

# TFY4235/FYS8904 : Computational Physics<sup>1</sup>

Ingve Simonsen

Department of Physics  
Norwegian University of Science and Technology  
Trondheim, Norway

Lecture series, Spring 2018

---

<sup>1</sup>Special thanks Profs. Alex Hansen and Morten Hjorth-Jensen for invaluable contributions

# General information

## Lectures

- Mon. 12:15–14:00 (EL6)
- Fri. 08:15–09:00 (R8)

## Exercises

- Fri. 09:15–10:00 (R8)

## Evaluation and grading

- Multi-day *take home* exam (counts 100% of final grade)
- Exam date to be decided<sup>a</sup>
- Exercises are not compulsory, but strongly recommended...
- Three assignments will be given during the semester
  - Solutions to one of the three will be randomly picked and part of the take-home exam

<sup>a</sup>Probably to take place just before the ordinary exam period

## Homepage

- <http://web.phys.ntnu.no/~ingves/Teaching/TFY4235/>
- All relevant information will be posted here!

# The aims of the course

## Main aim of the class

Provide you with a TOOLBOX for solving physical problems on the computer!

For a given problem, you should after completing this class, be able to

- identify suitable and efficient methods for solving it numerically
- write programs that use these algorithms
- to test you program to make sure it is correct
- to analyze the results
- have an overview of existing scientific numerical libraries

However, this class will not

- focus on the *details* of the various algorithms
- teach you how to program by following the lectures

# The aims of the course

In more detail:

- Develop a critical approach to *all* steps of a project
  - which natural laws and physical processes are important
  - sort out initial conditions and boundary conditions etc.
  - which methods are most relevant
- This means to teach you **structured scientific computing**, learn to structure a project.
- A critical understanding of central mathematical algorithms and methods from numerical analysis. Know their limits and stability criteria.
- **Always try to find good checks of your codes** (like closed-form solutions)
- To enable you to develop a critical view on the mathematical model and the physics.

# Programming languages

# Which programming language to use?

*In principle, you may use any programming language you wish for this class!*

## Recommended (by me)

- C
- C++
- Fortran 90/2003/2008
- python
- julia


## Not Recommended

- Matlab
- Java
- Fortran 77

## Comments

- Python is the definitely slowest of the recommended languages!
- Fortran 77 and C are regarded as slightly faster than C++ or Fortran<sup>2</sup> .

---

<sup>2</sup>For this class, Fortran means Fortran 95/2003/2008 (modern Fortran) 

# “The two-language problem”

“The two-language problem” is also known as *Outerhout's dichotomy* (after computer scientist John Outerhout's categorization scheme)

High-level programming languages tend to fall into two groups

- system programming languages
  - hard to use, **fast**
- scripting languages
  - easy to use, **slow**

Attempts to get the best of both worlds have tended to result in a bit of a mess.  
The best option today is Julia in my opinion!

## Strong features

- are widely available
- portable
- fast (C++ is slower than C)
- complex variables can also be defined in the new ANSI C++/C standards
- more and more numerical libraries exist (but still not as many as for Fortran 77)
- the efficiency of C++ can be close to that provided by Fortran
- C++ is rich (about 60 keywords)
- C++/C is used also outside the scientific/technical community
- C++ is an object-oriented language (C is not...)

## Weak features

- C++ is a complex language (takes time and experience to master)
- some parts of the language should NOT be used for numerical calculations since they are slow
- error prone dynamic memory management
- it is easy in C++ to write inefficient code (slow)



For this class, Fortran means Fortran 95/2003/2008 (modern Fortran)

## Strong features

- language made for numerical calculations
- large body of libraries for numerical calculations
- fairly easy to learn
- portable
- fast
- complex variables are native to the language
- array syntax ala Matlab  
e.g.  $A(3:5)$ ,  $\text{size}(A)$ ,  $\text{min}(A)$
- newest versions of Fortran is object-oriented

## Weak features

- Fortran is only used in the scientific/technical community
- Fortran is less rich then C++

## Strong features

- a rich scripting language
- fully object-oriented
- clean syntax gives fast code development
- free (non-commercial)
- bindings to many libraries exist
- **numpy** and **scipy** (use them!)
- easy integration of fast compiled C/C++/Fortran routines

## Weak features

- can be slow
- a scripting language

## Strong features

- fast : “Just in Time” compilation
- a rich scripting language
- fully object-oriented
- clean syntax gives fast code development
- free (non-commercial)
- bindings to many libraries exist
- interface an increasing number of libraries

## Weak features

- under development
- not so widespread

# Choose the “right” language for the job

During the 2014 exam a student found a speedup of a factor 350 when moving his code from Matlab to C++!



E.g. if the C-job took 15 min., Matlab will require 5250 h (about 4 days)!

The student said



It was painful to write out the program all over again at first, but after a while I got used to the syntax and came to understand exactly how much computing power can be saved. A relatively small simulation (1000 sweeps on a lattice with  $L = 100$ ) ran 343 times faster after re-writing the program in C++!

**The biggest lesson for me in this exam has definitely been the value of programming in an efficient language, and I only wish I had more time to make the C++ program even more efficient.**

The following material represents good reading material for this class:

-  Press, Flanery, Teukolsky and Vetterling, *Numerical Recipes: The Art of Scientific Computing*, 3rd ed., Cambridge University Press, 2007.
-  Morten Hjorth-Jensen, *Computational Physics*, unpublished, 2013  
Available from <http://web.phys.ntnu.no/~ingves/Teaching/TFY4235/Download/lectures2013.pdf>

For C++ programmers

-  J. J. Barton and L. R. Nackman, *Scientific and Engineering C++*, Addison Wesley, 3rd edition 2000.
-  B. Stoustrup, *The C++ programming language*, Pearson, 1997.

## Strong recommendation

- Use existing libraries whenever possible
- They are typically more efficient than what you can write yourself

Some important numerical libraries (to be mentioned and discussed later)

- LAPACK (Fortran 90) [wrapper LAPACK95]
- BLAS (Fortran 77)
- GNU Scientific Library (C)
- Slatec (Fortran 77)

Check out the list of numerical libraries at:

[http://en.wikipedia.org/wiki/List\\_of\\_numerical\\_libraries](http://en.wikipedia.org/wiki/List_of_numerical_libraries)

# A structured programming approach

- Before writing a single line of code, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g., the flow and organization of data before one starts writing.
- Always try to choose the simplest algorithm. Computational speed can be improved upon later.
- **Try to write an as clear program as possible.** Such programs are easier to debug, and although it may take more time, in the long run it may save you time. If you collaborate with other people, it reduces spending time on debugging and trying to understand what the codes do.

A clear program will also allow you to remember better what the program really does!

# A structured programming approach

- The planning of the program should be from top down to bottom, trying to keep the flow as linear as possible. Avoid jumping back and forth in the program. First you need to arrange the major tasks to be achieved. Then try to break the major tasks into subtasks. These can be represented by functions or subprograms. They should accomplish limited tasks and as far as possible be independent of each other. That will allow you to use them in other programs as well.
- **Try always to find some cases where an analytic solution exists** or where simple test cases can be applied. If possible, devise different algorithms for solving the same problem. If you get the same answers, you may have coded things correctly or made the same error twice or more.

## Warning

Remember, a compiling code does not necessarily mean a correct program!



# Section 1

## Introduction

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods
- 8 Numerical integration

# Outline II

- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations
- 12 Optimization

# What is Computational Physics (CP)?

- University of California at San Diego (UCSD)<sup>3</sup>

“Computational physics is a rapidly emerging new field covering a wide range of disciplines based on collaborative efforts of mathematicians, computer scientists, and researchers from many areas of pure and applied physics. This new approach has had a decisive influence on fields that traditionally have been computationally intensive, and is expected to change the face of disciplines that have not commonly been associated with high performance computation.

By its very nature, computational physics is strongly interdisciplinary, with methodologies that span the traditional boundaries between fields, allowing experts in this area a more flexible position in today’s competitive employment arena.”

- Wikipedia tries the following definition:

“Computational physics is the study and implementation of numerical algorithms to solve problems in physics for which a quantitative theory already exists” (not a good definition in my opinion)

---

<sup>3</sup>http:

# What is Computational Physics (CP)?

## Definition

*Computational physics* is the science of using computers to assist in the solution of physical problems, and to conduct further physics research

- 1 Discretized analytic calculations
  - 2 Algorithmic modelling
  - 3 Data treatment (*e.g.* CERN)
- 
- Computational physics is the “**third way**” of physics alongside experimental and theoretical physics
  - CP is a separate and independent branch of physics
  - Systems are studied by “**numerical experiments**”
  - Computational physics is *interdisciplinary*

# What is Computational Physics?

Some examples of areas that lie within the scope of computational physics

- Large scale quantum mechanical calculations in nuclear, atomic, molecular and condensed matter physics
- Large scale calculations in such fields as hydrodynamics, astrophysics, plasma physics, meteorology and geophysics
- Simulation and modelling of complex physical systems such as those that occur in condensed matter physics, medical physics and industrial applications
- Experimental data processing and image processing
- Computer algebra; development and applications
- The online interactions between physicist and the computer system
- Encouragement of professional expertise in computational physics in schools and universities

Source : Institute of Physics

# Why Computational Physics?

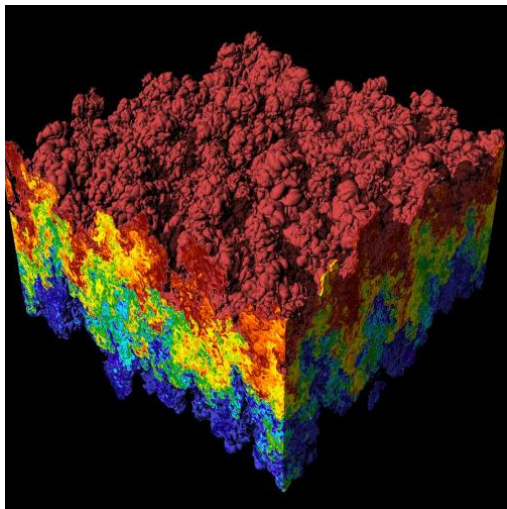
- **Physics problems are in general very difficult to solve exactly**
  - analytically solutions are the exceptions. . . not the rule
- Real-life problems often **cannot** be solved in closed form, due to
  - lack of algebraic and/or analytic solubility
  - complex and/or chaotic behavior
  - *too* many equations to render an analytic solution practical
- Some examples :
  - a bouncing ball
  - the physical pendulum satisfying the differential equation

$$\frac{d^2\theta(t)}{dt^2} + \frac{g}{\ell} \sin \theta(t) = 0$$

- system of interacting spheres (studies by *e.g.* molecular dynamics)
- the Navier-Stokes equations (non-linear equations)
- quantum mechanics of molecules
- *etc. etc*

**On the computer, one can study many complex real-life problems!**

# Why Computational Physics?



Simulations of Rayleigh-Taylor instability



Main class of approaches:

- 1 Discretized analytic calculations
  - Nature  $\rightarrow$  Continuous equations  $\rightarrow$  Discrete numerical model
  - A quantitative theory does exist
- 2 Algorithmic modeling
  - Nature  $\rightarrow$  Discrete numerical model (No analytic intermediate)
  - No quantitative theory used
- 3 Data treatment (*e.g.* CERN)

We now give some examples!

# Example 1: Laplace equation

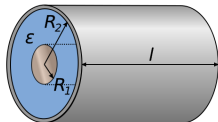
Problem: Find the electric field inside an annulus of inner radius  $R_1$  and outer radius  $R_2$  when the potential difference between these surfaces is  $V_0$ .

Mathematical formulation : Laplace equation

$$\nabla^2 V(\mathbf{r}) = 0$$

$$V(\mathbf{R}_1) = 0$$

$$V(\mathbf{R}_2) = V_0$$



Geometry under study

Discretization of space

$$\mathbf{r} \longrightarrow \{\mathbf{r}_{i,j}\}, \quad i, j = 1, \dots, N$$

$$V(\mathbf{r}) \longrightarrow V(\mathbf{r}_{i,j}) \rightarrow V_{i,j}$$

$$\nabla^2 V(\mathbf{r}) = 0 \longrightarrow V_{i,j+1} + V_{i,j-1} + V_{i+1,j} + V_{i-1,j} - 4V_{i,j} = 0$$

# Example 1: Laplace equation

- values of the potential on the boundary are known
  - $|\mathbf{r}_{i,j}| \approx R_1 : V_{i,j} = 0$
  - $|\mathbf{r}_{i,j}| \approx R_2 : V_{i,j} = V_0$
- this modifies the equations for points close to the surface

$$V_{i,j+1} + V_{i,j-1} + V_{i+1,j} + V_{i-1,j} - 4V_{i,j} = 0$$

so that known values gives raise to a right-hand-side

- a *linear system* (in  $\{V_{i,j}\}$ ) is formed

Linear system

Discretization of Laplace equation results in a linear system

Solving a linear system, solves the original continuous problem

$$\mathbf{A}\mathbf{v} = \mathbf{b} \quad \text{where } \mathbf{v} = [V_{11} \quad V_{21} \quad \dots \quad V_{NN}]^T$$

# Example 2: Diffusion-Limited Aggregation (DLA)

Consider the following system

- small (colloidal) particles diffuse in a liquid
- place a *sticky ball* in the liquid
- when a particle hits the surface of the ball it sticks

*Question* : What does the structure formed by this process look like?

## Challenge

How can one address this question?

# Example 2: Diffusion-Limited Aggregation (DLA)

## Model 1 : Discrete continuous model

- $C(\mathbf{r}, t)$  : particle concentration at position  $\mathbf{r}$  in the liquid at time  $t$
- $\partial B(t)$  : boundary of the (growing) “ball” at time  $t$
- $C_0$  : constant concentration at *long* distances  $r = R$  from the position of the ball at  $t = 0$

$$\nabla^2 C(\mathbf{r}, t) - \partial_t C(\mathbf{r}, t) = 0$$

$$C(\mathbf{r}, t) = 0 \text{ for } \mathbf{r} \text{ in } \partial B(t) \text{ (sticking boundary)}$$

$$C(\mathbf{r}, t) = C_0 \text{ for } r \approx R \gg |\partial B(t)|$$

Assumption: Surface growth proportional to concentration gradient,

$$\dot{\mathbf{S}}(\mathbf{r}, t) \propto \nabla C(\mathbf{r}, t)$$

- differential equation where the boundary conditions change with the solution
- the solution is unstable at all scales
- **problem can not be solved by solving a differential equation**

# Example 2: Diffusion-Limited Aggregation (DLA)

## Model 2 : Algorithmic modeling

- consider a large number of particles
- individual particles do *random walks*
- they stick to the boundary the first time they hit it

This model renders a description that fits quantitatively what is seen in nature (examples next slide)

### Algorithmic modeling

Nature is modeled directly!

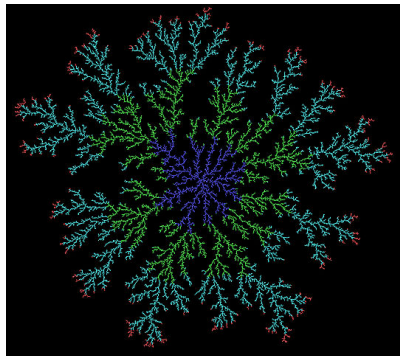
# Example 2: Diffusion-Limited Aggregation (DLA)

## Experiments



A DLA cluster grown from a copper sulfate solution in an electrodeposition cell

## Simulations



A DLA consisting about 33, 000 particles obtained by allowing random walkers to adhere to a seed at the center. Different colors indicate different arrival time of the random walkers.

DLA clusters in 2D have fractal dimension :  $D \approx 1.7$

# Example 3: Bak-Sneppen Model

Phys. Rev. Lett. **71**, 4083 (1993)

The model deals with evolutionary biology.

- a simple model of co-evolution between interacting species
- developed to show how self-organized criticality may explain key features of fossil records
  - the distribution of sizes of extinction events
  - the phenomenon of punctuated equilibrium

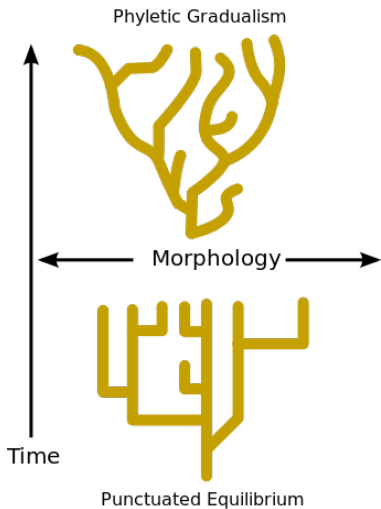
Reference :

- The “Bak-Sneppen” paper : Phys. Rev. Lett. **71**, 4083 (1993)
- See also : Phys. Rev. Lett. **76**, 348 (1996)



# Example 3: Bak-Sneppen Model

Phys. Rev. Lett. 71, 4083 (1993)



Two opposing theory of evolution:

- phyletic gradualism
- punctuated equilibrium (1972)

# Example 3: Bak-Sneppen Model

Phys. Rev. Lett. 71, 4083 (1993)

The “algorithm” used in the Bak-Sneppen model

- each species  $i$  is given a fitness parameter  $r_i$
- simplifications : species form a one dimensional chain with periodic BC

$$\dots - r_1 - r_2 - r_3 - r_4 - \dots$$

- *exchange dynamics* : How to update the system?

1 find lowest fitness

- $r_w = \min_i r_i, \quad r_{i(w)} = r_w$

2 update fitness

- $r_{i(w)} \rightarrow$  new random  $r_{i(w)}$
- $r_{i(w)\pm 1} \rightarrow$  new random  $r_{i(w)\pm 1}$

3 for next time step, repeat step 1 and 2 above

## Algorithmic modeling

Nature is modeled directly!

Normally quantitative theories do not exist for such problems

For instance, Diffusion-Limited Aggregation *cannot* be described by a differential equation!

# Topics

# Topics covered by the class

Tentative list and order

- Numerical precision
- Interpolation and extrapolation
- Numerical derivation and integration
- Random numbers and Monte Carlo integration
- Linear algebra
- Eigensystems
- Non-linear equations and roots of polynomials
- Fourier and Wavelet transforms
- Optimization (maximization/minimization)
- Monte Carlo methods in statistical physics
- Ordinary differential equations
- Partial differential equations
- Eigenvalue problems
- Integral equations
- Parallelization of codes (if time allows)

## Section 2

# Number representation and numerical precision

# Outline I

- 1 Introduction
- 2 Number representation and numerical precision**
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods
- 8 Numerical integration

# Outline II

- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations
- 12 Optimization

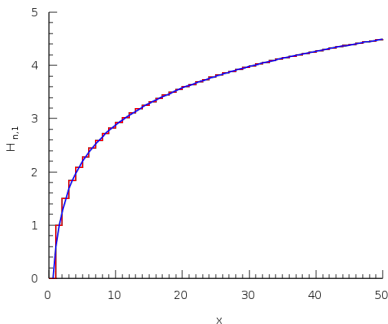


# An illustrative example

Harmonic numbers are defined by the sum

$$H_n = \sum_{k=1}^n \frac{1}{k} \sim \gamma + \ln(n) + \frac{1}{2n}$$

where the Euler constant is  $\gamma \approx 0.577215664\dots$



# An illustrative example

Results for  $H_n$  (in single precision) for different values of  $n$ :

- $n = 10$

Asymptotic	=	2.92980075	
Forward sum	=	2.92896843	-8.32319260E-04
Backward sum	=	2.92896843	-8.32319260E-04

- $n = 100\,000$

Asymptotic	=	12.0901461	
Forward sum	=	12.0908508	7.04765320E-04
Backward sum	=	12.0901527	6.67572021E-06

Question : Why is the backward sum more accurate?

# How numbers are represented

- Numbers  $\rightarrow$  words (*i.e.* strings of bits)
- May have length 32 or 64 or ...

## Consequence

Only a limited range of numbers may be represented with infinite precision. Otherwise, always an approximation.

# Finite numerical precision

Serious problem with the representation of numbers

A computer has *finite* numerical precision!

Potential problems in representing *integer*, *real*, and *complex* numbers

- Overflow
- Underflow
- Roundoff errors
- Loss of precision

# Typical limits for C/C++ and Fortran (on x86/x86\_64)

type in C/C++ and Fortran2008	bits	range
int/INTEGER (2)	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to 32767
int/long int/INTEGER(4)	32	-2147483648 to 2147483647
signed long int	32	-2147483648 to 2147483647
float/REAL(4)	32	$1.2 \times 10^{-38}$ to $3.4 \times 10^{+38}$
double/REAL(8)	64	$2.2 \times 10^{-308}$ to $1.8 \times 10^{+308}$
long double	64	$2.2 \times 10^{-308}$ to $1.8 \times 10^{+308}$

# Typical limits for C/C++ and Fortran (on x86/x86\_64)

How do we find these constants?

Fortran90 program:

```
program test_huge_tiny_epsilon
  implicit none
  write(*,*) huge(0), huge(0.0), huge(0.0d0)
  write(*,*)          tiny(0.0), tiny(0.0d0)
  write(*,*)          epsilon(0.0), epsilon(0.0d0)
end program test_huge_tiny_epsilon
```

Output:

```
~/Tmp tux => gfortran huge_tiny.f90 -o test_tiny_huge_epsilon
~/Tmp tux => test_tiny_huge_epsilon
2147483647    3.40282347E+38    1.7976931348623157E+308
              1.17549435E-38    2.2250738585072014E-308
              1.19209290E-07    2.2204460492503131E-016
```

# Representation of Integers

From decimal to binary representation

How is an integer number represented in the computer?

$$a_n 2^n + a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \dots + a_0 2^0.$$

In binary notation, we have e.g.  $(417)_{10} = (110100001)_2$  since

$$(110100001)_2 = 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

To see this, we have performed the following *integer* divisions by 2

417/2=208	remainder 1	coefficient of $2^0$ is 1
208/2=104	remainder 0	coefficient of $2^1$ is 0
104/2=52	remainder 0	coefficient of $2^2$ is 0
52/2=26	remainder 0	coefficient of $2^3$ is 0
26/2=13	remainder 1	coefficient of $2^4$ is 0
13/2= 6	remainder 1	coefficient of $2^5$ is 1
6/2= 3	remainder 0	coefficient of $2^6$ is 0
3/2= 1	remainder 1	coefficient of $2^7$ is 1
1/2= 0	remainder 1	coefficient of $2^8$ is 1

# Representation of floating-point numbers

A *floating-point* number,  $x$ , can be represented by:

$$x = (s, m, e)_b = (-1)^s \times m \times b^e$$

- $s$  the sign: positive ( $s = 0$ ) or negative ( $s = 1$ )
- $m$  the *mantissa* (significand or coefficient)
- $e$  the exponent
- $b$  the *base*:  $b = 2$  (binary) or  $b = 10$  (decimal)

Example :  $(1, 12345, -3)_{10} = (-1)^1 \times 12345 \times 10^{-3} = -12.345$

## Warning

Floating point representations vary from machine to machine!  
However, the IEEE 754 standard is quite common



# Representation of floating-point numbers

Some of the many floating-point arithmetic standard and their adoption

Standard	Architectures
IEEE 754	Intel x86, and all RISC systems <sup>4</sup>
VAX	Compaq/DEC
IBM S/390	IBM (in 1998, IBM added an IEEE 754 option to S/390)
Cray	X-MP, Y-MP, C-90 <sup>5</sup>

## Observation

Safe to assume that most modern CPUs (99%(?)) are IEEE 754 compliant. Only fairly exotic CPU architectures do today *not* use this standard!

<sup>4</sup>IBM Power and PowerPC, Compaq/DEC Alpha, HP PA-RISC, Motorola 68xxx and 88xxx, SGI (MIPS) R-xxxx, Sun SPARC, and others)

<sup>5</sup>Other Cray models have been based on Alpha and SPARC processors with IEEE-754 arithmetic

# Representation of floating-point numbers

In a typical computer **base  $b = 2$**  (binary representation) is used and one puts restrictions on  $m$  and  $e$  (imposed by the available word length).

## The mantissa

- *the leftmost binary digit of  $m$  is 1*
- this means,  $m$  is normalized; moved to the left as far as possible
- leading bit of  $m$  (always 1) is **not** stored  
(24 bits information storeable in 23 bits)

## The exponent

- $e$  is given uniquely from the requirements on  $m$
- add to  $e$  a *machine dependent exponential bias*  $e_0$  so that  $e + e_0 > 0$
- one store  $e + e_0 > 0$  in the floating-point representation

# Representation of floating-point numbers

Storage convention (for IEEE 754 floating-numbers)

$$x \longrightarrow \left| \underbrace{s}_{\text{sign}} \right| \left| \underbrace{e + e_0}_{\text{exponent}} \right| \left| \underbrace{m - 1}_{\text{mantissa}} \right| = \mathbf{s} \mathbf{e + e_0} \mathbf{m-1}$$

- $e + e_0$  is a positive integer

- $m$  the mantissa

$$m = (1.a_{-1}a_{-2}\dots a_{-23})_2 = 1 \times 2^0 + a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \dots + a_{-23} \times 2^{-23}$$

- only the fraction  $m - 1$  of the mantissa is stored, i.e. the  $a_n$ 's
- the leading bit of  $m$  is *not stored*)
- 23 bits used to represent 24 bits of information when using single precision

Storage size

Size in bits used in the IEEE 754 standard (most modern computers)

Type	Sign	Exponent	Mantissa	Total bits	Exponent bias	Bits precision	Decimal digits
Single	1	8	23	32	127	24	~7.2
Double	1	11	52	64	1024	53	~15.9
Quadruple	1	15	112	128	16383	113	~34.0

For more information see [http://en.wikipedia.org/wiki/Floating\\_point](http://en.wikipedia.org/wiki/Floating_point)

# Representation of floating-point numbers

Some examples: IEEE 754 floating-points

Example 1 Single precision representation (IEEE 754) of  $x = 1/4$

In single precision (32 bits),  $e_0 = 127$  and  $m$  is stored in 23 bits

$$x = 0.25 = 2^{-2} = (-1)^{\overbrace{0}^s} \times \overbrace{(2^{-2} \times 2^2)}^m \times 2^{\overbrace{-2}^e}$$

$$s = 0$$

$$m = (2^{-2} \times 2^2)_{10} = (1.000000000000000000000000)_2$$

$$e + e_0 = -2 + 127 = 125 = (01111101)_2$$

$$x \longrightarrow \mathbf{0} \mathbf{01111101} 000000000000000000000000$$

Exact (binary) representation exist for :  $x = 1/4$

# Representation of floating-point numbers

Some examples: IEEE 754 floating-points

## Example 2

Single precision representation (IEEE 754) of  $x = 2/3$

$$x = \frac{2}{3} = (-1)^{\overbrace{0}^s} \times \overbrace{(2/3 \times 2^1)}^m \times 2^{\overbrace{-1}^e}$$

$$x = (0.10101010\dots)_2 = \underbrace{(1.0101010\dots)_2}_m \times 2^{-1}$$

$$s = 0$$

$$m = (2/3 \times 2^1)_{10} = (1.01010101010101010\dots)_2$$

$$e + e_0 = -1 + 127 = 126 = (01111110)_2$$

$$x \longrightarrow \mathbf{0} \mathbf{01111110} \mathbf{0101010101010101010101011}$$

Representation of  $x = 2/3$  is an approximation!

# Representation of floating-point numbers

Some examples: IEEE 754 floating-points

## Example 3

Convert the following 32 bits (IEEE 754) binary number to decimal format:

$$x \longrightarrow 1 \ 01111101 \ 111010000000000000000000$$

This gives

$$\begin{aligned} s &= 1 \\ e + e_0 &= (01111101)_2 = 125 \quad \Rightarrow \quad e = 125 - 127 = -2 \\ m &= (1.111010000000000000000000)_2 \\ &= 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-5} \approx (1.906250)_{10} \end{aligned}$$

so that

$$x = (-1)^s \times m \times 2^e = (-0.4765625 \dots)_{10}$$

# Representation of floating-point numbers

Extra material

In the decimal system we would write a number like 9.90625 in what is called the normalized scientific notation.

$$9.90625 = 0.990625 \times 10^1,$$

and a real non-zero number could be generalized as

$$x = \pm r \times 10^n,$$

with  $r$  a number in the range  $1/10 \leq r < 1$ . In a similar way we can use represent a binary number in scientific notation as

$$x = \pm q \times 2^m,$$

with  $q$  a number in the range  $1/2 \leq q < 1$ . This means that the mantissa of a binary number would be represented by the general formula

$$(0.a_{-1}a_{-2}\dots a_{-n})_2 = a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \dots + a_{-n} \times 2^{-n}.$$

# Representation of floating-point numbers

## Extra material

In a typical computer, floating-point numbers are represented in the way described above, but with certain restrictions on  $q$  and  $m$  imposed by the available word length. In the machine, our number  $x$  is represented as

$$x = (-1)^s \times \text{mantissa} \times 2^{\text{exponent}},$$

where  $s$  is the sign bit, and the exponent gives the available range. With a single-precision word, 32 bits, 8 bits would typically be reserved for the exponent, 1 bit for the sign and 23 for the mantissa.



# Representation of floating-point numbers

Extra material

A modification of the scientific notation for binary numbers is to require that the leading binary digit 1 appears to the left of the binary point. In this case the representation of the mantissa  $q$  would be  $(1.f)_2$  and  $1 \leq q < 2$ . This form is rather useful when storing binary numbers in a computer word, since we can always assume that the leading bit 1 is there. One bit of space can then be saved meaning that a 23 bits mantissa has actually 24 bits. This means explicitly that a binary number with 23 bits for the mantissa reads

$$(1.a_{-1}a_{-2}\dots a_{-23})_2 = 1 \times 2^0 + a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \dots + a_{-23} \times 2^{-23}.$$

As an example, consider the 32 bits binary number

$$(10111110111101000000000000000000)_2,$$

where the first bit is reserved for the sign, 1 in this case yielding a negative sign. The exponent  $m$  is given by the next 8 binary numbers 01111101 resulting in 125 in the decimal system.

# Representation of floating-point numbers

## Extra material

However, since the exponent has eight bits, this means it has  $2^8 - 1 = 255$  possible numbers in the interval  $-128 \leq m \leq 127$ , our final exponent is  $125 - 127 = -2$  resulting in  $2^{-2}$ . Inserting the sign and the mantissa yields the final number in the decimal representation as

$$-2^{-2} (1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}) =$$
$$(-0.4765625)_{10}.$$

In this case we have an exact machine representation with 32 bits (actually, we need less than 23 bits for the mantissa).

If our number  $x$  can be exactly represented in the machine, we call  $x$  a machine number. Unfortunately, most numbers cannot and are thereby only approximated in the machine. When such a number occurs as the result of reading some input data or of a computation, an inevitable error will arise in representing it as accurately as possible by a machine number.

# How the computer performs elementary operations

## Addition and subtraction

### addition and subtraction

$x_1 \pm x_2$  is performed by *adjusting* the smallest exponent to equal the largest, add/subtract the scaled mantissas and multiply with the common exponent factor

Assuming

$$x_1 = (-1)^{s_1} m_1 2^{e_1}; \quad x_2 = (-1)^{s_2} m_2 2^{e_2}$$

with  $e_1 \geq e_2$ , the addition/subtraction operations are performed as

$$\begin{aligned} x_1 \pm x_2 &= (-1)^{s_1} m_1 2^{e_1} \pm (-1)^{s_2} m_2 2^{e_2} \\ &= [(-1)^{s_1} m_1 \pm (-1)^{s_2} m_2 2^{e_2 - e_1}] 2^{e_1} \end{aligned}$$

The factor  $2^{e_2 - e_1}$  pushes the mantissa of the scaled number to the **right**, causing loss of the least significant bits.

This phenomenon leads to **round-off error**, and is most pronounced when operating on numbers of different magnitude.

# How the computer performs elementary operations

## Addition and subtraction

Example (assuming single (32 bits) precision)

$$\begin{aligned}y &= (0.4765625)_{10} + (0.0000100)_{10} \\ &= (1.90625 \cdot 2^{-2})_{10} + (1.31072 \cdot 2^{-17})_{10} \\ &= 0 \text{ (125)}_{10} \text{ (0.90625)}_{10} + 0 \text{ (110)}_{10} \text{ (0.31072)}_{10} \\ &= 0 \text{ (125)}_{10} \text{ (0.90625 + 1.31072} \cdot 2^{-15})_{10} \\ &= 0 \text{ (125)}_{10} \text{ (0.90625 + 0.00004)}_{10} \\ &= 0 \text{ (125)}_{10} \text{ (0.90629)}_{10}\end{aligned}$$

# How the computer performs elementary operations

## Addition and subtraction

Using proper IEEE 754 floating point notation one gets

$$y = (0.4765625)_{10} + (0.0000100)_{10}$$

$$= 0 \text{ 01111101 } 111010000000000000000000 +$$

$$0 \text{ 01101110 } 01001111100010110101100 \quad (2^4 - 2^1 + 2^0 = 15)$$

$$= 0 \text{ 01111101 } 111010000000000000000000 +$$

$$0 \text{ 01111101 } 00000000000000101001111 \text{ 100010110101100 (lost bits)}$$

$$\approx 0 \text{ 01111101 } 11101000000000101010000 \text{ (rounding)}$$

### Right shifting

Since the mantissa is 23 bits, and we need to shift to the right GREATER THAN 23 bits in order to make the exponents equal (don't forget the hidden bit)

Converter : [http://www.binaryconvert.com/result\\_float.html](http://www.binaryconvert.com/result_float.html)

# How the computer performs elementary operations

## Addition and subtraction

Subtraction of almost identical numbers is dangerous (same for division)

Example (assuming single (32 bits) precision)

$$\begin{aligned} & 0 \ E \ 111 \dots 111 \ - \ 0 \ E \ 111 \dots 110 \\ &= 0 \ E \ 000 \dots 001 \\ &= 0 \ E \ -22 \ 1000 \dots 000 \end{aligned}$$

Only 50% accuracy!

# How the computer performs elementary operations

## Addition and subtraction

Example :  $y = 1 + 2^{-23}$ ;  $2^{-23} \approx 1.19209289550781e - 07$

$$\begin{aligned}y &= (1)_{10} + (2^{-23})_{10} \\ &= \begin{array}{l} 0 \text{ } 01111111 \text{ } 000000000000000000000000 \\ 0 \text{ } 01101000 \text{ } 000000000000000000000000 \\ 0 \text{ } 01111111 \text{ } 000000000000000000000001 \end{array} + \\ &= (1.00000012)_{10}\end{aligned}$$

Exponent used for scaling  $2^{-23}$  :  $2^4 + 2^2 + 2^1 + 2^0 = 23!$

$2^{-24}$  is numerical zero in single precision

**Note** :  $1 + 2^{-24}$  is one to single precision (32 bits)!

## Definition

The smallest number that can be added to 1 giving a result different from 1 ;  
*i.e.* smallest  $x$  such that  $1 + x > 1$

This results in the following machine precision :

- Single precision (32 bits) :  $2^{-23} \approx 1.1921 \cdot 10^{-7} \sim 10^{-7}$
- Double precision (64 bits) :  $2^{-52} \approx 2.2204 \cdot 10^{-16} \sim 10^{-16}$
- Quadruple precision (128 bits) :  $2^{-112} \approx 1.9259 \cdot 10^{-34} \sim 10^{-34}$   
(non-standard)

Fortran90 has inquiry functions for these numbers

- Single prec. : `epsilon(1.0)`
- Double prec. : `epsilon(1.0D0)!`



# How the computer performs elementary operations

## Multiplication and division

Multiplication/division of two numbers

$$x_1 = (-1)^{s_1} m_1 2^{e_1}; \quad x_2 = (-1)^{s_2} m_2 2^{e_2}$$

is done in the following way

$$\begin{aligned} x_1 \times x_2 &= [(-1)^{s_1} m_1 2^{e_1}] \times [(-1)^{s_2} m_2 2^{e_2}] \\ &= (-1)^{s_1+s_2} (m_1 m_2) 2^{e_1+e_2} \end{aligned}$$

In other words, it is done as we would have done it in mathematics!

However, on the computer one also tests for overflow/underflow.

# Loss of precision: Some examples

Problem : Calculate  $y = \frac{\cos(\pi/2)}{x}$  accurately for some small value  $x$

Direct calculation with  $x = 10^{-9}$  gives

- Single prec :  $y = -43.7113876$       **Wrong**
- Double prec :  $y = 6.1232339957367658E - 8 \neq 0$       **Inaccurate**

# Loss of precision: Some examples

Problem : For  $x = 0.0001$  calculate

$$y_1(x) = \frac{1 - \cos x}{\sin x}, \quad y_2(x) = \frac{\sin x}{1 + \cos x}$$

Analytically we have  $y_1(x) = y_2(x)$  which for the given value of  $x$  approximates to  $\approx 5.0000000041633333361 \dots \times 10^{-5}$ .

