Introduction to Computational Physics

Introduction & Julia

Andreas Adelmann

Paul Scherrer Institut, Villigen
E-mail: andreas.adelmann@psi.ch

https://moodle-app2.let.ethz.ch/course/view.php?id=18025

Plan for today I

1. General Information
   - Useful Addresses and Information
   - Who is the Target Audience of This Lecture?
   - Some Words About Me
   - Outline of the Course
   - What is Computational Physics all about?
   - One Example: Electron dynamics in a Penning Trap
2. The Programming Language Julia
   - Programming Languages / Environments
   - Which programming language should I learn?
   - Why Julia?
   - Technicalities
3. Random Numbers
   - Notation and Definitions
   - Definition of Uniform Random Numbers
   - Congruential RNG (Multiplicative)

Plan for today II

- Lagged Fibonacci RNG (Additive)
  - Implementation
- How Good is a RNG?
- Quasi Random Numbers or Deterministic Sequences
  - Discrepancy and Low-discrepancy Sequences
  - Halton Sequences
- Non-Uniform Distributions
  - Transformation Methods of Special Distributions
  - The Rejection Method
- Creating Random Numbers in Parallel
- Four categories of concurrent and parallel programming in Julia
  - Multi-threading or Thread Parallelism
- Distributed Computing
- Interlude: Performance and scalability
  - Speedup
  - Efficiency
  - Amdahl's law

1.1 Useful Addresses and Information I

- The content of this class including exercises is available online:
  https://moodle-app2.let.ethz.ch/course/view.php?id=18025
- Pdf-files of both the slides and the exercises are also provided on this page
- The script is constantly evolving and will be online, chapter per chapter on Monday before the lecture
- The script and the slides should be synchronized as much as possible (except this lecture ☺)

1.2 Who is the Target Audience of This Lecture? I

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

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

1.3 Some Words About Me I

I am 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 Tuesday's and Friday's in HPK G 28, ETH Hönggerberg, Zürich.
My PSI group web page is located at https://amas.web.psi.ch/.

1.5 Outline of the Course I

- 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 I

- You should have a basic understanding of the UNIX operating system and be able to work with it.
- You should ideally have some knowledge about a higher level programming language such as Python, Fortran, C/C++, Java, ... In particular you should also be able to write, compile and debug programs yourself.
- Requirements in mathematics:
  - You should know the basics of statistical analysis (averaging, distributions, etc.).
  - Furthermore, some knowledge of linear algebra, analysis 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? I

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. Another field of computational physics is the simulation of many-body/particle systems; in this area, a virtual reality is created which is sometimes also referred to as the 3rd branch of physics (between experiments and theory).

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

Another area in which computers are used in physics is the control of experiments. However, this area is not treated in the lecture.

1.7 What is Computational Physics all about? II

Computational physics plays an important role all fields of science, examples: Links to important books and journals are listed in the script

- Computational Fluid Dynamics (CFD): solve and analyze problems that involve fluid flows
- Classical Phase Transition: percolation, critical phenomena
- Condensed Matter Physics (Quantum Mechanics)
- High Energy Physics / Particle Physics: in particular Lattice Quantum Chromodynamics ("Lattice QCD")
- Astrophysics: many-body simulations of stars, galaxies
- Accelerator Physics (see also PAM-1 & 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

1.7 What is Computational Physics all about? III

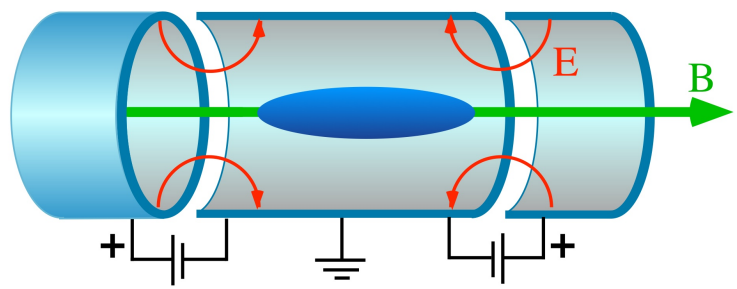
1.8 One Example: Electron dynamics in a Penning Trap I

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} f + \frac{q_e}{m_e} (\mathbf{E} + \mathbf{v} \times B_{ext}) \cdot \nabla_{\mathbf{v}} f = 0,$$

where $\mathbf{E} = \mathbf{E}_{sc} + \mathbf{E}_{ext}$, and the self-consistent fields due to space charge are given by

$$\mathbf{E}_{sc} = -\nabla \phi, \quad -\Delta \phi = \rho = \rho_e - \rho_i.$$

Charge density evolution in a 3D Penning trap. Penning traps are storage devices for charged particles, which uses a quadrupole electric field to confine the particles axially and a homogeneous axial magnetic field to confine the particles in the radial direction.



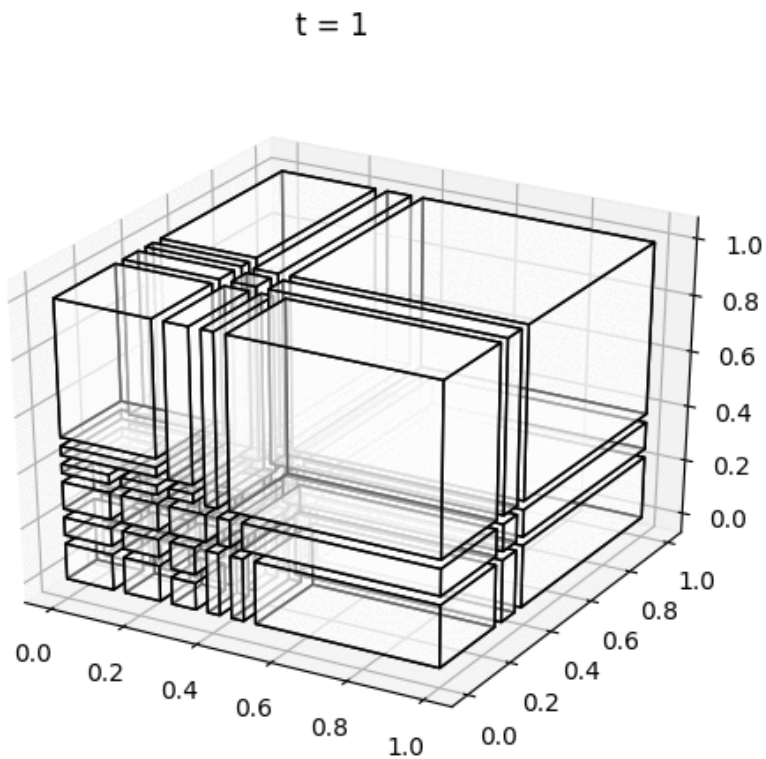
Source: Wikipedia.

