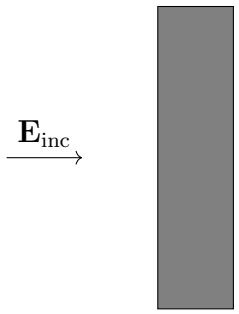
for $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \boldsymbol{r}^3$, then the "real solution" is

$$\mathrm{Re}(\mathbf{E}) = \boldsymbol{\alpha}\cos(\mathbf{k}\cdot\mathbf{x} - \omega t) - \boldsymbol{\beta}\sin(\mathbf{k}\cdot\mathbf{x} - \omega t).$$

Note that $\nabla\cdot\mathbf{E} = 0$ requires that $\mathbf{k}\cdot\boldsymbol{\alpha} = \mathbf{k}\cdot\boldsymbol{\beta}$. It is not difficult to see that this traces out an ellipse.

Definition (Elliptically polarized wave)**.** If $\mathbf{E}_0$ and $\mathbf{B}_0$ are complex, then it is said to be *elliptically polarized*. In the special case where $|\boldsymbol{\alpha}| = |\boldsymbol{\beta}|$ and $\boldsymbol{\alpha}\cdot\boldsymbol{\beta} = 0$, this is *circular polarization*.

We can have a simple application: why metals are shiny.
A metal is a conductor. Suppose the region $x > 0$ is filled with a conductor.



A light wave is incident on the conductor, ie

$$\mathbf{E}_{\text{inc}} = E_0 \hat{\mathbf{y}} \exp(i(kx + \omega t)),$$

with $\omega = ck$.
We know that inside a conductor, $\mathbf{E} = 0$, and at the surface, $\mathbf{E}_{\parallel} = 0$. So $\mathbf{E}_0 \cdot \hat{\mathbf{y}}|_{x=0} = 0$.
Then clearly our solution above does not satisfy the boundary conditions!
To achieve the boundary conditions, we add a reflected wave

$$\mathbf{E}_{\text{ref}} = -E_0 \hat{\mathbf{y}} \exp(i(-kx - \omega t)).$$

Then our total electric field is

$$\mathbf{E} = \mathbf{E}_{\text{inc}} + \mathbf{E}_{\text{ref}}.$$

Then this is a solution to Maxwell's equations since it is a sum of two solutions, and satisfies $\mathbf{E} \cdot \hat{\mathbf{y}}|_{x=0} = 0$ as required.
Maxwell's equations says $\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$. So

$$\mathbf{B}_{\text{inc}} = \frac{E_0}{c} \hat{\mathbf{z}} \exp(i(kx - \omega t))$$
$$\mathbf{B}_{\text{ref}} = \frac{E_0}{c} \hat{\mathbf{z}} \exp(i(-kx - \omega t))$$

This obeys $\mathbf{B} \cdot \hat{\mathbf{n}} = 0$, where $\hat{\mathbf{n}}$ is the normal to the surface. But we also have

$$\mathbf{B} \cdot \hat{\mathbf{z}}|_{x=0^-} = \frac{2E_0}{c} e^{-i\omega t},$$

So there *is* a magnetic field at the surface. However, we know that inside the conductor, we have $\mathbf{B} = 0$. This means that there is a discontinuity across the surface! We know that discontinuity happens when there is a surface current. Using the formula we've previously obtained, we know that the surface current is given by

$$\mathbf{K} = \pm \frac{2E_0}{\mu_0 c} \hat{\mathbf{y}} e^{-i\omega t}.$$

So shining a light onto a metal will cause an oscillating current. We can imagine the process as the incident light hits the conductor, causes an oscillating current, which generates a reflected wave (since accelerating charges generate light — cf. IID Electrodynamics)
We can do the same for light incident at an angle, and prove that the incident angle is equal to the reflected angle.

10.2.5 Poynting vector

Electromagnetic waves carry energy — that's how the Sun heats up the Earth! We will compute how much.
The energy stored in a field in a volume V is

$$U = \int_V \left(\frac{\varepsilon_0}{2} \mathbf{E} \cdot \mathbf{E} + \frac{1}{2\mu_0} \mathbf{B} \cdot \mathbf{B} \right) \, dV.$$

We have

$$\begin{aligned} \frac{dU}{dt} &= \int_V \left(\varepsilon_0 \mathbf{E} \cdot \frac{\partial \mathbf{E}}{\partial t} + \frac{1}{\mu_0} \mathbf{B} \cdot \frac{\partial \mathbf{B}}{\partial t} \right) \, dV \\ &= \int_V \left(\frac{1}{\mu_0} \mathbf{E} \cdot (\nabla \times \mathbf{B}) - \mathbf{E} \cdot \mathbf{J} - \frac{1}{\mu_0} \mathbf{B} \cdot (\nabla \times \mathbf{E}) \right) \, dV. \end{aligned}$$

But

$$\mathbf{E} \cdot (\nabla \times \mathbf{B}) - \mathbf{B} \cdot (\nabla \times \mathbf{E}) = \nabla \cdot (\mathbf{E} \times \mathbf{B}),$$

by vector identities. So

$$\frac{dU}{dt} = - \int_V \mathbf{J} \cdot \mathbf{E} \, dV - \frac{1}{\mu_0} \int_S (\mathbf{E} \times \mathbf{B}) \cdot \, d\mathbf{S}.$$