Calculation of  $y_1(x)$

- Single prec :  $y_1(x) = 0.00000000$  **Wrong**
- Double prec :  $y_1(x) = 4.9999999779459782E - 005$  **Inaccurate**

Calculation of  $y_2(x)$

- Single prec :  $y_2(x) = 4.99999987E - 05$  **Accurate**
- Double prec :  $y_2(x) = 5.0000000041666671E - 005$  **Inaccurate**

# Loss of precision: Some examples

Problem : Solve the quadratic equation

$$ax^2 + bx + c = 0$$

for parameters  $a = 1$ ,  $b = 200$ , and  $c = -0.000015$

The analytically solutions are  $x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ .

Real numerical result  $x_- = -200.000000075 \dots$  and  $x_+ = 0.000000075 \dots$

**Method 1:** Brute force numerical calculation (in double prec.) gives

$$x_- = -200.00000007500000 \quad x_+ = \underbrace{7.500000}_{7 \text{ digits}} \underbrace{2652814146}_{\text{inaccurate}} E - 008$$

Only  $x_-$  is accurate!

# Loss of precision: Some examples

**Method 2:** Rewrite the solution in the form

$$x_- = \frac{-b - \operatorname{sgn}(b) \sqrt{b^2 - 4ac}}{2a}, \quad x_+ = \frac{2c}{-b - \operatorname{sgn}(b) \sqrt{b^2 - 4ac}} = \frac{c}{ax_-}$$

$$x_- = -200.00000007500000 \quad x_+ = \underbrace{7.49999999}_{9 \text{ digits}} 71874996E - 008$$

Now  $x_+$  has 2 extra correct digits!

What is the problem?

When  $b^2 \gg 4ac$  like here, one has

- $x_-$  : result OK
- $x_+$  : catastrophic cancellation

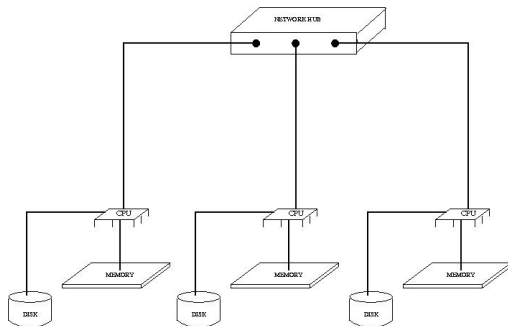
Reference :

D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Computing Surveys **23**, 5 (1991).

# Computer architectures

The main computer architectures of today are:

- *Distributed* memory computers
- *Shared* memory computers



## Section 3

# Finite differences and interpolation

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation**
  - Finite difference approximations
  - Interpolation schemes
  - Differentiation schemes
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods



# Outline II

- 8 Numerical integration
- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations
- 12 Optimization

# How to approximate derivatives

## The problem

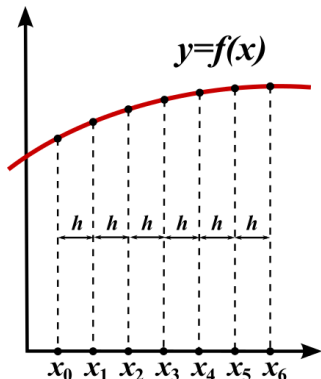
- the function  $f(x)$  is only known at a set of discrete points  $x_k$
- how can one then get information about derivatives?

## Some notation

$$x_k = x_{k-1} + h = x_0 + kh$$

$$f_k \equiv f(x_k)$$

$$f_{k+1} \equiv f(x_k + h) \text{ etc}$$



## Motivation

$$f^{(2)}(x_k) = \frac{f_{k+1} - 2f_k + f_{k-1}}{h^2} + \mathcal{O}(h^2)$$

# How to approximate derivatives

- Central difference approximation 1st order derivatives

$$f^{(1)}(x_k) = \frac{f_{k+1} - f_{k-1}}{2h} + \mathcal{O}(h^2)$$

- Forward difference approximation 1st order derivatives

$$f^{(1)}(x_k) = \frac{f_{k+1} - f_k}{h} + \mathcal{O}(h)$$

- Backward difference approximation 1st order derivatives

$$f^{(1)}(x_k) = \frac{f_k - f_{k-1}}{h} + \mathcal{O}(h)$$

- Central difference approximation 2nd order derivatives

$$f^{(2)}(x_k) = \frac{f_{k+1} - 2f_k + f_{k-1}}{h^2} + \mathcal{O}(h^2)$$

- Forward difference approximation 2nd order derivatives

$$f^{(2)}(x_k) = \frac{f_{k+2} - 2f_{k+1} + f_k}{h^2} + \mathcal{O}(h)$$

- ...

# How to approximate derivatives

Some higher order finite central difference approximations:

- 1st order derivatives

$$f^{(1)}(x_k) = \frac{-f_{k+2} + 8f_{k+1} - 8f_{k-1} + f_{k-2}}{12h} + \mathcal{O}(h^4)$$

- 2nd order derivatives

$$f^{(2)}(x_k) = \frac{-f_{k+2} + 16f_{k+1} - 30f_k + 16f_{k-1} - f_{k-2}}{12h^2} + \mathcal{O}(h^4)$$

Question: How to obtain such approximations?

# How to approximate derivatives

Say that we wanted a finite difference approximation to  $f^{(2)}(x_i)$  but now using the **three** points  $x_k$ ,  $x_{k+1}$  and  $x_{k+2}$ . How can this be done?

Write

$$f^{(2)}(x_k) = c_0 f_k + c_1 f_{k+1} + c_2 f_{k+2} + \underbrace{\epsilon(h)}_{\text{error}}$$

To determine the  $c_k$ 's, Taylor expand  $f_{k+n} = f(x_{k+n})$  around  $x_k$  to give the system

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & h & 2h \\ 0 & \frac{h^2}{2!} & \frac{(2h)^2}{2!} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \implies \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \frac{1}{h^2} \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

or

$$f^{(2)}(x_k) = \frac{f_k - 2f_{k+1} + f_{k+2}}{h^2} + \mathcal{O}(h)$$

3-point forward differences

# Finite Difference Operators

Define the following finite difference operators

- Forward differences

$$\Delta_h f_k = f_{k+1} - f_k = \Delta_h[f](x_k)$$

and higher-order differences obtained by induction  $[\Delta_h^2 f_k = \Delta_h(\Delta_h f_k)]$

$$\Delta_h^n f_k = \sum_{i=0}^n (-1)^i \binom{n}{i} f_{k+n-i} = \Delta_h^n[f](x_k)$$

- Backward differences

$$\nabla_h f_k = f_k - f_{k-1} = \nabla_h[f](x_k)$$

iterations gives

$$\nabla_h^n f_k = \sum_{i=0}^n (-1)^i \binom{n}{i} f_{k-n+i} = \nabla_h^n[f](x_k)$$

# Finite Difference Operators

- Central differences

$$\delta_h f_k = f_{k+1/2} - f_{k-1/2} = \delta_h[f](x_k)$$

iterations gives

$$\delta_h^n f_k = \sum_{i=0}^n (-1)^i \binom{n}{i} f_{k-n/2+i} = \delta_h^n[f](x_k)$$

Note that for odd  $n$  e.g.  $f_{k+1/2}$  and  $f_{k-n/2+i}$  are not known!

- Solution, introduce the central average

$$\mu_h f_k = \frac{1}{2}(f_{k+1/2} + f_{k-1/2})$$

and note that

$$\mu_h \delta_h f_k = \mu_h (f_{k+1/2} - f_{k-1/2}) = \frac{1}{2} (f_{k+1} - f_{k-1}) = \frac{\delta_{2h} f_k}{2}$$

# Finite Difference Operators

Definition of derivatives

$$f^{(1)}(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \equiv \lim_{h \rightarrow 0} \frac{\Delta_h[f](x)}{h}.$$

Finite difference approximations to derivatives of order  $n$  can be obtained by

$$f^{(n)}(x) = \frac{d^n f(x)}{dx^n} = \frac{\Delta_h^n[f](x)}{h^n} + \mathcal{O}(h^2) = \frac{\nabla_h^n[f](x)}{h^n} + \mathcal{O}(h^2) = \frac{\delta_h^n[f](x)}{h^n} + \mathcal{O}(h^2)$$

Higher-order differences can also be used to construct better approximations.  
Examples :

$$f^{(1)}(x) + \mathcal{O}(h^2) = \frac{\Delta_h[f](x) - \frac{1}{2}\Delta_h^2[f](x)}{h} = -\frac{f(x+2h) - 4f(x+h) + 3f(x)}{2h}$$

The best way to prove this is by Taylor expansion.



# Finite Difference Operators

*Finite-difference methods* are numerical methods for approximating the solutions to, e.g., differential equations using finite difference equations to approximate derivatives. We will later see in detail how this can be done.

Example : The 1D Diffusion equation

$$u_t = u_{xx}$$

$$u(0, t) = u(1, t) = 0 \quad (\text{boundary condition})$$

$$u(x, 0) = u_0(x) \quad (\text{initial condition})$$

Introducing  $u(x_j, t_n) = u_j^n$ , and using *forward difference* for time, and *central difference* for the space, one gets

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}.$$

or the explicit equation ( $\alpha = k/h^2$ )

$$u_j^{n+1} = (1 - 2\alpha)u_j^n + \alpha u_{j-1}^n + \alpha u_{j+1}^n$$

# Forwards difference

## Forwards difference

$$\Delta_h f_k = f_{k+1} - f_k$$

Consequence, by iteration:

$$\Delta_h^n f_k = \sum_{i=0}^n (-1)^i \binom{n}{i} f_{k+n-i}$$

## Backwards difference

$$\nabla_h f_k = f_k - f_{k-1}$$

Consequence, by iteration:

$$\nabla_h^n f_k = \sum_{i=0}^n (-1)^i \binom{n}{i} f_{k-n+i}$$

# Central difference

## Central difference

$$\delta_h f_k = f_{k+1/2} - f_{k-1/2}$$

Consequence, by iteration:

$$\delta_h^n f_k = \sum_{i=0}^n (-1)^i \binom{n}{i} f_{k-n/2+i}$$

But  $f_{n/2}$  is unknown for  $n$  odd! We introduce the **central average**.

## Central average

$$\mu_h f_k = \frac{1}{2}(f_{k+1/2} + f_{k-1/2})$$

Substitute all odd central differences by **central average of central differences**:

$$\delta_h f_k = f_{k+1/2} - f_{k-1/2} \rightarrow \mu_h \delta_h f_k = \frac{1}{2}(f_{k+1} - f_{k-1})$$

# Interpolation schemes

How to get as closely as possible to  $f(x)$  for any  $x$  given  $f(x_k) = f_k$ ?

We define:

$$u = \frac{x - x_k}{h}$$

and:

$$\binom{u}{l} = \frac{u(u-1)\cdots(u-l+1)}{l!}$$

Be careful, since  $u$  is not necessarily an integer here!

# Lagrange interpolation

Task : Given a set of  $N$  points,  $\{(x_n, y_n)\}_{n=1}^N$ , and we want to find an interpolating function for these points.

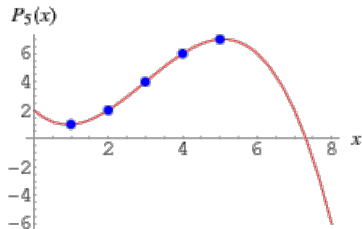
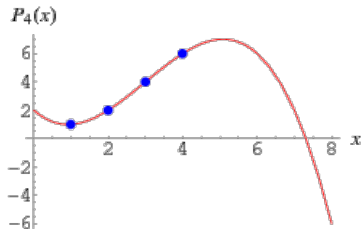
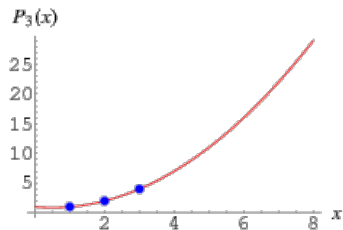
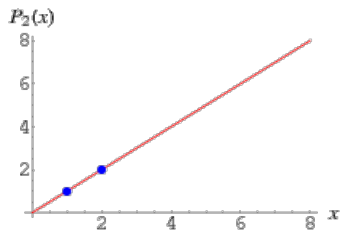
Lagrange interpolation is centered around constructing an *interpolating polynomial*  $P(x)$  of order  $\leq (N - 1)$  that passes through these points. *Lagrange interpolating polynomial* reads

$$P(x) = \sum_{n=1}^N P_n(x)$$

where

$$P_n(x) = y_n \prod_{\substack{k=1 \\ k \neq n}}^N \frac{x - x_k}{x_n - x_k}$$

# Lagrange interpolation



- The term “spline” is used to refer to a wide class of functions that are used in applications requiring data interpolation and/or smoothing
- Spline interpolation is a form of interpolation where the interpolant is a special type of piecewise polynomial called a spline
- The data may be **either one-dimensional or multi-dimensional**
- The splines are often chosen to be of a type that render derivatives continuous

For details see [https://en.wikipedia.org/wiki/Spline\\_\(mathematics\)](https://en.wikipedia.org/wiki/Spline_(mathematics))



## Newton-Gregory forward interpolation scheme

$$F_m(x) = f_k + \sum_{l=1}^m \binom{u}{l} \Delta_h^l f_k + \mathcal{O}(h^{m+1})$$

$F_m$  is a polynomial of order  $m$  that passes through  $m$  tabulated points  $(x_{k+i}, f_{k+i}), i \in \llbracket 0; m-1 \rrbracket$

Here  $F_m(x)$  represents a  $m$ th order polynomial approximation to  $f(x)$  for any coordinate  $x$ , i.e.  $F_m(x) \approx f(x)$ !

# Newton-Gregory forward interpolation scheme

Example  $m = 2$ :

$$\begin{aligned}F_2(x) &= f_k + \frac{\Delta_h f_k}{h}(x - x_k) + \frac{\Delta_h^2 f_k}{2h^2}(x - x_k)(x - x_{k+1}) + \mathcal{O}(h^3) \\&= f_k + \frac{1}{h}(f_{k+1} - f_k)(x - x_k) \\&\quad + \frac{1}{2h^2}(f_{k+2} - 2f_{k+1} + f_k)(x - x_k)(x - x_{k+1}) + \mathcal{O}(h^3)\end{aligned}$$

In particular:

$$F_2(x_k) = f_k$$

$$F_2(x_{k+1}) = f_{k+1}$$

$$F_2(x_{k+2}) = f_{k+2}$$

# Newton-Gregory backward interpolation scheme

## Newton-Gregory backward interpolation scheme

$$F_m(x) = f_k + \sum_{l=1}^m \binom{u+l-1}{l} \nabla_h^l f_k + \mathcal{O}(h^{m+1})$$

Example  $m = 2$ :

$$F_2(x) = f_k + \frac{\nabla_h f_k}{h} (x - x_k) + \frac{\nabla_h^2 f_k}{2h^2} (x - x_k)(x - x_{k-1}) + \mathcal{O}(h^3)$$

## Stirling interpolation scheme

$$F_{2n}(x) = f_k + \sum_{l=1}^n \binom{u+l-1}{2l-1} \left( \mu_h \delta_h^{2l-1} f_k + \frac{u}{2l} \delta_h^{2l} f_k \right) + \mathcal{O}(h^{2n+1})$$

Example  $n = 1$ :

$$F_2(x) = f_k + \frac{\mu_h \delta_h f_k}{h} (x - x_k) + \frac{\delta_h^2 f_k}{2h^2} (x - x_k)^2 + \mathcal{O}(h^3)$$

This is the **parabolic Stirling formula**.

# Difference quotients

It can be shown that

$$\frac{d}{du} \binom{u}{l} = \binom{u}{l} \sum_{i=0}^{l-1} \frac{1}{u-i}$$

$$\frac{d^2}{du^2} \binom{u}{l} = \begin{cases} 0 & \text{if } l = 1 \\ \binom{u}{l} \sum_{i=0}^{l-1} \sum_{j=0}^{l-1} \frac{1}{(u-i)(u-j)} & \text{if } l \geq 2 \end{cases}$$

These results will be useful when taking the derivatives of the Newton-Gregory schemes as will be done now!

# Difference Newton-Gregory forwards scheme

$$F'_m(x) = \frac{1}{h} \sum_{l=1}^m \Delta_h^l f_k \binom{u}{l} \sum_{i=0}^{l-1} \frac{1}{u-i} + \mathcal{O}(h^m)$$

$$F''_m(x) = \frac{1}{h^2} \sum_{l=2}^m \Delta_h^l f_k \binom{u}{l} \sum_{i=0}^{l-1} \sum_{j=0}^{l-1} \frac{1}{(u-i)(u-j)} + \mathcal{O}(h^{m-1})$$

If we set  $x = x_k$ :

$$F'_m(x_k) = \frac{1}{h} \sum_{l=1}^m (-1)^{l-1} \frac{\Delta_h^l f_k}{l} + \mathcal{O}(h^m)$$

$$F''_m(x_k) = \frac{1}{h^2} \sum_{l=2}^m (-1)^{l+1} \frac{\Delta_h^l f_k}{l} \sum_{i=1}^{l-1} \frac{1}{i} + \mathcal{O}(h^{m-1})$$

Example: Difference NG forward scheme with  $m = 2$ :

$$\begin{aligned} F'_2(x_k) &= \frac{1}{h} \left( \Delta_h f_k - \frac{\Delta_h^2 f_k}{2} \right) + \mathcal{O}(h^2) \\ &= \frac{1}{h} \left( -\frac{1}{2} f_{k+2} + 2f_{k+1} - \frac{3}{2} f_k \right) + \mathcal{O}(h^2) \end{aligned}$$

Example: Difference NG backward scheme with  $m = 2$ :

$$\begin{aligned} F'_2(x_k) &= \frac{1}{h} \left( \nabla_h f_k - \frac{\nabla_h^2 f_k}{2} \right) + \mathcal{O}(h^2) \\ &= \frac{1}{h} \left( \frac{3}{2} f_k - 2f_{k-1} + \frac{1}{2} f_{k-2} \right) + \mathcal{O}(h^2) \end{aligned}$$

# Differential Stirling formula

It can be shown that

$$F'_{2n}(x_k) = \frac{1}{h} \left( \mu_h \delta_h f_k - \frac{1}{6} \mu_h \delta_h^3 f_k + \frac{1}{30} \mu_h \delta_h^5 f_k - \frac{1}{140} \mu_h \delta_h^7 f_k + \dots \right) + \mathcal{O}(h^{2n})$$
$$F''_{2n}(x_k) = \frac{1}{h^2} \left( \delta_h^2 f_k - \frac{1}{12} \delta_h^4 f_k + \frac{1}{90} \delta_h^6 f_k - \frac{1}{560} \delta_h^8 f_k + \dots \right) + \mathcal{O}(h^{2n})$$

**Note:** in the DNGF and DNGB schemes, the error increases for each new order of derivation. This does not happen in the DST scheme!



# Differential Stirling formula

Example  $n = 1$ :

- 1st derivative

$$\begin{aligned}F_2'(x_k) &= \frac{1}{h} \mu_h \delta_h f_k + \mathcal{O}(h^2) \\ &= \frac{1}{2h} (f_{k+1} - f_{k-1}) + \mathcal{O}(h^2)\end{aligned}$$

- 2nd derivative

$$\begin{aligned}F_2''(x_k) &= \frac{\delta_h^2 f_k}{h^2} + \mathcal{O}(h^2) \\ &= \frac{1}{h^2} (f_{k+1} - 2f_k + f_{k-1}) + \mathcal{O}(h^2)\end{aligned}$$

**Note:** these results are the “standard” central difference approximations for the 1st and 2nd derivatives

# Why does it work so well?

Graphical representation (to come).

# Section 4

## Linear algebra

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra**
  - Direct methods
  - Iterative methods
  - Singular value decomposition
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods

# Outline II

- 8 Numerical integration
- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations
- 12 Optimization

# Basic Matrix Features

## Matrix Properties Reminder

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The inverse of a matrix is defined by

$$\mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{I}$$

# Basic Matrix Features

## Matrix Properties Reminder

Relations	Name	matrix elements
$\mathbf{A} = \mathbf{A}^T$	symmetric	$a_{ij} = a_{ji}$
$\mathbf{A} = (\mathbf{A}^T)^{-1}$	real orthogonal	$\sum_k a_{ik} a_{jk} = \sum_k a_{ki} a_{kj} = \delta_{ij}$
$\mathbf{A} = \mathbf{A}^*$	real matrix	$a_{ij} = a_{ij}^*$
$\mathbf{A} = \mathbf{A}^\dagger$	hermitian	$a_{ij} = a_{ji}^*$
$\mathbf{A} = (\mathbf{A}^\dagger)^{-1}$	unitary	$\sum_k a_{ik} a_{jk}^* = \sum_k a_{ki}^* a_{kj} = \delta_{ij}$

# Some famous Matrices

- 1 Diagonal if  $a_{ij} = 0$  for  $i \neq j$
- 2 Upper triangular if  $a_{ij} = 0$  for  $i > j$
- 3 Lower triangular if  $a_{ij} = 0$  for  $i < j$
- 4 Upper Hessenberg if  $a_{ij} = 0$  for  $i > j + 1$
- 5 Lower Hessenberg if  $a_{ij} = 0$  for  $i < j - 1$
- 6 Tridiagonal if  $a_{ij} = 0$  for  $|i - j| > 1$
- 7 Lower banded with bandwidth  $p$   $a_{ij} = 0$  for  $i > j + p$
- 8 Upper banded with bandwidth  $p$   $a_{ij} = 0$  for  $i < j - p$
- 9 Banded, block upper triangular, block lower triangular....



## Some Equivalent Statements

For an  $N \times N$  matrix  $\mathbf{A}$  the following properties are all equivalent

- 1 If the inverse of  $\mathbf{A}$  exists,  $\mathbf{A}$  is nonsingular.
- 2 The equation  $\mathbf{Ax} = 0$  implies  $\mathbf{x} = 0$ .
- 3 The rows of  $\mathbf{A}$  form a basis of  $R^N$ .
- 4 The columns of  $\mathbf{A}$  form a basis of  $R^N$ .
- 5  $\mathbf{A}$  is a product of elementary matrices.
- 6 0 is not eigenvalue of  $\mathbf{A}$ .

# Important Mathematical Operations

The basic matrix operations that we will deal with are addition and subtraction

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \implies a_{ij} = b_{ij} \pm c_{ij},$$

scalar-matrix multiplication

$$\mathbf{A} = \gamma \mathbf{B} \implies a_{ij} = \gamma b_{ij},$$

vector-matrix multiplication

$$\mathbf{y} = \mathbf{A}\mathbf{x} \implies y_i = \sum_{j=1}^n a_{ij} x_j,$$

matrix-matrix multiplication

$$\mathbf{A} = \mathbf{B}\mathbf{C} \implies a_{ij} = \sum_{k=1}^n b_{ik} c_{kj},$$

and transposition

$$\mathbf{A} = \mathbf{B}^T \implies a_{ij} = b_{ji}$$

# Important Mathematical Operations

Similarly, important vector operations that we will deal with are addition and subtraction

$$\mathbf{x} = \mathbf{y} \pm \mathbf{z} \implies x_i = y_i \pm z_i,$$

scalar-vector multiplication

$$\mathbf{x} = \gamma \mathbf{y} \implies x_i = \gamma y_i,$$

vector-vector multiplication (called Hadamard multiplication)

$$\mathbf{x} = \mathbf{y}\mathbf{z} \implies x_i = y_i z_i,$$

the inner or so-called dot product resulting in a constant

$$x = \mathbf{y}^T \mathbf{z} \implies x = \sum_{j=1}^n y_j z_j,$$

and the outer product, which yields a matrix,

$$\mathbf{A} = \mathbf{y}\mathbf{z}^T \implies a_{ij} = y_i z_j,$$

# Some notations before starting!

- We will denote matrices and vectors by capital letters, and their components by lowercase letters. Example: the matrix  $\mathbf{A}$ , its element  $a_{ij}$ .
- $\mathfrak{M}_{M,N}(\mathbb{R})$ : set of real matrices of size  $M \times N$ .
- $\mathbf{I}_N = 1_{\mathfrak{M}_N(\mathbb{R})}$  the identity matrix.
- $\mathbf{A}^T$  the transpose of the matrix  $\mathbf{A}$ .
- $\mathbf{GL}_N(\mathbb{R})$ : general linear group of matrices i.e. set of invertible real matrices of size  $N$ .
- $\mathbf{O}_N(\mathbb{R})$ : orthogonal group i.e.  $\{M \in \mathbf{GL}_N(\mathbb{R}), M^T M = M M^T = \mathbf{I}_N\}$
- $\text{Sp}(A)$ : spectrum of  $A$ , i.e. the set of eigenvalues of  $\mathbf{A}$ .
- $\ker(\mathbf{A})$ : kernel of  $\mathbf{A}$ , i.e.  $\ker(\mathbf{A}) = \{\mathbf{v} \in \mathbb{R}^N | \mathbf{A}\mathbf{v} = \mathbf{0}\}$

The problem addressed in linear algebra

- Solve the matrix system  $\mathbf{Ax} = \mathbf{b}$ , *i.e.*

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N = b_2$$

$$\vdots$$

$$a_{M1}x_1 + a_{M2}x_2 + \cdots + a_{MN}x_N = b_M$$

- $N$  unknowns,  $M$  equations.

Three possibilities:

- If  $N = M$ , two possible situations:
  - a **unique solution** exists and is given by  $\mathbf{x} = \mathbf{A}^{-1} \mathbf{B}$
  - Some of the equations are linear combinations of others
    - $\mathbf{A}$  is a singular matrix
    - $\Rightarrow$  **Degenerate problem.**

For a *degenerate problem* the matrix  $\mathbf{A}$  is singular i.e. cannot be inverted (in the normal sense). The solution of  $\mathbf{Ax} = \mathbf{b}$  then consists of a *particular solution*  $\mathbf{x}_p$  plus any linear combinations of zero-vectors  $\mathbf{x}_i^0$  such that  $\mathbf{Ax}_i^0 = 0$ .

$$\mathbf{x} = \mathbf{x}_p + \sum_{i=1}^D c_i \mathbf{x}_i^0$$

$D = \dim(\ker(\mathbf{A}))$  The particular solution  $\mathbf{x}_p$  can be found by the so-called

*Moore-Pendose pseudoinverse*,  $\mathbf{A}^+$ , also know as the *generalized inverse*. The pseudoinverse always exists.

- If  $N > M$ , the equation set is also degenerate
- If  $N < M$ , the system is overdetermined and probably no solution exists.
  - This is typically caused by the physical assumptions underlying the equations not being compatible. However, it is still possible to find a *best solution* given the circumstances.

We now assume:  $A \in \mathbf{GL}_N(\mathbb{R})$  or  $A \in \mathbf{GL}_N(\mathbb{C})$  (i.e.  $M = N$  and  $\mathbf{A}^{-1}$  exists).

Two classes of solvers

- Direct solvers
- Iterative solvers

When to use a Direct or an Iterative method to solve  $\mathbf{Ax} = \mathbf{b}$ ?

Direct methods

- will always find the solution
- typically used for **dense** matrices
- matrix inversion takes long
- Lapack useful

Iterative methods

- solution not necessarily found (no convergence)
- typically used for **sparse** matrices
- preconditioning useful
- can be fast



The most central methods in the two classes are:

## Direct methods

- Gauss elimination
- LU decomposition
- QR decomposition
- Tridiagonal matrices
- Toeplitz matrices

## Iterative methods

- Jacobi
- Gauss-Seidel
- SOR
- Steepest descent
- Powell
- Conjugate gradient
- BiCGStab
- GMRES

The system we want to solve is

$$\begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix}$$

## Gauss elimination (row reduction)

- 1 Find the largest element in abs-value of the 1<sup>st</sup> column. If this is located in row  $i$ , then switch the 1<sup>st</sup> and  $i^{\text{th}}$  row in  $\mathbf{A}$ ,  $\mathbf{x}$ , and  $\mathbf{b}$ . (**Partial pivoting**)
- 2 Subtract from rows 2 to  $N$  in  $\mathbf{A}$  and  $\mathbf{b}$  row 1 multiplied by a suitable factor such that so that  $\forall i \in [2; N] A_{i1} = 0$ .
- 3 Repeat the above steps for 2nd column and rows up to  $N - 1$ , and so on.

After completing this GE-process, the resulting matrix is in an *upper triangular form*.

The *upper triangular form* looks like this

$$\begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1N} \\ 0 & a'_{22} & \cdots & a'_{2N} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a'_{NN} \end{pmatrix} \begin{pmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_N \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_N \end{pmatrix}$$

This system of equations is solved by *backsubstitution*

## Backsubstitution

$$\begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1N} \\ 0 & a'_{22} & \cdots & a'_{2N} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a'_{NN} \end{pmatrix} \begin{pmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_N \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_N \end{pmatrix}$$

Solve the last equation first, then the next-to-last, and so on...

$$x'_N = \frac{1}{a'_{NN}} b'_N$$

$$x'_{N-1} = \frac{1}{a'_{N-1, N-1}} (b'_{N-1} - a'_{N-1, N} x'_N)$$

$\vdots$

$$x'_i = \frac{1}{a'_{ii}} \left( b'_i - \sum_{j=i+1}^N a'_{ij} x'_j \right)$$

Lower Upper (LU) decomposition with partial pivoting consists of writing  $\mathbf{A}$  as

$$\mathbf{A} = \mathbf{P}^{-1} \mathbf{L} \mathbf{U}$$

where  $\mathbf{P}$  is a permutation matrix;  $\mathbf{L}$  is *lower triangular* with unit diagonal, and  $\mathbf{U}$  is an *upper triangular* matrix<sup>6</sup>

In matrix notation this looks like (for  $N = 4$  and assuming  $\mathbf{P} = \mathbf{I}$ )

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix}$$

LU decomposition is often also called LUP due to the permutation matrix ( $\mathbf{P}$ )

LUP decomposition is always possible and it is unique!

<sup>6</sup>LU decomposition introduced by Alan Turing.

LU decomposition can be viewed as the matrix form of Gaussian elimination.

Several algorithms exist for performing LU decomposition

- Crout and LUP algorithms
- Doolittle algorithm
- Closed formula

Numerical complexity of the LU decomposition

- LU decomposition :  $2N^3/3$
- forward and backward substitution for solving a linear system:  $\propto N^2$

## The LU decomposition of $\mathbf{A}$ is useful for many tasks

- Solution of  $\mathbf{Ax} = \mathbf{b}$

- 1 since  $\mathbf{A} = \mathbf{P}^{-1}\mathbf{LU}$  the system to solve is  $\mathbf{LUx} = \mathbf{Pb}$
- 2 define  $\mathbf{y} = \mathbf{Ux}$  and solve  $\mathbf{Ly} = \mathbf{Pb}$  for  $\mathbf{y}$  (forward substitution)
- 3 solve  $\mathbf{Ux} = \mathbf{y}$  for  $\mathbf{x}$  (backward substitution)

We note that it is the LU decomposition that is the costly step computationally.

To solve for several right-hand-sides is almost free in comparison.

- Determinant  $|\mathbf{A}|$

$$|\mathbf{A}| = |\mathbf{P}^{-1}| |\mathbf{L}| |\mathbf{U}| = (-1)^S \underbrace{\left[ \prod_{i=1}^N l_{ii} \right]}_1 \left[ \prod_{i=1}^N u_{ii} \right] = (-1)^S \prod_{i=1}^N u_{ii}$$

where  $S$  is the no. exchanges in the decomposition

- Inverse matrix  $\mathbf{A}^{-1}$

Solve the systems

$$\mathbf{A}\mathbf{x}_n = \mathbf{b}_n \quad \text{for } n = 1, 2, \dots, N$$

with  $\mathbf{b}_n$  zero everywhere, except at row  $n$  where it has value 1. The inverse is then

$$\mathbf{A}^{-1} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N]$$

In practice this is solved as the matrix system  $\mathbf{A}\mathbf{X} = \mathbf{I}$  so that  $\mathbf{X} = \mathbf{A}^{-1}$



## Recommendation: Dense matrix solver

LU decomposition (or LU factorization) is the workhorse for solving dense matrix systems, finding determinants, and directly obtaining the matrix inverse.

As a general advice, use the **LU decomposition for dense matrices**.

## LAPACK

Do *not* implement the LU decomposition yourself, use the high performance library LAPACK (which depend on BLAS)!

You simply can not beat an optimized version of this library!

LAPACK routines (LAPACK95 equivalents: `la_getrs / la_getrf`)

- solve system (using LU decomp.) : `sgetrs, dgetrs, cgetrs, zgetrs`
- LU factorization : `sgetrf, dgetrf, cgetrf, zgetrf`

# QR decomposition

A general rectangular  $M$ -by- $N$  matrix  $\mathbf{A}$  has a QR decomposition into the product of an orthogonal  $M$ -by- $M$  square matrix  $\mathbf{Q}$  (where  $\mathbf{Q}^T \mathbf{Q} = \mathbf{I}$ ) and an  $M$ -by- $N$  right-triangular (upper-triangular) matrix  $\mathbf{R}$ ,

$$\mathbf{A} = \mathbf{QR}$$

## Usage

- solving  $\mathbf{Ax} = \mathbf{b}$  by back-substitution
- to compute an *orthonormal basis* for a set of vectors
  - The first  $N$  columns of  $\mathbf{Q}$  form an orthonormal basis for the range of  $\mathbf{A}$ ,  $\text{rang}(\mathbf{A})$ , when  $\mathbf{A}$  has full column rank.

- LAPACK — Linear Algebra Package

<http://www.netlib.org>

- The library provides routines for solving systems of linear equations and linear least squares, eigenvalue problems, and singular value decomposition.
- routines to handle both real and complex matrices in both single and double precision.
- originally written in FORTRAN 77, but moved to Fortran 90 from version 3.2 (2008)
- ScaLAPACK: Parallel (MPI) version
- source code freely available
- Fortran and C/C++ versions are available
- LAPACK is based on the older LINPACK and EISPACK

- LAPACK95 (for Fortran users)

- Generic and convenient (modern) Fortran interface to LAPACK
- See: [www.netlib.org/lapack95/](http://www.netlib.org/lapack95/)

- **BLAS — Basic Linear Algebra Subprograms**

<http://www.netlib.org>

- routines that provide standard building blocks for performing basic vector and matrix operations
- highly (manufactured) optimized versions exist  
(e.g. multi-threaded Intel MKL, or AMD ACML, or OpenBLAS libraries)
- BLAS routines exist at 3 levels
  - BLAS I : vector operations
  - BLAS II : vector-matrix operations
  - BLAS III : III matrix-matrix operations.
- also BLAS is freely available

- **GNU Scientific Library (GSL)**

<http://www.gnu.org/software/gsl/>

- general purpose numerical library (including linear algebra)
- freely available
- has BLAS support
- written in C
- C++ and Fortran wrappers exist

# Important Matrix and vector handling packages

- Armadillo: C++ linear algebra library  
<http://arma.sourceforge.net/>
  - optional integration with LAPACK
  - syntax (API) is deliberately similar to Matlab
  - library is open-source software

For C++ users, Armadillo is a useful tool!

# Important Matrix and vector handling packages

Armadillo, recommended!!

- Armadillo is a C++ linear algebra library (matrix maths) aiming towards a good balance between speed and ease of use. The syntax is deliberately similar to Matlab.
- Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided through optional integration with LAPACK, or one of its high performance drop-in replacements (such as the multi-threaded MKL or ACML libraries).
- A delayed evaluation approach is employed (at compile-time) to combine several operations into one and reduce (or eliminate) the need for temporaries. This is accomplished through recursive templates and template meta-programming.
- Useful for conversion of research code into production environments, or if C++ has been decided as the language of choice, due to speed and/or integration capabilities.
- The library is open-source software, and is distributed under a license that is useful in both open-source and commercial/proprietary contexts.