1.8 One Example: Electron dynamics in a Penning Trap II

Courtesy of Dr. Muralikrishnan Sriramkrishnan (Marie Curie Fellow PSI).

1.8 One Example: Electron dynamics in a Penning Trap III

- Physics Modelling (classic, spin, QM)
- Numerical Methods (PIC)
- High Performance Computing (load balancing)



2.0 The Programming Language Julia

- Getting started ++ https://julialang.org/learning/
- https://live.juliacon.org/
- Explore Talks: https://live.juliacon.org/viz

2.1 Programming Languages / Environments I

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.

- **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.
- **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.

2.1 Programming Languages / Environments II

- **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).
- **FORTRAN (FORmula TRANslator)** was the first scientific programming languages. 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.

2.1 Programming Languages / Environments III

- **Other procedural languages: C, Pascal, Modula,...** 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.
- **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.
- **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.

2.1 Programming Languages / Environments IV

- **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 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.

2.2 Which programming language should I learn? I

We recommend Julia for various reasons:

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

2.3 Why Julia? I

www.linode.com/docs/development/julia/why-learn-julia/#what-is-julia

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 and is still under development hence, the ecosystem is much less mature.
- 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.

2.3 Why Julia? II

www.linode.com/docs/development/julia/why-learn-julia/#what-is-julia

Compiling

- Julia is a compiled language (JIT Just In Time compilation) that's one of the reasons that it performs faster than interpreted languages.
- Julia is not strictly statically typed. It uses JIT 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, but that can achieve comparable speeds to compiled languages like C or Fortran.