Recall that the work done on a particle q moving with velocity v is $\delta W = q\mathbf{v} \cdot \mathbf{E} \, \delta t$. So the $\mathbf{J} \cdot \mathbf{E}$ term is the work done on a charged particles in V. We can thus write

Theorem (Poynting theorem)**.**

$$\underbrace{\frac{dU}{dt} + \int_V \mathbf{J} \cdot \mathbf{E} \, dV}_{\text{Total change of energy in } V \text{ (fields + particles)}} = \underbrace{-\frac{1}{\mu_0} \int_S (\mathbf{E} \times \mathbf{B}) \cdot dS}_{\text{Energy that escapes through the surface } S}.$$

10.3 FDTD Solution of Maxwells Equation

The Maxwell equations can be solved either in the time domain or the frequency domain. The numerical methods can be applied either on the PDE formulation of the Maxwell equations or in a boundary integral formulation.
Consider a general first order linear PDE system

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathcal{L}\mathbf{u} = f$$

where u is called a state variable, $\mathcal{L}$ is a linear operator depending on a set of parameters q, and f is a source term.
Examples are,

- $\mathcal{L} = c\frac{\partial}{\partial x}$ yields a wave equation.

- $\mathbf{u} = (H, E)^T$ and
$$\mathcal{L} = \begin{bmatrix} 0 & \frac{1}{\mu}\frac{\partial}{\partial x} \\ -\frac{1}{\epsilon}\frac{\partial}{\partial x} & 0 \end{bmatrix}$$
yields 1D Maxwell's equations in a dielectric, equivalent to the wave equation with speed $c = \sqrt{1/\epsilon\mu}$.

- $\mathbf{u} = (\mathbf{H}, \mathbf{E})^T$ and
$$\mathcal{L} = \begin{bmatrix} 0 & \frac{1}{\mu}\nabla\times \\ -\frac{1}{\epsilon}\nabla\times & 0 \end{bmatrix}$$
yields 3D Maxwell curl equations in a non-dispersive dielectric.

Also the following relations hold:

$$\begin{aligned} \mathbf{D} &= \epsilon\mathbf{E} \\ \mathbf{B} &= \mu\mathbf{H}. \end{aligned}$$

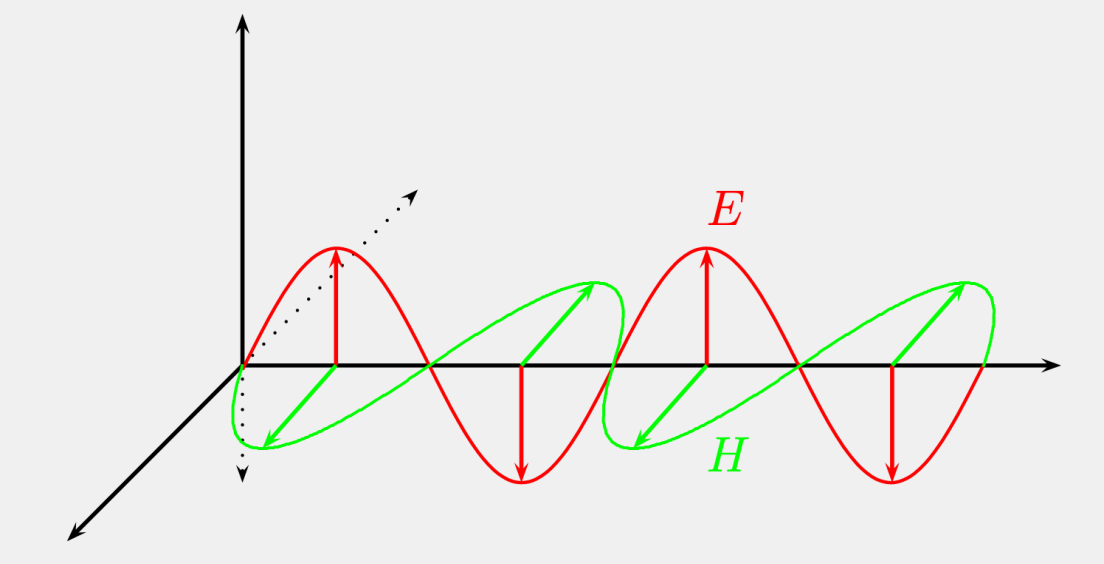
The time evolution of the fields is thus completely specified by the curl equations

$$\begin{aligned} \epsilon\frac{\partial \mathbf{E}}{\partial t} &= \nabla \times \mathbf{H} \\ \mu\frac{\partial \mathbf{H}}{\partial t} &= -\nabla \times \mathbf{E}. \end{aligned}$$

The system above can be combined to a single second order equation for $\mathbf{E}$

$$\epsilon\frac{\partial^2 \mathbf{E}}{\partial t^2} + \nabla \times \frac{1}{\mu}\nabla \times \mathbf{E} = 0.$$

This is often referred to as the curl-curl equation or the vector wave equation.
Assuming that the electric field is polarized to oscillate only in the y direction, propagate in the x direction, and there is uniformity in the z direction:



10.3.1 The Yee Scheme

In 1966 Kane Yee originated a set of finite-difference equations for the time dependent Maxwell's curl equations. The finite difference time domain (FDTD) or Yee algorithm solves for both the electric and magnetic fields in time and space using the coupled Maxwell's curl equations rather than solving for the electric field alone (or the magnetic field alone) with a wave equation. For this he introduced the staggered grid approach. First order derivatives are much more accurately evaluated on staggered grids, such that if a variable is located on the integer grid, its first derivative is best evaluated on the half-grid and vice-versa.