### Examples of compiling a program (on unix) that requires the use of a library

```
g++ -O2 -o RunMe program.cpp -larmadillo -llapack -lblas
```

```
gfortran -Imypath program.f90 -Lmylibpath -lmylib -llapack -lblas
```

### Options

```
-l : library you wish to link to!  
-L : search path for libraries  
-I : search path for include files  
-O2 : optimization flag
```

# Important Matrix and vector handling packages

## Armadillo, simple examples

```
#include <iostream>
#include "armadillo"
using namespace arma;
using namespace std;

int main(int argc, char** argv)
{
  // directly specify the matrix size (elements are uninitialised)
  mat A(2,3);
  // .n_rows = number of rows      (read only)
  // .n_cols = number of columns (read only)
  cout << "A.n_rows = " << A.n_rows << endl;
  cout << "A.n_cols = " << A.n_cols << endl;
  // directly access an element (indexing starts at 0)
  A(1,2) = 456.0;
  A.print("A:");
  // scalars are treated as a 1x1 matrix,
  // hence the code below will set A to have a size of 1x1
  A = 5.0;
  A.print("A:");
  // if you want a matrix with all elements set to a particular value
  // the .fill() member function can be used
  A.set_size(3,3);
  A.fill(5.0);  A.print("A:");
}
```



# Important Matrix and vector handling packages

## Armadillo, simple examples

```
mat B;  
  
// endr indicates "end of row"  
B << 0.555950 << 0.274690 << 0.540605 << 0.798938 << endr  
  << 0.108929 << 0.830123 << 0.891726 << 0.895283 << endr  
  << 0.948014 << 0.973234 << 0.216504 << 0.883152 << endr  
  << 0.023787 << 0.675382 << 0.231751 << 0.450332 << endr;  
  
// print to the cout stream  
// with an optional string before the contents of the matrix  
B.print("B:");  
  
// the << operator can also be used to print the matrix  
// to an arbitrary stream (cout in this case)  
cout << "B:" << endl << B << endl;  
// save to disk  
B.save("B.txt", raw_ascii);  
// load from disk  
mat C;  
C.load("B.txt");  
C += 2.0 * B;  
C.print("C:");
```

# Important Matrix and vector handling packages

## Fortran90 examples

```
program f90_example
  implicit none
  integer, parameter :: N=100
  integer, parameter :: wp=kind(1.0)
  real(wp)           :: one
  real(wp), dimension(N,N) :: A, B, C
  real(wp), dimension(N*N) :: vec

  ! Fill A and B with uniform random numbers [0,1>
  call random_number( A )
  call random_number( B )

  ! declare a variable
  one = 1._wp

  ! Adding element wise
  C = A + B

  ! Matrix multiplication
  C = matmul(A,B)

  ! convert matrix to vector
  vec = reshape(A, [N*N] )

  ! sum of elements
  Write(*,*) " Sum of A elements           : ", sum(A), sum( abs(A-0.5_wp) )

end program f90_example
```

Typically, a physical problem will be formulated by the interactions between  $N$  objects. Each object  $i$  is characterized by a variable  $x_i$ , and depends on the state of  $n$  other objects. Hence, we have

$$\begin{aligned}x_1 &= f_1(x_{1(1)}, x_{2(1)}, \dots, x_{n(1)}) \\x_2 &= f_2(x_{1(2)}, x_{2(2)}, \dots, x_{n(2)}) \\&\vdots \\x_N &= f_N(x_{1(N)}, x_{2(N)}, \dots, x_{n(N)})\end{aligned}$$

If this is a linear problem, it is on the form:

$$x_i = b_i + \sum_{j \in n(i)} c_{ij} x_j, \quad i = 1, 2, \dots, N$$

This can be formulated as  $\mathbf{Ax} = \mathbf{b}$ .

When  $n \ll N$ , the corresponding matrix  $\mathbf{A}$  is sparse.

- **Objective of iterative methods:**

to construct a sequence  $\{\mathbf{x}^{(k)}\}_{k=1}^{\infty}$ , so that  $\mathbf{x}^{(k)}$  converges to a fixed vector  $\mathbf{x}^*$ , where  $\mathbf{x}^*$  is the solution of the problem (e.g. a linear system)

- **General iteration idea**

Say that we want to solve the equations

$$\mathbf{g}(\mathbf{x}) = \mathbf{0},$$

and the equation  $\mathbf{x} = \mathbf{f}(\mathbf{x})$  has the same solution as it, then construct

$$\mathbf{x}^{(k+1)} = \mathbf{f}(\mathbf{x}^{(k)}).$$

If  $\mathbf{x}^{(k)} \rightarrow \mathbf{x}^*$ , then  $\mathbf{x}^* = \mathbf{f}(\mathbf{x}^*)$ , and the root of  $\mathbf{g}(\mathbf{x}) = \mathbf{0}$  is obtained.

When this strategy is applied to  $\mathbf{Ax} = \mathbf{b}$ , the functions  $\mathbf{f}(\cdot)$  and  $\mathbf{g}(\cdot)$  are linear operators.

### Some terminology for solving the system $\mathbf{Ax} = \mathbf{b}$ iteratively

Let  $\mathbf{x}$  be the exact solution, and  $\{\mathbf{x}^{(k)}\}$  a sequence of approximations to this solution. Then one defines

- The residual

$$\mathbf{R}^{(k)} = \mathbf{b} - \mathbf{Ax}^{(k)}$$

- The error

$$\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}$$

- The rate of convergence

$$r^{(k+1)} = \frac{\|\mathbf{e}^{(k+1)}\|}{\|\mathbf{e}^{(k)}\|}$$
$$\tilde{r}^{(k+1)} = \frac{\|\mathbf{e}^{(k+1)} - \mathbf{e}^{(k)}\|}{\|\mathbf{e}^{(k)}\|} = \frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|}{\|\mathbf{x}^{(k)} - \mathbf{x}\|}$$

for a suitable  $\|\cdot\|$  norm

## Basic idea behind iterative methods for solving $\mathbf{Ax} = \mathbf{b}$

- Start with an initial guess for the solution  $\mathbf{x}^{(0)}$ .
- Obtain  $\mathbf{x}^{(k+1)}$  from the knowledge of  $\mathbf{x}^{(k)}$  for  $k = 0, 1, \dots$
- If  $\|\mathbf{Ax}^{(k)} - \mathbf{b}\| \xrightarrow[k \rightarrow \infty]{} 0$  then  $\mathbf{x}^{(k)} \xrightarrow[k \rightarrow \infty]{} \mathbf{x}$ ; the solution of  $\mathbf{Ax} = \mathbf{b}$  is found

The two main classes of iterative methods (for linear systems) are:

### Stationary iterative methods

- Jacobi
- Gauss-Seidel
- SOR

### Krylov subspace methods

- Conjugate gradient
- BiCGStab
- GMRES
- Steepest descent
- Powell

# Jacobi algorithm

The Jacobi iteration method is one of the oldest and simplest methods

Decompose the matrix  $\mathbf{A}$  as follows

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U},$$

where

- $\mathbf{D}$  is diagonal
- $\mathbf{L}$  and  $\mathbf{U}$  are strict lower and upper triangular matrices

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{NN} \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1N} \\ 0 & 0 & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

## Jacobi iteration scheme

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1} \left[ \mathbf{b} - (\mathbf{L} + \mathbf{U}) \mathbf{x}^{(k)} \right], \quad k = 0, 1, 2, \dots,$$

which in component form reads

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[ b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right], \quad i = 1, 2, \dots, N.$$

If the matrix  $\mathbf{A}$  is positive definite ( $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$ ) or diagonally dominant ( $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \forall i$ ), one can show that this method will always converge to the exact solution.



# Other iterative methods

Sketch of the derivation

Let  $\mathbf{x}$  be the exact solution of  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{x}'$  the approximation and  $\delta\mathbf{x}$  the error

$$\underbrace{\mathbf{x}}_{\text{solution}} = \underbrace{\mathbf{x}'}_{\text{guess}} + \underbrace{\delta\mathbf{x}}_{\text{error}}$$

Substituting this expression into  $\mathbf{Ax} = \mathbf{b}$  gives

$$\begin{aligned}\mathbf{A}(\mathbf{x}' + \delta\mathbf{x}) &= \mathbf{b} \\ \mathbf{A}\delta\mathbf{x} &= \mathbf{b} - \mathbf{Ax}'\end{aligned}$$

This equation is the starting point for an iterative algorithm.

$$\mathbf{A} \underbrace{(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})}_{\delta\mathbf{x}} = \mathbf{b} - \mathbf{A} \underbrace{\mathbf{x}^{(k)}}_{\mathbf{x}'}$$

It is not practical to use this expression since we need  $\mathbf{A}^{-1}$  to find  $\mathbf{x}^{(k+1)}$  when we know  $\mathbf{x}^{(k)}$ .

# Other iterative methods

However, all we need is  $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \xrightarrow[k \rightarrow \infty]{} 0$ .

Hence, we need to find a matrix  $\mathbf{M}$  that also is simple to invert, and replace  $\mathbf{A}$  on the left-hand-side by  $\mathbf{M}$  to get the scheme

$$\mathbf{M} \left( \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right) = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$$

or after solving for  $\mathbf{x}^{(k+1)}$

$$\mathbf{x}^{(k+1)} = \mathbf{M}^{-1} \left[ \mathbf{b} - (\mathbf{A} - \mathbf{M}) \mathbf{x}^{(k)} \right]$$

The simplest choice is  $\mathbf{M} = \mathbf{D}$  and corresponds to the Jacobi iteration scheme.

Other choices for  $\mathbf{M}$  will give rise to other iterative methods (as we will see)!

- Spectral radius

The spectral radius of a matrix  $\mathbf{A}$  is defined as

$$\rho(\mathbf{A}) = \max_{\lambda} |\lambda(\mathbf{A})|$$

- Theorem (Spectral radius and iterative convergence)

If  $\mathbf{A} \in \mathbb{R}^n$ , then

$$\lim_{k \rightarrow \infty} \mathbf{A}^k = \mathbf{0} \iff \rho(\mathbf{A}) < 1$$

## Convergence analysis for the general case

- Linear system  $\mathbf{Ax} = \mathbf{b}$ 
  - Iterative solution :  $\mathbf{x}^{(k+1)} = \mathbf{Bx}^{(k)} + \mathbf{c}$
  - Exact solution :  $\mathbf{x}^* = \mathbf{Bx}^* + \mathbf{c}$
  - $\mathbf{B}$  is known as the **iteration matrix**
- Subtracting the two latter equations gives ( $\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}^*$ )

$$\mathbf{e}^{(k+1)} = \mathbf{B}\mathbf{e}^{(k)}$$

which by induction leads to

$$\mathbf{e}^{(k)} = \mathbf{B}^k \mathbf{e}^{(0)} \iff \|\mathbf{e}^{(k)}\| = \rho^k(\mathbf{B}) \|\mathbf{e}^{(0)}\| \quad (\text{when } k \rightarrow \infty)$$

- This means that

$$\mathbf{e}^{(k)} \rightarrow 0 \iff \mathbf{B}^k \rightarrow 0 \iff \rho(\mathbf{B}) < 1$$

This is the convergence criterion for general iterations.

## Jacobi method

$$x_1^{(k+1)} = (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11}$$

$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22}$$

$$x_3^{(k+1)} = (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33}$$

$$x_4^{(k+1)} = (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},$$

**Idea:** Update each component of  $x_i^{(k)}$  sequentially with the most *updated* information available

$$x_1^{(k+1)} = (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11}$$

$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22}$$

$$x_3^{(k+1)} = (b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)})/a_{33}$$

$$x_4^{(k+1)} = (b_4 - a_{41}x_1^{(k+1)} - a_{42}x_2^{(k+1)} - a_{43}x_3^{(k+1)})/a_{44}$$

This procedure leads to the **Gauss-Seidel method** and improves normally the convergence behavior.

# Gauss-Seidel relaxation

In component form one has

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[ b_i - \sum_{j < i} a_{ij} x_j^{(k+1)} - \sum_{j > i} a_{ij} x_j^{(k)} \right], \quad i = 1, 2, \dots, N.$$

Formally the Gauss-Seidel method corresponds to choosing  $\mathbf{M} = \mathbf{D} + \mathbf{L}$  (see previous equations under the Jacobi method)

$$\begin{aligned} \mathbf{M} \left( \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right) &= \mathbf{b} - \mathbf{A} \mathbf{x}^{(k)} \\ (\mathbf{D} + \mathbf{L}) \mathbf{x}^{(k+1)} &= \mathbf{b} - \mathbf{U} \mathbf{x}^{(k)} \end{aligned} \quad (1)$$

where we in the last transition have recalled that  $\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$ .

- To solve Eq. (1) for  $\mathbf{x}^{(k+1)}$ , one inverts  $\mathbf{D} + \mathbf{L}$  by **forward substitution**
- The iteration matrix is  $\mathbf{G} = (\mathbf{D} + \mathbf{L})^{-1} \mathbf{U}$
- Typically  $\rho_{\text{GS}}(\mathbf{G}) < \rho_{\text{J}}(\mathbf{J})$

# Gauss-Seidel relaxation

When will we have convergence?:

- Both Jacobi and Gauss-Seidel require  $\rho(\mathbf{M}) < 1$  to converge, *i.e.* the spectral radius of the respective iteration matrices is less than one
- For instance, this means that both methods converge when the matrix  $\mathbf{A}$  is symmetric, positive-definite, or is strictly or irreducibly diagonally dominant.
- Both methods sometimes converge even if these conditions are not satisfied.

# Successive Over Relaxation (SOR)

- SOR is an efficient algorithm that should be considered.
- **Idea:**  $\mathbf{x}^{(k+1)}$  is a *weighted average* of  $\mathbf{x}^{(k)}$  and  $\mathbf{x}_{\text{GS}}^{(k+1)}$

$$\mathbf{x}^{(k+1)} = \omega \mathbf{x}_{\text{GS}}^{(k+1)} + (1 - \omega) \mathbf{x}^{(k)} \quad (2)$$

where  $\omega$  is the **relaxation parameter**

- The special case  $\omega = 1$  corresponds to the Gauss-Seidel method
- choose  $\omega$  to accelerate the rate of convergence of the SOR method
- In terms of components we have

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left[ b_i - \sum_{j < i} a_{ij} x_j^{(k+1)} - \sum_{j > i} a_{ij} x_j^{(k)} \right] + (1 - \omega) x_i^{(k)}, \quad i = 1, 2, \dots, N.$$



# Successive Over Relaxation (SOR)

The matrix form of the SOR method is obtained from ( $i = 1, 2, \dots, N$ )

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left[ b_i - \sum_{j < i} a_{ij} x_j^{(k+1)} - \sum_{j > i} a_{ij} x_j^{(k)} \right] + (1 - \omega) x_i^{(k)},$$

by multiplying with  $a_{ii}$  and rearranging to get

$$a_{ii} x_i^{(k+1)} + \omega \sum_{j < i} a_{ij} x_j^{(k+1)} = \omega b_i - \omega \sum_{j > i} a_{ij} x_j^{(k)} + a_{ii} (1 - \omega) x_i^{(k)}$$

which in matrix notation may be expressed as

$$(\mathbf{D} + \omega \mathbf{L}) \mathbf{x}^{(k+1)} = \omega \mathbf{b} - [\omega \mathbf{U} + (\omega - 1) \mathbf{D}] \mathbf{x}^{(k)}.$$

This implies that the iteration matrix for the SOR method is

$$\mathbf{S} = -(\mathbf{D} + \omega \mathbf{L})^{-1} [\omega \mathbf{U} + (\omega - 1) \mathbf{D}].$$

Normally one finds

$$\rho(\mathbf{S}) < \rho(\mathbf{G}) < \rho(\mathbf{J}),$$

*i.e.* the SOR method converges faster than the Gauss-Seidel and Jacobi methods

# Successive Over Relaxation (SOR)

Question: What value to choose for  $\omega$ ?

- The SOR-sequence converges for  $0 < \omega < 2$  [Kahan (1958)]
- Optimal value for  $\omega$

$$\omega = \frac{2}{1 + \sqrt{1 - \rho(\mathbf{J})^2}},$$

but this value is not known in advance

- Frequently, some heuristic estimate is used, such as

$$\omega = 2 - \mathcal{O}(h)$$

where  $h$  is the mesh spacing of the discretization of the underlying physical domain.

# Successive Over Relaxation (SOR)

Alternative, motivation for the SOR iteration formula!

Starting from the linear system

$$\mathbf{Ax} = \mathbf{b},$$

multiplying by  $\omega$  and adding  $\mathbf{Dx}$  to both sides of the equation, leads to

$$(\mathbf{D} + \omega\mathbf{L})\mathbf{x} = \omega\mathbf{b} - [\omega\mathbf{U} + (\omega - 1)\mathbf{D}]\mathbf{x}$$

after some trivial manipulation

This equation is the starting point for an iterative scheme — the SOR method!

# Examples :iterative solvers

Problem: Solve the one-dimensional Poisson equation

$$\nabla^2 \phi(x) = s(x) \equiv x(x + 3) \exp(x) \quad 0 \leq x \leq 1$$

with boundary conditions  $\phi(0) = \phi(1) = 0$ .

Numerical solution : In order to solve this problem numerically we discretize

$$\begin{aligned} x &\rightarrow \{x_n\}_{n=1}^N, & \Delta x &= 1/(N - 1) \\ \phi(x) &\rightarrow \phi(x_n) \equiv \phi_n \end{aligned}$$

Using central differences to approximate the 2nd derivative of the potential at some internal point  $x_n$ , the equation satisfied by the  $\phi_n$ 's is

$$\frac{\phi_{n+1} - 2\phi_n + \phi_{n-1}}{(\Delta x)^2} = s(x_n) \quad n = 2, 3, \dots, N - 1$$

# Examples

Defining the vector  $\mathbf{x} = [\phi_2, \phi_3, \dots, \phi_{N-1}]^T$  results in the **tridiagonal system**

$$\mathbf{Ax} = \mathbf{b},$$

where

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ 0 & -1 & 2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -1 & 2 \end{bmatrix} \quad \mathbf{b} = -(\Delta x)^2 \begin{bmatrix} s(x_2) \\ s(x_3) \\ \vdots \\ \vdots \\ s(x_{N-1}) \end{bmatrix}.$$

It is this system of equations we will need to solve!

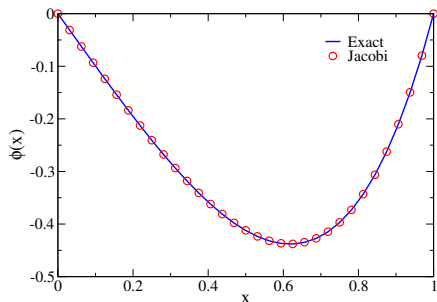
Direct solvers for tridiagonal systems (LAPACK): sgtrfs, dgtrfs, cgtrfs, zgtrfs

# Examples

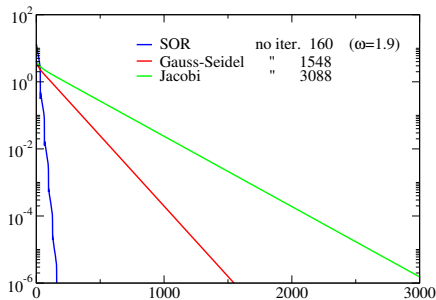
Exact solution

$$\phi(x) = x(x - 1) \exp(x)$$

Solution  $\phi(x)$



No. of iterations



Parameters:  $N = 33$ ; tolerance:  $\|\mathbf{b} - \mathbf{Ax}\| < 10^{-6}$ ;  $\mathbf{x}^{(0)} = \mathbf{0}$

Direct solvers for tridiagonal systems (LAPACK): sgtrfs, dgtrfs, cgtrfs, zgtrfs

# Krylov subspace methods

# Krylov subspace methods

Krylov subspace methods are also known as *conjugate gradient methods*!

What is a Krylov space?

In linear algebra, the order- $r$  Krylov subspace generated by an  $N \times N$  matrix  $\mathbf{A}$  and a vector  $\mathbf{b}$  of dimension  $N$  is the linear subspace spanned by the images of  $\mathbf{b}$  under the first  $r - 1$  powers of  $\mathbf{A}$  (starting from  $\mathbf{A}^0 = \mathbf{I}$ ), that is

$$\mathcal{K}_r(\mathbf{A}, \mathbf{b}) = \text{span} \{ \mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}^2\mathbf{b}, \dots, \mathbf{A}^{r-1}\mathbf{b} \}.$$

Working principle

Krylov subspace methods work by forming a basis of the sequence of successive matrix powers times the initial residual (the Krylov sequence). The approximations to the solution are then formed by minimizing the residual over the subspace formed.

The prototypical method in this class is the [conjugate gradient method \(CG\)](#). Other methods are the [generalized minimal residual method \(GMRES\)](#) and the [biconjugate gradient Stabilized method \(BiCGStab\)](#).



# Steepest descent

Task : Minimize the function  $f(\mathbf{x}_i)$ !

## Philosophy of the Steepest decent method

Start at some point  $\mathbf{x}_0$ . As many times as needed, move from point  $\mathbf{x}_i$  to point  $\mathbf{x}_{i+1}$  by minimizing along the line from  $\mathbf{x}_i$  in the direction of the local downhill gradient  $-\nabla f(\mathbf{x}_i)$

However, as we will see, this simple strategy is not very efficient, but a modified version of the main idea is!

# Steepest descent

The steepest decent strategy applied to the solution of the linear system  $\mathbf{Ax} = \mathbf{b}$

- Assume  $\mathbf{A}$  to be *symmetric* and *positive definite*<sup>7</sup>.
- Define the function  $f(\cdot)$  — a quadratic form

$$\begin{aligned} f(\mathbf{x}) &= \frac{1}{2} \mathbf{x}^T \mathbf{Ax} - \mathbf{x} \cdot \mathbf{b}, & f : \mathbb{R}^N &\rightarrow \mathbb{R}^+ \\ &= \frac{1}{2} \langle \mathbf{x}, \mathbf{Ax} \rangle - \langle \mathbf{x}, \mathbf{b} \rangle \end{aligned}$$

- $f(\mathbf{x})$  has a minimum when (remember  $\mathbf{A}$  is symmetric):

$$\nabla f(\mathbf{x}) = \mathbf{Ax} - \mathbf{b} = \mathbf{0}$$

- The solution of  $\mathbf{Ax} = \mathbf{b}$  corresponds to **the minimum** of  $f$ .

---

<sup>7</sup>That is  $\mathbf{A}$  is symmetric and  $\mathbf{x}^T \mathbf{Ax} > 0$ , or equivalently, all its eigenvalues are positive ( $\forall \mathbf{x} \in \mathfrak{M}_{N,1}^*, \mathbf{x}^T \mathbf{Ax} > 0 \Leftrightarrow \text{Sp}(\mathbf{A}) \subset ]0; +\infty[$ )

# Cauchy steepest descent method

The algorithm

- 1 Choose an initial vector  $\mathbf{x}_0$ .
- 2 If  $\nabla f(\mathbf{x}_k) \neq 0$  move in the *direction* of  $\mathbf{d}_k = -\nabla f(\mathbf{x}_k)$ .
- 3 Stop at the minimum along direction  $\mathbf{d}_k$ , *i.e.* solve:

$$\frac{d}{d\alpha_k} f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) = 0 \quad \Longrightarrow \quad \alpha_k = \frac{\mathbf{d}_k \cdot \mathbf{r}_k}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k} \quad [\mathbf{r}_k = \mathbf{b} - \mathbf{A} \mathbf{x}_k]$$

- 4 Define new minimizer :  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$
- 5 Let  $k \rightarrow k + 1$  and repeat the above steps

Comment : the method can be generalized to matrices  $\mathbf{A}$  that are not symmetric and positive definite. One simply applies the algorithm to the linear system  $\mathbf{A}^T \mathbf{A} \mathbf{x} = \mathbf{A}^T \mathbf{b}$  instead; However, better to use more sophisticated methods (since  $\mathbf{A}^T \mathbf{A}$  is badly conditioned)!

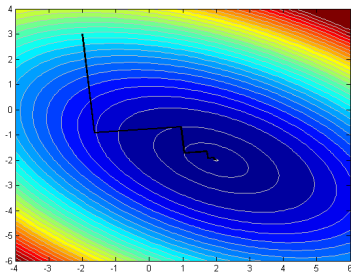
# Cauchy steepest descent method

## Steepest decent

Solving  $\mathbf{Ax} = \mathbf{b}$



Finding the minimum of  $f(\mathbf{x})$



- $\mathbf{A}$  and  $\mathbf{b}$  contain  $N(N + 1)$  parameters
- no. of minimizations  $\sim N^2$

Problem:

- Changes in direction are all *orthogonal* to each other  
otherwise  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$  would not be a minimum along direction  $\mathbf{d}_k = -\nabla f(\mathbf{x}_k)$
- Result : **zig-zag steps**

# “Ordinary” Conjugate Gradient (CG) method

Assumption:  $\mathbf{A}$  is symmetric and positive-definite

Idea: similar to the steepest decent (SD) method, but with more optimal directions  $\mathbf{d}_k$ ; CG uses conjugate directions ( $\mathbf{A}$ -orthogonal)  $\mathbf{d}_i^T \mathbf{A} \mathbf{d}_j = 0$ .

Algorithm :

- 1 Initial guess  $\mathbf{x}_0$ ; get residual  $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$  and define direction  $\mathbf{d}_0 = \mathbf{r}_0$ .
- 2 For  $k = 1, 2, \dots$  do

$$\alpha_k = \frac{\mathbf{r}_k \cdot \mathbf{r}_k}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k \quad (\text{improved minimizer})$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{d}_k \quad (\text{new residual})$$

$$\beta_{k+1} = \frac{\mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}}{\mathbf{r}_k \cdot \mathbf{r}_k}$$

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k \quad (\text{new search direction})$$

- 3 Stop when  $\|\mathbf{r}_{k+1}\| < \varepsilon$  or  $k = N_{\max}$ ; else  $k \rightarrow k + 1$  and repeat step 2

For details see: <http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf>

# “Ordinary” Conjugate Gradient (CG) method

## Comments

- CG completes after  $N$  iterations (exact arithmetic's) [if  $\mathbf{A}$  is positive-definite]
- In practice, set max. no. of interactions to  $2N$  due to roundoff errors
- The search directions are built from the residuals
- Search directions are orthogonal to previous residuals, *i.e.* orthogonal to the (Krylov) space spanned by previous residuals
  - This means that we will find the minimum of  $f(\mathbf{x})$  — the solution to  $\mathbf{Ax} = \mathbf{b}$  — in maximum  $N$  iterations
- Note: If  $\mathbf{A} = \mathbf{A}^T$  is symmetric, but not necessarily positive-definite, the CG method is still found in practice to work well
  - However, the convergence in  $N$  steps is no longer guaranteed (not even in theory)

# “Ordinary” Conjugate Gradient (CG) method

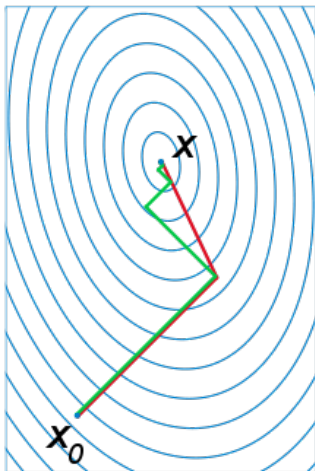
- The vectors  $\mathbf{r}_k$  and  $\mathbf{d}_k$  satisfy (can be shown)

$$\begin{array}{lll} \mathbf{r}_i \cdot \mathbf{r}_j = 0, & i < j & \text{orthogonality} \\ \mathbf{r}_i \cdot \mathbf{d}_j = \mathbf{d}_i \cdot \mathbf{r}_j = 0, & i < j & \text{mutual orthogonality} \\ \mathbf{d}_i^T \mathbf{A} \cdot \mathbf{d}_j = 0, & i < j & \text{conjugacy} \end{array}$$

Better name for the method may have been : Conjugate Directions  
(since the all gradients (or residuals),  $\nabla f(\mathbf{x}_k) = -\mathbf{r}_k$ , are not conjugate)

# “Ordinary” Conjugate Gradient (CG) method

How the CG method works





# Biconjugate Gradient (BiCG) method

Assumption :  $\mathbf{A}$  is a general  $N \times N$  matrix, i.e.  $\mathbf{A} \in \mathfrak{M}_N(\mathbb{F})$  with ,  $\mathbb{F} = \mathbb{R}, \mathbb{C}$

Algorithm :

- 1 Initial guess  $\mathbf{x}_0$ ; get residual  $\mathbf{r}_0$  and define  $\bar{\mathbf{r}}_0 = \mathbf{d}_0 = \bar{\mathbf{d}}_0$ .
- 2 For  $k = 1, 2, \dots$  do

$$\alpha_k = \frac{\bar{\mathbf{r}}_k \cdot \mathbf{r}_k}{\bar{\mathbf{d}}_k^T \mathbf{A} \mathbf{d}_k}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k \quad (\text{improved minimizer})$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{d}_k \quad (\text{new residual})$$

$$\bar{\mathbf{r}}_{k+1} = \bar{\mathbf{r}}_k - \alpha_k \mathbf{A}^T \bar{\mathbf{d}}_k$$

$$\beta_{k+1} = \frac{\bar{\mathbf{r}}_{k+1} \cdot \mathbf{r}_{k+1}}{\bar{\mathbf{r}}_k \cdot \mathbf{r}_k}$$

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k \quad (\text{new search direction})$$

$$\bar{\mathbf{d}}_{k+1} = \bar{\mathbf{r}}_{k+1} + \beta_{k+1} \bar{\mathbf{d}}_k$$

- 3 Stop when  $\|\mathbf{r}_{k+1}\| < \varepsilon$  or  $k = N_{\max}$ ; else  $k \rightarrow k + 1$  and repeat step 2

# Biconjugate Gradient (BiCG) method

- The vectors  $\mathbf{r}_k$ ,  $\mathbf{d}_k$ ,  $\bar{\mathbf{r}}_k$ , and  $\bar{\mathbf{d}}_k$ , (can be shown) to satisfy for  $i < j$

$$\bar{\mathbf{r}}_i \cdot \mathbf{r}_j = \mathbf{r}_i \cdot \bar{\mathbf{r}}_j = 0 \quad \text{biorthogonality}$$

$$\bar{\mathbf{r}}_i \cdot \mathbf{d}_j = \mathbf{r}_i \cdot \bar{\mathbf{d}}_j = 0 \quad \text{mutual orthogonality}$$

$$\bar{\mathbf{d}}_i^T \mathbf{A} \mathbf{d}_j = \mathbf{d}_i^T \mathbf{A}^T \bar{\mathbf{d}}_j = 0 \quad \text{biconjugacy}$$

## Observations

- The algorithm constructs two vector sets  $\{\mathbf{r}_k, \mathbf{d}_k\}$  and  $\{\bar{\mathbf{r}}_k, \bar{\mathbf{d}}_k\}$
- BiCG  $\rightarrow$  CG when  $\mathbf{A} = \mathbf{A}^T$  and  $\bar{\mathbf{r}}_k = \mathbf{r}_k$  and  $\bar{\mathbf{d}}_k = \mathbf{d}_k$
- No equivalent minimization problem exist (like for the CG)
- Even in theory, the iteration scheme is not guaranteed to converge, but this is rare in practice!

# Biconjugate Gradient Stabilized method (BiCGStab)

Recommended modern implementation:

## Biconjugate Gradient Stabilized method (BiCGStab)

A variant of the biconjugate gradient method (BiCG) but with faster and smoother convergence [1,2].

BiCGStab can be viewed as a combination of BiCG and GMRES (to be discussed later) where each BiCG step is followed by a GMRES(1) step.

### References

- 1 H.A. van der Vorst, SIAM J. Sci. Stat. Comput. **13**, 631 (1992) doi:10.1137/0913035
- 2 Y. Saad, (2003). "Iterative Methods for Sparse Linear Systems" (2nd ed.), paragraph 7.4.2 "BICGSTAB", SIAM. pp. 231. (2003) doi:10.2277/0898715342
- 3 See also : [http://en.wikipedia.org/wiki/Biconjugate\\_gradient\\_stabilized\\_method](http://en.wikipedia.org/wiki/Biconjugate_gradient_stabilized_method)

# Minimum residual method (MINRES)

Assumption:  $\mathbf{A}$  symmetric

Discussion: If we try to minimize the function

$$\Phi(\mathbf{x}) = \frac{1}{2} \mathbf{r} \cdot \mathbf{r} = \frac{1}{2} |\mathbf{Ax} - \mathbf{b}|^2,$$

instead of the quadratic form,  $f(\mathbf{x})$ , done in the CG method, the MINRES method results.

Note that

$$\nabla \Phi(\mathbf{x}) = \mathbf{A}^T (\mathbf{Ax} - \mathbf{b}) = -\mathbf{Ar}$$

Algorithm: similar to the BiCG algorithm, but choose

- $\bar{\mathbf{r}}_k = \mathbf{Ar}_k$  and  $\bar{\mathbf{d}}_k = \mathbf{Ad}_k$
- BiCG  $\rightarrow$  CG but with  $\mathbf{a} \cdot \mathbf{b}$  replaced by  $\mathbf{a}^T \mathbf{Ab}$

# Generalized minimum residual method (GMRES)

Assumption :  $\mathbf{A}$  is a general nonsymmetric matrix

## Generalized Minimum Residual method (GMRES)

Generalization and extension of the MINRES to nonsymmetric general matrices. The method is fairly robust!

- GMRES approximates the exact solution of  $\mathbf{Ax} = \mathbf{b}$  by the vector  $\mathbf{x}_k \in \mathcal{K}_k(\mathbf{A}, \mathbf{r}_0)$  that minimizes the Euclidean norm of the residual  $\|\mathbf{r}_k\| = \|\mathbf{b} - \mathbf{Ax}_k\|$ .
- Also GMRES should converge to the solution (in theory) after  $N$  iterations
- The cost of the iterations grows as  $\mathcal{O}(k^2)$ , where  $k$  is the iteration number
- GMRES is therefore often “restarted” after  $m$  steps  $\implies$  GMRES( $m$ )

### References

- 1 Y. Saad, “Iterative Methods for Sparse Linear Systems, 2nd edition, Society for Industrial and Applied Mathematics, 2003.
- 2 See also : [http://en.wikipedia.org/wiki/Generalized\\_minimal\\_residual\\_method](http://en.wikipedia.org/wiki/Generalized_minimal_residual_method)

# Preconditioning

GOAL : Transform the system matrix to a simpler form (closer to the identity matrix), so that fewer iterations will be needed to find the solution

How can this be done?

The system

$$\mathbf{Ax} = \mathbf{b}$$

can be preconditioned with left ( $\mathbf{P}_L$ ) and right ( $\mathbf{P}_R$ ) precondition matrices as

$$\mathbf{P}_L \mathbf{A} \mathbf{P}_R^{-1} \mathbf{P}_R \mathbf{x} = \mathbf{P}_L \mathbf{b}$$

In general, a good preconditioner should meet the following requirements:

- The preconditioned system should be easy to solve
- The preconditioner should be cheap to construct and apply

Ref: Journal of Computational Physics **182**, 418 (2002)

[http://www.mathcs.emory.edu/~benzi/Web\\_papers/survey.pdf](http://www.mathcs.emory.edu/~benzi/Web_papers/survey.pdf)

# Preconditioning

Iterative solvers are typically combined with *preconditioners*

GOAL : to reduce the *condition number* of the left or right preconditioned system matrix

Instead of solving the original linear system  $\mathbf{Ax} = \mathbf{b}$ , one may solve either the right preconditioned system:

$$\mathbf{AP}^{-1}\mathbf{Px} = \mathbf{b}$$

via solving

$$\mathbf{AP}^{-1}\mathbf{y} = \mathbf{b}$$

for  $\mathbf{y}$  and

$$\mathbf{Px} = \mathbf{y}$$

for  $\mathbf{x}$ ; or the left preconditioned system:

$$\mathbf{P}^{-1}(\mathbf{Ax} - \mathbf{b}) = \mathbf{0}$$

both of which give the same solution as the original system as long as the preconditioning matrix  $\mathbf{P}$  is nonsingular.

Examples of preconditioners:

- Jacobi (or diagonal)  
[ $P = \text{diag}(A)$ ]
- Sparse Approximate Inverse
- Incomplete Cholesky factorization
- Incomplete LU factorization
- Successive over-relaxation
- Symmetric successive over-relaxation
- Multigrid
- Fourier Acceleration

Which method to use depend on the problem at hand, and the computational cost of  $P^{-1}$ !

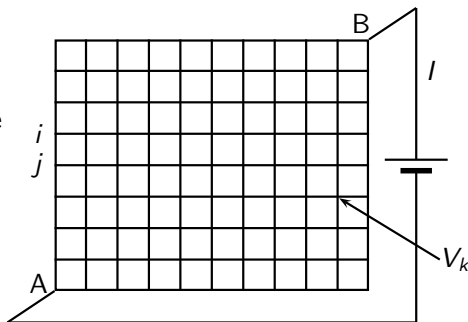
Hence, no general best answer exists!



# Fourier acceleration

Examples : a random resistor network — *Phys. Rev. Lett.* 57, 1336 (1986).

variable conductance  
from link to link:  $g_{ij}$



# Fourier acceleration

The Kirchhoff equations:

$$\sum_j g_{ij}(V_i - V_j) + I_i = 0$$

with

$$I_i = \begin{cases} 0 & \text{if } i \neq A, B \\ I & \text{if } i = A, B \end{cases}$$

On matrix form

$$\mathbf{GV} + \mathbf{I} = 0$$

A very simple iterative process to solve this equation, is to set

$$\frac{dV}{dt} = \mathbf{GV} + \mathbf{I}$$

Solution of the problem as  $t \rightarrow \infty$

$$\frac{dV}{dt} \rightarrow 0$$

Now, imagine all  $g_{ij} = 1$  (they are in general not!) so that we have:

$$\sum_j (V_i - V_j) + I_i = 0$$

Details to be presented on the backboard!

# Recommended methods for solving linear systems

Our recommendations are

- Direct solvers
  - LU-decomposition and backs substitution (use LAPACK/BLAS)!
- Iterative solvers
  - $\mathbf{A} = \mathbf{A}^T$  symmetric
    - Conjugate Gradient (CG)
  - $\mathbf{A}$  is a general matrix
    - Biconjugate Gradient Stabilized method (BiCGStab)
    - Generalized minimum residual method (GMRES)

For iterative solvers, try out various preconditioners!