2.3 Why Julia? III

www.linode.com/docs/development/julia/why-learn-julia/#what-is-julia

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)
```

2.3 Why Julia? IV

www.linode.com/docs/development/julia/why-learn-julia/#what-is-julia

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.

2.4 Technicalities I

ucidatascienceinitiative.github.io/IntroToJulia/Html/WhyJulia

Core Idea: Multiple Dispatch + Type Stability → Speed + Readability

- with JIT compilation every statement is run using compiled functions
- but this is not the main reason for speed.
- 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.

2.4 Technicalities II

ucidatascienceinitiative.github.io/IntroToJulia/Html/WhyJulia

Multiple Dispatch

- Type stability is the idea that there is only **1 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 spit out 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 spit out floats.

Julia provides code introspection macros (`@code_llvm 2*5`) so that way you can see what your code actually compiles to.

2.4 Technicalities III

ucidatascienceinitiative.github.io/IntroToJulia/Html/WhyJulia

Type Stability

- 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.
- $\rightarrow$ compile the function with the full **amount of optimizations**.
- multiple-dispatch works into this story because it means that `*` can be a type-stable function. Why? 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`.
- propagate the type information $\rightarrow$ knowing all of the types along the way $\rightarrow$ full optimiziations.

Introduction to Computational Physics

Random Numbers

Andreas Adelmann

Paul Scherrer Institut, Villigen
E-mail: andreas.adelmann@psi.ch

https://moodle-app2.let.ethz.ch/course/view.php?id=18025

Plan for today I

1. General Information
   - Useful Addresses and Information
   - Who is the Target Audience of This Lecture?
   - Some Words About Me
   - Outline of the Course
   - What is Computational Physics all about?
   - One Example: Electron dynamics in a Penning Trap
2. The Programming Language Julia
   - Programming Languages / Environments
   - Which programming language should I learn?
   - Why Julia?
   - Technicalities
3. Random Numbers
   - Notation and Definitions
   - Definition of Uniform Random Numbers
   - Congruential RNG (Multiplicative)

Plan for today II

- Lagged Fibonacci RNG (Additive)
  - Implementation
- How Good is a RNG?
- Quasi Random Numbers or Deterministic Sequences
  - Discrepancy and Low-discrepancy Sequences
  - Halton Sequences
- Non-Uniform Distributions
  - Transformation Methods of Special Distributions
  - The Rejection Method
- Creating Random Numbers in Parallel
- Four categories of concurrent and parallel programming in Julia
  - Multi-threading or Thread Parallelism
- Distributed Computing
- Interlude: Performance and scalability
  - Speedup
  - Efficiency
  - Amdahl's law

3 Random Numbers I

Random numbers (RNs) are an important tool for scientific simulations.

- Simulate random events 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 i.e. uncertanty quantification
- Base of all Monte Carlo (MC) methods

3.0.1 Notation and Definitions I

$\mathbb{N} = 0, 1, 2, \ldots, \mathbb{N}^+ = 1, 2, 3, \ldots$
$\mathbb{Z} = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ 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 Definition of Uniform Random Numbers I

Definition

the probability that a given number occurs next in the sequence is always the same

Physical systems can produce random events, for example in electronic circuits ("electronic flicker noise") or in systems where quantum effects play an important role (such as for example radioactive decay or the photon emission from a semiconductor). However, physical random numbers can have **correlations** and they are **not reproducible**
The algorithmic creation of random numbers is problematic:

- computer is completely deterministic $\rightarrow$ sequence should be non-deterministic
- one therefore considers the creation of pseudo-random numbers
- deterministic algorithm, but in such a way that the numbers are almost homogeneously, randomly distributed and reproducible

3.1 Definition of Uniform Random Numbers II

These numbers should follow a well-defined distribution and should have long periods. Furthermore, they should be calculated quickly.

> A very important tool in the creation of pseudo-random numbers is the modulo-operator **mod** (in Julia & C++ **%**), which determines the remainder of a division of one integer number with another one.

Given two numbers a (dividend) and n (divisor), we write a **modulo** n or a **mod** n which stands for the remainder of division of a by n. More precise: we consider a number $q \in \mathbb{Z}$, and the two integers a and n we then get

$$a = nq + r$$

with $0 \leq r < |n|$, where r is the the remainder. The **mod**-operator is useful because one obtains both big and small numbers when starting with a big number.