- Maxwell equations have a very special geometric structure. The electric field $\mathbf{E}$ is a vector while the magnetic field $\mathbf{B}$ is a bivector (this is disguised in the usual formulations of Maxwell equations).
- In spacetime formulations the complete electromagnetic field is represented as a single bivector in 4D space-time.
- The fact that we are dealing with two objects of different geometric types indicates that the discrete Maxwell equations should also inherit this somehow.
- The Yee algorithm, often called the finite-difference time-domain algorithm is the most successful (and simple) algorithm that accounts of this geometric structure. It is implemented in most PIC codes, though recent research has focused on structure preserving finite-element and other methods.

10.3.2 The 1D Case

Let's introduce the staggered grids with constant mesh size h such that:

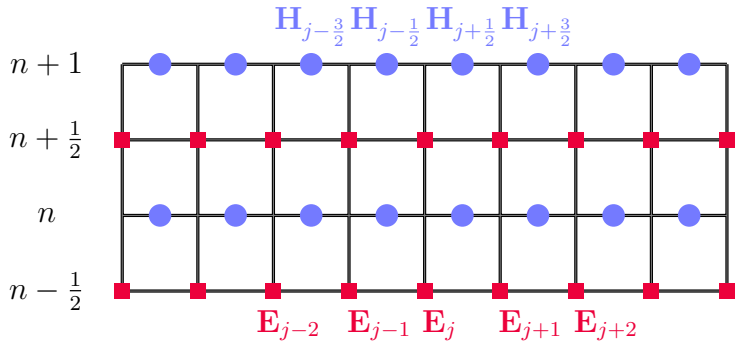
- the primary grid is defined by $G_p = z_j = jh$ with $j \in \mathcal{Z}$.
- the dual grid is defined by $G_d = z_{j+\frac{1}{2}} = (j + \frac{1}{2})h$ with $j \in \mathcal{Z}$.

In the one-dimensional case, we can use only the E_x and H_y components i.e.
Yee's scheme consists in considering E_x and H_y shifted in space by half a cell and in time by half a time step when considering a central difference approximation of the derivatives.

$$\frac{E_j^{n+\frac{1}{2}} - E_j^{n-\frac{1}{2}}}{\Delta t} = -\frac{1}{\epsilon}\frac{H_{j+\frac{1}{2}}^n - H_{j-\frac{1}{2}}^n}{h}$$

$$\frac{H_{j+\frac{1}{2}}^{n+1} - H_{j+\frac{1}{2}}^n}{\Delta t} = -\frac{1}{\mu}\frac{E_{j+1}^{n+\frac{1}{2}} - E_j^{n+\frac{1}{2}}}{h}$$

that represents a plane wave traveling in the z direction.



To avoid computational problems due to the very different amplitudes of E and H, Taflove introduced a normalization of the E field:

$$\tilde{\mathbf{E}} = \sqrt{\frac{\epsilon_0}{\mu_0}}\mathbf{E}.$$

If we now substitute $E \leftarrow \tilde{\mathbf{E}}_x$

$$E_j^{n+\frac{1}{2}} = E_j^{n-\frac{1}{2}} + \frac{1}{\sqrt{\epsilon_0\mu_0}}\frac{\Delta t}{h}\left(H_{j-\frac{1}{2}}^n - H_{j+\frac{1}{2}}^n\right)$$

$$H_{j+\frac{1}{2}}^{n+1} = H_{j+\frac{1}{2}}^n + \frac{1}{\sqrt{\epsilon_0\mu_0}}\frac{\Delta t}{h}\left(E_{j+1}^{n+\frac{1}{2}} - E_j^{n+\frac{1}{2}}\right)$$

This gives an explicit second order accurate scheme in both time and space. It is conditionally stable with the CFL condition

$$\nu = \frac{c\Delta t}{h}$$

with $c = 1/\sqrt{\epsilon\mu}$ and ν is the Courant number.
Once the cell size has been chosen, the time step is also chosen according to stability considerations. For stability reasons, a field component cannot propagate more than one cell size in the time step Δt. It can be proven that, in general, the stability condition is given by

$$\Delta t \leq \frac{h}{\sqrt{d}c},$$

with d the dimension of the problem.