- LIS — Library of Iterative Solvers for Linear Systems
  - Source : <http://www.netlib.org/misc/lis/>
  - *parallel* library for solving linear equations and eigenvalue problems
  - freely available
  - C and Fortran wrappers exist
- AGMG — Iterative solution with AGgregation-based algebraic MultiGrid
  - Source : <http://agmg.eu/>
  - Fortran 90 with Matlab wrapper
- PETSc — Portable, Extensible Toolkit for Scientific Computation
  - Source : <https://www.mcs.anl.gov/petsc/>
  - Extensive library geared towards the solution of PDEs
  - Contains more than just iterative solvers
  - freely available

A more detailed list of available software can be found at  
<http://www.netlib.org/utk/people/JackDongarra/la-sw.html>.

# Singular value decomposition (SVD)

Question : What to do with  $\mathbf{Ax} = \mathbf{b}$  when  $\mathbf{A}$  is singular?

Lets assume that the linear set of equations is on the form (with  $M \neq N$  in general)

$$\mathbf{Ax} = \mathbf{b}$$

with  $\mathbf{A} \in \mathfrak{M}_{M,N}(\mathbb{F})$ ,  $\mathbf{x} \in \mathfrak{M}_{N,1}(\mathbb{F})$ ,  $\mathbf{b} \in \mathfrak{M}_{M,1}(\mathbb{F})$  all over fields  $\mathbb{F} = \mathbb{R}$  or  $\mathbb{C}$

Question : Could the answer be something like

$$\mathbf{x} = \mathbf{A}^+ \mathbf{b},$$

and if so, what is the meaning of  $\mathbf{A}^+$  (to be called pseudoinverse)

The singular value decomposition (SVD) is useful this context!

# Singular value decomposition (SVD)

## Definition: Singular values

Let  $\mathbf{A} \in \mathbb{C}^{M \times N}$ . A **non-negative real number**  $s$  is a **singular value** for  $\mathbf{A}$  iff there exist unit-vectors  $\hat{\mathbf{u}} \in \mathbb{C}^M$  and  $\hat{\mathbf{v}} \in \mathbb{C}^N$  such that

$$\mathbf{A}\hat{\mathbf{v}} = s\hat{\mathbf{u}} \quad \mathbf{A}^\dagger\hat{\mathbf{u}} = s\hat{\mathbf{v}}.$$

The vectors  $\hat{\mathbf{u}}$  and  $\hat{\mathbf{v}}$  are called left-singular and right-singular vectors for  $s$ , respectively.

The singular values of  $\mathbf{A}$  will be denoted by  $s_1, s_2, \dots, s_{\min(M,N)}$ , and it is customary to list them in decreasing order

$$s_1 \geq s_2 \geq \dots \geq s_{\min(M,N)}$$

Here  $\mathbf{A}^\dagger$  denotes the *conjugate transpose* of  $\mathbf{A}$ , i.e.  $\mathbf{A}^\dagger = (\mathbf{A}^*)^T$

# Singular value decomposition (SVD)

## Relation to eigenvalues

Since  $\mathbf{A}^\dagger \mathbf{A} \mathbf{v} = s^2 \mathbf{v}$  the *singular values* of  $\mathbf{A}$  are also the square roots of the eigenvalues of  $\mathbf{A}^\dagger \mathbf{A}$ .

Since

$$\left(\mathbf{A}^\dagger \mathbf{A}\right)^\dagger = \mathbf{A}^\dagger \mathbf{A}$$

the matrix  $\mathbf{A}^\dagger \mathbf{A}$  is Hermitian and all eigenvalues are real. Therefore, the singular values of any real or complex matrix are real as the definition states!

Technically  $\mathbf{u} = \mathbf{A} \mathbf{v} / s$  so all SVD information can be obtained by solving the eigensystem  $\mathbf{A}^\dagger \mathbf{A} \mathbf{v} = s^2 \mathbf{v}$ .



# Singular value decomposition (SVD)

## Full SVD decomposition

Decompose a matrix  $\mathbf{A} \in \mathfrak{M}_{M,N}(\mathbb{C})$  into the form

$$\mathbf{A} = \mathbf{U} \mathbf{S} \mathbf{V}^\dagger$$

where

$$\mathbf{U}^\dagger \mathbf{U} = \mathbf{I} \quad (\text{unitary}) \quad \mathbf{U} \in \mathbb{C}^{M \times M}$$

$$\mathbf{V}^\dagger \mathbf{V} = \mathbf{I} \quad (\text{unitary}) \quad \mathbf{V} \in \mathbb{C}^{N \times N}$$

$$\mathbf{S} = \text{diag}(s_1, s_2, \dots, s_{\min(M,N)}) \quad \mathbf{S} \in \mathbb{R}^{M \times N}$$

Here  $\mathbf{U}^\dagger$  denotes the *conjugate transpose* or *adjoint* of  $\mathbf{U}$ .

Note : When  $\mathbf{A} \in \mathbb{R}^{M \times N}$ , i.e. is a **real** matrix,  $\mathbf{U}^\dagger = \mathbf{U}^T$  and  $\mathbf{V}^\dagger = \mathbf{V}^T$  are real and orthogonal matrices!

# Singular value decomposition (SVD)

## Warning

Several so-called **reduced SVD decompositions** exist where not *all* details of the **full SVD** decomposition are calculated

The **Thin SVD** (e.g. used by numerical recipes)

$$\mathbf{A} = \mathbf{U}_N \mathbf{S}_N \mathbf{V}^\dagger$$

where  $\mathbf{U}_N \in \mathbb{C}^{M \times N}$ ;  $\mathbf{S}_N \in \mathbb{R}^{N \times N}$ ; and (as before)  $\mathbf{V} \in \mathbb{C}^{N \times N}$ .

## Software

- Numerical Recipes calculates the thin SVD
- LAPACK calculates the full SVD
- Matlab/Octave calculate full SVD
- Python (via numpy) can calculate both full and reduced SVDs

# Singular value decomposition (SVD)

## Comments

- SVD is **always** possible and the decomposition is **unique** (except for permutations of columns in  $\mathbf{U}$  and rows in  $\mathbf{V}^T$ ) and their signs
- All singularities are collected in  $\mathbf{S}$  and  $s_{ji} = s_i$  are the singular values of  $\mathbf{A}$
- For  $\mathbf{A} \in \mathbb{C}^{M \times N}$  there are  $\min(M, N)$  singular values
- If a square matrix  $\mathbf{A}$  is singular, then at least one of its singular values are zero
  
- The columns of  $\mathbf{V}$  are eigenvectors of  $\mathbf{A}^\dagger \mathbf{A}$
- The columns of  $\mathbf{U}$  ( $= \mathbf{A}\mathbf{V}$ ) are eigenvectors of  $\mathbf{A}\mathbf{A}^\dagger$
  
- Numerical complexity of the full SVD-algorithm is  $\mathcal{O}(4MN^2 + 22N^3)$ 
  - This is much more than the complexity of the LU decomp:  $\mathcal{O}(2N^3/3)$

# Singular value decomposition (SVD)

Example : Full SVD decomposition of the  $2 \times 3$  matrix

$$\mathbf{A} = \begin{bmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{bmatrix}$$

The full SVD decomposition of this matrix gives

$$\mathbf{A} = \mathbf{USV}^\dagger$$

with

$$\mathbf{U} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} \sqrt{12} & 0 & 0 \\ 0 & \sqrt{10} & 0 \end{bmatrix} \quad \mathbf{V} = \begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{30}} \\ \frac{2}{\sqrt{6}} & \frac{-1}{\sqrt{5}} & \frac{2}{\sqrt{30}} \\ \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{5}} & \frac{-5}{\sqrt{30}} \\ \frac{1}{\sqrt{6}} & 0 & \frac{1}{\sqrt{30}} \end{bmatrix}$$

# Singular value decomposition (SVD)

Example : Obtaining this result with python (using numpy)

```
>>> import numpy as np
>>> A=np.reshape([3,1,1,-1,3,1],[2,3])
>>> A
array([[ 3,  1,  1],
       [-1,  3,  1]])
```

Performing the *full* SVD decomposition

```
>>> U, S, V = np.linalg.svd(A)
>>> S
array([ 3.46410162,  3.16227766])
>>> U
array([[ -0.70710678, -0.70710678],
       [-0.70710678,  0.70710678]])
```

# Singular value decomposition (SVD)

Example : Using python to calculating the SVD decomposition of a matrix

Initialize numpy and a complex  $9 \times 6$  matrix

```
>>> import numpy as np
>>> a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
```

Reconstruction based on full SVD (convention used by python:  $\mathbf{a} = \mathbf{USV}$ )

```
>>> U, s, V = np.linalg.svd(a, full_matrices=True)
>>> U.shape, V.shape, s.shape
((9, 9), (6, 6), (6,))
>>> S = np.zeros((9, 6), dtype=complex)
>>> S[:6, :6] = np.diag(s)
>>> np.allclose(a, np.dot(U, np.dot(S, V)))      # check  $\mathbf{a} = \mathbf{U} \mathbf{S} \mathbf{V}^H$ 
True
```

Reconstruction based on reduced SVD

```
>>> U, s, V = np.linalg.svd(a, full_matrices=False)
>>> U.shape, V.shape, s.shape
((9, 6), (6, 6), (6,))
>>> S = np.diag(s)
>>> np.allclose(a, np.dot(U, np.dot(S, V)))
True
```

# Singular value decomposition (SVD)

## Example : Using Fortran90 with LAPACK95 for SVD-calculation

```
program la_gesvd_example

  use la_precision, only : wp = dp
  use f95_lapack, only   : la_gesvd

  implicit none
  real(wp),      allocatable :: S(:)
  complex(wp),  allocatable :: A(:, :), U(:, :), VT(:, :)
  .
  .
  ! Allocate storage
  allocate ( A(M,N), AA(M,N), U(M,M), VT(N,N), WW(1:MN-1) )
  .
  .
  write( *, * )'Details of LA_ZGESVD LAPACK Subroutine Results.'
  call la_gesvd( A, S, U, VT, WW, 'N', INFO )
  ! NOTE : V^dagger is returned not V
  .
  .
end program la_gesvd_example
```

### SYNTAX:

```
LA_GESVD( A, S, [U=u], [VT=vt], [WW=ww], [JOB=job], [INFO=info] )
```

# Singular value decomposition (SVD)

## Syntax : Routine LA\_GESVD from LAPACK95 for SVD-calculations

```
SUBROUTINE LA_GESVD / LA_GESDD ( A, S, U=u, VT=vt, WW=ww, JOB=job, INFO=info )
```

```
type(wp),          INTENT(INOUT)          :: A(:, :)
REAL(wp),          INTENT(OUT)            :: S(:)
type(wp),          INTENT(OUT), OPTIONAL :: U(:, :)
type(wp),          INTENT(OUT), OPTIONAL :: VT(:, :)
REAL(wp),          INTENT(OUT), OPTIONAL :: WW(:)
CHARACTER(LEN=1), INTENT(IN),  OPTIONAL :: JOB
INTEGER,           INTENT(OUT), OPTIONAL :: INFO
```

where

```
type ::= REAL / COMPLEX
```

```
wp   ::= KIND(1.0) / KIND(1.0D0)
```

More information available from the LAPACK95 homepage :

<http://www.netlib.org/lapack95/lug95/>



# Singular value decomposition (SVD)

## Syntax : Routine LA\_GESVD from LAPACK95 for SVD-calculations

```
! A      (input/output) REAL or COMPLEX array, shape (:, :) with
!      size(A, 1) = m and size(A, 2) = n.
!      On entry, the matrix A.
!      On exit, if JOB = 'U' and U is not present, then A is
!      overwritten with the first min(m, n) columns of U (the left
!      singular vectors, stored columnwise).
!      If JOB = 'V' and VT is not present, then A is overwritten with
!      the first min(m, n) rows of V^H (the right singular vectors,
!      stored rowwise).
!      In all cases the original contents of A are destroyed.

! WW     Optional (output) REAL array, shape (:) with size(WW) =
!      min(m, n) - 1
!      If INFO > 0, WW contains the unconverged superdiagonal elements
!      of an upper bidiagonal matrix B whose diagonal is in SIGMA (not
!      necessarily sorted). B has the same singular values as A.
!      Note: WW is a dummy argument for LA_GESDD.

! JOB    Optional (input) CHARACTER(LEN=1).
!      = 'N': neither columns of U nor rows of V^H are returned in
!      array A.
!      = 'U': if U is not present, the first min(m, n) columns of U
!      (the left singular vectors) are returned in array A;
!      = 'V': if VT is not present, the first min(m, n) rows of V^H
!      (the right singular vectors) are returned in array A;
!      Default value: 'N'.
```

# Singular value decomposition (SVD)

- $M > N$ 
  - $S(i, :) = 0 \quad \forall i \in [N + 1; M] \quad \text{i.e. some rows are zero}$
- $M < N$ 
  - $S(:, i) = 0 \quad \forall i \in [M + 1; N] \quad \text{i.e. some columns are zero}$
- $M = N$ 
  - If  $\mathbf{A} \in \mathbf{GL}_N(\mathbb{F})$ , i.e.  $\mathbf{A}$  is invertable, then SVD gives

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^\dagger \implies \mathbf{A}^{-1} = \mathbf{V} \underbrace{\mathbf{S}^{-1}}_{\text{diag}(1/s_i)} \mathbf{U}^\dagger$$

- If  $\mathbf{A} \notin \mathbf{GL}_N(\mathbb{F})$ , then at least one  $s_i = 0$  for  $i \in [1, N]$
- Trouble occurs if  $s_i = 0$  (or if  $s_i \approx 0$ , i.e.  $\mathbf{A}$  is ill-conditioned)  
This motivates the introduction of the **condition number**.

## Condition number of matrix $\mathbf{A}$

$$\kappa(\mathbf{A}) = \frac{\max_i(s_i)}{\min_i(s_i)}$$

The number  $\kappa(\mathbf{A})$  measures how singular the matrix is.

Question: When is the matrix  $\mathbf{A}$  singular?

Answer: When the condition number  $\kappa(\mathbf{A})$  is of the order of:

- $\sim 10^6$  for single precision (32 bits)
- $\sim 10^{15}$  for double precision (64 bits)

the matrix  $\mathbf{A}$  is singular (to working precision).

A matrix is singular (numerically) if  $\kappa^{-1}(\mathbf{A})$  is comparable to (say 10 times) the machine precision!

# Singular value decomposition (SVD)

What can we use the SVD decomposition for?

Application of the SVD decomposition

- Othonormal basis for range and null space of  $\mathbf{A}$
- Calculate the condition number of  $\mathbf{A}$
- Moore-Penrose pseudoinverse
- least square calculations

# Singular value decomposition (SVD)

Recall from Linear Algebra

- The range of  $\mathbf{A}$  is the vector space  $\mathcal{R}(\mathbf{A}) = \{\mathbf{b} : \mathbf{Ax} = \mathbf{b}\}$  (row space)
- The null space (or kernel) of  $\mathbf{A}$  is the space  $\mathcal{N}(\mathbf{A}) = \{\mathbf{x} : \mathbf{Ax} = \mathbf{0}\}$
- The **rank** of  $\mathbf{A}$  is the dimension of its range :  $\text{rank}(\mathbf{A}) = \dim(\mathcal{R}(\mathbf{A}))$
- The **nullity** of  $\mathbf{A}$  is the dimension of its null space:  $\text{nullity}(\mathbf{A}) = \dim(\mathcal{N}(\mathbf{A}))$

## The rank-nullity theorem

Let  $\mathbf{A} \in \mathbb{F}^{M \times N}$ , then

$$\text{rank}(\mathbf{A}) + \text{nullity}(\mathbf{A}) = N$$

You will also see the notation:

- $\mathcal{R}(\mathbf{A}) = \text{range}(\mathbf{A})$
- $\mathcal{N}(\mathbf{A}) = \text{null}(\mathbf{A})$

# Singular value decomposition (SVD)

## Orthogonal basis

- Columns of  $\mathbf{U}$  corresponding to  $s_{ij} \neq 0$  span  $\mathcal{R}(\mathbf{A})$
- Columns of  $\mathbf{V}$  corresponding to  $s_{ij} \approx 0$  span  $\mathcal{N}(\mathbf{A})$

- **Non-Singular matrix** : If  $\mathbf{A} \in \mathbf{GL}_N(\mathbb{F})$  then:

$$\begin{aligned}\mathcal{R}(\mathbf{A}) &= \mathbb{F}^N, & \text{rank}(\mathbf{A}) &= N \\ \mathcal{N}(\mathbf{A}) &= \mathbf{0}_{\mathbb{F}^N}, & \text{nullity}(\mathbf{A}) &= 0\end{aligned}$$

- **Singular matrix** : If  $\mathbf{A}$  is singular,  $\mathbf{Ax} = \mathbf{b}$  has no unique solution. However, one may ask for the vector  $\mathbf{x}$  which minimizes the norm  $\|\mathbf{Ax} - \mathbf{b}\|$ . This vector  $\mathbf{x}$  has the form

$$\mathbf{x} = \sum_{i=1}^{\text{nullity}(\mathbf{A})} c_i \mathbf{x}_i^{(0)} + \mathbf{x}_p$$

where  $\mathbf{Ax}_i^{(0)} = \mathbf{0}$  and the “particular solution”  $\mathbf{x}_p$  — “best fit” (least squares) solution to a system of linear equations — is obtained from the so-called *pseudoinverse*  $\mathbf{A}^+$  (to be defined)

$$\mathbf{x}_p = \mathbf{A}^+ \mathbf{b}.$$

# Singular value decomposition (SVD)

## The Moore-Penrose pseudoinverse

A pseudoinverse (or generalized inverse)  $\mathbf{A}^+$  of a matrix  $\mathbf{A}$  is a generalization of the inverse matrix.

If  $\mathbf{A}$  has the SVD decomposition

$$\mathbf{A} = \mathbf{USV}^\dagger$$

then its pseudoinverse  $\mathbf{A}^+$  is obtained from

$$\mathbf{A}^+ = \mathbf{VS}^+ \mathbf{U}^\dagger$$

The pseudoinverse  $\mathbf{S}^+$  of the diagonal matrix  $\mathbf{S}$  is obtained by taking the reciprocal of each non-zero element on the diagonal, leaving the zeros in place and taking the transpose of the result (if  $\mathbf{S}$  is not a square matrix), *i.e.*

$$\mathbf{S} \implies \mathbf{S}^+$$

$$s_{ii} \neq 0 \implies 1/s_{ii}$$

In Matlab and NumPy (numpy.linalg.pinv) the function **pinv** calculates  $\mathbf{A}^+$ !

# Singular value decomposition (SVD)

## Least square solution

If  $\mathbf{Ax} = \mathbf{b}$  then

$$\mathbf{x} = \mathbf{A}^+ \mathbf{b}$$

represents the least-squares approximate solutions that minimized the norm  $\|\mathbf{Ax} - \mathbf{b}\|$ .

This can be used to do linear regression by finding the linear least square solution.



## Section 5

# How to install libraries on a Linux system

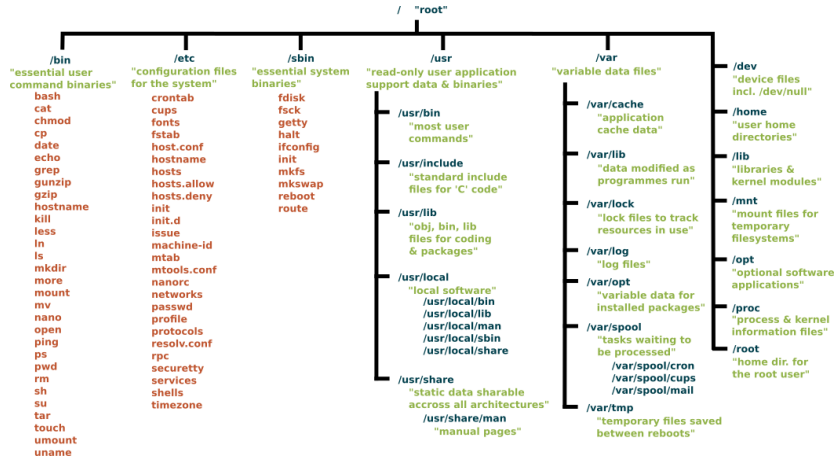
# Outline I

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system**
- 6 Eigenvalue problems
- 7 Spectral methods
- 8 Numerical integration

# Outline II

- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations
- 12 Optimization

# Unix file system



- `\home` : stores user files; mine is usually `/home/ingves`
- `\usr\lib` : system wide libraries (package manager installed)
- `\usr\local\lib` : system wide libraries (manually installed)

# How to install libraries

- Install pre-compiled versions via the *package manager* if it exists
  - the simplest option
  - example : lapack
  - Debian/Ubuntu (deb based): `apt-get install lapack`
  - Fedora (rpm based): `dnf install lapack`
  
- Compile it yourself from source
  - a more challenging option
  - two types of libraries
    - static libraries
    - shared libraries

# Different types of libraries

- **static** libraries

- binary code included in the executable
- Example gcc math library : `libm.a`
- command : `ar rvc "objectfiles"`

- **shared** libraries

- binary code **not** included the executable
- binary code takes from the shared library during execution
- the shared library needs to be installed for program execution
- Ex : `libm.so`

# How to generate, make and use libraries (on Unix)

The main steps to generate a library are

- 1 generate object files by compiling source files
- 2 create the library
  - the method is different for static and shared libraries
- 3 use the library when linking the object files

# How to generate, make and use libraries (on Unix)

Two functions in separate files `add_1.f90` and `add_2.f90`

```
! File add_1.f90  
!  
function add_1(x) result(res)  
  ! Adding one : res = x + 1  
  real, intent(in) :: x  
  real                :: res  
  res = x + 1.  
end function add_1
```

```
! File : add_2.f90  
!  
function add_2(x) result(res)  
  ! Adding two :  
  !   res = x + 1 + 1  
  real, intent(in) :: x  
  real                :: res  
  !  
  !real, external    :: add_1  
  res = add_1(x)  
  res = add_1(res)  
end function add_2
```



# How to generate, make and use libraries (on Unix)

The main program calling the function `add_2 ()`

```
program main
  implicit none
  real                :: x
  real, external    :: add_2
  !
  x = 2.
  write(*,*) " x          = ", x
  write(*,*) " x + 2     = ", add_2(x)
  !
end program main
```

# How to generate, make and use libraries (on Unix)

Compiling and running the program without the use of libraries

```
tux => ls  
add_1.f90  add_2.f90  main.f90
```

```
tux => gfortran -o main add_2.f90 add_1.f90 main.f90
```

```
tux => ls  
add_1.f90  add_2.f90  main  main.f90
```

```
tux => ./main  
x          =      2.00000000  
x + 2      =      4.00000000
```

```
tux =>
```

# How to generate, make and use libraries (on Unix)

## Static library

```
tux => gfortran -c add_1.f90 add_2.f90
```

```
tux => ls  
add_1.f90  add_1.o  add_2.f90  add_2.o  main.f90
```

```
tux => ar rvc libmylib.a  add_1.o add_2.o  
a - add_1.o  
a - add_2.o
```

```
tux => ls  
add_1.f90  add_1.o  add_2.f90  add_2.o  libmylib.a  main.f90
```

```
tux => gfortran -o main main.f90 -L. -lmylib
```

```
tux => ./main  
x      =      2.00000000  
x + 2  =      4.00000000
```

# How to generate, make and use libraries (on Unix)

## Shared library

- 1 make object files with *Position Independent Code* flag set
  - `gfortran -c -fPIC add_1.f90 add_2.f90`
- 2 Creating a shared library from the object file
  - `gfortran -shared -o libMyLib.so add_1.o add_2.o`
- 3 Linking with a shared library
  - `gfortran -o Main main.f90 -L. -lMyLib`
  - However this requires that the compiler can find the shared library `libMyLib.so`
- 4 Making the library available at runtime
  - `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.`

# How to generate, make and use libraries (on Unix)

## Shared library

```
tux => ls  
add_1.f90  add_2.f90  main.f90
```

```
tux => gfortran -c -fPIC add_1.f90 add_2.f90
```

```
tux => ls  
add_1.f90  add_1.o  add_2.f90  add_2.o  main.f90
```

```
tux => gfortran -shared -o libMyLib.so add_1.o add_2.o
```

```
tux => ls  
add_1.f90  add_1.o  add_2.f90  add_2.o  libMyLib.so  main.f90
```

```
tux => gfortran -o Main main.f90 -IMyLib  
/usr/bin/ld: cannot find -IMyLib
```

```
tux => gfortran -o Main main.f90 -L. -IMyLib
```

```
tux => ls  
add_1.f90  add_1.o  add_2.f90  add_2.o  libMyLib.so  Main  main
```

# How to generate, make and use libraries (on Unix)

## Shared library

```
tux => ./Main
./Main: error while loading shared libraries: libMyLib.so:
cannot open shared object file: No such file or directory
```

```
tux => ldd Main
linux-vdso.so.1 (0x00007ffedd91b000)
libMyLib.so => not found
libgfortran.so.3 => /usr/lib/x86_64-linux-gnu/
libgfortran.so.3 (0x00002af52c91f000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0
x00002af52cc45000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
(0x00002af52cf49000)
libquadmath.so.0 => /usr/lib/x86_64-linux-gnu/
libquadmath.so.0 (0x00002af52d160000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0
x00002af52d3a1000)
/lib64/ld-linux-x86-64.so.2 (0x000055d52b067000)
```

```
tux => LD_LIBRARY_PATH=$LD_LIBRARY_PATH:. ./Main
x      =    2.00000000
x + 2  =    4.00000000
```

# How to generate, make and use libraries (on Unix)

## Shared library

```
tux => export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

```
tux => ldd Main
```

```
linux-vdso.so.1 (0x00007ffff4576000)
```

```
libMyLib.so (0x00002b8745f1c000)
```

```
libgfortran.so.3 => /usr/lib/x86_64-linux-gnu/  
libgfortran.so.3 (0x00002b874613d000)
```

```
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0  
x00002b8746463000)
```

```
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0  
x00002b8746767000)
```

```
libquadmath.so.0 => /usr/lib/x86_64-linux-gnu/  
libquadmath.so.0 (0x00002b874697e000)
```

```
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0  
x00002b8746bbf000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x0000558b81992000)
```

```
tux => Main
```

```
x = 2.00000000
```

```
x + 2 = 4.00000000
```

# How to generate, make and use libraries (on Unix)

Compile your own library from source

Example : OpenBLAS — An optimized BLAS (and LAPACK) library

[urlwww.openblas.net/](http://www.openblas.net/)

- 1 Downloaded the latest version of the library from github

```
git clone https://github.com/xianyi/OpenBLAS.git
```

- 2 Generation of serial library, do

```
make USE_THREAD=0 FC=gfortran
```

- 3 Installed the library to `~/Tmp/OpenBLAS` (or whatever you prefer)

```
make PREFIX=~/Tmp/OpenBLAS/ install
```

- 4 (optional) OpenBLAS contains both the BLAS and LAPACK.

- However, most of my Makefiles have separate BLAS and LAPACK libraries. Therefore, for compatibility reasons, I prefer to make the following two links in the installation directory