3.1 Definition of Uniform Random Numbers III

The pseudo-random number generators (RNG) can be divided into two classes: the multiplicative and the additive generators.

- The **multiplicative** ones are simpler and faster to program and execute, but do not produce very good sequences.
- The **additive** ones are more difficult to implement and take longer to run, but produce much better random sequences.

3.2 Congruential RNG (Multiplicative) I

The algorithm (Lehmer in 1948) is based on the properties of the **mod** -operator.

Congruential RNG (Multiplicative)

Let us assume that we choose two integer numbers c and p and a seed value x_0 with $c, p, x_0 \in \mathbb{Z}$. We then create the sequence $x_i \in \mathbb{Z}$, $i \in \mathbb{N}$ iteratively by

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

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

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

3.2 Congruential RNG (Multiplicative) II

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

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

In 1910, R. D. Carmichael proved that the maximal period can be obtained if p is a Mersenne prime number and if the number is at the same time the smallest integer number for which the following condition holds:

$$c^{p-1} \,\mathbf{mod}\, p = 1.$$

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

3.2 Congruential RNG (Multiplicative) III

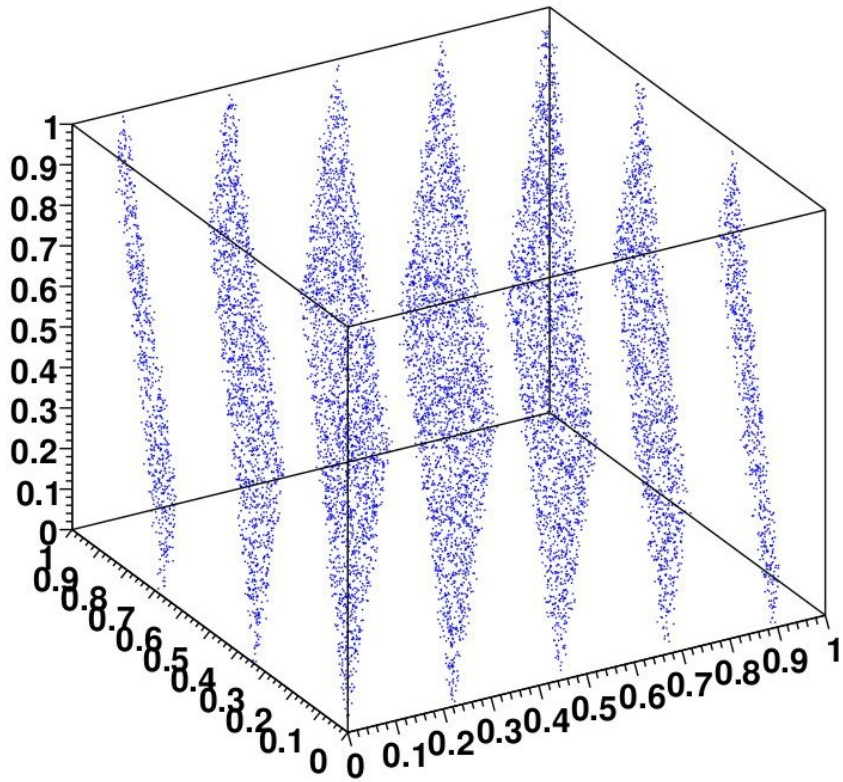
```
const  p=2147483647
const  c=16807
int rnd=42
rnd q = (c * rnd) % p
```

The number p is of course a Mersenne prime with the maximal length of an integer (32 bit): $2^{31} - 1$. Side note: as of today, we know 51 Mersenne prime numbers, the largest one is $2^{82,589,933} - 1$ (24 862 048 digits). Conjectured are of course ∞ number of Mersenne prime numbers exists.

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. It is of course also possible to do this kind of visualization for three consecutive numbers (x_i, x_{i+1}, x_{i+2}) in 3D. There is

3.2 Congruential RNG (Multiplicative) IV

even a theorem which quantifies the patterns observed. Let us call the normalized numbers of the pseudo-random sequence $\{z_i\} = \{x_i\}/p$, $i \in \mathbb{N}$. Let then $\pi_1 = (z_1, ..., z_n)$, $\pi_2 = (z_2, ..., z_{n+1})$, $\pi_3 = (z_3, ..., z_{n+2})$, ... be the points of the unit n-cube formed from n successive z_i.



3.3 Lagged Fibonacci RNG (Additive) I

A more complicated version of a RNG is the Lagged Fibonacci algorithm proposed by Tausworth in 1965. Lagged Fibonacci type generators permit extremely large periods and even allow for advantageous predictions about correlations.

Lagged Fibonacci generators often use integer values, but in the following we shall focus on binary values.

Consider a sequence of of binary numbers $x_i \in \mathbb{B}$, $1 \leq i \leq b$. The next bit in our sequence, x_{b+1} is then given by

$$x_{b+1} = (\sum_{j \in \mathcal{J}} x_{b+1-j}) \, \mathbf{mod} \, 2$$

with $\mathcal{J} \subset [1, .., b]$. In other words, the sum includes only a subset of all the other bits, so the new bit could for instance simply be based on the

3.3 Lagged Fibonacci RNG (Additive) II

first and third bit, $x_{b+1} = (x_1 + x_3) \mathbf{mod}\, 2$ (or of course any other subset!).

Consider two natural numbers $c, d \in \mathbb{N}$ with $d \leq c$, and we define our sequence recursively as

Let $c, d \in \{1, \ldots, i-1\}$ with $i \in \mathbb{N}$ and $d \leq c$, and we define our sequence recursively as

$$x_{i+1} = (x_{i-c} + x_{i-d}) \mathbf{mod}\, 2$$

Initial conditions:

- initial seed sequence of at least c bits to start from
- one usually uses a congruential generator to obtain the seed sequence

3.3 Lagged Fibonacci RNG (Additive) III

Much as in the case of congruential generators, there are conditions for the choice of the numbers c and d. In this case, c and d must satisfy the Zierler-Trinomial condition which states that

$$T_{c,d}(z) = 1 + z^c + z^d$$

cannot be factorized in subpolynomials, where z is a binary number. The number c is chosen up to $100,000$ and it can be shown that the maximal period is $2^c - 1$, which is much larger than for congruential generators. The smallest numbers satisfying the Zierler conditions are $(c, d) = (250, 103)$. The generator is named after the discoverers of the

3.3 Lagged Fibonacci RNG (Additive) IV

numbers, Kirkpatrick and Stoll (1981). The following pairs (c, d) are known:

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

3.3.1 Implementation I

There are two methods to convert the obtained binary sequences to natural numbers (e.g. 32 bit unsigned variables):

- One runs 32 Fibonacci generators in parallel (this can be done very efficiently). The problem with this method is the initialization, as the 32 initial sequences 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.