The Absorbing Boundary Condition

Absorbing boundary conditions are necessary to keep outgoing **E** and **H** fields from being reflected back into the problem space. Normally, in calculating the **E** field, we need to know the surrounding **H** values. This is a fundamental assumption of the FDTD method. At the edge of the problem space we will not have the value of one side. However, we have an advantage because we know that the fields at the edge must be propagating outward. We will use this fact to estimate the value at the end by using the value next to it. Suppose we are looking for a boundary condition at the end where $j = 0$. If a wave is going toward

a boundary in free space, it is traveling at c, the speed of light. For simplicity we choose $c\Delta t/h = 0.5$ hence, in one time step of the FDTD algorithm, it travels a distance

$$l = \Delta t \cdot c = c\frac{h}{2c} = \frac{h}{2}.$$

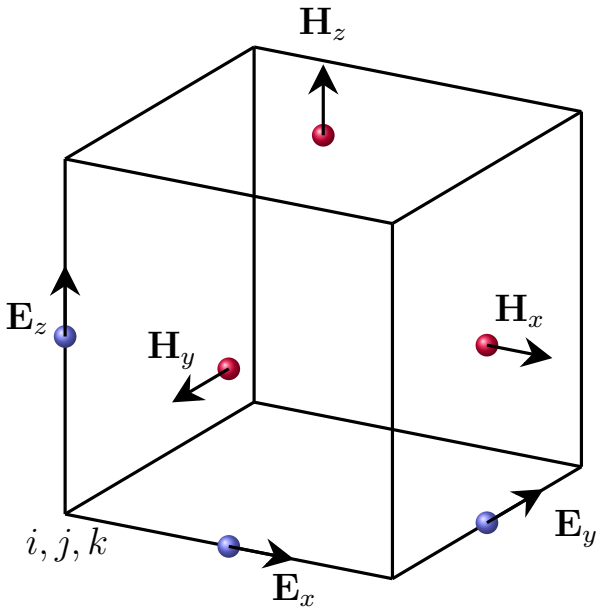
This equation shows that it takes two time steps for the field to cross one cell. So this tells us that an acceptable boundary condition might be

$$E^n_{j=0} = E^{n-2}_{j=1}.$$

10.3.3 The 3D Case

On the Yee-cell the difference approximation to Maxwell equations "falls out", almost like magic. The updates are staggered in time and use two different discrete curl operators, one for the edges and one for the faces.

$$\begin{aligned}
\mathbf{E}^{n+\frac{1}{2}} &= \mathbf{E}^{n-\frac{1}{2}} + \frac{1}{\sqrt{\epsilon_0\mu_0}}\frac{\Delta t}{h}\nabla_F \times \mathbf{B}^n \\
\mathbf{H}^{n+1} &= \mathbf{H}^n + \frac{-1}{\sqrt{\epsilon_0\mu_0}}\frac{\Delta t}{h}\nabla_E \times \mathbf{E}^n
\end{aligned}$$



Here the symbols ∇_E and ∇_F are the discrete curl operators:

- ∇_F takes face-centered magnetic field and computes it curl. This operator puts the result on cell edges.
- ∇_E takes edge-centered electric field and computes it curl. This operator puts the result on cell faces.

- The structure of Yell-cell also indicates that currents must be co-located with the electric field and computed at half time-steps.

This duality neatly reflects the underlying geometry of Maxwell equations. The staggering in time reflects the fact that in 4D the electromagnetic field is a bivector in spacetime.

Divergence relations are exactly maintained

We can show that the discrete Maxwell equations on a Yee-cell maintain the divergence relations exactly:

$$\begin{aligned} \nabla_F \cdot \mathbf{B}^{n+\frac{1}{2}} &= 0 \\ \nabla_E \cdot \mathbf{E}^{n} &= 0 \end{aligned}$$

There is an additional constraint of Maxwell equations in a plasma, that is, the current conservation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{J} = 0$$

where ρ is the charge density and $\mathbf{J}$ the current density. On the Yee-cell we get

$$\frac{\rho^{n+1} - \rho^{n}}{\delta t} + \nabla_E \cdot \mathbf{J}^{\mathbf{n}+\frac{\mathbf{1}}{\mathbf{2}}} = 0.$$

One must ensure that current from particles is computed carefully to ensure that this expression is satisfied.

10.4 Frequency-domain methods

This is from the lecture notes of P. Arbenz, please ignore, the numbering and the literature citations.
In physics, eigenvalues are usually connected to vibrations. Objects like violin strings, drums, bridges, sky scrapers can swing. They do this at certain frequencies. And in some situations they swing so much that they are destroyed. On November 7, 1940 the Tacoma narrows bridge collapsed, less than half a year after its opening. Strong winds excited the bridge so much that the platform in reinforced concrete fell into pieces. A few years ago the London millennium footbridge started wobbling in a way that it had to be closed. The wobbling had been excited by the pedestrians passing the bridge. These are prominent examples of vibrating structures. https://www.youtube.com/watch?v=j-zczJXSxnw
But eigenvalues appear in many other places. Electric Fields in cyclotrons, a special form of particle accelerators, have to vibrate in a precise manner, in order to accelerate the charged particles that circle around its center. The solutions of the Schrödinger equation from quantum physics and quantum chemistry have solutions that correspond to vibrations of the, say, molecule it models. The eigenvalues correspond to energy levels that molecule can occupy. Many characteristic quantities in science are eigenvalues:

- decay factors

- frequencies
- norms of operators or matrices
- singular values
- condition numbers