```
• ln -s libopenblas.a libblas.a
```

```
• ln -s libopenblas.a liblapack.a
```

For more details see <https://pleiades.ucsc.edu/hyades/OpenBLAS>



## Section 6

# Eigenvalue problems

# Outline I

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems**
- 7 Spectral methods
- 8 Numerical integration

# Outline II

- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations
- 12 Optimization

# Eigenvalue problems

## Definition: Eigenvalues and eigenvectors

If for a matrix  $\mathbf{A} \in \mathbb{F}^{N \times N}$ , a vector  $\mathbf{x} \in \mathbb{F}^N$ , and a scalar  $\lambda \in \mathbb{F}$

$$\mathbf{Ax} = \lambda\mathbf{x}, \quad \mathbf{x} \neq \mathbf{0}$$

then  $\lambda$  is an *eigenvalue* with corresponding *eigenvector*  $\mathbf{x}$  of the matrix  $\mathbf{A}$  ( $\mathbb{F} = \mathbb{C}$  or  $\mathbb{R}$ ).

The above definition concerns the *right eigenvalue problem*; similarly, a left eigenvector is defined as a row vector  $\mathbf{x}_L$  satisfying  $\mathbf{x}_L\mathbf{A} = \lambda\mathbf{x}_L$ .

Unless otherwise stated, the term eigenvector will here mean the right eigenvector.

- The above right eigenvector problem is equivalent to a set of  $N$  equations in  $N$  unknowns  $\mathbf{x}_i$  (and unknown  $\lambda_i$ ).
- The eigenvalue problem consists in finding all  $N$  **eigenvalues** (distinct or not) and the corresponding **eigenvectors**.

# Eigenvalue problems

The eigenvalue problem can be rewritten as

$$(\mathbf{A} - \lambda \mathbf{I}) \mathbf{x} = 0,$$

where  $\mathbf{I}$  denotes the identity matrix.

This equation provides a solution to the problem if and only if the determinant is zero, namely

$$|\mathbf{A} - \lambda \mathbf{I}| = 0,$$

which in turn means that the determinant is a polynomial of degree  $N$  in  $\lambda$  and in general we will have  $N$  distinct zeros.

# Eigenvalue problems

The eigenvalues of a matrix  $\mathbf{A} \in \mathbb{C}^{N \times N}$  are thus the  $N$  roots of its characteristic polynomial

$$P(\lambda) = \det(\lambda \mathbf{I} - \mathbf{A}),$$

or

$$P(\lambda) = \prod_{i=1}^N (\lambda_i - \lambda).$$

The set of these roots is called the **spectrum** of  $\mathbf{A}$  and is denoted as  $\lambda(\mathbf{A})$ . If  $\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \dots, \lambda_N\}$  then we have

$$\det(\mathbf{A}) = \lambda_1 \lambda_2 \dots \lambda_N,$$

and if we define the trace of  $\mathbf{A}$  as

$$\text{Tr}(\mathbf{A}) = \sum_{i=1}^n a_{ii}$$

then  $\text{Tr}(\mathbf{A}) = \lambda_1 + \lambda_2 + \dots + \lambda_n$ .

# Eigenvalue problems

How can we numerically solve the eigenvalue problem?

To illustrate the procedure, we will in the following assume (for simplicity) that our matrix is real and symmetric, that is,  $\mathbf{A} \in \mathbb{R}^{N \times N}$  and  $\mathbf{A} = \mathbf{A}^T$ . The matrix  $\mathbf{A}$  has  $N$  eigenvalues  $\lambda_1 \dots \lambda_N$  (distinct or not). Let  $\mathbf{D}$  be the diagonal matrix with the eigenvalues on the diagonal

$$\mathbf{D} = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \lambda_{n-1} & \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_N \end{pmatrix} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_N).$$

If  $\mathbf{A}$  is real and symmetric then there exists a real orthogonal matrix  $\mathbf{S}$  such that

$$\mathbf{S}^{-1} \mathbf{A} \mathbf{S} = \mathbf{S}^T \mathbf{A} \mathbf{S} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_N),$$

and for  $j \in [1, N]$  we have  $\mathbf{A} \mathbf{S}(:, j) = \lambda_j \mathbf{S}(:, j)$ .

# Eigenvalue problems

We say that a matrix  $\mathbf{B}$  is a *similarity transform* of  $\mathbf{A}$  if

$$\mathbf{B} = \mathbf{S}^{-1} \mathbf{A} \mathbf{S},$$

Similarity transformations preserves the eigenvalues

If matrices  $\mathbf{A}$  and  $\mathbf{B}$  are related by the similarity transformation  $\mathbf{A} = \mathbf{S} \mathbf{B} \mathbf{S}^{-1}$ , then

$$\lambda(\mathbf{A}) = \lambda(\mathbf{B}),$$

that is, the two matrices have the same eigenvalues.

Note: The eigenvectors of  $\mathbf{A}$  and  $\mathbf{B}$  are in general different!



# Eigenvalue problems

To prove this we start with the eigenvalue problem and a similarity transformed matrix  $\mathbf{B}$ .

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \quad \text{and} \quad \mathbf{B} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S}.$$

We multiply the first equation on the left by  $\mathbf{S}^{-1}$  and insert  $\mathbf{S}^{-1}\mathbf{S} = \mathbf{I}$  between  $\mathbf{A}$  and  $\mathbf{x}$ . Then we get

$$(\mathbf{S}^{-1}\mathbf{A}\mathbf{S})(\mathbf{S}^{-1}\mathbf{x}) = \lambda\mathbf{S}^{-1}\mathbf{x}, \quad (3)$$

which is the same as

$$\mathbf{B}(\mathbf{S}^{-1}\mathbf{x}) = \lambda(\mathbf{S}^{-1}\mathbf{x}).$$

The variable  $\lambda$  is an eigenvalue of  $\mathbf{B}$  as well, but with eigenvector  $\mathbf{S}^{-1}\mathbf{x}$ .

**Note :** The above holds true for any invertible matrix  $\mathbf{S}$  not only an orthogonal matrix.

# Eigenvalue problems

Strategy for obtaining the eigenvalues of  $\mathbf{A} \in \mathbb{F}^{N \times N}$  numerically

- 1 Perform a series of **similarity transformations** on the original matrix  $\mathbf{A}$ , in order to reduce it either into a **diagonal form** or into a **tridiagonal form**.
- 2 Thereafter, use specialized routines for tridiagonal systems!

Various slightly different algorithms are used to find the eigenvalues and eigenvectors of the matrix  $\mathbf{A}$  depending on its properties, *e.g.* depending on the matrix being

- real symmetric / real non-symmetric
- Hermitian / non-Hermitian
- tridiagonal
- upper / lower Hessenberg
- etc.

# Eigenvalue problems

In short, the basic philosophy of finding eigenvalues/eigenvectors are:

- either apply subsequent similarity transformations (direct method) so that

$$\mathbf{S}_N^T \dots \mathbf{S}_1^T \mathbf{A} \mathbf{S}_1 \dots \mathbf{S}_N = \mathbf{D},$$

- or apply subsequent similarity transformations so that  $\mathbf{A}$  becomes tridiagonal (Householder) or upper/lower triangular (**QR** method). Thereafter, techniques for obtaining eigenvalues from tridiagonal matrices can be used.
- or use so-called power methods
- or use iterative methods (Krylov, Lanczos, Arnoldi). These methods are popular for huge matrix problems.

# Eigenvalue problems

General recommendation for numerically solving eigenvalue problems

- Dense matrices of order “ $N \leq 10^5$ ”
  - use library routines (LAPACK)
  - Computational complexity:  $\mathcal{O}(N^3)$  when  $\mathbf{A} \in \mathbb{F}^{N \times N}$
- Sparse matrices/dense (“ $N \geq 10^5$ ”)
  - use iterative techniques (ARPACK and TRLAN; FORTRAN libraries)
    - Lanczos
    - Arnoldi
    - Krylov

The field of solving eigensystems is a complex matter, so we will not cover the rather technical details here!

# Eigenvalue problems

**Direct or non-iterative methods** require  $\mathcal{O}(N^3)$  operations for an  $N \times N$  matrix.

Historic overview:

Year	$N$	Library
1950	20	(Wilkinson)
1965	200	(Forsythe et al.)
1980	2000	Linpac
1995	20000	Lapack
2012	$\sim 10^5$	Lapack

Ratio of computational complexity :

$$\frac{N_{2012}^3}{N_{1950}^3} \sim (10^4)^3 = 10^{12}$$

Progress in computer hardware during period 1950–2012

$$\sim 1 \text{ flop (1950)} \longrightarrow \text{petaflops} = 10^{15} \text{ flops (2012)}$$

# Eigenvalue problems

## Iterative methods for eigensystems

- If the matrix to diagonalize is large and sparse, direct methods simply become impractical, also because many of the direct methods tend to destroy sparsity.
- As a result large dense matrices may arise during the diagonalization procedure
- The idea behind iterative methods is to project the  $N$ -dimensional problem onto a lower dimensional vector spaces, so-called Krylov subspaces

Matrix	$\mathbf{Ax} = \mathbf{b}$	$\mathbf{Ax} = \lambda \mathbf{x}$
$\mathbf{A} = \mathbf{A}^\dagger$	Conjugate gradient	Lanczos algorithm
$\mathbf{A} \neq \mathbf{A}^\dagger$	GMRES etc	Arnoldi algorithm

## Software for eigenvalue calculations

- **LAPACK** : <http://www.netlib.org/lapack/>
- **LAPACK95** : <http://www.netlib.org/lapack95/>
- **ARPACK** : <http://www.caam.rice.edu/software/ARPACK/> or <https://en.wikipedia.org/wiki/ARPACK>
- Efficient special routines exist for calculating only the largest eigenvalues (and corresponding eivenectors); see ARPACK for details

# Eigenvalue problems

## Generalized Eigensystems

Let  $\mathbf{A}$  and  $\mathbf{B}$  be matrices of the compatible dimensions.

Then following system is referred to as the **generalized eigensystem**

$$\mathbf{Ax} = \lambda \mathbf{Bx}$$

Technically it is equivalent to solving  $\mathbf{B}^{-1}\mathbf{Ax} = \lambda \mathbf{x}$ , but  $\mathbf{B}^{-1}$  is computationally expensive to obtain.

These problems are solved by LAPACK in an efficient way (without first calculating  $\mathbf{B}^{-1}$ )!



The remaining slides of this section were not covered in the lectures

# Largest eigenvalue (not covered in the lectures)

- If only one eigenvalue is to be determined — or just a few — including corresponding eigenvectors, one may use simple iterative methods.
- We assume  $\text{Sp}(A) \subset \mathbb{R}$  for simplicity. Let  $\lambda_m = \max(\text{Sp}(A))$ .

## Iteration scheme

$$X'_k = AX_{k-1}$$

$$X_k = \frac{X'_k}{\|X'_k\|}$$

- We will denote  $V_i$  a normalized eigenvector associated to the eigenvalue  $\lambda_i$ .

# Largest eigenvalue

Start with a decomposition on  $(V_i)_{i \in \llbracket 1; N \rrbracket}$ :

$$X_{k-1} = \sum_{i=1}^N c_{k-1}^i V_i$$

$$X'_k = AX_{k-1} = \sum_{i=1}^N c_{k-1}^i \lambda_i V_i$$

$$X_k = \frac{X'_k}{\|X'_k\|} = \sum_{i=1}^N c_k^i \lambda_i V_i$$

$$c_k^i = \frac{c_{k-1}^i}{\sqrt{\sum_{i=1}^N (c_{k-1}^i \lambda_i)^2}}$$

$$c_k^i \xrightarrow{k \rightarrow \infty} \delta_{im}$$

when  $\lambda_m = X_{k-1} \cdot X'_k$ ,  $X_k = V_m$ .

# Determining any eigenvalue

We still assume  $\text{Sp}(A) \subset \mathbb{R}$ .

$$(A - \lambda I_N)^{-1} V_j = \frac{1}{\lambda_j - \lambda} V_j$$

## Iteration scheme

$$X'_k = (A - \lambda I_N)^{-1} X_{k-1}$$

$$X_k = \frac{X'_k}{\|X'_k\|}$$

Converges to the eigenvalue closest to  $\lambda$

$$\lambda_j = \lambda + \frac{1}{X_{k-1} \cdot X'_k}$$

# Lambert-Weaire's algorithm (not covered in the lectures)

*Lambert and Wearie, Phys. Stat. Solidii B, 101, 591 (1980)*

*Burton and Lambert, Europhys. Lett. 5, 161 (1988)*

- The problem is still to find  $AV_i = \lambda_i V_i$ .
- This is an algorithm to count the number of eigenvalues being smaller than some specific value  $\lambda$ .
- The aim is to eliminate one by one the  $N$  directions of the original problem, by transforming the  $N \times N$  matrix  $A$  into a  $(N - 1) \times (N - 1)$  matrix  $A'$  having  $\lambda_i, i \in \llbracket 1; N - 1 \rrbracket$  as eigenvalues.

# Lambert-Weaire's algorithm

Example:  $N = 2$

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \lambda \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

The system of equations reads:

$$(a_{11} - \lambda)v_1 + a_{12}v_2 = 0$$

$$a_{21}v_1 + (a_{22} - \lambda)v_2 = 0$$

We eliminate  $v_2$ .

$$(a_{11} - \lambda)v_1 - \frac{a_{12}a_{21}}{a_{22} - \lambda}v_1 = 0$$

We rewrite this as

$$\left( a_{11} - \frac{a_{12}a_{21}}{a_{22} - \lambda} \right) v_1 = \lambda v_1$$

This is now a  $1 \times 1$  eigenvalue problem with  $A' = a_{11} - \frac{a_{12}a_{21}}{a_{22} - \lambda}$ .

# Lambert-Weaire's algorithm

The  $N \times N$  case.  $\forall i \in \llbracket 1, N \rrbracket$ ,

$$\sum_{j=1}^N a_{ij} v_j = \lambda v_i$$

We eliminate  $v_N$  and find

$$\sum_{j=1}^{N-1} a'_{ij} v_j = \lambda v_i$$

where  $a'_{ij} = a_{ij} - \frac{a_{iN} a_{Nj}}{a_{NN} - \lambda}$ . We do this recursively:

(\*)

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{i,N-k+1}^{(k-1)} a_{N-k+1,j}^{(k-1)}}{a_{N-k+1,N-k+1}^{(k-1)} - \lambda}$$

# Lambert-Weaire's algorithm

If we now choose a value  $\lambda$ , the **negative eigenvalue theorem** states that the number of eigenvalues verifying  $\lambda_i < \lambda$  is equal to the number of times the denominator in (\*) -  $(a_{N-k+1, N-k+1}^{(k-1)} - \lambda)$  - has been negative for  $k = 1, \dots, N$ . This is a simple, powerful, and largely forgotten algorithm!



# Section 7

## Spectral methods

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods**
  - Fourier transform
  - Wavelet transform

# Outline II

- 8 Numerical integration
- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations
- 12 Optimization

# Fourier Transform

The Fourier transform is employed to transform signals between the time (or spatial) domain and the frequency (wave vector) domain

$$f(t) \xrightarrow{\mathcal{F}} \hat{f}(\omega) = \mathcal{F}[f](\omega)$$

The formal definition reads:

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} dt f(t) e^{i\omega t} \quad (\text{forward transform})$$

$$f(t) = \int_{-\infty}^{\infty} \frac{d\omega}{2\pi} \hat{f}(\omega) e^{-i\omega t} \quad (\text{inverse transform})$$

**Note:** there exist numerous conventions regarding how the factor of  $2\pi$  are distributed among the forward and backward transform. Check your implementation!

# Discretely sampled data

- $f(t)$  is sampled evenly at  $\Delta$  time intervals [ $f_k \equiv f(t_k)$  with  $t_k = k\Delta$ ]
- Nyquist critical frequency:

$$\omega_c \equiv \frac{\pi}{\Delta} \qquad \left[ f_c \equiv \frac{1}{2\Delta} \right]$$

- The Nyquist-Shannon sampling theorem:  
If a continuous function  $f(t)$ , sampled at interval  $\Delta$ , is *bandwidth limited* to frequencies smaller in magnitude than  $\omega_c$ , i.e., if  $\hat{f}(\omega) = 0$  for all  $|\omega| > \omega_c$ , then  $f(t)$  is completely *determined* by its samples  $\{f_k\}_{k=0}^{N-1}$ .
- If the sampled function is *not* bandwidth limited to less than the Nyquist critical frequency, then the amplitudes for the frequencies outside this interval is spuriously moved into that range. This is called *aliasing*.

# Discrete Fourier Transform

- Assume  $N$  consecutive sampled values, with sampling interval  $\Delta$ ,

$$f_k \equiv f(t_k), \quad t_k \equiv k\Delta, \quad k = 0, 1, 2, \dots, N-1.$$

where  $N$  is assumed even (for simplicity).

- The discrete Fourier transform can then be used to find the Fourier transform at frequencies:

$$\omega_n \equiv 2\pi \frac{n}{N\Delta}, \quad n = -\frac{N}{2}, \dots, \frac{N}{2}.$$

- The inverse Fourier transform,  $\hat{f}(\omega_n)$ , can then be approximated by a discrete sum:

$$\hat{f}(\omega_n) = \int_{-\infty}^{\infty} dt f(t) e^{i\omega_n t} \approx \sum_{k=0}^{N-1} \Delta f_k e^{i\omega_n t_k} = \Delta \sum_{k=0}^{N-1} f_k e^{2\pi i k n / N}$$

# Discrete Fourier Transform

- The summation in (the continuous Fourier transform)

$$\hat{f}(\omega_n) \approx \Delta \sum_{k=0}^{N-1} f_k e^{2\pi i k n / N}$$

is known as the discrete Fourier transform (DFT)

$$\hat{f}_n = \sum_{k=0}^{N-1} f_k e^{2\pi i k n / N} \quad \left[ \hat{f}(\omega_n) \approx \Delta \hat{f}_n \right]$$

- *Inverse* discrete Fourier transform (IDFT):

$$f_k = \frac{1}{N} \sum_{n=0}^{N-1} \hat{f}_n e^{-2\pi i k n / N}$$

Note : the prefactor  $1/N$  is not always included in the IDFT  
(normalized/non-normalized IDFT)

# Discrete Fourier Transform

How much computation is involved in doing the discrete Fourier transform?

$$W_N \equiv e^{2\pi i/N} \quad \implies \quad \hat{f}_n = \sum_{k=0}^{N-1} W_N^{nk} f_k$$

- The vector of  $f_k$ 's is multiplied by a matrix with  $(n,k)$ th element equal  $W_N$  to the power  $nk$ , resulting in the vector of  $\hat{f}_n$ 's. This matrix multiplication, and therefore the discrete Fourier transform, is an  $\mathcal{O}(N^2)$  process.
- Using an algorithm called the *fast Fourier transform* (FFT), it is possible to compute the discrete Fourier transform in  $\mathcal{O}(N \log_2 N)$  operations.



# Fast Fourier Transform (FFT)

The basic idea behind the (radix-2) FFT<sup>8</sup> is to write the Fourier transform over  $N$  points as two Fourier transforms over  $N/2$  points.<sup>9</sup>

One way this can be derived is through the Danielson-Lanczos lemma:

$$\begin{aligned}\hat{f}_n &= \sum_{k=0}^{N-1} f_k W_N^{nk}, & W_N &= e^{2\pi i/N} \text{ (N'th root of unity)} \\ &= \sum_{k=0}^{N/2-1} f_{2k} W_N^{n(2k)} + \sum_{k=0}^{N/2-1} f_{2k+1} W_N^{n(2k+1)} \\ &= \sum_{k=0}^{N/2-1} f_{2k} W_{N/2}^{nk} + W_N^n \sum_{k=0}^{N/2-1} f_{2k+1} W_{N/2}^{nk} \\ &= \hat{f}_n^e + W_N^n \hat{f}_n^o\end{aligned}$$

---

<sup>8</sup>For more info see Wikipedia.

<sup>9</sup>The FFT-method was popularized by a publication of J. W. Cooley and J. W. Tukey in 1965, but it was later (1984) discovered that those two authors had independently re-invented an algorithm known to Carl Friedrich Gauss around 1805.

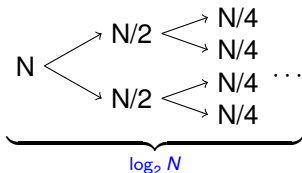
# Fast Fourier Transform (FFT)

Hence

$$\hat{f}_n = \hat{f}_n^e + W_N^n \hat{f}_n^o$$

- $\hat{f}_n^e$ : even components, length  $N/2$
- $\hat{f}_n^o$ : odd components, length  $N/2$

Can be used recursively until the problem is reduced to finding the Fourier transform of length 1 (assume  $N = 2^p$ ,  $p \in \mathbb{Z}^+$ ):

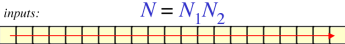


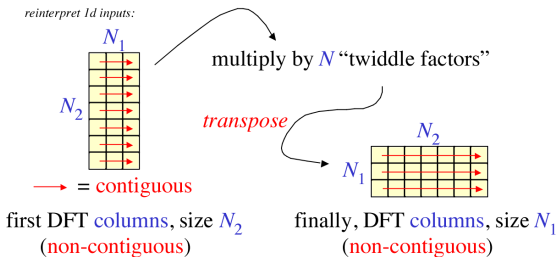
Each layer requires  $N$  operations, and there is a total of  $\log_2 N$  layers resulting in  $N \log_2 N$  operations.

# Cooley-Tukey FFT algorithm

Generalization of the algorithm above is:

- Perform  $N_1$  DFTs of size  $N_2$ .
- Multiply by complex roots of unity called twiddle factors.
- Perform  $N_2$  DFTs of size  $N_1$ .

1d DFT of size  $N$ :   
 $= \sim$  2d DFT of size  $N_1 \times N_2$



# Examples FFT

numpy: real-to-complex

```
>>> import numpy as np
>>> x=np.random.rand(8)
>>> x
array ([ 0.54528432,  0.53861951,  0.44141222,  0.30055361,
         0.66794224,  0.53216542,  0.78834189,  0.77609837])

>>> X=np.fft.rfft(x)
>>> X
array ([ 4.59041758+0.j           ,  0.21816672+0.67862686j ,
        -0.01652755+0.00586704j , -0.46348258-0.01523247j ,
         0.29554377+0.j           ])

>>> np.fft.irfft(X)
array ([ 0.54528432,  0.53861951,  0.44141222,  0.30055361,
         0.66794224,  0.53216542,  0.78834189,  0.77609837])

>>> x.sum()
4.5904175817116109
```

The transform `np.fft.rfft(x)` is normalized!

# Examples FFT

numpy: complex-to-complex

```
>>> import numpy as np
>>> x=np.random.rand(8)
>>> x
array ([ 0.54528432,  0.53861951,  0.44141222,  0.30055361,
         0.66794224,  0.53216542,  0.78834189,  0.77609837])

>>> X=np.fft.fft(x)
>>> X
array ([ 4.59041758+0.j           ,  0.21816672+0.67862686j ,
        -0.01652755+0.00586704j , -0.46348258-0.01523247j ,
         0.29554377+0.j           , -0.46348258+0.01523247j ,
        -0.01652755-0.00586704j ,  0.21816672-0.67862686j ] )

>>> np.fft.ifft(X) # removed the last two digits
array ([ 0.545284 +0.000000e+00j ,  0.538619 -1.509072e-17j ,
         0.441412 +1.387778e-16j ,  0.300553 +1.052963e-16j ,
         0.667942 +0.000000e+00j ,  0.532165 +1.266485e-17j ,
         0.788341 -1.387778e-16j ,  0.776098 -1.028704e-16j ] )
```

# Fast Fourier Transform (FFT)

Depending on implementation, FFTs may be

- normalized
- unnormalized

FFTs exists in different flavors

- real-to-complex
- complex-to-complex

Before using an FFT-library, check its

- normalization convention (*i.e.* factors of  $2\pi$ )
- sign convention
- storage convention

Figuring out the conventions used can be a pain!

Use a library for doing FFTs!

Software for doing FFTs:

- **FFTW** : Fastest Fourier Transform in the west — [www.fftw.org/](http://www.fftw.org/)
- **FFTPACK** : [www.netlib.org/fftpack/](http://www.netlib.org/fftpack/)
- **GSL**: GNU scientific Library — [www.gnu.org/software/gsl/](http://www.gnu.org/software/gsl/)
- **SLATEC** : [www.netlib.org/slatec/](http://www.netlib.org/slatec/)
- **scipy** : [www.netlib.org/slatec/](http://www.netlib.org/slatec/)

# Why are Fourier transforms so useful?

- Physical description is sometimes simpler in the Fourier domain (FD)
- Some mathematical operations are simpler in the FD (e.g. ODEs)
- Periodic signals are often of interest.
- Harmonic dependence/Superposition
- Some numerical operations can be done much faster via FFTs (e.g. convolutions)



Typical situation in experiments:

Measure a signal  $s(t)$  with a given tool to produce the measured signal  $m(t)$ . Ideally  $s(t)$  and  $m(t)$  should be equal but the measuring tool has a *response function*  $r(t)$  so that

$$m(t) = (r * s)(t) = \int_{-\infty}^{\infty} d\tau r(\tau) s(t - \tau)$$

- $m(t)$  — measured signal [ $m(t) \neq s(t)$ ]
- $s(t)$  — signal to measure
- $r(t)$  — response function of the tool

Question : How can we obtain  $s(t)$  from the knowledge of  $m(t)$ ?

Convolution theorem

$$\mathcal{F}[r * s] = \mathcal{F}[r]\mathcal{F}[s] = \hat{r}(\omega)\hat{s}(\omega)$$

The convolution operation

$$(r * s)(t) = \int_{-\infty}^{\infty} d\tau r(\tau) s(t - \tau)$$

can be discretized and defined like:

$$s(t) \rightarrow s_j, \quad r(t) \rightarrow r_j,$$

$$(r * s)_j \equiv \sum_{k=-M/2+1}^{M/2} r_k s_{j-k}.$$

Here  $M$  is the finite duration of the response function.

$$r_k \neq 0 \text{ for } -M/2 \leq k \leq M/2.$$

# Convolution - Interpretation of $r_j$

Input signal  $s_j$  in channel  $j$ :

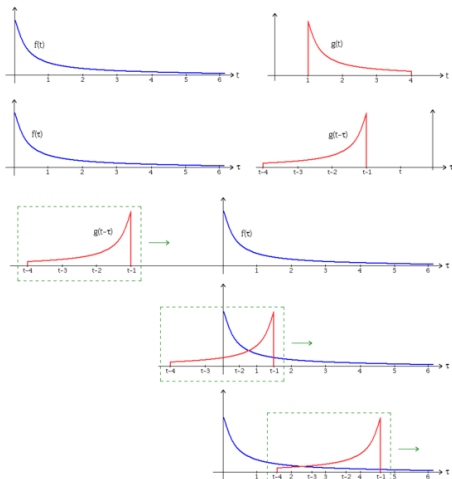
- $r_0$  gives how much of  $s_j$  remains in channel  $j$ .
- $r_1$  gives how much of  $s_j$  is sent into channel  $j + 1$ .
- And so on...

Identity response:

$$r_0 = 1, r_k = 0, k \neq 0$$

# Convolution - Interpretation of $r_j$

$$(f * g)(t) = \int_{-\infty}^{\infty} d\tau f(\tau) g(t - \tau)$$



# Discrete Convolution Theorem

The *discrete convolution theorem*: If a signal  $s_j$  is periodic with period  $N$ , so that it is completely determined by the  $N$  values  $s_0, \dots, s_{N-1}$ , then its discrete convolution with a response function of *finite duration*  $N$  is a member of the discrete Fourier pair,

$$\sum_{k=-N/2+1}^{N/2} r_k s_{j-k} \iff S_n R_n$$

- $S_n$ : is the discrete Fourier transform of the values  $s_j$  ( $j = 0, \dots, N - 1$ )
- $R_n$ : is the discrete Fourier transform of the values  $r_k$  ( $k = 0, \dots, N - 1$ )

# Treatment of end effects by zero padding

- Almost always, one is interested in a response function whose duration  $M$  is much shorter than the length of the data set  $N$ . In this case one can simply extend the response function to length  $N$  by padding it with zeros.
- If  $s$  is not periodic with period  $N$ , then data from “higher” channels will be folded into the 0 channel. This can be fixed by adding zero padding:

$$s_k = 0, k = N + 1, \dots, N + M.$$

# Deconvolution

To try and undo the smearing done to a dataset from a known response function, the function

$$\underbrace{(r * s)_j}_{\text{known}} \equiv \sum_{k=-N/2+1}^{N/2} \underbrace{r_k}_{\text{known}} \underbrace{s_{j-k}}_{\text{unknown}}$$

is solved with respect to  $s_j$ . This is a set of simultaneous linear equations that is unrealistic to solve in most cases.

- The problem can be solved very quickly using the FFT. Just divide the transform of the convolution by the transform of the response to get the transform of the deconvoluted signal.
- This procedure won't work if  $R_n = 0$ . This indicated that the original convolution has lost all information at that one frequency. The process is also generally sensitive to noise in the input and to how well the response  $r_k$  is known.

# Wavelet transform

To be written!



# Section 8

## Numerical integration

# Outline I

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods
- 8 Numerical integration**
  - One-dimensional integration

# Outline II

- Multidimensional integrals

9 Random numbers

10 Ordinary differential equations

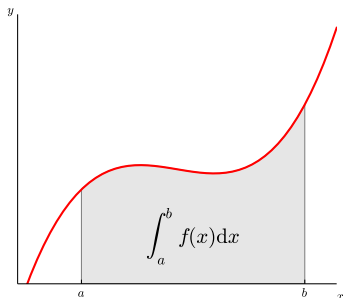
11 Partial differential equations

12 Optimization

# Numerical integration

## Main classes of integration routines

- 1 One-dimensional numerical integration (or "quadratures")
  - Newton-Cotes quadratures
  - Gaussian quadratures
- 2 Multi-dimensional numerical integration (or "cubatures")
  - Sparse grid methods
  - Monte-Carlo integration



# One-dimensional numerical integration

Statement of the problem in 1D

The generic problem is to numerically calculate (evaluate)

$$I(a, b) = \int_a^b dx f(x)$$

The choice of integration method has to consider

- trivial, but first ask yourself, does the integral really exist
- are the integration limits finite or not
- is  $f(x)$  continuous and known for all  $x \in [a, b]$  or only on  $\{x_i\}_{i=1}^N$
- does  $f(x)$  have any singularities in the interval  $a$  to  $b$
- is  $f(x)$  oscillatory
- is  $f(x)$  computationally expensive to obtain
- what accuracy is needed

# One-dimensional numerical integration

A change of variable may be useful!

Definite integrals

$$\int_a^b dx f(x) = \frac{b-a}{2} \int_{-1}^1 du f\left(\frac{b-a}{2}u + \frac{b+a}{2}\right); \quad x = \frac{b-a}{2}u + \frac{b+a}{2}$$

Improper integrals

$$\int_0^\infty dx f(x) = \int_0^1 du \frac{f\left(\frac{1}{u} - 1\right)}{u^2}; \quad x = \frac{1}{u} - 1$$

$$\int_{-\infty}^\infty dx f(x) = \int_0^\infty dx [f(x) + f(-x)]$$

or use the transformation  $x = (1 - u)/u$  (which maps the entire real line on  $[0, 1]$ ).

# Quadrature rules

Consider the integral

$$I(a, b) = \int_a^b dx f(x)$$

where it is assumed that (i)  $f(x)$  is continuous; and (ii)  $a$  and  $b$  are finite

A **quadrature formula** is a linear combination of the  $f(x_n)$  such that

$$I(a, b) = \int_a^b dx f(x) \approx \sum_n w_n f(x_n)$$

where

- $w_n$  are called **weights** (normally  $w_n > 0$  to void cancellations)
- $x_n$  are called **abscissas** (or mesh points;  $x_n \in [a, b]$ )

Two main classes:

- **Newton-Cotes quadratures**

the *abscissas*  $x_n$  are equally spaced, and the *weights*  $w_n$  are determined so that the formula is exact for polynomials of as high degree as possible.

- $x_n$  given and  $w_n$  determined

- **Gaussian quadrature**

both the *abscissas*  $x_n$  and the *weights*  $w_n$  are chosen so that the formula is exact for polynomials of as high a degree as possible.

This leads to non-equidistant  $x_n$ 's!

- both  $x_n$  and  $w_n$  are determined



## Comments

- If the function  $f(x)$  is given explicitly instead of simply being tabulated at the values  $x_n$ , the best numerical method is the [Gaussian quadratures](#)
- By picking the *abscissas*  $x_n$  to sample the function, the Gaussian quadrature produces more accurate approximations for the same number of points
- However, the Gaussian quadratures are significantly more complicated to implement than the Newton-Cotes quadratures.

# Newton-Cotes quadratures

The Newton-Cotes quadratures are extremely useful and straightforward methods for calculating

$$I(a, b) = \int_a^b dx f(x)$$

Method: To integrate  $f(x)$  over  $[a, b]$  using a Newton-Cotes quadrature do

- choose the  $x_n$  *equally spaced* throughout  $[a, b]$  and calculate  $f_n = f(x_n)$
- find a (Lagrange interpolating) polynomial which approximates the tabulated function
- choose  $w_n$  so that the formula is exact for polynomials of as high a degree as possible

Note: Newton-Cotes quadratures may be "closed" or "open" depending on endpoints being included or not

# Newton-Cotes quadratures

- Closed rules

The closed formulas use the end points of subintervals. If we use  $N$  subintervals, the stepsize is  $h = (b - a)/N$ , and we get  $N + 1$  points

$$x_0 = a, \quad x_1 = a + h, \quad \dots, \quad x_{N-1} = b - h, \quad x_N = b$$

- Open rules

The open formulas use the midpoints of subintervals. If we use  $N$  subintervals, the stepsize is again  $h = (b - a)/N$ , and we get  $N$  points

$$x_{1/2} = a + h/2, \quad x_{3/2} = a + 3h/2, \quad \dots, \quad x_{N-1/2} = b - h/2$$

# Newton-Cotes quadratures

## Comments:

- Newton-Cotes formulas are not recommended for more than 7 points, because the weights become large, and some of them are negative, which leads to cancellation.
- There is still an easy way to use more points, for higher accuracy: we simply subdivide  $[a, b]$  into smaller intervals, and use a lower order Newton-Cotes formula on each subinterval. These are the [repeated, extended or compound Newton-Cotes formulas](#)
- Using lower order rules repeatedly, together with some simple extrapolation, is actually more efficient than using higher order rules. If you ever need the higher order rules, you can look them up in a book.

# Newton-Cotes quadratures

## Some Classical Newton-Cotes quadratures

- Midpoint rule: open 1 point Newton-Cotes quadrature ( $N = 1$ )
- Trapezoidal rule: closed 2 point Newton-Cotes quadrature ( $N = 1$ )
- Simpson's rule: closed 3 point Newton-Cotes quadrature ( $N = 2$ )

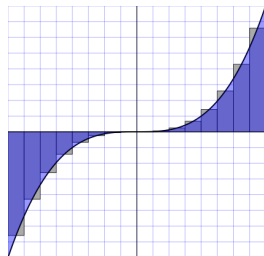
We will now revisit these methods!

# Midpoint rule (or Rectangle method)

Local rule: With only one point, we can only interpolate a polynomial of degree 0, i.e. a constant.

$$\int_a^b f(x) dx \approx (b-a) f\left(\frac{a+b}{2}\right)$$

Local error :  $\frac{(b-a)^3}{24} f^{(2)}(\xi)$ ,  $\xi \in [a, b]$



Extended formula (open); Repeated midpoint rule ( $x_0 = a$ ;  $x_N = b$ )

$$\int_a^b f(x) dx = \int_{x_0}^{x_1} f(x) dx + \int_{x_1}^{x_2} f(x) dx + \dots + \int_{x_{N-1}}^{x_N} f(x) dx; \quad h = \frac{b-a}{N}$$
$$\approx h [f(x_{1/2}) + f(x_{3/2}) + \dots + f(x_{N-1/2})] = h \sum_{n=1}^N f(x_{n-1/2}),$$

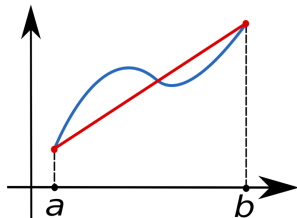
Global error :  $\frac{(b-a)h^2}{24} f^{(2)}(\xi) = \mathcal{O}(h^2)$

# Trapezoidal rule

Local rule: We have two points, so we can interpolate a polynomial of degree 1, i.e. a straight line.

$$\int_a^b f(x) dx \approx \frac{b-a}{2} [f(a) + f(b)]$$

Local error :  $-\frac{(b-a)^3}{12} f^{(2)}(\xi)$ ,  $\xi \in [a, b]$



Extended formula (closed)

$$\int_a^b f(x) dx \approx h \left[ \frac{1}{2} f(x_0) + f(x_1) + f(x_2) + \dots + f(x_{N-1}) + \frac{1}{2} f(x_N) \right],$$
$$x_n = x_0 + nh$$

Global error :  $-\frac{(b-a)h^2}{12} f^{(2)}(\xi) = \mathcal{O}(h^2)$

# Trapezoidal rule

Easy to implement numerically through the following simple algorithm

$$\int_a^b f(x) dx \approx h \left[ \frac{1}{2}f(x_0) + f(x_1) + f(x_2) + \dots + f(x_{N-1}) + \frac{1}{2}f(x_N) \right],$$
$$x_n = x_0 + nh$$

- Choose the number of mesh points and fix the step.
- Calculate  $f(a)$  and  $f(b)$
- Perform a loop over  $n = 1$  to  $n - 1$  ( $f(a)$  and  $f(b)$  are known) and sum up the terms  $f(a + h) + f(a + 2h) + f(a + 3h) + \dots + f(b - h)$ . Each step in the loop corresponds to a given value  $a + nh$ .
- Add  $f(a)/2$  and  $f(b)/2$  to the sum.
- Multiply the final result by  $h$ .



# Trapezoidal rule

C++ code for the Trapezoidal rule

```
double trapezoidal_rule(double a, double b, int n,  
                        double (*func)(double))  
{  
    double trapez_sum;  
    double fa, fb, x, step;  
    int j;  
    step=(b-a)/((double) n);  
    fa=(*func)(a)/2. ;  
    fb=(*func)(b)/2. ;  
    trapez_sum=0.;  
    for (j=1; j <= n-1; j++){  
        x=j*step+a;  
        trapez_sum+=(*func)(x);  
    }  
    trapez_sum=(trapez_sum+fb+fa)*step;  
    return trapez_sum;  
} // end function for trapezoidal rule
```

# Trapezoidal rule

Pay attention to the way we transfer the name of a function. This gives us the possibility to define a general trapezoidal method, where we give as input the name of the function.

```
double trapezoidal_rule(double a, double b, int n,  
                        double (*func)(double))
```

We call this function simply as something like this

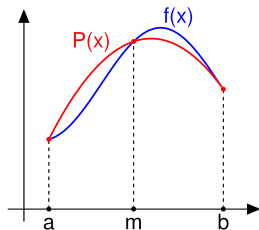
```
integral = trapezoidal_rule(a, b, n, mysuperduperfunction);
```

# Simpson's rule

Local rule : Simpson's rule is the three-point closed Newton-Cotes quadrature

$$\int_a^b f(x) dx \approx \frac{b-a}{6} \left[ f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right].$$

$$\text{Local error : } -\frac{(b-a)^5}{90} f^{(4)}(\xi), \quad \xi \in [a, b]$$



Extended formula (closed)

$$\int_a^b f(x) dx \approx \frac{h}{3} \left[ f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \dots + 4f(x_{N-1}) + f(x_N) \right].$$

$$\text{Global error : } -\frac{(b-a)h^4}{180} f^{(4)}(\xi) = \mathcal{O}(h^4)$$

# General order- $N$ Newton-Cotes quadrature

The basic idea behind all integration methods is to approximate the integral

$$I = \int_a^b f(x) dx \approx \sum_{n=1}^N w_n f(x_n),$$

where  $w_n$  and  $x_n$  are the weights and the chosen mesh points, respectively.

Simpson's rule gives

$$w_n = \{h/3, 4h/3, 2h/3, 4h/3, \dots, 4h/3, h/3\},$$

for the weights, while the trapezoidal rule resulted in

$$w_n = \{h/2, h, h, \dots, h, h/2\}.$$

In general, an integration formula which is based on a Taylor series using  $N$  points, will integrate exactly a polynomial  $P$  of degree  $N - 1$ . That is, **the  $N$  weights  $w_n$  can be chosen to satisfy  $N$  linear equations.**

# General order- $N$ Newton-Cotes quadrature

Given  $N + 1$  distinct points  $x_0, \dots, x_N \in [a, b]$  and  $N + 1$  values  $y_0, \dots, y_N$  there exists a unique polynomial  $P_N$  with the property

$$P_N(x_j) = y_j \quad j = 0, \dots, N$$

One such possibility is the [Lagrange interpolation polynomial](#) given by

$$P_N(x) = \sum_{k=0}^N l_k(x) y_k,$$

with the Lagrange factors

$$l_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^N \frac{x - x_i}{x_k - x_i} \quad k = 0, \dots, N$$

Example:  $N = 1$

$$P_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} = \frac{y_1 - y_0}{x_1 - x_0} x - \frac{y_1 x_0 + y_0 x_1}{x_1 - x_0},$$

which we recognize as the equation for a straight line.

# General order- $N$ Newton-Cotes quadrature

The polynomial interpolatory quadrature of order  $N$  with equidistant quadrature points  $x_k = a + kh$  and step  $h = (b - a)/N$  is called the Newton-Cotes quadrature formula of order  $N$ .

The integral is

$$\int_a^b f(x) dx \approx \int_a^b P_N(x) dx = \sum_{k=0}^N w_k f(x_k)$$

with

$$w_k = h \frac{(-1)^{N-k}}{k!(N-k)!} \int_0^N \prod_{\substack{j=0 \\ j \neq k}}^N (z - j) dz,$$

for  $k = 0, \dots, N$ .

For details check the literature!

QUADPACK is a library providing several Newton-Cotes quadratures

It is part of or being used by

- Slatec
- GSL
- Matlab (quad function)
- Scientific Python (scipy.integrate.quad)

More information on QUADPACK :

<https://en.wikipedia.org/wiki/QUADPACK>

<http://nines.cs.kuleuven.be/software/QUADPACK/>

QUADPACK can be found here:

<http://www.netlib.org/quadpack/>

[https://people.sc.fsu.edu/~jburkardt/f\\_src/quadpack/quadpack.html](https://people.sc.fsu.edu/~jburkardt/f_src/quadpack/quadpack.html)

## Gaussian quadratures



# Gaussian quadratures

- Newton-Cotes methods based on Taylor series using  $N + 1$  points will integrate exactly a polynomial  $P$  of degree  $N$ . If a function  $f(x)$  can be approximated with a polynomial of degree  $N$

$$f(x) \approx P_N(x),$$

with  $N + 1$  mesh points we should be able to integrate exactly the polynomial  $P_N$ .

- Gaussian quadrature methods promise more than this. We can get a better polynomial approximation with order greater than  $N + 1$  to  $f(x)$  and still get away with only  $N + 1$  mesh points.

More precisely, we approximate

$$f(x) \approx P_{2N+1}(x),$$

and with only  $N + 1$  mesh points these methods promise that

$$\int f(x) dx \approx \int P_{2N+1}(x) dx = \sum_{i=0}^N w_i P_{2N+1}(x_i),$$

# Gaussian Quadratures

A quadrature formula

$$\int_a^b W(x)f(x)dx \approx \sum_{i=0}^N w_i f(x_i),$$

with  $N + 1$  distinct quadrature points (mesh points) is called a **Gaussian quadrature** formula if it **integrates all polynomials  $p \in P_{2N+1}$  exactly**, that is

$$\int_a^b W(x)p(x)dx = \sum_{i=0}^N w_i p(x_i),$$

It is assumed that  $W(x)$  is continuous and positive and that the integral

$$\int_a^b W(x)dx$$

exists. Note that the replacement of  $f \rightarrow Wg$  is normally a better approximation due to the fact that we may isolate possible singularities of  $W$  and its derivatives at the endpoints of the interval.

# Gaussian quadratures

But which  $x_i$  and  $w_i$  to use?

- The theoretical foundation behind Gaussian quadratures is interesting, but will not be covered here!
- The abscissas  $x_i$  are related to orthogonal polynomials (e.g. Legendre polynomials) in the sense that they are the zeroes for these polynomials
- The weights  $w_i$  are determined so that the integrator is exact for a polynomial up to a given order

Library routines give you the sets  $\{w_i\}$  and  $\{x_i\}$  for a given Gaussian quadrature

The homepage of John Burkardt offers many of these and other routines:

<http://people.sc.fsu.edu/~jburkardt/>

# Gaussian quadratures

Examples: Gauss-Legendre quadrature on  $[-1, 1]$  (here  $W(x) = 1$ )

Abscissas values:

- Oder 1:

$$x_i = 0$$

- Oder 2:

$$x_i = -0.5773502691896257 \\ 0.5773502691896257$$

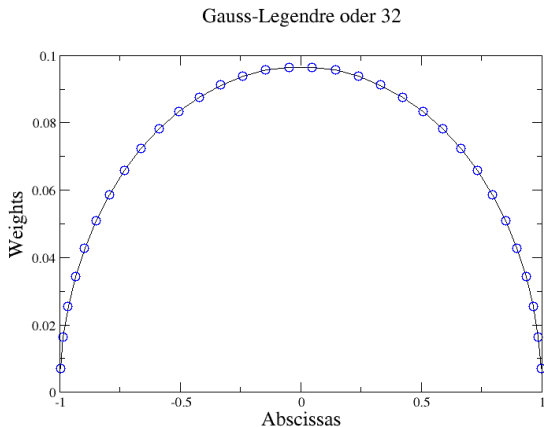
- Oder 4:

$$x_i = -0.8611363115940526 \\ -0.3399810435848563 \\ 0.3399810435848563 \\ 0.8611363115940526$$

After : [http://people.sc.fsu.edu/~jburkardt/f\\_src/int\\_exactness\\_legendre/int\\_exactness\\_legendre.html](http://people.sc.fsu.edu/~jburkardt/f_src/int_exactness_legendre/int_exactness_legendre.html)

# Gaussian quadratures

Examples: Gauss-Legendre quadrature on  $[-1, 1]$  (not equidistant sampling)



After : [http://people.sc.fsu.edu/~jburkardt/f\\_src/int\\_exactness\\_legendre/int\\_exactness\\_legendre.html](http://people.sc.fsu.edu/~jburkardt/f_src/int_exactness_legendre/int_exactness_legendre.html)

# Gaussian quadratures

What if the integrand is not smooth?

Even though the integrand is not smooth, we could render it smooth by extracting from it the weight function of an orthogonal polynomial, i.e., we are rewriting

$$I = \int_a^b f(x) dx = \int_a^b W(x)g(x) dx \approx \sum_{i=0}^N w_i g(x_i),$$

where  $g$  is smooth and  $W$  is the weight function, which is to be associated with a given orthogonal polynomial.

# Gaussian quadratures

- The weight function  $W$  is non-negative in the integration interval  $x \in [a, b]$  such that for any  $N \geq 0$ ,  $\int_a^b |x|^N W(x) dx$  is integrable.
- The naming weight function arises from the fact that it may be used to give more emphasis to one part of the interval than another.

Weight function	Interval	Polynomial
$W(x) = 1$	$x \in [a, b]$	Legendre
$W(x) = e^{-x^2}$	$-\infty \leq x \leq \infty$	Hermite
$W(x) = e^{-x}$	$0 \leq x \leq \infty$	Laguerre
$W(x) = 1/(\sqrt{1-x^2})$	$-1 \leq x \leq 1$	Chebyshev

$$I = \int_{-1}^1 f(x) dx$$

$$a(1 - x^2)P - b^2P + (1 - x^2)\frac{d}{dx} \left( (1 - x^2)\frac{dP}{dx} \right) = 0$$

Here  $a$  and  $b$  are constants. For  $b = 0$  we obtain the Legendre polynomials as solutions, whereas  $b \neq 0$  yields the so-called associated Legendre polynomials. The corresponding polynomials  $P$  are

$$L_k(x) = \frac{1}{2^k k!} \frac{d^k}{dx^k} (x^2 - 1)^k \quad k = 0, 1, 2, \dots,$$

which, up to a factor, are the Legendre polynomials  $L_k$ . The latter fulfil the orthogonality relation

$$\int_{-1}^1 L_i(x)L_j(x) dx = \frac{2}{2i + 1} \delta_{ij},$$

and the recursion relation

$$(j + 1)L_{j+1}(x) + jL_{j-1}(x) - (2j + 1)xL_j(x) = 0.$$



$$I = \int_0^{\infty} f(x) dx = \int_0^{\infty} x^{\alpha} e^{-x} g(x) dx.$$

These polynomials arise from the solution of the differential equation

$$\left( \frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2} \right) \mathcal{L}(x) = 0,$$

where  $l$  is an integer  $l \geq 0$  and  $\lambda$  a constant. They fulfil the orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x} \mathcal{L}_m(x) \mathcal{L}_n(x) dx = \delta_{mn},$$

and the recursion relation

$$(n+1)\mathcal{L}_{n+1}(x) = (2n+1-x)\mathcal{L}_n(x) - n\mathcal{L}_{n-1}(x).$$

In a similar way, for an integral which goes like

$$I = \int_{-\infty}^{\infty} f(x) dx = \int_{-\infty}^{\infty} e^{-x^2} g(x) dx.$$

we could use the Hermite polynomials in order to extract weights and mesh points. The Hermite polynomials are the solutions of the following differential equation

$$\frac{d^2 H(x)}{dx^2} - 2x \frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0.$$

They fulfil the orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_m(x) H_n(x) dx = \sqrt{\pi} 2^n n! \delta_{mn},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

# Some examples

Integral : Evaluate the integral

$$\int_0^3 dx \frac{1}{2+x^2} = \frac{1}{\sqrt{2}} \arctan\left(\sqrt{18/4}\right) \approx 0.7992326575\dots$$

Results :

N	Trapezoidal	Simpson	Gauss-Legendre
10	0.798861	0.799231	0.799233
20	0.799140	0.799233	0.799233
40	0.799209	0.799233	0.799233
100	0.799229	0.799233	0.799233
1000	0.799233	0.799233	0.799233

Python : Calculate the integral over  $f(x)$  from  $a$  to  $b$  via

- `scipy.integrate.quadrature(f,a,b)`
- `scipy.integrate.quad(f,a,b)`

# Some examples

Integral : Evaluate the integral

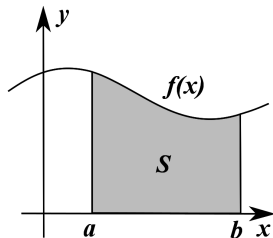
$$\int_1^{100} dx \frac{\exp(-x)}{x} \approx 0.2193839344 \dots$$

Results :

N	Trapezoidal	Simpson	Gauss-Legendre
10	1.821020	1.214025	0.1460448
20	0.912678	0.609897	0.2178091
40	0.478456	0.333714	0.2193834
100	0.273724	0.231290	0.2193839
1000	0.219984	0.219387	0.2193839

# Some Terminology : Numerical integration

- **Adaptive integrators** : a process in which the integral of a function  $f(x)$  is approximated using static quadrature rules on adaptively refined subintervals of the integration domain.
- **Automatic integrators** : tries to obtain a given tolerance automatically
- **Nested integrators**: An efficient calculation chooses the sequence of  $N$  in such a way that previous function values can be reused. This is called nesting.



# Quadratures not discussed

- Gauss-Kronrod quadrature

a variant of Gaussian quadrature, in which the evaluation points are chosen so that an accurate approximation can be computed by re-using the information produced by the computation of a less accurate approximation, *i.e.* Gauss-Kronrod quadrature are nested.

- Clenshaw-Curtis quadrature (Fejér quadratures)

Based on an expansion of the integrand in terms of Chebyshev polynomials. A change of variables  $x = \cos \theta$  is done and a discrete cosine transform (DCT) is used to calculate the resulting series.

Clenshaw-Curtis quadrature naturally leads to nested quadrature rules.

Special Clenshaw-Curtis quadratures exist for integrands  $W(x)f(x)$  with a weight function  $W(x)$  that is highly oscillatory, e.g. a sinusoid or Bessel functions. These methods are useful for high-accuracy integration of rapid oscillating integrands.

Due to the nested nature of the Gauss-Kronrod and Clenshaw-Curtis quadratures they are frequently used for [adaptive quadratures!](#)

# Some examples

Use Python's scipy package to calculate  $\int_0^4 dx x^2$