- One extracts a 32 bit long part from the sequence. This method is relatively slow, as for each random number one needs to generate 32 new elements in the binary sequence. Furthermore, it has been shown that random numbers produced in this way show strong correlations.

3.4 How Good is a RNG? I

There are many possibilities to test how random a sequence generated by a given RNG really is. There is an impressive collection of possible tests for a given sequence $\{s_i\}$, $i \in \mathbb{N}$, for instance:

1. Square test: the plot of two consecutive numbers $(s_i, s_{i+1})\,\forall i$ should be distributed homogeneously. Any sign of lines or clustering shows the non-randomness and correlation of the sequence $\{s_i\}$.
2. Cube test: this test is similar to the square test, but this time the plot is three-dimensional with the tuples (s_i, s_{i+1}, s_{i+2}). Again the tuples should be distributed homogeneously.
3. Fluctuation of the mean value (χ^2-test): the distribution around the mean value should behave like a Gaussian distribution.

3.4 How Good is a RNG? II

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

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

   So the more numbers are averaged, the better $\frac{1}{2}$ will be approximated.

5. Spectral analysis (Fourier analysis): If we assume that the $\{s_i\}$ are 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).

3.4 How Good is a RNG? III

6. Serial correlation test: R_k = Autocovariance at lag k

$$R_k = \frac{1}{n-k}\sum_{i=1}^{n}(x_i - \frac{1}{2})(x_{i+k} - \frac{1}{2}).$$

For a good RNG, R_k should be around 0. For large n, R_k is normally distributed with a mean of zero and a variance of $1/[144(n-k)]$. For this test you can easy get the $100(1-\alpha)\%$ confidence interval for R_k as

$$R_k \pm z_{1-\frac{1}{\alpha}}/(12\sqrt{n-k}).$$

7. ...

Very famous are Marsaglia's "Diehard" tests for random numbers. These Diehard tests are a battery of statistical tests for measuring the quality of a set of random numbers. They were developed over many years and published for the first time by Marsaglia on a CD-ROM with random numbers in 1995 https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/.

3.5 Quasi Random Numbers or Deterministic Sequences I

Let N be the number of samples in a Monte Carlo (MC) method:

- the expected error of MC sampling is

$$\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$$

- the error is fairly independent of the problem dimension
- random point sets generated by MC 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.

3.5 Quasi Random Numbers or Deterministic Sequences II

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

$$\mathcal{O}((logN)^d \times N^{-1})$$

where d is the problem dimension and N is again the number of samples generated.

$\Rightarrow$ when the number of samples is large enough, quasi MC methods are theoretically superior to MC sampling

$\Rightarrow$ in quasi MC methods their error bounds are deterministic

3.5.1 Discrepancy and Low-discrepancy Sequences I

Discrepancy is a measure of non-uniformity of a sequence of points placed in a unary hypercube $[0,1]^d$. The most widely studied distance measure is the D-star discrepancy:

$$D_N^*(x_1,\ldots,x_n) = \sup_{0\leq v_j\leq 1, j=1,\ldots,d} \left| \frac{1}{N}\sum_{i=1}^{N}\prod_{j=1}^{d} 1_{0\leq x_j^i\leq v_j} - \prod_{j=1}^{d} v_j \right|$$

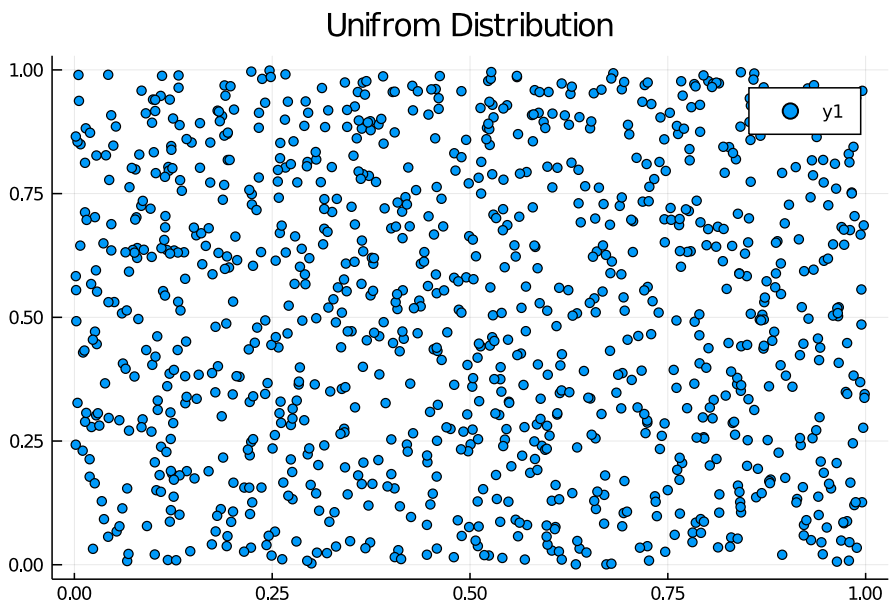
For every subset E of $[0,1]^d$ of the form $[0,v_1)\times\ldots\times[0,v_d)$, we divide the number of points $x_k \in E$ by N and take the absolute difference between this quotient and the volume of E.
A sequence of points $(x_1,\ldots,x_n)\in[0,1]^d$ is a low-discrepancy sequence if for any $N>1$

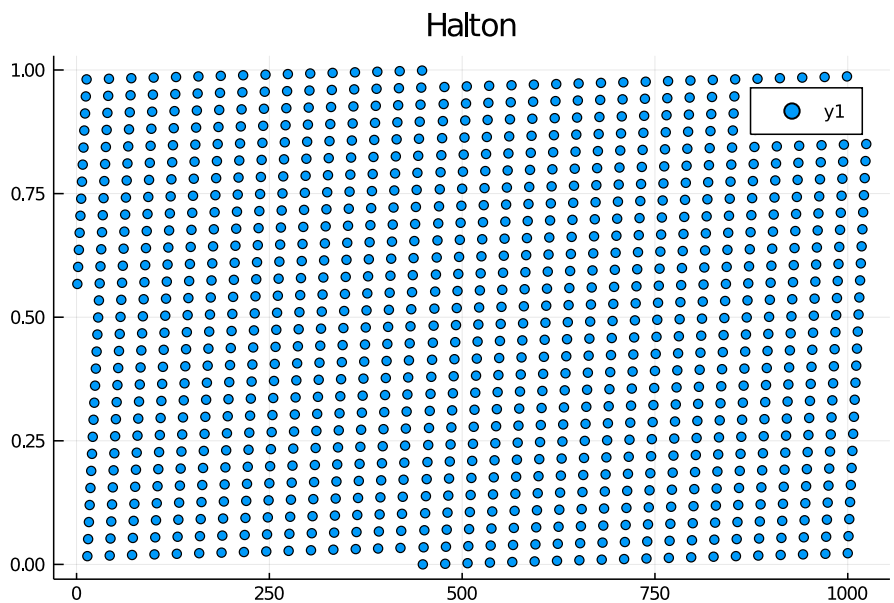
$$D_N^*(x_1,\ldots,x_n) \leq c(d)\cdot\frac{(\log N)^d}{N}$$

holds, 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

3.5.1 Discrepancy and Low-discrepancy Sequences II

points within any subset $E \in [0,1]^d$ of the form $[0, v_1) \times ... \times [0, v_d)$ 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. In the illustration below a two-dimensional projection of a random sequence and of a low-discrepancy sequence is shown.





3.5.1 Discrepancy and Low-discrepancy Sequences III

Lets assume, we want to estimate

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

With Monte Carlo sampling, we first generate a random sequence of independent vectors $x_1, x_2, \ldots, x_N$ from the uniform distribution in $[0, 1]^d$, then use

$$\frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

as the estimator of I. The error bound for Monte Carlo sampling is probabilistic with order $\mathcal{O}(N^{-1/2})$.
In quasi-Monte Carlo methods, we use a low-discrepancy sequence $x_1, x_2, \ldots, x_N$ instead of a random sequence to estimate I. The integration

3.5.1 Discrepancy and Low-discrepancy Sequences IV

accuracy for quasi-Monte Carlo methods relates to D^* discrepancy by the Koksma-Hlawka inequality (see Niederreiter, 1992b)