1.2 Example 1: The vibrating string

1.2.1 Problem setting

Let us consider a string as displayed in Fig. 1.1. The string is clamped at both ends,

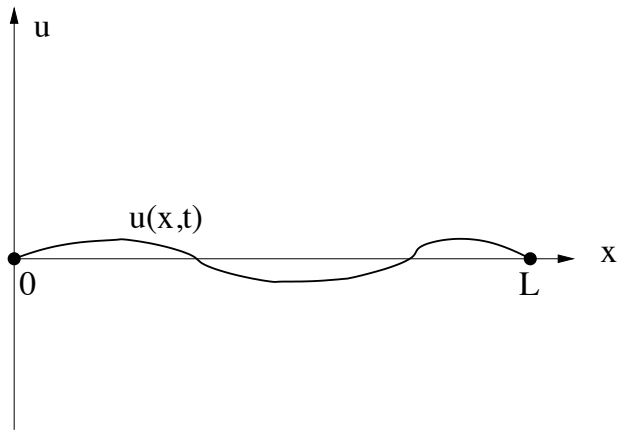


Figure 1.1: A vibrating string clamped at both ends.

at $x = 0$ and $x = L$. The x-axis coincides with the string's equilibrium position. The displacement of the rest position at x, $0 < x < L$, and time t is denoted by $u(x, t)$.

We will assume that the spatial derivatives of u are not very large:

$$\left|\frac{\partial u}{\partial x}\right| \quad \text{is small.}$$

This assumption entails that we may neglect terms of higher order.

Let $v(x, t)$ be the velocity of the string at position x and at time t. Then the kinetic energy of a string section ds of mass $dm = \rho\, ds$ is given by

$$dT = \frac{1}{2} dm\, v^2 = \frac{1}{2}\rho\, ds\, \left(\frac{\partial u}{\partial t}\right)^2. \tag{1.1}$$

From Fig. 1.2 we see that $ds^2 = dx^2 + \left(\frac{\partial u}{\partial x}\right)^2 dx^2$ and thus

$$\frac{ds}{dx} = \sqrt{1 + \left(\frac{\partial u}{\partial x}\right)^2} = 1 + \frac{1}{2}\left(\frac{\partial u}{\partial x}\right)^2 + \text{ higher order terms.}$$

Plugging this into (1.1) and omitting also the second order term (leaving just the number 1) gives

$$dT = \frac{\rho\, dx}{2}\left(\frac{\partial u}{\partial t}\right)^2.$$

The kinetic energy of the whole string is obtained by integrating over its length,

$$T = \int_0^L dT(x) = \frac{1}{2}\int_0^L \rho(x)\left(\frac{\partial u}{\partial t}\right)^2 dx$$

The potential energy of the string has two components

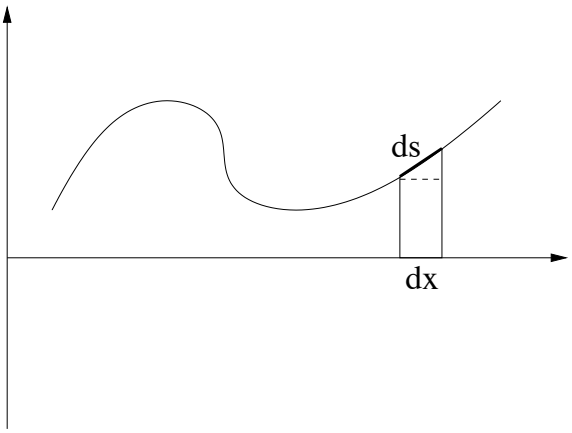


Figure 1.2: A vibrating string, local picture.

1. the stretching times the exerted strain τ.

$$\tau \int_0^L ds - \tau \int_0^L dx = \tau \int_0^L \left(\sqrt{1 + \left(\frac{\partial u}{\partial x} \right)^2} - 1 \right) dx$$
$$= \tau \int_0^L \left(\frac{1}{2} \left(\frac{\partial u}{\partial x} \right)^2 + \text{ higher order terms} \right) dx$$

2. exterior forces of density f

$$- \int_0^L f u dx$$

Summing up, the kinetic energy of the string becomes

$$V = \int_0^L \left(\frac{\tau}{2} \left(\frac{\partial u}{\partial x} \right)^2 - f u \right) dx \tag{1.2}$$

To consider the motion (vibration) of the string in a certain time interval $t_1 \leq t \leq t_2$ we form the integral

$$\begin{aligned} I(u) &= \int_{t_1}^{t_2} (T - V)\, dt \\ &= \frac{1}{2} \int_{t_1}^{t_2} \int_0^L \left[\rho(x) \left(\frac{\partial u}{\partial t} \right)^2 - \tau \left(\frac{\partial u}{\partial x} \right)^2 - f u \right] dx\, dt \end{aligned} \tag{1.3}$$

Here functions $u(x,t)$ are admitted that are differentiable with respect to x and t and satisfy the **boundary conditions (BC)** that correspond to the clamping,

$$u(0,t) = u(L,t) = 0, \qquad t_1 \leq t \leq t_2, \tag{1.4}$$

as well as given **initial conditions** and **end conditions**,

$$\begin{aligned} u(x,t_1) &= u_1(x), \\ u(x,t_2) &= u_2(x), \end{aligned} \qquad 0 < x < L. \tag{1.5}$$

According to the **principle of Hamilton** a mechanical system with kinetic energy T and potential energy V behaves in a time interval $t_1 \le t \le t_2$ for given initial and end positions such that

$$I = \int_{t_1}^{t_2} L\,dt, \qquad L = T - V,$$

is minimized.