```
>>> from scipy import integrate
>>> func = lambda x: x**2
>>> scipy.integrate.quad(func, 0, 4)
(21.333333333333332, 2.3684757858670003e-13)
>>> print(4**3 / 3.) # analytical result
21.3333333333
```

# Romberg quadrature

- Romberg integration is a powerful method
- It consists of repeatably using an integrator for stepsizes, say,  $h$ ,  $h/2$ ,  $h/4$  etc. and than extrapolating to  $h \rightarrow 0$
- Romberg integration is “Richardson extrapolation” (a sequence acceleration method) applied to integration
- Romberg was a professor in Trondheim (NTH)

Example: see *e.g.*

[https://en.wikipedia.org/wiki/Romberg%27s\\_method](https://en.wikipedia.org/wiki/Romberg%27s_method)



# Ways of increasing integration accuracy

Question : How can one increase the accuracy of a numerical integration?

- Increasing the number of points (not always a good idea, why not?)
- Using a more accurate integrator
- Using subdivision of the integration domain
- Using an automatic **adaptive integrator** (which typically uses subdivision)

We will now look at some of these possibilities!

Assume that we want to compute an integral using say the trapezoidal rule. We limit ourselves to a one-dimensional integral. Our integration domain is defined by  $x \in [a, b]$ . The algorithm goes as follows

- We compute our first approximation by computing the integral for the full domain. We label this as  $I^{(0)}$ . It is obtained by calling our previously discussed function **trapezoidal\_rule** as

```
I0 = trapezoidal_rule(a, b, n, function);
```

- In the next step we split the integration in two, with  $c = (a + b)/2$ . We compute then the two integrals  $I^{(1L)}$  and  $I^{(1R)}$

```
I1L = trapezoidal_rule(a, c, n, function);  
I1R = trapezoidal_rule(c, b, n, function);
```

With a given defined tolerance, being a small number provided by us, we estimate the difference  $|I^{(1L)} + I^{(1R)} - I^{(0)}| < \text{tolerance}$ . If this test is satisfied, our first approximation is satisfactory.

- If not, set up a **recursive procedure** where the integral is split into subsequent subintervals until our tolerance is satisfied.

# Adaptive methods

```
function [I,nf] = adaptsimp(f,a,b,tol);
% ADAPTSIMP calls the recursive function ADAPTREC to compute the integral of the
% vector-valued function f over [a,b]; tol is the desired absolute accuracy;
% nf is the number of function evaluations.
% G. Dalquist and A Bjorck, Numerical Methods in Scientific Computing, (SIAM,
% 2008).
ff = feval(f,[a, (a+b)/2, b]);
nf = 3; % Initial Simpson approximation
I1 = (b - a)*[1, 4, 1]*ff'/6;
% Recursive computation
[I,nf] = adaptrec(f,a,b,ff,I1,tol,nf);
function [I,nf] = adaptrec(f,a,b,ff,I1,tol,nf);
h = (b - a)/2;
fm = feval(f, [a + h/2, b - h/2]);
nf = nf + 2;
% Simpson approximations for left and right subinterval
fR = [ff(2); fm(2); ff(3)];
fL = [ff(1); fm(1); ff(2)];
IL = h*[1, 4, 1]*fL/6;
IR = h*[1, 4, 1]*fR/6;
I2 = IL + IR;
I = I2 + (I2 - I1)/15;
% Extrapolated approximation
if abs(I - I2) > tol
% Refine both subintervals
[IL,nf] = adaptrec(f,a,a+h,fL,IL,tol/2,nf);
[IR,nf] = adaptrec(f,b-h,b,fR,IR,tol/2,nf);
I = IL + IR;
end
```

Example: Use the MATLAB routine `adaptsimp` to integrate

$$I = \int_{-4}^4 \frac{dx}{1+x^2} = 2.65163532733607.$$

Results

I	tolerance	n	Error
2.6516250211	$10^{-4}$	41	$1.0 \cdot 10^{-5}$
2.6516352064	$10^{-5}$	81	$1.2 \cdot 10^{-7}$
2.651635327353	$10^{-6}$	153	$-1.7 \cdot 10^{-11}$

Here  $n$  is the no. of function calls.

# Adaptive methods

The recommended canned integration routines are typically adaptive

For instance, **QAGS** from QUADPACK uses an adaptive Gauss-Kronrod quadratures, and it is the main routine used in

- `scipy.integrate.quad`
- Matlab/Octave : `quad`
- GSL: `gsl_integration_qags`

# Numerical integration one-dimension: Recommendations

The general recommendation I would give for numerical integration is:

- Adaptive Gauss-Kronrod or Clenshaw-Curtis quadratures if  $f(x)$  is known for any argument  $x$ 
  - e.g. QAGS from quadpack (available through various libraries)
- Newton-Cotes if  $f(x)$  is tabulated

# Software: One-dimensional numerical integration

The following software packages provide numerical integration routines:

- QUADPACK library (part of Slatec)
  - <http://en.wikipedia.org/wiki/QUADPACK>
- GNU Scientific Library (contains QUADPACK)
- Boost C++ libraries [www.boost.org/](http://www.boost.org/)
- Python scipy
- NAG Numerical Libraries (commercial)

Another good source is

- John Burkardt homepage:  
<http://people.sc.fsu.edu/~jburkardt/>

- **QAGS**: uses global adaptive quadrature based on 21-point Gauss-Kronrod quadrature within each subinterval, with acceleration by Peter Wynn's epsilon algorithm
- **QAGI**: is the only general-purpose routine for infinite intervals, and maps the infinite interval onto the semi-open interval  $(0, 1]$  using a transformation that uses the same approach as QAGS, except with 15-point rather than 21-point Gauss-Kronrod quadrature.

For an integral over the whole real line, the transformation used is  $x = (1 - t)/t$

$$\int_{-\infty}^{+\infty} f(x) dx = \int_0^1 \frac{dt}{t^2} \left[ f\left(\frac{1-t}{t}\right) + f\left(-\frac{1-t}{t}\right) \right].$$

QUADPACK also contains routines for singular integrands (**QAGP**), and Cauchy principle value integrals (**QAWC**) (to be discussed next)

See : <http://en.wikipedia.org/wiki/QUADPACK>



# Numerical integration of singular integrands

Question : How can one evaluate the integral

$$I(a, b) = \int_a^b dx f(x)$$

if  $f(x)$  has a singularity in the interval  $[a, b]$ ?

First question to ask:

- where is the singularity located
- what kind of singularity; algebraic, logarithmic, etc.

Various approaches:

- “ignore” the singularity (not in general a good idea)
- specially designed Gaussian quadrature handling the singularity as a weighting function
- “**transformation trick**”: transform it away analytically
- “**subtraction trick**”: subtract off the singularity and treat it analytically

# Numerical integration of singular integrands

Numerical integration of singular integrands, may, or-may-not, be done successfully with a normal quadrature rule if we avoid the singular point.

However, we stress such strategy **may not** be successful, so real care should be taken. For instance for oscillatory integrands, the method of “ignoring the singularity” does not work well.

**Homework:** Try this approach to the integrals (eg. using Simpson and/or Gauss-Legendre quadratures)

- Integral 1

$$I_1 = \int_0^1 dx \frac{1}{\sqrt{x}} = 2$$

- Integral 2

$$I_2 = \int_0^1 dx \frac{1}{x} \sin\left(\frac{1}{x}\right) = 0.624713 \dots$$

# Numerical integration of singular integrands

- First advice : investigate if one can transform or modify the given problem *analytically* to make it more suitable for numerical integration

Example 1: Numerically calculate (singularity at  $x = 0$ )

$$I = \int_0^1 dx \frac{e^x}{\sqrt{x}}$$

However, make a change of variable  $x = t^2$ , to get

$$I = 2 \int_0^1 dt \exp(t^2)$$

which is well-behaved.

Alternatively one may do a few integrations by part!

# Numerical integration of singular integrands

Alternatively one may construct a special Newton-Cotes rule the integral with weight function  $W(x) = 1/\sqrt{x}$

$$\int_0^{2h} dx \frac{f(x)}{\sqrt{x}} \approx \sqrt{2h} [w_0 f(0) + w_1 f(h) + w_2 f(2h)].$$

Using the method of undetermined coefficients for polynomials  $f(x) = 1, x, x^2$  gives

$$\int_0^{2h} dx \frac{f(x)}{\sqrt{x}} \approx \sqrt{2h} \left[ \frac{12}{15} f(0) + \frac{16}{15} f(h) + \frac{2}{15} f(2h) \right].$$

Show this yourself!

# Numerical integration of singular integrands

Example 2: A more general form

Numerically calculate for  $f(x) \in \mathbb{C}[0, 1]$

$$I = \int_0^1 dx \frac{f(x)}{x^{1/\alpha}}, \quad \alpha \geq 2$$

Here a change of variable  $x = t^\alpha$  results in

$$I = \alpha \int_0^1 dt t^{\alpha-2} f(t^\alpha)$$

which is also well-behaved and better suited for numerical evaluation.

# Numerical integration of singular integrands

Example 3: What to do with this integral for  $f(x) \in \mathbb{C}[0, 1]$

$$I = \int_0^1 dx \ln(x) f(x)$$

Question: What to do here regarding the logarithmic singularity at  $x = 0$ ?

# Numerical integration of singular integrands

Example 3: What to do with this integral for  $f(x) \in C[0, 1]$

$$I = \int_0^1 dx \ln(x) f(x)$$

Question: What to do here regarding the logarithmic singularity at  $x = 0$ ?

Possible answer : A change of variable  $x = e^{-t}$ , will do the job since it implies

$$I = - \int_0^{\infty} dt t e^{-t} f(e^{-t})$$

which is a well behaved improper integral (that can be transformed to a finite integration domain).

# Numerical integration of singular integrands

- **Second advice** : If the singular points are known, but can not be transformed away, then the integral should first be broken up into several pieces so that all the singularities are located at one (or both) ends of the interval  $[a, b]$ .

Say we have a singularity at  $c \in [a, b]$ , then do

$$\int_a^b dx f(x) = \int_a^c dx f(x) + \int_c^b dx f(x)$$

Many integrals can then be treated by weighted quadrature rules, i.e., the singularity is incorporated into the weight function

Comments: Many canned integration routines, like **QAGS** from QUADPACK, do this for you automatically!



# Numerical integration of singular integrands

Elimination of the singularity by subtracting it off and treat it analytically

Consider again the integral

$$I = \int_a^b dx f(x)$$

where  $f(x)$  has an *integrable* singularity in  $[a, b]$  that can not be transformed away. Assume that  $g(x)$  has a similar behavior to  $f(x)$  *around* the singularity.

Then do

$$I = \int_a^b dx g(x) + \int_a^b dx [f(x) - g(x)] \equiv I_A + I_N$$

where  $I_A$  is treated analytically, and  $I_N$  numerically since it does *not* have a singularity.

This methods only works if a function  $g(x)$  can be found that allows an analytic treatment.

# Numerical integration of singular integrands

Example: Elimination of the singularity by subtracting

$$\begin{aligned} I &= \int_0^1 dx \frac{\cos x}{\sqrt{x}} \\ &= \int_0^1 dx \frac{1}{\sqrt{x}} + \int_0^1 dx \frac{\cos x - 1}{\sqrt{x}} \\ &= 2 + \int_0^1 dx \frac{\cos x - 1}{\sqrt{x}}. \end{aligned}$$

In evaluating the last integral numerically one has to explicitly use that

$$\cos x - 1 \approx -\frac{x^2}{2} + \dots$$

**Note:** How small the argument has to be for this to be true depends on the precision (single/double) that the calculation is done in!

# Cauchy principle value integrals

Cauchy principle value integrals do appear in physics when dealing with *e.g.* dispersion relations and Greens functions for scattering problems.

## Definition: Cauchy principle value

If the following integral, for  $b \in [a, c]$  and  $f(x)$  singular at  $x = b$ , is finite

$$I = \mathcal{P} \int_a^c dx f(x) = \lim_{\varepsilon \rightarrow 0^+} \left[ \int_a^{b-\varepsilon} f(x) dx + \int_{b+\varepsilon}^c f(x) dx \right]$$

then it is the Cauchy principle value (denoted  $\mathcal{P}$ ).

Note first of all that one has to be careful in how the limit is taken:

$$\lim_{a \rightarrow 0^+} \left( \int_{-1}^{-a} \frac{dx}{x} + \int_a^1 \frac{dx}{x} \right) = 0,$$

$$\lim_{a \rightarrow 0^+} \left( \int_{-1}^{-2a} \frac{dx}{x} + \int_a^1 \frac{dx}{x} \right) = \ln 2$$

Only the first integral is a Cauchy principle value integral (a distribution)! 

# Cauchy principle value integrals

Example: Calculate the Cauchy principle value of

$$\mathcal{P} \int_0^{\infty} \frac{dx}{x^2 + x - 2}$$

Analytically one finds (by straightforwardly using the definition):

$$\begin{aligned} \mathcal{P} \int_0^{\infty} \frac{1}{x^2 + x - 2} &= \frac{1}{3} \mathcal{P} \int_0^{\infty} \left( \frac{1}{x-1} - \frac{1}{x+2} \right) dx \\ &= \frac{1}{3} \lim_{\epsilon \rightarrow 0} \left[ \int_0^{1-\epsilon} \left( \frac{1}{x-1} - \frac{1}{x+2} \right) dx + \int_{1+\epsilon}^{\infty} \left( \frac{1}{x-1} - \frac{1}{x+2} \right) dx \right] \\ &= \frac{1}{3} \lim_{\epsilon \rightarrow 0} \left[ [\log(1-x) - \log(x+2)]_0^{1-\epsilon} + [\log(x-1) - \log(x+2)]_{1+\epsilon}^{\infty} \right] \\ &= \frac{1}{3} \lim_{\epsilon \rightarrow 0} [\log(\epsilon) - \log(3-\epsilon) + \log(2) - \log(\epsilon) + \log(3+\epsilon)] \\ &= \frac{\log(2)}{3} \end{aligned}$$

where we have used that  $\lim_{R \rightarrow \infty} \log\left(\frac{R-1}{R+2}\right) = 0$ .

# Cauchy principle value integrals

How can one calculate Cauchy principle value integrals numerically?

Consider the integral where  $y \in [a, b]$  and  $f(x) \in \mathbb{C}[a, b]$  (Hilbert transform)

$$I(y) = \mathcal{P} \int_a^b dx \frac{f(x)}{x-y} = \lim_{\varepsilon \rightarrow 0^+} \left[ \int_a^{y-\varepsilon} \frac{f(x)}{x-y} dx + \int_{y+\varepsilon}^b \frac{f(x)}{x-y} dx \right]$$

However, only the region around the singularity at  $x = y$  is problematic:

$$\mathcal{P} \int_a^b dx \frac{f(x)}{x-y} = \int_a^{y-\Delta} \frac{f(x)}{x-y} dx + \mathcal{P} \int_{y-\Delta}^{y+\Delta} dx \frac{f(x)}{x-y} + \int_{y+\Delta}^b \frac{f(x)}{x-y} dx$$

The first and last integral are calculated by standard methods.

For the Principle value integral we make a change of variable  $x = u\Delta + y$  to get

$$I_{\Delta}(y) = \mathcal{P} \int_{y-\Delta}^{y+\Delta} dx \frac{f(x)}{x-y} = \mathcal{P} \int_{-1}^1 du \frac{f(u\Delta + y)}{u}$$

# Cauchy principle value integrals

Now we use the “subtraction trick” to calculate  $I_{\Delta}(y)$

$$\begin{aligned} I_{\Delta}(y) &= \mathcal{P} \int_{-1}^1 du \left[ \frac{f(u\Delta + y) - f(y)}{u} + \frac{f(y)}{u} \right] \\ &= \mathcal{P} \int_{-1}^1 du \left[ \frac{f(u\Delta + y) - f(y)}{u} \right] + f(y) \underbrace{\mathcal{P} \int_{-1}^1 du \frac{1}{u}}_0 \\ &= \int_{-1}^1 du \left[ \frac{f(u\Delta + y) - f(y)}{u} \right] \end{aligned}$$

where we in the last transition have taken advantage of the integrand no longer is singular since

$$\lim_{u \rightarrow 0} [f(\Delta u + y) - f(y)] = 0$$

Calculate  $I_{\Delta}(y)$  by Gauss-Legendre quadratures of EVEN order (why?)

# Examples : Cauchy principle value integrals

The routine **QAWC** from QUADPACK calculates Cauchy principle value integrals, and here we demonstrate the scipy interface to this library

Example: Verify that

$$\mathcal{P} \int_{-1}^1 dx \frac{1 + \sin(x)}{x} = 2 \int_0^1 \frac{\sin(x)}{x} = 2 \operatorname{Si}(1) \approx 1.8921661407343662$$

Scipy code

```
import numpy as np
import scipy.integrate
# P \int_{-1}^1 dx (1 + sin(x))/(x - wvar)
numerator=lambda x: 1 + np.sin(x);
scipy.integrate.quad(numerator, -1, 1, weight='cauchy', wvar=0)
(1.8921661407343664, 2.433608869978343e-13)
```

# Multidimensional integrals

The multidimensional integral

$$I_n(\mathbf{a}, \mathbf{b}) = \int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \dots \int_{a_n}^{b_n} dx_n f(\mathbf{x})$$

- can be calculated as repeated one-dimensional integrals
- This is valid for any numerical integration method
- Numerical integration over more than one dimension is sometimes described as “cubatures”
- Different integration methods can be applied in different dimensions

**Challenge:** “Curse of dimensionality”

The exponentially rising cost associated with simple product grids

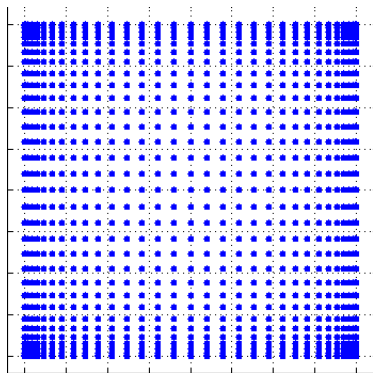
Ex: With  $10^6$  function eval. leaves about 2 points per direction in  $20D$ !

This cost is unaffordable, but worse, it is *unnecessary*!

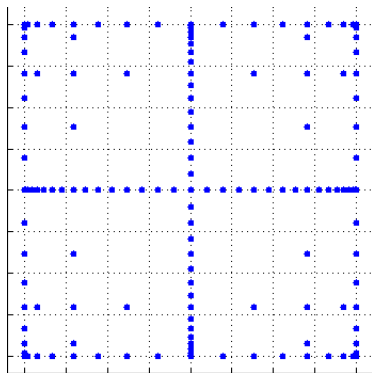


# Sparse grid methods — Smolyak quadratures

Sparse grids avoid the exponentially rising cost associated with simple product grids, and can produce more accurate results than a Monte Carlo approach if the function has bounded derivatives of sufficient order.



Tensor product grid



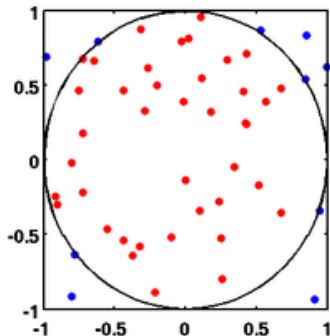
Sparse grid

# Sparse grid methods — Smolyak quadratures

- developed by the Russian mathematician Sergey A. Smolyak
- based on a sparse tensor product construction
- Smolyak's quadrature rule : more efficient method of integrating multidimensional functions
- **Sparse Grid = Sum of *Selected* Product Rule** (more efficient)
  
- **Tutorial:** <http://www.math.tu-berlin.de/~garcke/paper/sparseGridTutorial.pdf>
- **Software** <http://people.sc.fsu.edu/~jburkardt>

# Monte-Carlo integration

Trivial example : Calculate  $\pi$  by random numbers!



$$\pi = 4 \frac{A_{\circ}}{A_{\square}} = 4 \lim_{N_{\square} \rightarrow \infty} \frac{N_{\circ}}{N_{\square}}$$

# Monte-Carlo integration

Define

$$I = \int_{\Omega} d^n x f(\mathbf{x}); \quad V = \int_{\Omega} d^n x$$

Then we do the approximation

$$I \approx Q_N \equiv V \langle f(\mathbf{x}) \rangle = V \frac{1}{N} \sum_{i=1}^N f(\mathbf{x}_i)$$

Given the estimation of  $I$  from  $Q_N$ , the error bars of  $Q_N$  can be estimated by the sample variance using the unbiased estimate of the variance:

$$\sigma_N^2 = \frac{1}{N-1} \sum_{i=1}^N (f(\mathbf{x}_i) - \langle f \rangle)^2.$$

- The error in the Monte Carlo integration is however independent of  $D$  and scales as  $\sigma_N \sim 1/\sqrt{N}$ , always!
- The aim in Monte Carlo calculations is to have  $\sigma_N$  as small as possible after  $N$  samples. The results from one sample represents, since we are using concepts from statistics, a “measurement”.

# The volume of an n-dimensional sphere

- The n-dimensional sphere of radius  $R$ , the “Euclidean ball”, is the volume bounded by the surface  $|\mathbf{x}| = R$  where  $\mathbf{x} \in \mathbb{R}^n$ .
- The volume of such a sphere is

$$V_n(R) = \int_{|\mathbf{x}| < R} d^n x$$

- Homework: Calculate the volume  $V_n(R)$  for different values of  $n \in \mathbb{N}_0!$

## Section 9

# Random numbers

# Outline I

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods
- 8 Numerical integration

# Outline II

- 9 Random numbers
  - Uniform pseudo random number generators
  - Nonuniform pseudo random number generators

10 Ordinary differential equations

11 Partial differential equations

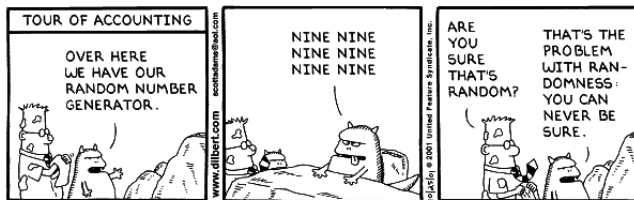
12 Optimization



# Random numbers

A computer is a *deterministic* machine!

Are random numbers possible to generate on a computer?



Copyright © 2001 United Feature Syndicate, Inc.

## Application of random numbers

- Scientific Computing
- Statistics
- Cryptography
- Gambling

## Physics

- Monte Carlo simulations
- Statistical physics
- Algorithmic modeling

## Use of Random Number Generators (RNGs)

In principle, RNGs are needed in any application where **unpredictable** results are required.

- For most applications it is desirable to have fast RNGs that produce numbers that are as random as possible
- However, these two properties are often inversely proportional to each other
- **Excellent RNGs are often slow, whereas poor RNGs are typically fast**

Good introduction :

Helmut G. Katzgraber, *Random Numbers in Scientific Computing: An Introduction*, arXiv:1005.4117, 2010; <http://arxiv.org/abs/1005.4117>

## Main classes of RNGs:

- True RNGs

- *True* random numbers are generated
- There are **no** correlations in the sequence of numbers
- special hardware typically needed (exception `/dev/random`)
- True RNGs are generally slow; limited use for large-scale computer simul.
- Debugging difficult; No restart of the sequence
- Buy random numbers from `www.random.org/`

- Pseudo RNGs (PRNGs)

- are fast (no need for post-processing)
- do not require special hardware and therefore are very portable
- easy debugging: the exact sequence of RNs may be reproduced
- have finite sequence lengths
- the numbers produced may be correlated (*i.e.* know your generator!)

# Pseudo Random Number Generators

Good pseudo RNGs should be

- Random
- Reproducible
- Portable
- Efficient

Random numbers from any distribution, takes *uniform random numbers*, denoted  $U(0, 1)$ , as the starting point.

## No universal PRNG!

There is NO universal pseudo random number generator (PRNG).  
Always test your PRNG in the application you have in mind!

The field of PRNGs is evolving quickly;

A generator that was considered good in the past, may no longer be “good”.

## Algorithm: Uniform PRNGs

Generate a sequence of numbers  $x_1, x_2, x_3, \dots$ , using a recurrence of the form

$$x_i = f(x_{i-1}, x_{i-2}, \dots, x_{i-n})$$

where  $n$  initial numbers (seed block) are needed to start the recurrence.

All PRNGs have this structure; the magic lies in finding a function  $f(\cdot)$  that produces numbers that are “as random as possible”.

The initial values — the **seed** — determines the sequence of random numbers.

Methods for generating uniform pseudo random numbers  $U(0, 1)$ :

- Linear congruential generators
- Lagged Fibonacci generators
- Other commonly-used PRNGs

### Linear congruential generators

These generators, in their simplest implementation, are of the form

$$l_{i+1} = (al_i + c) \pmod{m} \quad (\text{integer})$$

$$r_{i+1} = \frac{l_{i+1}}{m} \in [0, 1) \quad (\text{float})$$

with

- $l_0$ : a seed value  $0 \leq l_0 < m$
- $a$ : the multiplier  $0 \leq a < m$
- $c$ : the increment  $0 \leq c < m$
- $m$ : the modulus; (typically a large integer)

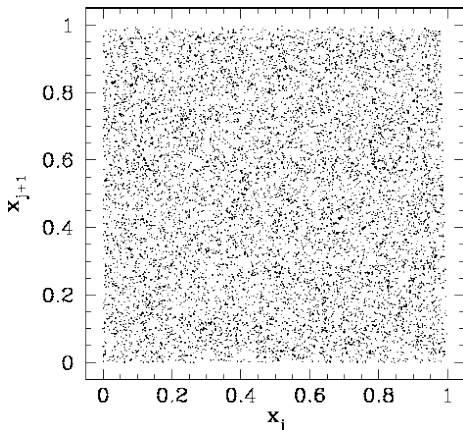
- $l_i$  is an integer between 0 and  $m - 1$
- period  $\leq m$  hopefully period  $\approx m$

- The choice of constants  $(I_0, a, c, m)$  influence the quality of the generator
- Mathematicians have partly guided us how to pick “magic numbers”
- sequential correlations show up
- Park-Miller generator :  $c = 0$

Some choices:

- “Minimal Standard Generator” (Lewis, Goodman and Miller (1969))
  - $a = 16\,807$ ;  $c = 0$ ; and  $m = 2^{31} - 1 = 2147483647 \sim 10^9$  [using 32-bits integer]
  - good for generating unsigned 32-bit random integers
  - very fast and full period of  $2^{31} - 1 \sim 10^9$  (often too short for floats)
  - little sequential correlations
- `drand48( )`
  - $a = 25214903917$ ,  $c = 11$  and  $m = 2^{48} \sim 10^{14}$

“Minimal Standard Generator” does not show sequential correlations



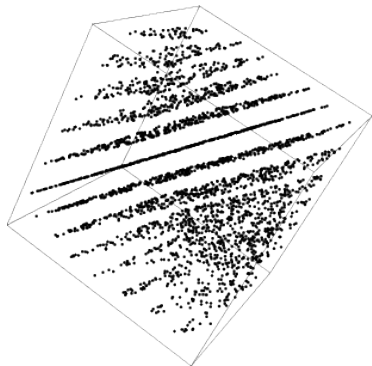
Plotting  $\mathbf{R}_j(n) = (x_j, x_{j+n})$  reveals no pattern whatsoever in the plotted points.



# Uniform deviates

Example of a bad generator

Example of a bad generator: RANDU ( $a = 65539$ ,  $c = 0$ , and  $m = 2^{31}$ ) used on the IBM mainframe computers in the 1960s



$10^3$  triplets of successive random numbers produced with RANDU plotted in three-dimensional space.

### Lagged Fibonacci generators

These generators are of the form

$$x_i = (x_{i-j} \odot x_{i-k}) \pmod{m}, \quad 0 < j < k,$$

where  $\odot$  denotes a binary operator, *i.e.* addition, multiplication or exclusive OR (XOR; “one or the other but not both”)

- Typically  $m = 2^M$  with  $M = 32$  or  $64$ .
- Requires a seed block of size  $k$  to be initialized (often with a Linear congruential generator)
- When  $\odot$  is a multiplication [addition] one talks about multiplicative [additive] lagged Fibonacci generators

The name derives from the similarity to the Fibonacci series

$$x_i = x_{i-1} + x_{i-2} \quad \rightarrow \quad \{1, 1, 2, 3, 5, 8, 13, 21, \dots\}$$

Lagged Fibonacci generators are intended as an improvement over linear congruential generators and, in general, they are not only fast but most of them pass all standard empirical random number generator tests.

Examples of Lagged Fibonacci generators are

- $r1279()$ 
  - *multiplicative* generator with  $k = 1279$
  - period approximately  $10^{394}$
  - has passed all known PRNG tests
  - the “standard” generator in *e.g.* GNU Scientific Library
- $r250()$ ; not a so good generator!
  - $k = 250$ ;  $\odot = XOR$
  - For many years the “standard generator” in numerical simulations
  - **Problem:** Energy per spin in the Ising model at the critical temperature was approximately 42 standard deviations off the known exact result (1992)

- **Mersenne Twister**; `mt19937()`, `mt19937-64()`
  - Developed in 1997, and passed almost all known tests
  - among the most frequently used generators of today
  - fast generator
  - Period  $2^{19937} - 1 \approx 10^{6001}$
  - period determined by Mersenne primes ( $M_n = 2^n - 1$ ,  $n \in \mathbb{N}$ )
  - available in 32 and 64-bits architectures
  - available in C/C++, Fortran, GSL, Matlab, Python, Boost etc.
  - [en.wikipedia.org/wiki/Mersenne\\_twister](http://en.wikipedia.org/wiki/Mersenne_twister)
- **WELL generators** — Well Equidistributed Long-period Linear
  - originally developed by Panneton, L'Ecuyer and Matsumoto
  - The idea is to provide better equidistribution and bit mixing with an equivalent period length and speed as the Mersenne Twister

# How to test the quality of PRNGs?

Various different *test suites* have been developed with the sole purpose of testing PRNGs.

- Simple tests for  $N$  PRNs from  $U(0, 1)$ 
  - correlation test :  $\langle x_i x_{i+n} \rangle - \langle x \rangle^2 < \mathcal{O}(1/\sqrt{N})$
  - moment test :  $\langle x^k \rangle - 1/(k+1) < \mathcal{O}(1/\sqrt{N})$
  - Graphical test : plot  $\mathbf{R}_i = (x_i, x_{i+1})$  etc.
- **DIEHARD** by George Marsaglia
  - consists of 16 different tests
  - For more details visit  
[http://en.wikipedia.org/wiki/Diehard\\_tests](http://en.wikipedia.org/wiki/Diehard_tests)

There is NO ultimate test!

Therefore run your code with different PRNGs.

If the results agree within error bars and the PRNGs used are from different families, the results are likely to be trusted.

# Recommendations for $U(0, 1)$ PRNGs

For any scientific applications we recommend :

- Good (and fast) generators
  - Mersenne Twister
  - multiplicative lagged Fibonacci generator such as *r1279()*
  - WELL generators
- Avoid
  - any home-cooked routines
  - use of linear congruential generators
  - the family of Unix built-in generators *drand48()*
  - Numerical Recipes *ran0()*, *ran1()* and *ran2()*

**Warning:** Implementation of PRNGs on a computer is not always trivial, because of the different numerical range of the integers specified by the computer language or hardware.

PRNGs are therefore not necessarily easily portable!

# Final recommendations

Dealing with random numbers can be a delicate issue. Therefore...

- Always try to run your simulations with two different PRNGs from different families, at least for small testing instances. One option would be to use an excellent but slow PRNG versus a very good but fast PRNG. For the production runs then switch to the fast one.
- To ensure data provenance always store the information of the PRNG as well as the seed used (better even the whole code) with the data. This will allow others to reproduce your results.
- Use trusted PRNG implementations. As much as it might feel good to make your own PRNG, rely on those who are experts in creating these delicate generators.
- Know your PRNG's limits: How long is the period? Are there known problems for certain applications? Are there correlations at any time during the sequence?
- Be careful with parallel simulations.

Recommendations from : <http://arxiv.org/abs/1005.4117>

# Examples

## Fortran 90