$$\left| \int_{[0,1]^d} f(x)dx - \frac{1}{N}\sum_{i=1}^{N} f(x_i) \right| \leq V(f)D_N^*(x_1, x_2, \ldots, x_N) \leq V(f)c(d)\frac{(\log N)^d}{N}$$

With $V(f) < \infty$ is the variation of f in the sense of Hardy and Krause (see Niederreiter, 1992b). It is easy to see that with an increase in N quasi-Monte Carlo methods may offer better convergence rates than Monte Carlo sampling. Another advantage of quasi Monte Carlo methods is that we obtain deterministic error bounds $\mathcal{O}(\frac{(\log N)^d}{N})$.

3.5.2 Halton Sequences I

Let $\pi_1, \ldots, \pi_d$ the first d prime numbers. The Halton d-dimensional sequence is defined as

$$x_n = (\Phi_{\pi_1}(n), \ldots, \Phi_{\pi_j}(n), \ldots, \ldots, \Phi_{\pi_d}(n)),$$

where $\Phi_{\pi_j}(n)$ is the jth radical inverse function

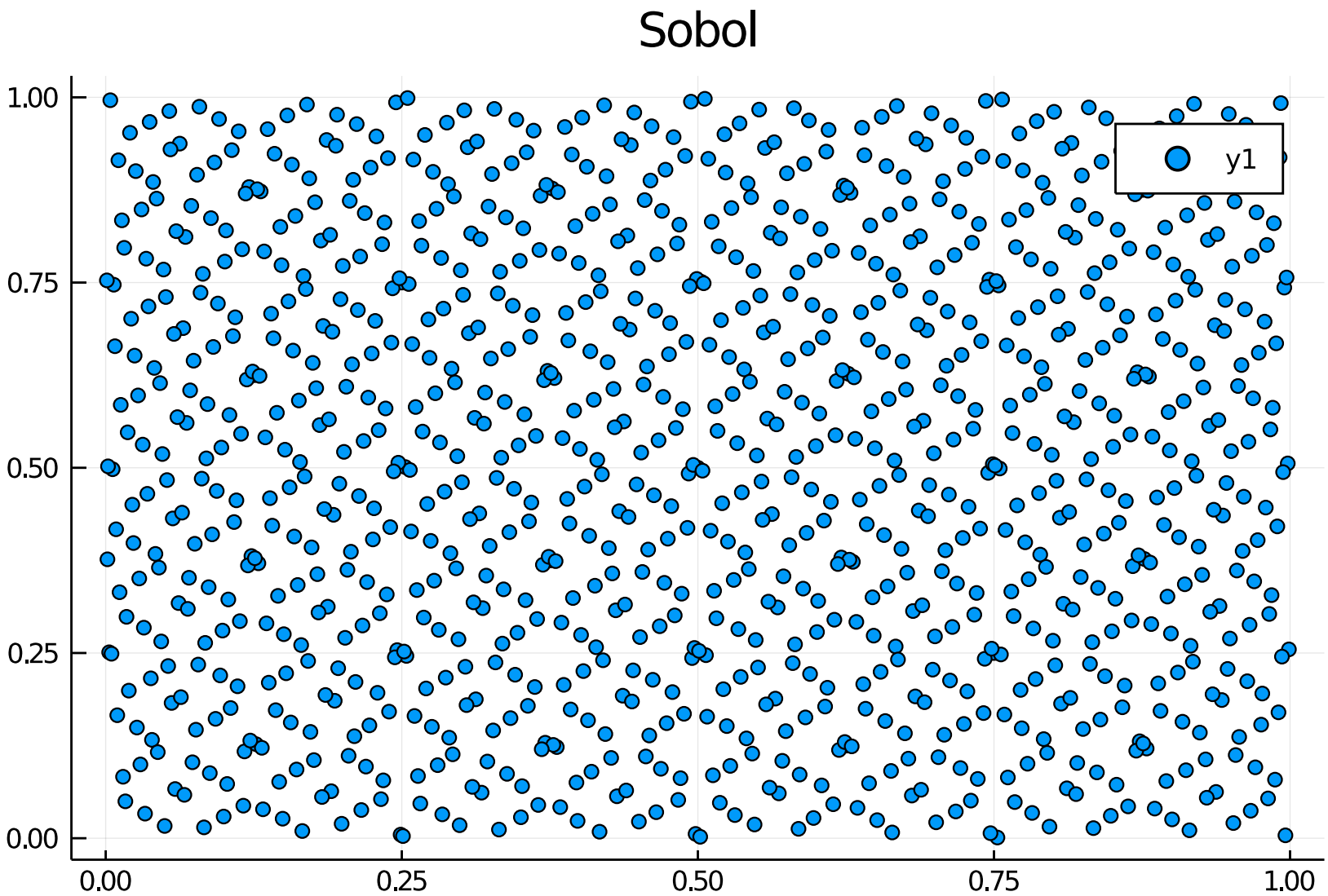
$$\Phi_{\pi_j}(n) = \sum_{i=0}^{l(j)} \frac{a_i(j,n)}{\pi_j^{-i}}.$$

This sum is finite with the integer coefficients $a_i(j,n) \in [0, \pi_j - 1]$ coming from the digit expansion of the integer n in base π_j, i.e.

$$n = \sum_{i=0}^{l(j)} a_i(j,n)\pi_i^j, \quad l(j) = \lceil \log_{\pi_j} n \rceil.$$

Many more quasi random sequences exists, for example the Solol' sequences on the next page.

3.5.2 Halton Sequences II



3.6 Non-Uniform Distributions I

- so far we only considered the uniform distribution of pseudo-random numbers
- the congruential and lagged Fibonacci RNG produce numbers in $\mathbb{N}$ which can easily be mapped to the interval $[0, 1[$ or any other interval by simple shifts and multiplications
- however, if the goal is to produce random numbers which are distributed according to a certain distribution (e.g. Gaussian), the algorithms presented so far are not very well suited.

There are essentially two different ways to perform this transformation:

- if we are looking at a distribution whose analytic description is known, it may be possible to apply a mapping
- however, if the analytic description is unknown (or the transformation cannot be applied), we have to use the so-called rejection method.

3.6.1 Transformation Methods of Special Distributions I

- for a certain class of distributions it is possible to create pseudo-random numbers from uniformly distributed random numbers by finding a mathematical transformation
- the transformation method works particularly nicely for the most common distributions (exponential, Poisson and normal distribution)
- it is not always feasible - this depends on the analytical description of the distribution.

The idea is to find the equivalence between area slices of the uniform distribution P_u and the distribution of interest. The uniform distribution is written as

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

3.6.1 Transformation Methods of Special Distributions II

Let us now consider the distribution $P(y)$. If we compare the areas of intergration, we find

$$z = \int_0^y P(y')dy' = \int_0^z P_u(z')dz' \tag{1}$$

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

$$y = I_P^{-1}(z)$$

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

1. $I_P(y) = \int_0^y P(y')dy'$ can be solved analytically in a closed form
2. there exists an analytic inverse of $z = I_P(y)$ such that $y = I_P^{-1}(z)$

3.6.1 Transformation Methods of Special Distributions III

Of course, these conditions can be overcome to a certain extent by precalculating/tabulating and inverting $I_P(y)$ numerically, if the integral is well-behaved (i.e. is non-singular). Then, with a little help from precalculated tables, it is possible to transform the uniform numbers numerically.

We are now going to demonstrate this method for the two most commonly used distributions: the Exponential distribution and the Gaussian distribution. We are already going to see in the case of the Gaussian distribution that quite a bit of work is required to create such a transformation.

The Exponential Distribution

The Exponential distribution is defined as

$$P(y) = ke^{-yk}.$$

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

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

thus

$$z = -e^{-y'k}|_0^y = 1 - e^{-yk}.$$

Solving for y yields

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

The Gaussian Distribution I

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

The Gaussian or normal distribution is written as

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

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

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

However, Box and Muller[1] have introduced the following elegant trick to circumvent this restriction. Let us assume we take two (uncorrelated) uniform random variables z_1 and z_2. Of course we can apply the area

The Gaussian Distribution II

equality of eq. (**??**) again but this time we write it as a product of the two random variables

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

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