Let $u(x,t)$ be such that $I(u) \le I(w)$ *for all* w, that satisfy the initial, end, and boundary conditions. Let $w = u + \varepsilon\, v$ with

$$v(0,t) = v(L,t) = 0, \qquad v(x,t_1) = v(x,t_2) = 0.$$

v is called a *variation*. We now consider $I(u + \varepsilon\, v)$ as a function of ε. Then we have the equivalence

$$I(u) \text{ minimal} \quad \Longleftrightarrow \quad \boxed{\frac{dI}{d\varepsilon}(u) = 0 \text{ for all admitted } v.}$$

Plugging $u + \varepsilon\, v$ into eq. (1.3) we obtain

$$\begin{aligned} I(u+\varepsilon\, v) &= \frac{1}{2}\int_{t_1}^{t_2}\int_{0}^{L}\left[\rho(x)\left(\frac{\partial(u+\varepsilon\, v)}{\partial t}\right)^2 - \tau\left(\frac{\partial(u+\varepsilon\, v)}{\partial x}\right)^2 - 2f(u+\varepsilon\, v)\right]dx\,dt \\ &= I(u) + \varepsilon\int_{t_1}^{t_2}\int_{0}^{L}\left[\rho(x)\frac{\partial u}{\partial t}\frac{\partial v}{\partial t} - \tau\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + 2fv\right]dx\,dt + \mathcal{O}(\varepsilon^2). \end{aligned} \tag{1.6}$$

Thus,

$$\frac{\partial I}{\partial \varepsilon} = \int_{t_1}^{t_2}\int_{0}^{L}\left[-\rho\frac{\partial^2 u}{\partial t^2} + \tau\frac{\partial^2 u}{\partial x^2} + 2\,f\right]v\,dx\,dt = 0$$

for all admissible v. Therefore, the bracketed expression must vanish,

$$-\rho\frac{\partial^2 u}{\partial t^2} + \tau\frac{\partial^2 u}{\partial x^2} + 2\,f = 0. \tag{1.7}$$

This last differential equation is named **Euler-Lagrange equation**.

Next we want to solve a differential equation of the form

$$\boxed{\begin{array}{c} -\rho(x)\dfrac{\partial^2 u}{\partial t^2} + \dfrac{\partial}{\partial x}\left(p(x)\dfrac{\partial u}{\partial x}\right) + q(x)u(x,t) = 0. \\ u(0,t) = u(1,t) = 0 \end{array}} \tag{1.8}$$

which is a generalization of the Euler-Lagrange equation (1.7) Here, $\rho(x)$ plays the role of a mass density, $p(x)$ of a locally varying elasticity module. We do not specify initial and end conditions for the moment.

From physics we know that $\rho(x) > 0$ and $p(x) > 0$ for all x. These properties are of importance also from a mathematical view point! For simplicity, we assume that $\rho(x) = 1$.

1.2.2 The method of separation of variables

For the solution u in (1.8) we make the *ansatz*

$$u(x,t) = v(t)w(x). \tag{1.9}$$

Here, v is a function that depends only on the time t, while w depends only on the spacial variable x. With this ansatz (1.8) becomes

$$v''(t)w(x) - v(t)(p(x)w'(x))' + q(x)v(t)w(x) = 0. \tag{1.10}$$

Now we *separate* the variables depending on t from those depending on x,

$$\frac{v''(t)}{v(t)} = \frac{1}{w(x)}(p(x)w'(x))' + q(x).$$

This equation holds for any t and x. We can vary t and x independently of each other without changing the value on each side of the equation. Therefore, each side of the equation must be equal to a constant value. We denote this value by $-\lambda$. Thus, from the left side we obtain the equation

$$-v''(t) = \lambda v(t). \tag{1.11}$$

This equation has the well-known solution $v(t) = a \cdot \cos(\sqrt{\lambda}t) + b \cdot \sin(\sqrt{\lambda}t)$ where $\lambda > 0$ is assumed. The right side of (1.10) gives a so-called **Sturm-Liouville problem**

$$\boxed{-(p(x)w'(x))' + q(x)w(x) = \lambda w(x), \qquad w(0) = w(1) = 0} \tag{1.12}$$

A value λ for which (1.12) has a *non-trivial* solution w is called an **eigenvalue**; w is a corresponding **eigenfunction**. It is known that all eigenvalues of (1.12) are positive. By means of our ansatz (1.9) we get

$$u(x,t) = w(x)\left[a \cdot \cos(\sqrt{\lambda}t) + b \cdot \sin(\sqrt{\lambda}t)\right]$$

as a solution of (1.8). It is known that (1.12) has infinitely many real positive eigenvalues $0 < \lambda_1 \le \lambda_2 \le \cdots$, $(\lambda_k \underset{k\to\infty}{\longrightarrow} \infty)$. (1.12) has a non-zero solution, say $w_k(x)$ *only* for these particular values λ_k. Therefore, the general solution of (1.8) has the form

$$u(x,t) = \sum_{k=0}^{\infty} w_k(x)\left[a_k \cdot \cos(\sqrt{\lambda_k}\,t) + b_k \cdot \sin(\sqrt{\lambda_k}\,t)\right]. \tag{1.13}$$

The coefficients a_k and b_k are determined by initial and end conditions. We could, e.g., require that

$$u(x,0) = \sum_{k=0}^{\infty} a_k w_k(x) = u_0(x),$$
$$\frac{\partial u}{\partial t}(x,0) = \sum_{k=0}^{\infty} \sqrt{\lambda_k}\, b_k w_k(x) = u_1(x),$$

where u_0 and u_1 are given functions. It is known that the w_k form an orthogonal basis in the space of square integrable functions $L_2(0,1)$. Therefore, it is not difficult to compute the coefficients a_k and b_k.

In concluding, we see that the difficult problem to solve is the eigenvalue problem (1.12). Knowing the eigenvalues and eigenfunctions the general solution of the time-dependent problem (1.8) is easy to form.

Eq. (1.12) can be solved analytically only in very special situation, e.g., if all coefficients are constants. In general a *numerical method* is needed to solve the Sturm-Liouville problem (1.12).

1.3 Numerical methods for solving 1-dimensional problems

In this section we consider three methods to solve the Sturm-Liouville problem.

1.3.1 Finite differences

We approximate $w(x)$ by its values at the discrete points $x_i = ih$, $h = 1/(n+1)$, $i = 1, \ldots, n$.

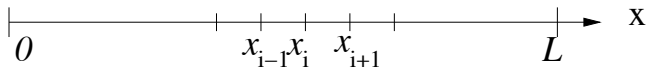


Figure 1.3: Grid points in the interval $(0, L)$.

At point x_i we approximate the derivatives by **finite differences**. We proceed as follows. First we write

$$\frac{d}{dx}g(x_i) \approx \frac{g(x_{i+\frac{1}{2}}) - g(x_{i+\frac{1}{2}})}{h}.$$

For $g = p\frac{dw}{dx}$ we get

$$g(x_{i+\frac{1}{2}}) = p(x_{i+\frac{1}{2}})\frac{w(x_{i+1}) - w(x_i)}{h}$$

and finally, for $i = 1, \ldots, n$,

$$\begin{aligned} -\frac{d}{dx}\left(p\frac{dw}{dx}(x_i)\right) &\approx -\frac{1}{h}\left[p(x_{i+\frac{1}{2}})\frac{w(x_{i+1}) - w(x_i)}{h} - p(x_{i-\frac{1}{2}})\frac{w(x_i) - w(x_{i-1})}{h}\right] \\ &= \frac{1}{h^2}\left[p(x_{i-\frac{1}{2}})w_{i-1} + (p(x_{i-\frac{1}{2}}) + p(x_{i+\frac{1}{2}}))w_i - p(x_{i+\frac{1}{2}})w_{i+1}\right]. \end{aligned}$$

Note that at the interval endpoints $w_0 = w_{n+1} = 0$.

We can collect all equations in a matrix equation,

$$\begin{bmatrix} \frac{p(x_{\frac{1}{2}})+p(x_{\frac{3}{2}})}{h^2} + q(x_1) & -p(x_{\frac{3}{2}}) & & \\ -p(x_{\frac{3}{2}}) & \frac{p(x_{\frac{3}{2}})+p(x_{\frac{5}{2}})}{h^2} + q(x_2) & -p(x_{\frac{5}{2}}) & \\ & -p(x_{\frac{5}{2}}) & \ddots & \ddots \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix} = \lambda \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix}$$

or, briefly,

$$A\mathbf{w} = \lambda\mathbf{w}. \tag{1.14}$$

By construction, A is symmetric and tridiagonal. One can show that it is positive definite as well.

1.3.2 The finite element method

We write (1.12) in the form

Find a twice differentiable function w with $w(0) = w(1) = 0$ such that

$$\int_0^1 \left[-(p(x)w'(x))' + q(x)w(x) - \lambda w(x)\right]\phi(x)dx = 0$$

for all smooth functions ϕ that satisfy $\phi(0) = \phi(1) = 0$.

Contents

Bibliography

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[2] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," 2015.

[3] Linode, "Why you should learn julia." https://www.linode.com/docs/guides/why-learn-julia/, 2020. Accessed: 2021-08-23.

[4] U. D. S. Initiative, "Why does julia work so well?." https://ucidatascienceinitiative.github.io/IntroToJulia/Html/WhyJulia. Accessed: 2021-08-23.

[5] JuliaLang, "Noteworthy differences from other languages." https://docs.julialang.org/en/v1/manual/noteworthy-differences/. Accessed: 2021-08-23.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 1998.

[7] J. E. Gentle, *Random Number Generation and Monte Carlo Methods*. Springer New York, 2003.

[8] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992.

[9] G. Marsaglia, "Random numbers fall mainly in the planes," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 61, no. 1, p. 25, 1968.