$$\begin{aligned} r^2 &= y_1^2 + y_2^2 \\ \tan\phi &= \frac{y_1}{y_2} \end{aligned}$$

with

$$dy_1' dy_2' = r' dr' d\phi'$$

The Gaussian Distribution III

Substituting in eq. (**??**) leads to

$$
\begin{aligned}
z_1 \cdot z_2 &= \frac{1}{\pi\sigma} \int_0^{\phi} \int_0^{r} e^{-\frac{r'^2}{\sigma}} r' \, dr' \, d\phi' \\
&= \frac{\phi}{\pi\sigma} \int_0^{r} e^{-\frac{r'^2}{\sigma}} r' \, dr' \\
&= \frac{\phi}{\pi\sigma} \cdot \frac{\sigma}{2} \left(1 - e^{-\frac{r^2}{\sigma}} \right) \\
z_1 \cdot z_2 &= \underbrace{\frac{1}{2\pi} \arctan\left(\frac{y_1}{y_2} \right)}_{\equiv z_1} \cdot \underbrace{\left(1 - e^{-\frac{y_1^2 + y_2^2}{\sigma}} \right)}_{\equiv z_2}
\end{aligned}
$$

The Gaussian Distribution IV

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

$$\begin{aligned} y_1^2 + y_2^2 &= -\sigma \ln(1 - z_2) \\ \frac{y_1}{y_2} &= \tan(2\pi z_1) = \frac{\sin(2\pi z_1)}{\cos(2\pi z_1)} \end{aligned}$$

Solving these two coupled equations finally yields

$$\begin{aligned} y_1 &= \sqrt{-\sigma \ln(1 - z_2)} \sin(2\pi z_1) \\ y_2 &= \sqrt{-\sigma \ln(1 - z_2)} \cos(2\pi z_1) \end{aligned}$$

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

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

3.6.2 The Rejection Method I

As we have seen previously, there are two conditions that have to be satisfied in order to apply the transformation method:

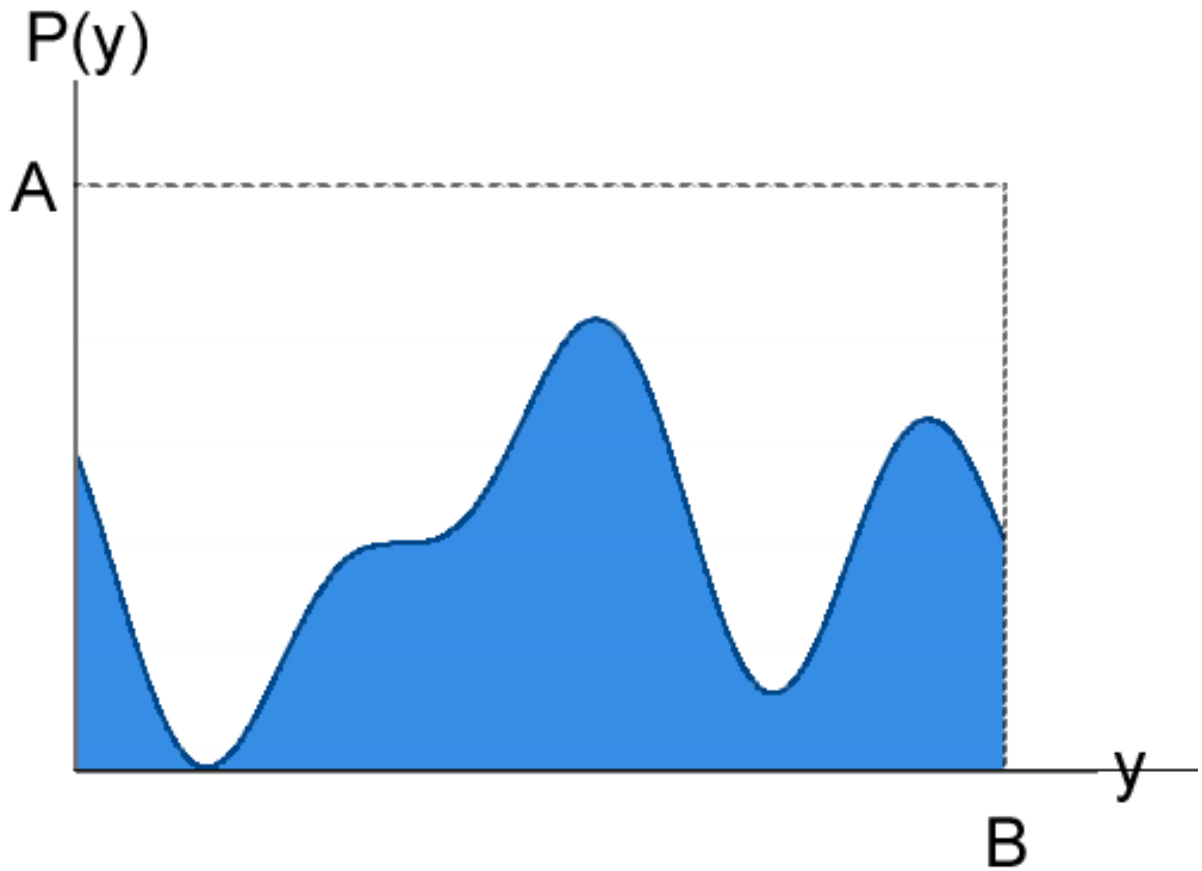
1. integrability
2. invertibility

If either of these conditions is not satisfied, there exists no analytical method to obtain random numbers in this 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 to obtain arbitrarily distributed random numbers, which is called the rejection method.

Let $P(y)$ be the distribution of which we would like to obtain random numbers. A necessary condition for the rejection method to work is that $P(y)$ is well-behaved, in this case "well-behaved" means that $P(y)$ is finite over the domain of interest $P(y) < \infty$ for

3.6.2 The Rejection Method II

$y \in [0, B]$, with $A, B \in \mathbb{R}$ and $A, B < \infty$. We then define an upper bound to be the box with edge length B and A.



3.6.2 The Rejection Method III

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

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

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

3.6.2 The Rejection Method IV

- While the method is sound, in practice it is often faster to invert $P(y)$ numerically as mentioned already in subsection **??**.
- There is a method to make the rejection method faster (but also more complicated): We use N boxes to cover $P(y)$ and define the individual box with side length A_i and $b_i = B_{i+1} - B_i$ for $I \leq i \leq N$. Then, the approximation of $P(y)$ is much better (this is related to the idea of the Riemann-integral)

3.7 Creating Random Numbers in Parallel I

Motivation

- need to create initial conditions $x_i \in \mathbb{R}^6$
- $i = 1 \ldots 10^9$ or more
- the computational domain is most likely distributed among P processors/cores

Difficulty

- time complexity
- space complexity

Parallelize the problem

- Thread Parallelism
- Distributed Computing

3.8 Four categories of concurrent and parallel programming in Julia I

- Asynchronous "tasks", or coroutines: Julia Tasks allow suspending and resuming computations for I/O, event handling, producer-consumer processes etc. Tasks can synchronize through operations like wait and fetch, and communicate via Channels. While strictly not parallel computing by themselves, Julia lets you schedule Tasks on several threads.
- Multi-threading: Julia's multi-threading provides the ability to schedule Tasks simultaneously on more than one thread or CPU core, sharing memory. This is usually the easiest way to get parallelism on one's PC or on a single large multi-core server. Julia's multi-threading is composable. When one multi-threaded function calls another multi-threaded function, Julia will schedule all the threads globally on available resources, without oversubscribing.

3.8 Four categories of concurrent and parallel programming in Julia II

- Distributed computing: Distributed computing runs multiple Julia processes with separate memory spaces. These can be on the same computer or multiple computers. The **Distributed** standard library provides the capability for remote execution of a Julia function. With this basic building block, it is possible to build many different kinds of distributed computing abstractions. Packages like **DistributedArrays.jl** are an example of such an abstraction. On the other hand, packages like **MPI.jl** and **Elemental.jl** provide access to the existing MPI ecosystem of libraries.
- GPU computing: The Julia GPU compiler provides the ability to run Julia code natively on GPUs. There is a rich ecosystem of Julia packages that target GPUs.

3.8.1 Multi-threading or Thread Parallelism I

https://julialang.org/blog/2019/07/multithreading/

```
$ export JULIA_NUM_THREADS=4;  jupyter notebook
```

```
Threads.nthreads()
4
```

3.8.1 Multi-threading or Thread Parallelism II

https://julialang.org/blog/2019/07/multithreading/

```
function parallelTask()
   thid = Threads.threadid()
   println("Remote computation with thread $thid")
   start = thid;
   for i in 1:10000000
      start += rand()
   end
   return start
end
N = 10
M = zeros(N,N)
@time for t in 1:10
    @sync for i in 1:N
        for j in 1:N
            Threads.@spawn begin
                M[i, j] = parallelTask()
            end
        end
    end
```

3.9 Distributed Computing I

```
using Distributed
addprocs(length(Sys.cpu_info()))
```

- use `@everywhere` macro to call the expressions following to be called on the allocated resources
- make (`@everywhere`) sure that the packages needed to execute my process are used or imported into each of the processors

3.10 Interlude: Performance and scalability I

Measuring the run time of a sequential algorithm is simple. The execution time depends on the volume of the input data.
Here, we simply count the number of floating point operations (**flop**).

$$\boxed{T_{\text{seq}} = (\text{number of flops}) \times t_{\text{flop}}}$$

t_{flop} is the time in seconds per flop.
Example: dot product of two n-vectors:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i \, y_i \Longrightarrow T_{\text{seq}} = (2n - 1) \, t_{\text{flop}}$$

Remark: t_{flop} is variable in general.
An other and more complicated measure is the number of memory accesses.

3.10 Interlude: Performance and scalability II

What does execution time mean on a parallel processor? On a distributed arrangement of processors there is no common or synchronized clock. (Maybe different processors even run in different time zones.)

- One may choose the maximal execution time of the program on the processors involved in the computation.
- If we actually measure time in parallel programs we do this on one processor (P_0).
- For the theoretical considerations of this section, we simply assume that all processors (say p) start in the same moment. The execution time $T(p)$ then is the period of time from this moment until the moment when the last of the p processors finishes its computation.
- $T(1)$ is the execution time of the best sequential algorithm.

3.10.1 Speedup I

Speedup measures the gain in (wall-clock) time that is obtained by parallel execution of a program.

$$S(p) = \frac{T(1)}{T(p)}$$

Clearly: $S(p) \leq p$.
If $S(p) > p$, we could run the parallel algorithm on one processor to obtain $pT(p) < T(1)$. This contradicts our assumption on $T(1)$ to be the execution time of the best sequential algorithm.

Remark

Some people report super-linear speedup. This is usually an artefact of the hardware (problem size vs. cache/memory size) used.
It is possible to have $S(p) < 1$ for $p > 1$. Of course it is not advisable to run an algorithm with this property on p processors. Nevertheless, memory requirements may force one to do so.

3.10.2 Efficiency I

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)}$$

As $S(p) \leq p$ we have

$$E(p) \leq 1$$

Efficiency gives the fraction of time for which a processor is doing useful work.

3.10.3 Amdahl's law I

Let us assume to have an algorithm that consists of two portions. One of them parallelizes optimally the other sequential portion not at all. (The sequential portion of a program may does initialization, or I/O or do some other work.) Then

$$T(p) = \left(\alpha + \frac{1-\alpha}{p}\right) T(1),$$

where $0 \leq \alpha \leq 1$ is the fraction of the sequential part of the algorithm. This formula is called Amdahl's law[2].

[2] Amdahl was an important pioneer of vector processing. His company was bought by Fujitsu in the 80s.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. A
was some unprocessed data that should have been added to the
this extra page has been added to receive it.
If you rerun the document (without altering it) this surplus page
away, because LaTeX now knows how many pages to expect for t
document.