[10] J. Cheng and M. J. Druzdzel, "Computational investigation of low-discrepancy sequences in simulation algorithms for bayesian networks," 2013.

[11] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[12] L. Kuipers and H. Niederreiter, *Uniform Distribution of Sequences*. Dover, 2006.

[13] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958.

[14] S. R. Broadbent and J. M. Hammersley, "Percolation processes: I. crystals and mazes," in *Mathematical proceedings of the Cambridge philosophical society*, vol. 53, pp. 629–641, Cambridge University Press, 1957.

[15] D. Stauffer and A. Aharony, *Introduction to percolation theory*. CRC press, 2018.

[16] M. Seyrich and D. Sornette, "Micro-foundation using percolation theory of the finite time singular behavior of the crash hazard rate in a class of rational expectation bubbles," *International Journal of Modern Physics C*, vol. 27, no. 10, p. 1650113, 2016.

[17] S. Galam, "Sociophysics: A review of galam models," *International Journal of Modern Physics C*, vol. 19, no. 03, pp. 409–440, 2008.

[18] A. S. Perelson, "Immune network theory," *Immunological Reviews*, vol. 110, no. 1, pp. 5–36, 1989.

[19] M. Sahini and M. Sahimi, *Applications of percolation theory*. CRC Press, 1994.

[20] G. Grimmett, *Percolation*. Springer, 1999.

[21] B. Bollobás and O. Riordan, *Percolation*. Cambridge University Press, 2006.

[22] L. Böttcher and H. J. Herrmann, *Computational Statistical Physics*. Cambridge University Press, 2021.

[23] M. H. Kalos and P. A. Whitlock, *Monte carlo methods*. John Wiley & Sons, 2009.

[24] J. Hammersley, *Monte carlo methods*. Springer Science & Business Media, 2013.

[25] K. Binder, D. Heermann, L. Roelofs, A. J. Mallinckrodt, and S. McKay, "Monte carlo simulation in statistical physics," *Computers in Physics*, vol. 7, no. 2, pp. 156–157, 1993.

[26] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo method*, vol. 10. John Wiley & Sons, 2016.

[27] S. Asmussen and P. W. Glynn, *Stochastic simulation: algorithms and analysis*, vol. 57. Springer Science & Business Media, 2007.

[28] M. B. Giles, "Multilevel monte carlo path simulation," *Operations research*, vol. 56, no. 3, pp. 607–617, 2008.

[29] M. B. Giles, "Multilevel monte carlo methods," 2013.

[30] M. Giles, "Improved multilevel monte carlo convergence using the milstein scheme," in *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pp. 343–358, Springer, 2008.

[31] S. Heinrich, "Monte carlo complexity of global solution of integral equations," *Journal of Complexity*, vol. 14, no. 2, pp. 151–175, 1998.

[32] S. Heinrich and E. Sindambiwe, "Monte carlo complexity of parametric integration," *Journal of Complexity*, vol. 15, no. 3, pp. 317–341, 1999.

[33] S. Heinrich, "The multilevel method of dependent tests," in *Advances in stochastic simulation methods*, pp. 47–61, Springer, 2000.

[34] S. Heinrich, "Multilevel monte carlo methods," in *International Conference on Large-Scale Scientific Computing*, pp. 58–67, Springer, 2001.

[35] E. Agullo, M. Altenbernd, H. Anzt, L. Bautista-Gomez, T. Benacchio, L. Bonaventura, H.-J. Bungartz, S. Chatterjee, F. M. Ciorba, N. DeBardeleben, D. Drzisga, S. Eibl, C. Engelmann, W. N. Gansterer, L. Giraud, D. Göddeke, M. Heisig, F. Jezequel, N. Kohl, X. S. Li, R. Lion, M. Mehl, P. Mycek, M. Obersteiner, E. S. Quintana-Orti, F. Rizzi, U. Rüde, M. Schulz, F. Fung, R. Speck, L. Stals, K. Teranishi, S. Thibault, D. Thönnes, A. Wagner, and B. Wohlmuth, "Resiliency in numerical algorithm design for extreme scale simulations," *The International Journal of High Performance Computing Applications*, vol. 36, no. 2, pp. 251–285, 2022.

[36] G. Dahlquist and k. Bjrck, *Numerical Methods in Scientific Computing: Volume 1*. USA: Society for Industrial and Applied Mathematics, 2008.

[37] L. D. Landau and E. M. Lifshitz, *Fluid Mechanics, Second Edition: Volume 6 (Course of Theoretical Physics)*. Course of theoretical physics / by L. D. Landau and E. M. Lifshitz, Vol. 6, Butterworth-Heinemann, 2 ed., 1987.

[38] J. C. Strikwerda, *Finite difference schemes and partial differential equations*. SIAM, 2004.

[39] E. Isaacson and H. B. Keller, *Analysis of numerical methods*. Courier Corporation, 2012.

[40] J. Crank and P. Nicolson, "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 43, pp. 50–67, Cambridge University Press, 1947.

[41] J. G. Charney, R. Fjörtoft, and J. Von Neumann, "Numerical integration of the barotropic vorticity equation," in *The Atmosphere—A Challenge*, pp. 267–284, Springer, 1990.