```
program random
  implicit none
  real, dimension(10) :: vec

  ! set a random seed
  call random_seed()
  ! generate a sequence of 10 pseudo-random numbers
  call random_number( vec )
  write(*, '(5F10.6)') vec(1:5)
  write(*, '(5F10.6)') vec(6:10)
end program random
```

## output

```
0.997560  0.566825  0.965915  0.747928  0.367391
0.480637  0.073754  0.005355  0.347081  0.342244
```

In gfortran, the built in PRNG is the KISS (Keep It Simple Stupid) random number generator!



# Examples

numpy

```
>>> import numpy as np
>>> np.random.rand(3,2)
array([[ 0.90526062,  0.88377706],
       [ 0.97982437,  0.27525055],
       [ 0.69037667,  0.2779478 ]])
```

In addition to the uniform and Gaussian distribution `np.random` contains about 35 different distributions!

# Nonuniform PRNGs

Often in scientific applications one needs other pdf's than the uniform  $U(0, 1)$ .

There are mainly two methods for generating such random deviates:

- Transformation methods
- Rejection method

They both require that one or several  $U(0, 1)$  random numbers are generated.

# Transformation of random variables

- Given a probability distribution function  $p(x)$
- The corresponding *cumulative distribution function* reads

$$P(x) = \int_{-\infty}^x dx' p(x'), \quad 0 \leq P(x) \leq 1$$

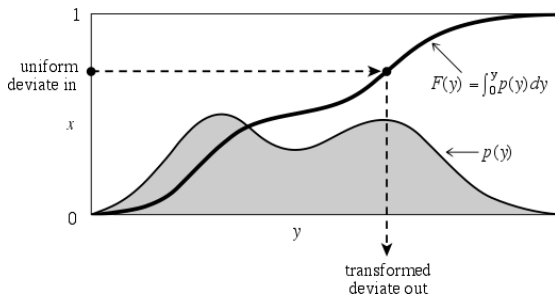
- If we are able to find the inverse of  $P(x)$ , we may construct the random deviates from  $p(x)$  as

$$P(x) = \xi \quad \Longrightarrow \quad x = P^{-1}(\xi),$$

where  $\xi = U(0, 1)$  are *uniformly distributed* random numbers

- The numbers  $x$  are then from the pdf  $p(x)$

# Transformation of random variables



## Algorithm

- 1 Generate  $\xi = U(0, 1)$
- 2 Calculate  $x = P^{-1}(\xi)$ , which is distributed according to  $p(x)$

This *hinges* on having an explicit expression for  $P^{-1}(\cdot)$ .

This is not always possible!

# Exponential deviates

- $p(x) = e^{-x} \quad 0 < x < \infty$
- $P(x) = 1 - e^{-x}$
- Transformation (with  $\xi = \xi' = U(0, 1)$ )

$$P(x) = \xi \quad \implies \quad x = -\ln(1 - \xi) = -\ln(\xi')$$

Code: from Numerical Recipes in C

```
#include <math.h>
float expdev(long *idum)
// Returns an exponentially distributed, positive,
// random deviate of unit mean, using
// ran1(idum) as the source of uniform deviates.
{
    float ran1(long *idum);
    float dum;
    do
        dum=ran1(idum);
    while (dum == 0.0);
    return -log(dum);
}
```

## Example 1

- $p(x) = \beta x^{\beta-1}$       $0 < x < 1$ ;    $\beta > 0$
- $P(x) = x^\beta$
- Transformation [with  $\xi = U(0, 1)$ ]

$$P(x) = \xi \quad \Longrightarrow \quad x = P^{-1}(\xi) = (\xi)^{1/\beta}$$

## Example 2

- $p(x) = \alpha x^{-\alpha-1}$       $1 < x < \infty$ ;    $\alpha > 0$
- $P(x) = 1 - x^{-\alpha}$
- Transformation

$$P(x) = \xi \quad \Longrightarrow \quad x = (1 - \xi)^{-1/\alpha} = (\xi')^{-1/\alpha}$$

- Probability density function

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

mean  $\mu$  and standard deviation  $\sigma$

- Cumulative distribution function (cdf)

$$P(x) = \int_{-\infty}^x dx' \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x'-\mu)^2}{2\sigma^2}\right)$$

is not known in closed form (an error-function).

- However, we do not know how to calculate  $P^{-1}(\cdot)$  in closed form!

# Gaussian deviates — The Box-Müller algorithm

The Box-Müller algorithm

Trick : go from 1D to 2D and convert to polar coordinates  $(x_1, x_2) \rightarrow (r, \phi)$   
(we here for simplicity assume  $\mu = 0$  and  $\sigma = 1$ )

$$\frac{1}{2\pi} \exp\left(-\frac{x_1^2 + x_2^2}{2}\right) dx_1 dx_2 = \frac{1}{2\pi} \exp\left(-\frac{r^2}{2}\right) r dr d\phi = p_\phi(\phi) p_r(r) dr d\phi$$
$$p_\phi(\phi) = \frac{1}{2\pi} \quad p_r(r) = r \exp\left(-\frac{r^2}{2}\right)$$

The cumulative distribution function for  $p_r(r)$

$$P_r(r) = \int_0^r dr' r' \exp\left(-\frac{r'^2}{2}\right) = 1 - \exp\left(-\frac{r^2}{2}\right)$$

can be inverted to give  $[P_r(r) = \xi]$

$$r = \sqrt{-2 \ln(1 - \xi)} = \sqrt{-2 \ln \xi'}$$



# Gaussian deviates — The Box-Müller algorithm

To generate two Gaussian random numbers we do

- 1 Generate  $\xi_1 = U(0, 1)$  and  $\xi_2 = U(0, 1)$  (uncorrelated)
- 2 Calculate

$$x_1 = r \cos(\phi) = \sqrt{-2 \ln \xi_1} \cos(2\pi\xi_2)$$
$$x_2 = r \sin(\phi) = \underbrace{\sqrt{-2 \ln \xi_1}}_r \underbrace{\sin(2\pi\xi_2)}_{\sin(\phi)}$$

- 3 The numbers  $x_1$  and  $x_2$  are uncorrelated Gaussian (or Normal) random deviates of mean 0 and standard deviation 1

Standard Gaussian random variables are typically denoted  $N(0, 1)$

Try this out for yourself! How can you make sure that the result is correct?

# Rejection method

The rejection method (von Neumann 1947) is a *general* and *powerful* technique for generating random deviates from a distribution  $p(x)$ .

In principle  $p(x)$  can be *any* pdf!

## Assumptions

- $p(x)$  is computable (and is a pdf)
- $p(x) \leq f(x)$  for  $\forall x$  ( $f$  is a comparison function)
- $F(x) = \int_{-\infty}^x dx' f(x')$  exists and is *invertible*

Note : It is always possible to choose

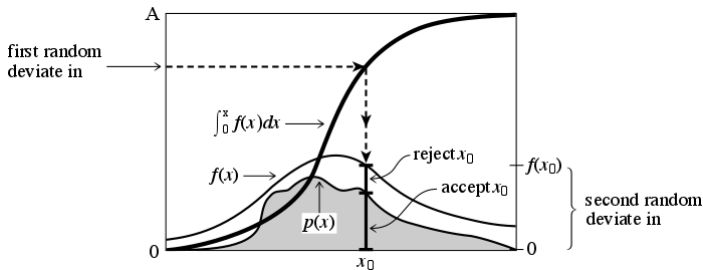
$$f(x) = \max p(x)$$

even if it may not be the best choice!

# Rejection method

## Algorithm

- 1 Generate  $\xi_1 = U(0, A)$ , where  $A = F(\infty)$
- 2 Calculate  $x_0 = F^{-1}(\xi_1)$
- 3 Generate  $\xi_2 = U(0, f(x_0))$
- 4 Number  $x_0$  should be
  - **accepted** if  $\xi_2 \leq p(x_0)$
  - **rejected** if  $\xi_2 > p(x_0)$  (and start from point 1 again)



# Rejection method

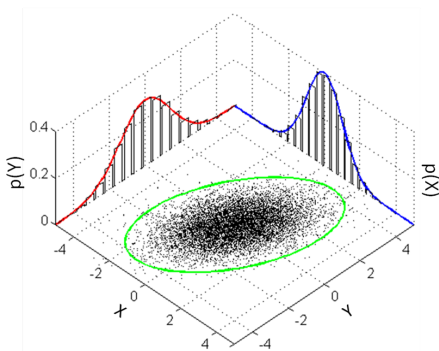
## Comments

- make sure to choose a comparison function  $f(x)$  for which  $F^{-1}(\xi)$  is known *analytically*
- the fraction of rejected points just depends on the ratio of the area under the comparison function to the area under  $p(x)$  (the details of  $F$  and  $p$  do not matter)

Note the special case where  $f(x) = p_m = \max_x p(x)$

- $\xi_1 = U(0, F(\infty))$  and  $\xi_2 = U(0, p_m)$

# Multivariate Gaussian variables



Assume

- Multi-dimensions :  $\mathbf{x} = (x_1, x_2, \dots, x_N)$
- Zero-mean :  $\langle x_i \rangle = 0$
- Covariance :  $\Sigma_{ij} = \langle x_i x_j \rangle$

# Multivariate Gaussian variables

Define the *covariance matrix*

$$\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} & \Sigma_{13} & \dots \\ \Sigma_{21} & \Sigma_{22} & \Sigma_{23} & \dots \\ \Sigma_{31} & \Sigma_{32} & \Sigma_{33} & \dots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Then the multivariate Gaussian distribution reads

$$p(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^N |\Sigma|}} \exp\left(-\frac{1}{2} \mathbf{x}^T \Sigma^{-1} \mathbf{x}\right),$$

In particular if  $\Sigma$  is diagonal it follows

$$p(\mathbf{x}) = \prod_{i=1}^N \frac{1}{\sqrt{2\pi \Sigma_{ii}}} \exp\left(-\frac{x_i^2}{2\Sigma_{ii}}\right)$$

*i.e.* , product of  $N$  uncorrelated 1D Gaussian distributions.

# Multivariate Gaussian variables

In general the quadratic form  $\mathbf{x}^T \boldsymbol{\Sigma}^{-1} \mathbf{x}$  is non-diagonal.

However, it can be diagonalized by a linear transformation  $\mathbf{x} = \mathbf{T} \mathbf{y}$  where  $\mathbf{T}$  is an orthogonal (rotation) matrix;  $\mathbf{T}^T \mathbf{T} = \mathbf{1}$

$$\begin{aligned} \mathbf{x}^T \boldsymbol{\Sigma}^{-1} \mathbf{x} &= \mathbf{y}^T \left( \mathbf{T}^T \boldsymbol{\Sigma}^{-1} \mathbf{T} \right) \mathbf{y} \\ &= \mathbf{y}^T \tilde{\boldsymbol{\Sigma}}^{-1} \mathbf{y} \\ &= \sum_{i=1}^N \frac{y_i^2}{\tilde{\Sigma}_{ii}} \end{aligned}$$

where we have defined

$$\tilde{\boldsymbol{\Sigma}} = \mathbf{T}^T \boldsymbol{\Sigma} \mathbf{T} = \mathbf{T}^{-1} \boldsymbol{\Sigma} \mathbf{T} = \left( \mathbf{T}^{-1} \boldsymbol{\Sigma}^{-1} \mathbf{T} \right)^{-1} = \left( \mathbf{T}^T \boldsymbol{\Sigma}^{-1} \mathbf{T} \right)^{-1}$$

Hence,  $\tilde{\boldsymbol{\Sigma}} = \mathbf{T}^T \boldsymbol{\Sigma} \mathbf{T}$  is diagonal for a proper choice of  $\mathbf{T}$ , and the diagonal elements are  $\tilde{\Sigma}_{ii} = \sigma_i^2$ .

# Multivariate Gaussian variables

## Algorithm: Generation of multivariate Gaussian variables

- 1 Covariance matrix  $\Sigma$  is given
- 2 Diagonalize  $\Sigma$ , that is, find its
  - eigenvalues  $\{\sigma_i^2\}_{i=1}^N$
  - eigenvector  $\{\mathbf{v}_i\}_{i=1}^N$ ;
- 3 Construct  $\mathbf{T} = [\mathbf{v}_i]$  with  $\|\mathbf{v}_i\| = 1$  (main axis transformation)
- 4 Generate  $N$  independent (uncorrelated) Gaussian random variables  $y_i$  with variance  $\sigma_i^2$ ; construct  $\mathbf{y}$
- 5 Make the transformation  $\mathbf{x} = \mathbf{T}\mathbf{y}$



# Section 10

## Ordinary differential equations

# Outline I

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods
- 8 Numerical integration

# Outline II

- 9 Random numbers
- 10 Ordinary differential equations**
- 11 Partial differential equations
- 12 Optimization

# Ordinary differential equations (ODEs)

## General definition

Consider  $N \in \mathbb{N}$ , an open subset  $U \subset \mathbb{R}^{N+2}$ ,  $I \subset \mathbb{R}$ , a function  $y : I \mapsto \mathbb{R}$ , and  $f \in \mathcal{C}^1(U, \mathbb{R})$ .  $(I, y)$  is a solution of the differential equation

$$f(x, y, y', \dots, y^{(N)}) = 0$$

if:

- $y \in \mathcal{C}^N(I, \mathbb{R})$
- $\forall x \in I, (x, y(x), y'(x), \dots, y^{(N)}(x)) \in U$
- $\forall x \in I, f(x, y(x), y'(x), \dots, y^{(N)}(x)) = 0$

This is a  $N^{\text{th}}$  order differential equation. We can further classify ODEs.

- linear
- non-linear
- homogeneous

To be made even more general

# A short classification

- An ODE is linear if  $f$  can be written as

$$f(x, y, y', \dots, y^{(N)}) = \sum_{i=0}^N \lambda_i y^{(i)} + r$$

where  $(\lambda_i)_{i \in \llbracket 0; N \rrbracket}$  and  $r$  are functions.

- A linear ODE is said homogeneous if  $r = 0$ .
- Examples:

**Linear homogeneous**

Free oscillator

$$y'' + \omega^2 y = 0$$

**Linear**

Forced oscillator

$$y'' + \omega^2 y = s$$

**Non-linear**

Pendulum equation

$$y'' + \omega^2 \sin y = 0$$

# From a $N^{\text{th}}$ order ODE to a set of $1^{\text{st}}$ order ODEs

Consider a normalized  $N^{\text{th}}$  order linear ODE of the form:

$$y^{(N)} = \sum_{i=0}^{N-1} \lambda_i y^{(i)} + r \quad (4)$$

We introduce the applications  $\mathbf{y}$ ,  $\mathbf{A}$ ,  $\mathbf{r}$ :

$$\mathbf{y} = \begin{pmatrix} y \\ y' \\ \vdots \\ y^{(N-1)} \end{pmatrix} \quad \mathbf{A} = \begin{pmatrix} 0 & 1 & & \mathbb{O} \\ & \ddots & \ddots & \\ \mathbb{O} & & 0 & 1 \\ \lambda_0 & \cdots & \lambda_{N-2} & \lambda_{N-1} \end{pmatrix} \quad \mathbf{r} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ r \end{pmatrix}$$

Then  $y$  is solution of Eq. (4) on  $I$  if and only if  $\mathbf{y}$  is solution of the set of  $1^{\text{st}}$  order ODE:

$$\mathbf{y}' = \mathbf{A}\mathbf{y} + \mathbf{r}$$

# From a $N^{\text{th}}$ order ODE to a set of $1^{\text{st}}$ order ODEs

What about non-linear ODEs? The same procedure is still possible but the set of  $1^{\text{st}}$  order ODEs is non-linear. In general the  $N^{\text{th}}$  order ODE:

$$f(x, y, y', \dots, y^{(N)}) = 0$$

can be written as

$$f(x, Y_0, Y_1, \dots, Y_{N-1}, Y'_{N-1}) = 0$$

by introducing  $Y_0 = y$  and  $\forall i \in \llbracket 1, N-1 \rrbracket, Y_i = y^{(i)}$ .

**Linear example:** 2nd Newton's law for the damped oscillator with external excitation.

$$\ddot{x}(t) = -\lambda\dot{x}(t) - \omega_0^2 x(t) + a \sin(\omega t)$$

We set

$$\mathbf{x}(t) = \begin{pmatrix} x(t) \\ \dot{x}(t) \end{pmatrix} \quad \mathbf{A} = \begin{pmatrix} 0 & 1 \\ -\omega_0^2 & -\lambda \end{pmatrix} \quad \mathbf{r}(t) = \begin{pmatrix} 0 \\ a \sin(\omega t) \end{pmatrix}$$

**Non-linear example:** 2nd Newton's law for the pendulum.

$$\ddot{\theta} = -\omega^2 \sin \theta$$

we set:  $\vartheta_0 = \theta$ , and  $\vartheta_1 = \dot{\theta}$ , hence

$$\dot{\vartheta}_0 = \vartheta_1$$

$$\dot{\vartheta}_1 = -\omega^2 \sin \vartheta_0$$



Since we can go from a  $N^{\text{th}}$  order ODE to a set of coupled 1st order ODEs, we are interested in solving a 1st order ODE of the form  $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$  on  $I \times \mathcal{C}^1(\mathbb{R}^N)$ .

- **Initial value problem:** we are looking for solution(s) verifying the initial condition

$$\mathbf{y}(x_0) = \mathbf{y}_0$$

**Note:** the condition  $\mathbf{y}(x_0) = \mathbf{y}_0$  correspond to the condition

$$y(x_0) = \tilde{y}_0, y'(x_0) = \tilde{y}_1, \dots, y^{(N-1)}(x_0) = \tilde{y}_{N-1}$$

for the corresponding  $N^{\text{th}}$  order ODE.

# Initial and boundary value problems

- **Boundary value problem:** we are looking for solution(s) verifying the boundary condition

$$\mathbf{B}(\mathbf{x}, \mathbf{y}) = 0$$

where  $\mathbf{x} = (x_1, \dots, x_m)$  is a set of boundary points,  $\mathbf{B}$  a  $N$ -dimensional algebraic operator. In practice, we have a set of  $n_1$  boundary equations at a point  $a$  and  $n_2 = N - n_1$  equations at another point  $b$ . This reads:

$$\mathbf{B}_{a,p}(a, \mathbf{y}) = 0, \quad p \in \llbracket 1, n_1 \rrbracket$$

$$\mathbf{B}_{b,q}(b, \mathbf{y}) = 0, \quad q \in \llbracket 1, n_2 \rrbracket$$

- A simple/common class of boundary conditions is to give the value of  $n_1$  components of  $\mathbf{y}$  at  $a$  and  $n_2$  components at  $b$ . For example:

$$y_p(a) = y_{a,p}, \quad p \in \llbracket 1, n_1 \rrbracket$$

$$y_q(b) = y_{b,q}, \quad q \in \llbracket n_1 + 1, N \rrbracket$$

In general, the boundary value problem is related to the initial value problem in the sense that if  $y$  is solution of the boundary value problem then there exists a set  $\{\tilde{y}_i\}_{\llbracket 1, N \rrbracket} \subset \mathbb{R}$  for which  $y$  is solution of the initial value problem associated to it.

# Cauchy-Lipschitz Theorem

Consider the Cauchy problem

$$\begin{aligned}y' &= f(x, y) \\ y(x_0) &= \tilde{y}_0\end{aligned}$$

If  $f$  is locally Lipschitz continuous with respect to the second variable, then a unique (maximal) solution exists to the Cauchy problem.

$F : U \rightarrow V$  is Lipschitz continuous if,

$$\exists k \geq 0, \quad \forall (x, y) \in U^2, \quad \|F(x) - F(y)\|_V \leq k \|x - y\|_U$$

# How to solve numerically the Cauchy problem?

We consider now a Cauchy problem in 1D. The resolution of higher dimension ODE will use what follow for each equation of the set.

The key ideas to solve the Cauchy problem are:

- Start from the initial point  $(x_0, y_0)$ .
- Discretize the interval  $I$  on which the problem must be solve. This defines a step  $h$ . More advanced method can also involve adaptive step.
- Compute the next point  $(x_{n+1}, y_{n+1})$  based on a scheme involving the previous step  $(x_n, y_n)$  (single step methods) or a set a previous steps (multiple steps methods) and the corresponding evaluation of  $f$  at those previous steps (or at intermediate steps).
- The scheme is often based on Taylor expansions and interpolation techniques.

# Single and multiple step methods

Some single step methods:

- Euler's methods
- Modified Euler's methods (e.g. midpoint method)
- Runge-Kutta methods

Multiple steps methods:

- Adams Bashforth methods (explicit)
- Adams Moulton methods (implicit)

# Euler's method - explicit and implicit

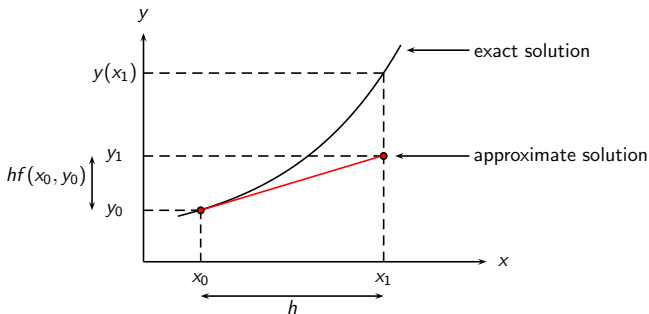
Euler's methods use the 1st order Taylor expansion of  $y$ .

$$y(x_n + h) = y(x_n) + hy'(x_n) + \mathcal{O}(h^2)$$

Knowing  $y_n$ , we construct  $y_{n+1}$  as:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

The local error from a step to another is of order  $\mathcal{O}(h^2)$ .



Euler's method is not recommended in practice.

- requires many steps  $\implies$  small  $h \implies$  numerical round off error.
- not stable

**Simple example:**  $y' = -\lambda y$  with  $\lambda > 0$

Exact solution:  $y(x) = y_0 e^{-\lambda x}$ . Hence  $y(x) \xrightarrow{x \rightarrow \infty} 0$

Approximate solution given by Euler's explicit method:  $y_n = y_0(1 - \lambda h)^n$ .  
Hence  $y_n \xrightarrow{n \rightarrow \infty} 0 \iff |1 - \lambda h| < 1$ .

Stability condition:  $h < \frac{2}{\lambda}$

Stability can be improved by using an **implicit** scheme instead.

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$

Back to previous example:

$$y_{n+1} = y_n - \lambda h y_{n+1} \iff y_{n+1} = \frac{y_n}{1 + \lambda h}$$

Hence the approximate solution reads:

$$y_n = \frac{y_0}{(1 + \lambda h)^n} \xrightarrow{n \rightarrow \infty} 0 \text{ unconditionally.}$$

**Drawback:** difficult to implement when  $f$  does not allow an explicit calculation of  $y_{n+1}$ .



# How to improve Euler's method?

To improve Euler's method one can:

- Use a higher order Taylor expansion.

**Drawback:** requires to evaluate partial derivatives of  $f$ . This method is hence not often use.

- Use intermediate points.

Midpoint method

Runge-Kutta method

# Midpoint method

Consider the midpoint between  $x_n$  and  $x_{n+1}$ :  $x_{n+1/2} = \frac{x_n + x_{n+1}}{2} = x_n + \frac{h}{2}$   
Two Taylor expansions around the midpoint:

$$y(x_n + h) = y(x_{n+1/2}) + \frac{h}{2}y'(x_{n+1/2}) + \frac{h^2}{8}y''(x_{n+1/2}) + \mathcal{O}(h^3)$$

$$y(x_n) = y(x_{n+1/2}) - \frac{h}{2}y'(x_{n+1/2}) + \frac{h^2}{8}y''(x_{n+1/2}) + \mathcal{O}(h^3)$$

By subtraction:

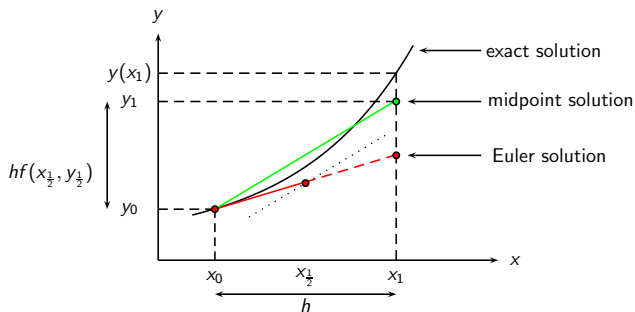
$$y(x_n + h) - y(x_n) = hy'(x_{n+1/2}) + \mathcal{O}(h^3)$$

Hence the midpoint method reads:

- 1 Use Euler's scheme with a step  $h/2$  to construct  $y_{n+1/2}$ .
- 2 Compute the slope at that midpoint  $f(x_{n+1/2}, y_{n+1/2})$ .
- 3 Use the midpoint slope to increment  $y_{n+1}$  from  $y_n$ .

The local error made with the midpoint method is  $\mathcal{O}(h^3)$ .

# Midpoint method



In practice the algorithm reads:

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$y_{n+1} = y_n + k_2$$

# Runge-Kutta methods

Euler's and the midpoint methods enters in a wider class a methods known as Runge-Kutta methods. As in the midpoint rule, the idea of higher order Runge-Kutta methods is to cancel the error terms order by order. A largely used method is the 4th-order Runge-Kutta method. The algorithm reads:

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$

$$k_4 = hf(x_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

The local error is  $\mathcal{O}(h^5)$ , but the cost is 4 evaluations of  $f$  per step.

# Multiple step methods

We have seen that steps close to the initial point are usually the most trustful, since errors add up from step to step. The key idea of multiple steps methods is to use several previous step to compute the next one.

## Adams-Bashforth explicit $m^{\text{th}}$ order scheme

We want to use an explicit scheme which is linear in the  $m$  previous step.

$$y_{n+1} = y_n + h \sum_{i=1}^m \alpha_i f(x_{n-i+1}, y_{n-i+1})$$

What should we choose for  $(\alpha_i)_{i \in \llbracket 1, m \rrbracket}$ ?

Consider the ODE:

$$y' = f(x, y)$$

and integrate it between  $x_n$  and  $x_{n+1}$ .

$$y(x_{n+1}) - y(x_n) = \int_{x_n}^{x_{n+1}} dx f(x, y(x))$$

Then the key idea is to **approximate  $f$  by its Lagrange interpolator polynomial on the  $m^{\text{th}}$  previous points**  $\{(x_{n-i+1}, y_{n-i+1})\}_{i \in \llbracket 1, m \rrbracket}$ .

The Lagrange polynomial is given by:

$$P(x) = \sum_{i=1}^m f(x_{n-i+1}, y_{n-i+1}) p_{n-i+1}(x) \quad \text{where} \quad p_i(x) = \prod_{k=1, k \neq i}^m \frac{x - x_k}{x_i - x_k}$$

Hence the approximation:

$$y(x_{n+1}) - y(x_n) \approx \int_{x_n}^{x_{n+1}} dx P(x) = \sum_{i=1}^m f(x_{n-i+1}, y_{n-i+1}) \int_{x_n}^{x_{n+1}} dx p_{n-i+1}(x)$$

Then we identify the  $(\alpha_i)_{i \in \llbracket 1, m \rrbracket}$ .

$$\alpha_i = \frac{1}{h} \int_{x_n}^{x_{n+1}} dx p_{n-i+1}(x)$$

# Adams-Bashforth methods

Note:

- The integral in  $\alpha_j$  can be computed analytically (polynomial integrand).
- For a given order  $m$ ,  $\alpha_j$  is independent of  $h$ . The coefficients can be computed once and for all.
- **The  $\{f(x_{n-i+1}, y_{n-i+1})\}_{i \in \llbracket 1, m \rrbracket}$  should be stored.**
- The local error is  $\mathcal{O}(h^m)$
- This method is the equivalent of Newton-Cotes quadrature seen previously in the course.

$m$	$\beta$	$\alpha_1/\beta$	$\alpha_2/\beta$	$\alpha_3/\beta$	$\alpha_4/\beta$
1	1	1			
2	1/2	3	-1		
3	1/12	23	-16	5	
4	1/24	55	-59	37	-9



## Adams-Moulton implicit $m + 1^{\text{th}}$ order scheme

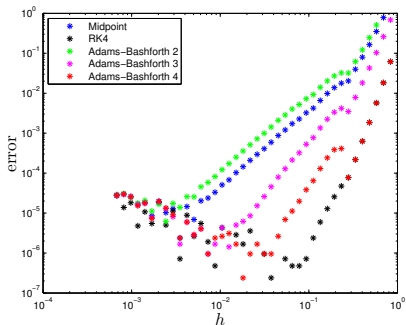
We want to use a scheme which is linear in the  $m$  previous step and  $f(x_{n+1}, y_{n+1})$ .

$$y_{n+1} = y_n + h \sum_{i=0}^m \alpha_i f(x_{n-i+1}, y_{n-i+1})$$

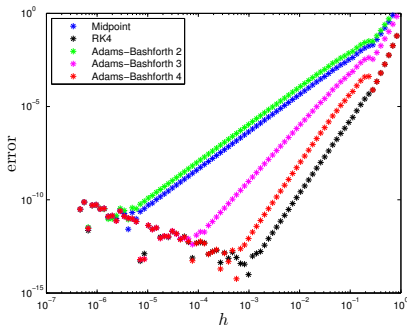
The derivation of the  $(\alpha_i)_{i \in \llbracket 1, m \rrbracket}$  is similar to the explicit method.

$m$	$\beta$	$\alpha_0/\beta$	$\alpha_1/\beta$	$\alpha_2/\beta$	$\alpha_3/\beta$
0	1	1			
1	1/2	1	1		
2	1/12	5	8	-1	
3	1/24	9	19	-5	1

# Comparison in a simple case



(a) Single precision



(b) Double precision

Cauchy problem:

$$y' = y, \quad y(0) = 1, \quad \implies y(x) = e^x$$

Error at  $x = 1$ :

$$\text{error} = |e - y_{\text{approx}}(1)|$$

The key idea is **PEC** - **P**redict **E**stimate **C**orrect:

- **P**redict  $y_{n+1}$  by using an explicit scheme.
- **E**stimate  $f(x_{n+1}, y_{n+1})$  using the predicted value of  $y_{n+1}$ .
- **C**orrect  $y_{n+1}$  by using  $f(x_{n+1}, y_{n+1})$  in an implicit scheme.

We can also estimate and correct several times within the same step until a certain accuracy is reached. These methods are then often denoted **P(EC)<sup>k</sup>E** if a fixed number of iteration is chosen or **P(EC)<sup>∞</sup>E** if a convergence condition is used. The later is preferred for stiff ODEs, in order to benefit the stabilization of the implicit scheme.

**Example:** 3<sup>rd</sup> order Adams-Bashforth-Moulton predictor-corrector method.  
Assume we have computed  $y_n, y_{n-1}, y_{n-2}$ .

- Prediction: Adams-Bashforth scheme

$$y_{n+1}^P = y_n + \frac{h}{12} [23 f(x_n, y_n) - 16 f(x_{n-1}, y_{n-1}) + 5 f(x_{n-2}, y_{n-2})]$$

- Estimate  $f_{n+1}^P = f(x_{n+1}, y_{n+1}^P)$  and correction: Adams-Moulton scheme

$$y_{n+1}^C = y_n + \frac{h}{12} [5 f_{n+1}^P + 8 f(x_n, y_n) - 1 f(x_{n-1}, y_{n-1})]$$

- Repeat **EC** using the new  $y_{n+1}^C$  until  $|y_{n+1}^{k+1} - y_{n+1}^k| < \epsilon$
- Terminate with an estimate of  $f(x_{n+1}, y_{n+1}^{\text{last}})$ .

**Principle:** adapt locally the step to reach a set precision.

To come

# Two point boundary problem - Shooting method

We know now how to solve the Cauchy problem.

Can we use the same techniques to solve a boundary problem?

Yes, this is the key idea of the **shooting method**!

- A two point boundary problem can be seen as an initial value problem.
- The condition  $y(a) = y_a$  and  $y(b) = y_b$  relates to the condition  $y(a) = y_a$  and  $y'(a) = \tilde{y}_a$ .
- Problem: how to find  $\tilde{y}_a$ ?

Consider the two problems

$$y'' = f(x, y, y'), \quad y(a) = y_a, \quad y(b) = y_b \quad (\text{BVP})$$

$$y'' = f(x, y, y'), \quad y(a) = y_a, \quad y'(a) = \tilde{y}_a \quad (\text{IVP})$$

We will denote  $y(\cdot | \tilde{y}_a)$  a solution of the IVP. Consider the difference  $F(\tilde{y}_a) = y(b | \tilde{y}_a) - y_b$ . Then the two problems are equivalent on the roots of  $F$ .

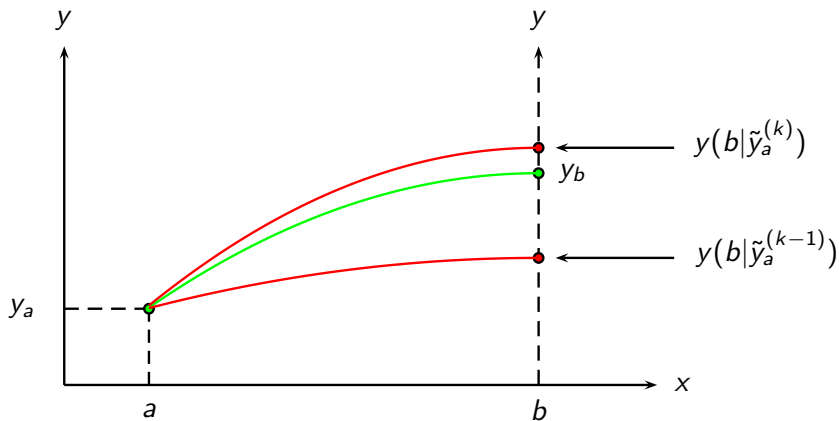
Then we have to deal with the problem of finding the roots of  $F$ , but  $F$  is not known analytically. How to proceed?

- 1 Solve the IVP with two arbitrary distinct values of  $\tilde{y}_a$ , say  $\tilde{y}_a^{(1)}$  and  $\tilde{y}_a^{(2)}$ .
- 2 Evaluate  $F(\tilde{y}_a^{(1)})$  and  $F(\tilde{y}_a^{(2)})$ . If one of them is close to 0 within a tolerance  $\epsilon$ , then problem solved (lucky you!). Otherwise
- 3 Construct a new initial condition  $\tilde{y}_a^{(k+1)}$  by the secant method (since in general  $F'$  is not known this is an alternative to Newton-Raphson method).

$$\tilde{y}_a^{(k+1)} = \tilde{y}_a^{(k)} - \frac{\tilde{y}_a^{(k)} - \tilde{y}_a^{(k-1)}}{F(\tilde{y}_a^{(k)}) - F(\tilde{y}_a^{(k-1)})} F(\tilde{y}_a^{(k)})$$

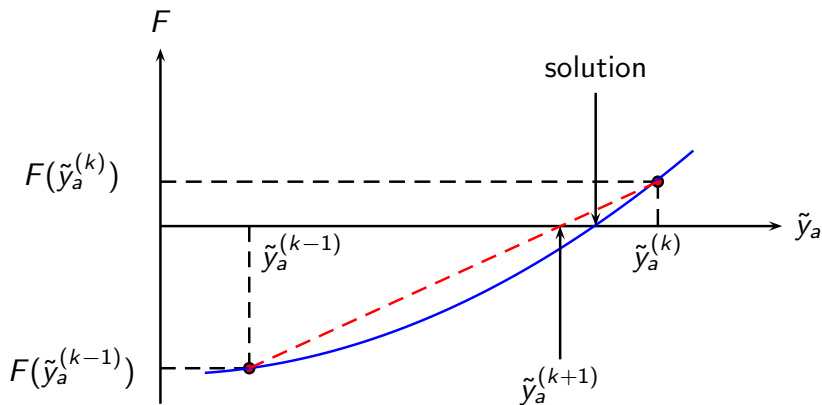
- 4 Repeat 2 and 3 until  $|F(\tilde{y}_a^{(k)})| < \epsilon$ .

# Shooting method





# Shooting method



## Note:

- You can show as an exercise that in the linear case, the solution is given by one iteration.
- In the non-linear case, neither the existence or the unicity of a solution is *a priori* guaranty in general.

# Shooting method in $N$ -dimension

- We have treated the particular ODE  $y'' = f(x, y, y')$ .
- The same idea can be applied in a  $N^{\text{th}}$ -order ODE:  $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$ .
- Consider a boundary condition of the form:

$$B_{a,p}(a, \mathbf{y}) = 0, \quad p \in \llbracket 1, n_1 \rrbracket$$
$$B_{b,q}(b, \mathbf{y}) = 0, \quad q \in \llbracket 1, n_2 \rrbracket.$$

This can be for example to specify the values of  $n_1$  components of  $\mathbf{y}$  at  $a$  and  $n_2 = N - n_1$  components at  $b$ .

- In order to use IVP methods we need to specify the  $N$  components of  $\mathbf{y}$  at  $a$ . Then we impose the  $n_1$  components with their true boundary values and the  $n_2$  remaining are set 'randomly', and seen as free parameters.
- Introduce the vector of free parameters  $\mathbf{v} = (v_1, \dots, v_{n_2})$ .

# Shooting method in $N$ -dimension

- The initial point  $\mathbf{y}(a)$  is then given by:

$$\mathbf{y}(a) = \tilde{\mathbf{y}}(a, \mathbf{v})$$

where  $\tilde{\mathbf{y}}$  already contains the  $n_1$  true boundary values at  $a$ .

- Then given this initial condition, we can use IVP methods to solve this particular Cauchy problem and determine  $\mathbf{y}(b|\mathbf{v})$ .
- Using the boundary functional  $B_{b,i}$ ,  $i \in \llbracket 1, n_2 \rrbracket$ , we can construct a discrepancy vector  $\mathbf{F}$  such that:

$$F_i(\mathbf{v}) = B_{b,i}(b, \mathbf{y}(b|\mathbf{v}))$$

- We must then use an iteration scheme to find  $\mathbf{v}$  that cancels  $\mathbf{F}$ .
- ND Newton-Raphson scheme

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \delta\mathbf{v}, \quad \mathbf{J} \delta\mathbf{v} = -\mathbf{F}$$

$$J_{ij} = \frac{\partial F_i}{\partial v_j} \approx \frac{F_i(v_1, \dots, v_j + \Delta v_j, \dots, v_{n_2}) - F_i(v_1, \dots, v_{n_2})}{\Delta v_j}$$

- ODEs approximate by finite-difference equations (FDEs) on a grid.
- Example:

$$y' = f(x, y) \quad \rightarrow \quad \frac{y_n - y_{n-1}}{x_n - x_{n-1}} = f\left(\frac{1}{2}(x_n + x_{n-1}), \frac{1}{2}(y_n + y_{n-1})\right)$$

- For a set of  $N$  coupled 1st order ODEs, we want to determine values of  $\mathbf{y}$  on  $M$  mesh points (i.e.  $N \times M$  values in total).
- The idea: start with a guess of the  $N \times M$  values and make them converge to the true value using an adequate iterative scheme.
- How to find an adequate iterative scheme?

- Consider a mesh of  $M$  points,  $k \in \llbracket 1, M \rrbracket$ , with associate abscissa  $x_k$ ,  $x_1 = a$  and  $x_M = b$ .
- Define:

$$\mathbf{E}_1 = \mathbf{B}_a(x_1, \mathbf{y}_1)$$

$$\mathbf{E}_k = \mathbf{y}_k - \mathbf{y}_{k-1} - (x_k - x_{k-1}) \mathbf{f}_k(x_k, x_{k-1}, \mathbf{y}_k, \mathbf{y}_{k-1}), \quad k \in \llbracket 2, M \rrbracket$$

$$\mathbf{E}_{M+1} = \mathbf{B}_b(x_M, \mathbf{y}_M)$$

where  $\mathbf{f}_k$  is scheme dependent, here implicit and evaluated from two points.

- A satisfactory solution is reached when it cancels  $\mathbf{E}$ .
- From a guess  $\mathbf{y}$ , we construct a new candidate  $\mathbf{y} + \Delta\mathbf{y}$  satisfying  $\mathbf{E}(\mathbf{y} + \Delta\mathbf{y}) = 0$ . We hence Taylor expand  $\mathbf{E}(\mathbf{y} + \Delta\mathbf{y})$  and solve for  $\Delta\mathbf{y}$ .

# Relaxation method

Taylor expansion for  $k \in \llbracket 2, M \rrbracket$ :

$$\begin{aligned} \mathbf{E}_k(\mathbf{y}_k + \Delta\mathbf{y}_k, \mathbf{y}_{k-1} + \Delta\mathbf{y}_{k-1}) &\approx \mathbf{E}_k(\mathbf{y}_k, \mathbf{y}_{k-1}) \\ &+ \sum_{n=1}^N \frac{\partial \mathbf{E}_k}{\partial y_{n,k-1}} \Delta y_{n,k-1} + \sum_{n=1}^N \frac{\partial \mathbf{E}_k}{\partial y_{n,k}} \Delta y_{n,k} \end{aligned}$$

Solving for  $\Delta\mathbf{y}$ :

$$\sum_{n=1}^N S_{j,n} \Delta y_{n,k-1} + \sum_{n=N+1}^{2N} S_{j,n} \Delta y_{n-N,k} = -E_{j,k}, \quad j \in \llbracket 1, N \rrbracket$$

where

$$S_{j,n} = \frac{\partial E_{j,k}}{\partial y_{n,k-1}}, \quad S_{j,n+N} = \frac{\partial E_{j,k}}{\partial y_{n,k}}, \quad n \in \llbracket 1, N \rrbracket$$

$\mathbf{S} \in \mathfrak{M}_{N,2N}(\mathbb{R})$  for each  $k$ .

The Taylor expansion at the boundary gives:

- At  $k = 1$

$$\sum_{n=1}^N S_{j,n} \Delta y_{n,1} = -E_{j,1}, \quad j \in \llbracket n_2 + 1, N \rrbracket$$

with

$$S_{j,n} = \frac{\partial E_{j,1}}{\partial y_{n,1}}, \quad n \in \llbracket 1, N \rrbracket$$

- At  $k = M + 1$

$$\sum_{n=1}^N S_{j,n} \Delta y_{n,M} = -E_{j,M+1}, \quad j \in \llbracket 1, n_2 \rrbracket$$

with

$$S_{j,n} = \frac{\partial E_{j,M+1}}{\partial y_{n,M}}, \quad n \in \llbracket 1, N \rrbracket$$



We have then a linear system to solve for  $\Delta\mathbf{y}$ .

# Section 11

## Partial differential equations

# Outline I

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods
- 8 Numerical integration

- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations**
  - Parabolic PDEs
  - Hyperbolic PDEs
  - Elliptic PDEs
  - Other methods
- 12 Optimization

# Partial differential equations (PDEs) in physics

In physics, PDEs appear in numerous contexts; Some examples are

- Diffusion (or heat) equation

$$D\nabla^2\phi(\mathbf{x}, t) = \frac{\partial\phi(\mathbf{x}, t)}{\partial t}$$

- Wave equation

$$\nabla^2 u(\mathbf{x}, t) = \frac{1}{c^2} \frac{\partial^2 u(\mathbf{x}, t)}{\partial t^2}$$

- Poisson (or Laplace) equation

$$\nabla^2\varphi(\mathbf{x}) = -\frac{\rho(\mathbf{x})}{\epsilon}$$

- Schrödinger equation

$$-\frac{\hbar^2}{2m}\nabla^2\Psi(\mathbf{x}, t) + V(\mathbf{x})\Psi(\mathbf{x}, t) = i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{x}, t)$$

**Question:** How can one numerically solve such equations?

# Classification of PDEs

Consider the general 2nd order PDE

$$A(x, y) \frac{\partial^2 u}{\partial x^2} + 2B(x, y) \frac{\partial^2 u}{\partial x \partial y} + C(x, y) \frac{\partial^2 u}{\partial y^2} = F(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y})$$

PDEs are classified as:

- **Parabolic**

- $B^2 - AC = 0$
- Diffusion/heat equation :  $u_{xx} = u_t$

- **Hyperbolic**

- $B^2 - AC > 0$
- Wave equation :  $u_{xx} - u_{tt} = 0$

- **Elliptic**

- $B^2 - AC < 0$
- Laplace's equation :  $u_{xx} + u_{yy} = 0$

# Classification of PDEs

## Some comments about PDEs in physics

- Parabolic problems in most cases describe an evolutionary process that leads to a steady state described by an elliptic equation.
- Hyperbolic equations describe the transport of some physical quantities or information, such as waves.
- Elliptic equations describe a special state of a physical system, which is characterized by the minimum of certain quantity (often energy).

The numerical solution methods for PDEs depend strongly on what class the PDE belongs to!

# Parabolic PDEs



In one dimension we have thus the following equation

$$\nabla^2 u(x, t) = \frac{\partial u(x, t)}{\partial t}, \quad \text{or} \quad u_{xx} = u_t,$$

with initial conditions, i.e., the conditions at  $t = 0$ ,

$$u(x, 0) = g(x), \quad 0 \leq x \leq L$$

with  $L = 1$  the length of the  $x$ -region of interest.

The boundary conditions are

$$\begin{aligned} u(0, t) &= a(t), & t \geq 0, \\ u(L, t) &= b(t), & t \geq 0, \end{aligned}$$

where  $a(t)$  and  $b(t)$  are two functions which depend on time only, while  $g(x)$  depends only on the position  $x$ .

# Diffusion equation: Explicit Scheme, Forward Euler

Parabolic PDEs

**Notation :**  $u(x_i, t_j) \equiv u_{i,j}$

Finite difference approximations

$$\frac{\partial u(x_i, t_j)}{\partial t} \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t},$$

and

$$\frac{\partial^2 u(x_i, t_j)}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}.$$

The one-dimensional diffusion equation can then be rewritten in its discretized version as

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}.$$

Defining  $\alpha = \Delta t / \Delta x^2$  results in the explicit scheme

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} + \alpha u_{i+1,j}.$$

# Diffusion equation: Explicit Scheme

Parabolic PDEs

Defining  $U_j = [u_{1,j}, u_{2,j}, \dots, u_{N,j}]^T$  gives

$$U_{j+1} = AU_j$$

with

$$A = \begin{pmatrix} 1 - 2\alpha & \alpha & 0 & 0 \dots \\ \alpha & 1 - 2\alpha & \alpha & 0 \dots \\ \dots & \dots & \dots & \dots \\ 0 \dots & 0 \dots & \alpha & 1 - 2\alpha \end{pmatrix}$$

yielding

$$U_{j+1} = AU_j = \dots = A^j U_0$$

The vector  $U_0$  is known from the initial conditions!

The explicit scheme, although being rather simple to implement has a very weak stability condition given by the CFL (Courant, Friedrichs, Levy) condition (we have used  $D = 1$ )

$$D \frac{\Delta t}{\Delta x^2} \leq 1/2$$

# Diffusion equation: Implicit Scheme

Parabolic PDEs

Choose now

$$\frac{\partial u(x_i, t_j)}{\partial t} \approx \frac{u(x_i, t_j) - u(x_i, t_j - k)}{k}$$

and

$$\frac{\partial^2 u(x_i, t_j)}{\partial x^2} \approx \frac{u(x_i + h, t_j) - 2u(x_i, t_j) + u(x_i - h, t_j)}{h^2}$$

Define  $\alpha = k/h^2$ , gives

$$u_{i,j-1} = -\alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} - \alpha u_{i+1,j}$$

Here  $u_{i,j-1}$  is the only known quantity.

# Diffusion equation: : Implicit Scheme

Parabolic PDEs

In matrix form one has

$$AU_j = U_{j-1}$$

with

$$A = \begin{pmatrix} 1 + 2\alpha & -\alpha & 0 & 0 \dots \\ -\alpha & 1 + 2\alpha & -\alpha & 0 \dots \\ \dots & \dots & \dots & \dots \\ 0 \dots & 0 \dots & -\alpha & 1 + 2\alpha \end{pmatrix}$$

which gives

$$U_j = A^{-1} U_{j-1} = \dots = A^{-j} U_0$$

Need only to invert a (tridiagonal) matrix.

The implicit Euler scheme is **unconditionally stable** (for the Euclidian and Infinity norms).

# Diffusion equation: Brute Force Implicit Scheme, inefficient algorithm

Parabolic PDEs

Some Fortran code:

```
! now invert the matrix
CALL matinv( a, ndim, det)
DO i = 1, m
  DO l=1, ndim
    u(l) = DOT_PRODUCT( a(l,:), v(:) )
  ENDDO
  v = u
  t = i*k
  DO j=1, ndim
    WRITE(6,*) t, j*h, v(j)
  ENDDO
ENDDO
```

**Question:** Why is this algorithm inefficient?

# Diffusion equation: Brief Summary of the Explicit and the Implicit Methods

Parabolic PDEs

- Explicit methods is straightforward to code, but avoid doing the matrix vector multiplication since the matrix is tridiagonal

$$u_t \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} = \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}$$

- Implicit method can be applied in a brute force way as well as long as the element of the matrix are constants

$$u_t \approx \frac{u(x, t) - u(x, t - \Delta t)}{\Delta t} = \frac{u(x_i, t_j) - u(x_i, t_j - \Delta t)}{\Delta t}$$

- However, it is more efficient to use a linear algebra solver for tridiagonal matrices.

# Diffusion equation: Crank-Nicolson scheme

Parabolic PDEs

The Crank-Nicolson scheme combines the explicit and implicit scheme!

It is defined by (with  $\theta = 1/2$ )

$$\begin{aligned} \frac{\theta}{\Delta x^2} (u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) + \frac{1-\theta}{\Delta x^2} (u_{i+1,j-1} - 2u_{i,j-1} + u_{i-1,j-1}) \\ = \frac{1}{\Delta t} (u_{i,j} - u_{i,j-1}), \end{aligned}$$

- $\theta = 0$  yields the forward formula for the first derivative and the explicit scheme
- $\theta = 1$  yields the backward formula and the implicit scheme
- For  $\theta = 1/2$  we obtain a new scheme after its inventors, Crank and Nicolson!



# Diffusion equation: Crank-Nicolson scheme

## Parabolic PDEs

Using our previous definition of  $\alpha = \Delta t / \Delta x^2$  we can rewrite the latter equation as

$$-\alpha u_{i-1,j} + (2 + 2\alpha) u_{i,j} - \alpha u_{i+1,j} = \alpha u_{i-1,j-1} + (2 - 2\alpha) u_{i,j-1} + \alpha u_{i+1,j-1},$$

or in matrix-vector form as

$$(2I + \alpha B) U_j = (2I - \alpha B) U_{j-1},$$

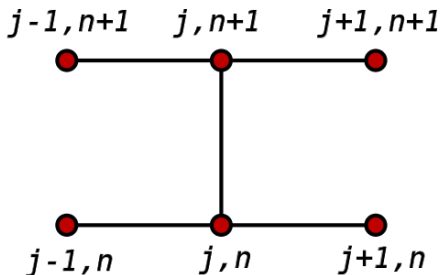
where the vector  $U_j$  is the same as defined in the implicit case while the matrix  $B$  is

$$B = \begin{pmatrix} 2 & -1 & 0 & 0 \dots \\ -1 & 2 & -1 & 0 \dots \\ \dots & \dots & \dots & \dots \\ 0 \dots & 0 \dots & \dots & 2 \end{pmatrix}$$

# Diffusion equation: Crank-Nicolson scheme

Parabolic PDEs

The Crank-Nicolson stencil using notation  $u(x_j, t_n)$



We now want to take a look at stability and accuracy!

Some relevant questions are

- How accurate is a discretization scheme?
- Is it stable for some or any choice of  $\Delta x$  and  $\Delta t$ ?
- How can this information be obtained (if relevant)?
- *etc*

Accuracy is studied by Taylor expanding the scheme around a common point!

# Diffusion equation: Analysis

## Parabolic PDEs

We start with the forward Euler scheme and Taylor expand  $u(x, t + \Delta t)$ ,  $u(x + \Delta x, t)$  and  $u(x - \Delta x, t)$  about  $(x, t)$

$$u(x + \Delta x, t) = u(x, t) + \frac{\partial u(x, t)}{\partial x} \Delta x + \frac{\partial^2 u(x, t)}{2\partial x^2} \Delta x^2 + \mathcal{O}(\Delta x^3),$$

$$u(x - \Delta x, t) = u(x, t) - \frac{\partial u(x, t)}{\partial x} \Delta x + \frac{\partial^2 u(x, t)}{2\partial x^2} \Delta x^2 + \mathcal{O}(\Delta x^3),$$

$$u(x, t + \Delta t) = u(x, t) + \frac{\partial u(x, t)}{\partial t} \Delta t + \mathcal{O}(\Delta t^2).$$

With these Taylor expansions the approximations for the derivatives takes the form

$$\left[ \frac{\partial u(x, t)}{\partial t} \right]_{\text{approx}} = \frac{\partial u(x, t)}{\partial t} + \mathcal{O}(\Delta t),$$
$$\left[ \frac{\partial^2 u(x, t)}{\partial x^2} \right]_{\text{approx}} = \frac{\partial^2 u(x, t)}{\partial x^2} + \mathcal{O}(\Delta x^2).$$

It is easy to convince oneself that the backward Euler method must have the same truncation errors as the forward Euler scheme. □

# Diffusion equation: Analysis

## Parabolic PDEs

For the Crank-Nicolson scheme we also need to Taylor expand  $u(x + \Delta x, t + \Delta t)$  and  $u(x - \Delta x, t + \Delta t)$  around  $t' = t + \Delta t/2$ .

$$u(x + \Delta x, t + \Delta t) = u(x, t') + \frac{\partial u(x, t')}{\partial x} \Delta x + \frac{\partial u(x, t')}{\partial t} \frac{\Delta t}{2} + \frac{\partial^2 u(x, t')}{2\partial x^2} \Delta x^2 + \frac{\partial^2 u(x, t')}{2\partial t^2} \frac{\Delta t^2}{4} + \frac{\partial^2 u(x, t')}{\partial x \partial t} \frac{\Delta t}{2} \Delta x + \mathcal{O}(\Delta t^3)$$

$$u(x - \Delta x, t + \Delta t) = u(x, t') - \frac{\partial u(x, t')}{\partial x} \Delta x + \frac{\partial u(x, t')}{\partial t} \frac{\Delta t}{2} + \frac{\partial^2 u(x, t')}{2\partial x^2} \Delta x^2 + \frac{\partial^2 u(x, t')}{2\partial t^2} \frac{\Delta t^2}{4} - \frac{\partial^2 u(x, t')}{\partial x \partial t} \frac{\Delta t}{2} \Delta x + \mathcal{O}(\Delta t^3)$$

$$u(x + \Delta x, t) = u(x, t') + \frac{\partial u(x, t')}{\partial x} \Delta x - \frac{\partial u(x, t')}{\partial t} \frac{\Delta t}{2} + \frac{\partial^2 u(x, t')}{2\partial x^2} \Delta x^2 + \frac{\partial^2 u(x, t')}{2\partial t^2} \frac{\Delta t^2}{4} - \frac{\partial^2 u(x, t')}{\partial x \partial t} \frac{\Delta t}{2} \Delta x + \mathcal{O}(\Delta t^3)$$

$$u(x - \Delta x, t) = u(x, t') - \frac{\partial u(x, t')}{\partial x} \Delta x - \frac{\partial u(x, t')}{\partial t} \frac{\Delta t}{2} + \frac{\partial^2 u(x, t')}{2\partial x^2} \Delta x^2 + \frac{\partial^2 u(x, t')}{2\partial t^2} \frac{\Delta t^2}{4} + \frac{\partial^2 u(x, t')}{\partial x \partial t} \frac{\Delta t}{2} \Delta x + \mathcal{O}(\Delta t^3)$$

$$u(x, t + \Delta t) = u(x, t') + \frac{\partial u(x, t')}{\partial t} \frac{\Delta t}{2} + \frac{\partial^2 u(x, t')}{2\partial t^2} \Delta t^2 + \mathcal{O}(\Delta t^3)$$

$$u(x, t) = u(x, t') - \frac{\partial u(x, t')}{\partial t} \frac{\Delta t}{2} + \frac{\partial^2 u(x, t')}{2\partial t^2} \Delta t^2 + \mathcal{O}(\Delta t^3)$$

# Diffusion equation: Analysis

## Parabolic PDEs

We now insert these expansions in the approximations for the derivatives to find

$$\left[ \frac{\partial u(x,t')}{\partial t} \right]_{\text{approx}} = \frac{\partial u(x,t')}{\partial t} + \mathcal{O}(\Delta t^2),$$
$$\left[ \frac{\partial^2 u(x,t')}{\partial x^2} \right]_{\text{approx}} = \frac{\partial^2 u(x,t')}{\partial x^2} + \mathcal{O}(\Delta x^2).$$

Hence the truncation error of the Crank-Nicolson scheme is  $\mathcal{O}(\Delta t^2)$  and  $\mathcal{O}(\Delta x^2)$ , i.e. 2nd order in both space and time!

The following table summarizes the three methods.

<i>Scheme:</i>	<i>Truncation Error:</i>	<i>Stability requirements:</i>
Crank-Nicolson	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t^2)$	Stable for all $\Delta t$ and $\Delta x$ .
Backward Euler	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t)$	Stable for all $\Delta t$ and $\Delta x$ .
Forward Euler	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t)$	$\Delta t \leq \frac{1}{2} \Delta x^2$

**Table:** Comparison of the different schemes.

The stability criterion is derived by using the von Neumann stability criterion<sup>10</sup> where the error is decomposed into Fourier series.

A finite difference scheme is stable if the errors made at one time step of the calculation do not cause the errors to increase as the computations are continued.

<sup>10</sup>[http://en.wikipedia.org/wiki/Von\\_Neumann\\_stability\\_analysis](http://en.wikipedia.org/wiki/Von_Neumann_stability_analysis)

# Diffusion equation: Analysis

Parabolic PDEs

Illustration of the von Neumann stability criteria for the forward Euler scheme.

To be added.....!

In the mean time consult e.g. [http:](http://en.wikipedia.org/wiki/Von_Neumann_stability_analysis)

[//en.wikipedia.org/wiki/Von\\_Neumann\\_stability\\_analysis!](http://en.wikipedia.org/wiki/Von_Neumann_stability_analysis)



# Diffusion equation: Analysis

## Parabolic PDEs

It cannot be repeated enough, it is always very useful to find cases where one can compare the numerics and the developed algorithms and codes with analytic solution.

The above case is also particularly simple. We have the following partial differential equation

$$\nabla^2 u(x, t) = \frac{\partial u(x, t)}{\partial t},$$

with initial conditions

$$u(x, 0) = g(x) \quad 0 < x < L.$$

The boundary conditions are

$$u(0, t) = 0 \quad t \geq 0, \quad u(L, t) = 0 \quad t \geq 0,$$

# Diffusion equation: Analysis

Parabolic PDEs

We assume that we have solutions of the form (separation of variable)

$$u(x, t) = F(x)G(t),$$

which inserted in the partial differential equation results in

$$\frac{F''}{F} = \frac{G'}{G},$$

where the derivative is with respect to  $x$  on the left hand side and with respect to  $t$  on right hand side. This equation should hold for all  $x$  and  $t$ . We must require the rhs and lhs to be equal to a constant.

We call this constant  $-\lambda^2$ . This gives us the two differential equations,

$$F'' + \lambda^2 F = 0; \quad G' = -\lambda^2 G,$$

with general solutions

$$F(x) = A \sin(\lambda x) + B \cos(\lambda x); \quad G(t) = C e^{-\lambda^2 t}.$$

# Diffusion equation: Analysis

## Parabolic PDEs

To satisfy the boundary conditions we require  $B = 0$  and  $\lambda = n\pi/L$ . One solution is therefore found to be

$$u(x, t) = A_n \sin(n\pi x/L) e^{-n^2 \pi^2 t/L^2}.$$

But there are infinitely many possible  $n$  values (infinite number of solutions).

Moreover, the diffusion equation is linear and because of this we know that a superposition of solutions will also be a solution of the equation.

We may therefore write

$$u(x, t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L) e^{-n^2 \pi^2 t/L^2}.$$

# Diffusion equation: Analysis

Parabolic PDEs

The coefficient  $A_n$  is in turn determined from the initial condition. We require

$$u(x, 0) = g(x) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L).$$

The coefficient  $A_n$  is the Fourier coefficients for the function  $g(x)$ . Because of this,  $A_n$  is given by (from the theory on Fourier series)

$$A_n = \frac{2}{L} \int_0^L dx g(x) \sin(n\pi x/L)$$

Different  $g(x)$  functions will obviously result in different results for  $A_n$ .

## Numerical examples:

Let look at the following diffusion equation

$$\begin{aligned}\partial_t C(x, t) &= \partial_x [D(x)\partial_x C(x, t)] \\ C(x, 0) &= C_0 \delta(x)\end{aligned}$$

where now the diffusion constant,  $D(x)$ , is position dependent (and  $C_0$  some constant).

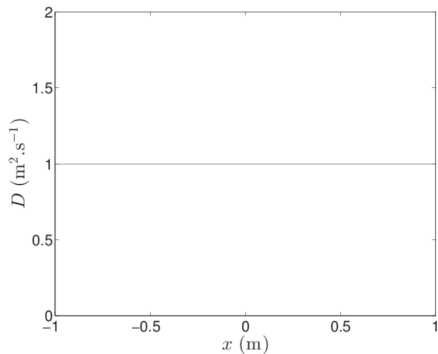
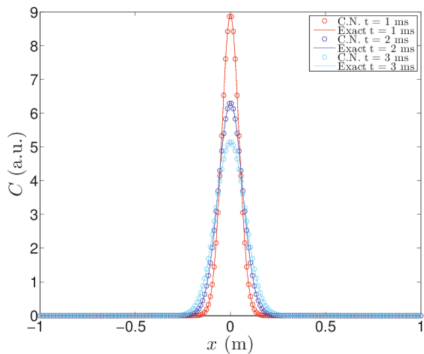
So how should the discretization scheme for this PDE be?

In particular, how should the spatial derivatives be discretized?

# Diffusion equation: Some examples

Parabolic PDEs

Simulation results: Diffusion with  $D(x) = 1$  (everywhere)

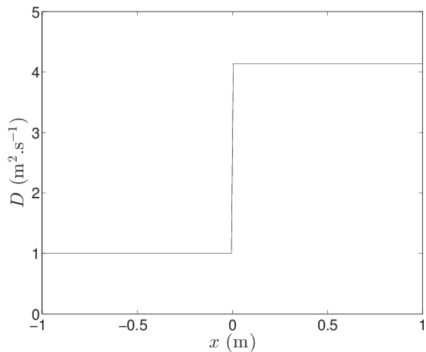
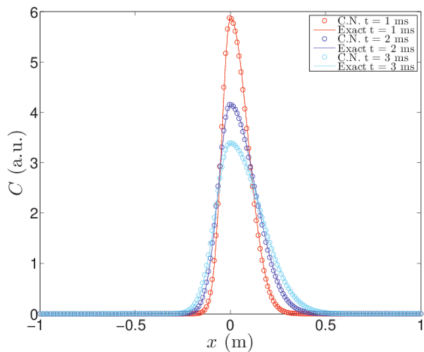


(using Crank-Nicolson)

# Diffusion equation: Some examples

Parabolic PDEs

Simulation results: Diffusion with piecewise constant  $D(x)$



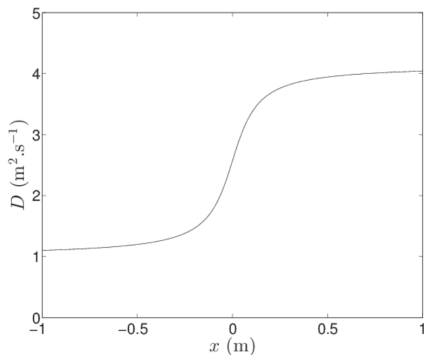
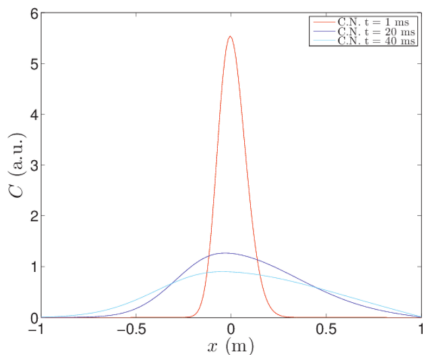
**Exercise:** Reproduce these results!

# Diffusion equation: Some examples

Parabolic PDEs

Simulation results: Crank-Nicolson for the position dependent diffusion constant

$$D(x) = 1 + \frac{\pi}{2} + \arctan\left(\frac{x - x_0}{\Delta}\right)$$



Parameters used :  $\Delta = 0.1\text{m}$  and  $x_0 = 0\text{m}$



# Diffusion equation: Some examples

## Parabolic PDEs

**Subroutine** Crank\_NicolsonDx (A,B,dx,dt)

**Implicit** None

**Integer** :: i, n

**Complex(wp)** :: dt, dx, Dp, Dm, alpha

**Complex(wp)** :: A(:, :), B(:, :)

*!A: matrix in front of  $U^{(n+1)}$*

*!B: matrix in front of  $U^{(n)}$*

*! Martices in 3-column form*

n = **size**(A,1)

alpha = dt/(2.\_wp\*dx\*\*2)

*! Diffusion constant D(x)*

Dp = diffusivity((1-n/2)\*dx+dx/2.\_wp)

Dm = diffusivity((1-n/2)\*dx-dx/2.\_wp)

*!*

A(1,1) = 0.\_wp

A(1,2) = 1.\_wp + alpha\*(Dp + Dm)

A(1,3) = - alpha\*Dp

B(1,1) = 0.\_wp

B(1,2) = 1.\_wp - alpha\*(Dp + Dm)

B(1,3) = alpha\*Dp

# Diffusion equation: Some examples

Parabolic PDEs

Code continues:

```
do i=2,n-1
  Dp = diffusivity((i-n/2)*dx+dx/2._wp)
  Dm = diffusivity((i-n/2)*dx-dx/2._wp)
```

```
  A(i,1) = - alpha*Dm
  A(i,2) = 1._wp + alpha*(Dp + Dm)
  A(i,3) = - alpha*Dp
```

```
  B(i,1) = alpha*Dm
  B(i,2) = 1._wp - alpha*(Dp + Dm)
  B(i,3) = alpha*Dp
```

```
end do
```

```
end Subroutine Crank_NicolsonDx
```

# Hyperbolic PDEs

# Two-dimensional wave equation

## Hyperbolic PDEs

Consider first the two-dimensional wave equation for a vibrating square membrane given by the following initial and boundary conditions

$$\begin{aligned}\lambda \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) &= \frac{\partial^2 u}{\partial t^2} & x, y \in [0, 1], \quad t \geq 0 \\ u(x, y, 0) &= \sin(\pi x) \sin(2\pi y) & x, y \in (0, 1) \\ u &= 0 \text{ boundary} & t \geq 0 \\ \partial u / \partial t|_{t=0} &= 0 & x, y \in (0, 1)\end{aligned}$$

The boundary is defined by  $x = 0$ ,  $x = 1$ ,  $y = 0$  and  $y = 1$ . For simplicity we set  $\lambda = 1$ .

# Two-dimensional wave equation

## Hyperbolic PDEs

Our equations depend on three variables whose discretized versions are now

$$\begin{cases} t_l = l\Delta t & l \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n_x \\ y_j = j\Delta y & 0 \leq j \leq n_y \end{cases},$$

and we will let  $\Delta x = \Delta y = h$  and  $n_x = n_y$  for the sake of simplicity.

We have now the following discretized partial derivatives [ $u(x_i, y_j, t_l) = u_{i,j}^l$ ]

$$u_{xx} \approx \frac{u_{i+1,j}^l - 2u_{i,j}^l + u_{i-1,j}^l}{h^2},$$

$$u_{yy} \approx \frac{u_{i,j+1}^l - 2u_{i,j}^l + u_{i,j-1}^l}{h^2},$$

and

$$u_{tt} \approx \frac{u_{i,j}^{l+1} - 2u_{i,j}^l + u_{i,j}^{l-1}}{\Delta t^2}.$$

# Two-dimensional wave equation

## Hyperbolic PDEs

We merge this into the discretized 2 + 1-dimensional wave equation as

$$u_{i,j}^{l+1} = 2u_{i,j}^l - u_{i,j}^{l-1} + \frac{\Delta t^2}{h^2} \left( u_{i+1,j}^l - 4u_{i,j}^l + u_{i-1,j}^l + u_{i,j+1}^l + u_{i,j-1}^l \right),$$

where again we have an explicit scheme with  $u_{i,j}^{l+1}$  as the only unknown quantity. It is easy to account for different step lengths for  $x$  and  $y$ . The partial derivative is treated in much the same way as for the one-dimensional case, except that we now have an additional index due to the extra spatial dimension, viz., we need to compute  $u_{i,j}^{-1}$  through

$$u_{i,j}^{-1} = u_{i,j}^0 + \frac{\Delta t}{2h^2} \left( u_{i+1,j}^0 - 4u_{i,j}^0 + u_{i-1,j}^0 + u_{i,j+1}^0 + u_{i,j-1}^0 \right),$$

in our setup of the initial conditions.

# Two-dimensional wave equation: Code example

## Hyperbolic PDEs

We show here how to implement the two-dimensional wave equation

```
// After initializations and declaration of variables
for ( int i = 0; i < n; i++ ) {
    u[i] = new double [n];
    uLast[i] = new double [n];
    uNext[i] = new double [n];
    x[i] = i*h;
    y[i] = x[i];
}
// initializing
for ( int i = 0; i < n; i++ ) { // setting initial step
    for ( int j = 0; j < n; j++ ) {
        uLast[i][j] = sin(Pl*x[i])*sin(2*Pl*y[j]);
    }
}
```

# Two-dimensional wave equation: Code example

Hyperbolic PDEs

```
for ( int i = 1; i < (n-1); i++ ) { // setting first step
    using the initial derivative
    for ( int j = 1; j < (n-1); j++ ) {
        u[i][j] = uLast[i][j] - ((tStep*tStep)/(2.0*h*h))*
            (4*uLast[i][j] - uLast[i+1][j] - uLast[i-1][j] - uLast[
                i][j+1] - uLast[i][j-1]);
    }
    u[i][0] = 0; // setting boundaries once and for all
    u[i][n-1] = 0;
    u[0][i] = 0;
    u[n-1][i] = 0;

    uNext[i][0] = 0;
    uNext[i][n-1] = 0;
    uNext[0][i] = 0;
    uNext[n-1][i] = 0;
}
```



# Two-dimensional wave equation: Code example

## Hyperbolic PDEs

```
// iterating in time
double t = 0.0 + tStep;
int iter = 0;

while ( t < tFinal ) {
    iter ++;
    t = t + tStep;

    for ( int i = 1; i < (n-1); i++ ) { // computing next step
        for ( int j = 1; j < (n-1); j++ ) {
            uNext[i][j] = 2*u[i][j] - uLast[i][j] - ((tStep*tStep)
                /(h*h))*
                (4*u[i][j] - u[i+1][j] - u[i-1][j] - u[i][j+1] - u[i
                ][j-1]);
        }
    }
}
```

# Two-dimensional wave equation: Code example

Hyperbolic PDEs

```
for ( int i = 1; i < (n-1); i++ ) { // shifting results
    down
    for ( int j = 1; j < (n-1); j++ ) {
        uLast[i][j] = u[i][j];
        u[i][j] = uNext[i][j];
    }
}
```

# Wave equation : Closed form solution

## Hyperbolic PDEs

We develop here the analytic solution for the 2 + 1 dimensional wave equation with the following boundary and initial conditions

$$\begin{aligned}c^2(u_{xx} + u_{yy}) &= u_{tt} & x, y \in (0, L), t > 0 \\u(x, y, 0) &= f(x, y) & x, y \in (0, L) \\u(0, 0, t) &= u(L, L, t) = 0 & t > 0 \\\frac{\partial u}{\partial t} \Big|_{t=0} &= g(x, y) & x, y \in (0, L)\end{aligned}$$

# Wave equation : Closed form solution

Hyperbolic PDEs

Our first step is to make the ansatz

$$u(x, y, t) = F(x, y)G(t),$$

resulting in the equation

$$FG_{tt} = c^2(F_{xx}G + F_{yy}G),$$

or

$$\frac{G_{tt}}{c^2G} = \frac{1}{F}(F_{xx} + F_{yy}) = -\nu^2.$$

The lhs and rhs are independent of each other and we obtain two differential equations

$$F_{xx} + F_{yy} + F\nu^2 = 0,$$

and

$$G_{tt} + Gc^2\nu^2 = G_{tt} + G\lambda^2 = 0,$$

with  $\lambda = c\nu$ .

# Wave equation : Closed form solution

## Hyperbolic PDEs

We can in turn make the following ansatz for the  $x$  and  $y$  dependent part

$$F(x, y) = H(x)Q(y),$$

which results in

$$\frac{1}{H}H_{xx} = -\frac{1}{Q}(Q_{yy} + Q\nu^2) = -\kappa^2.$$

Since the lhs and rhs are again independent of each other, we can separate the latter equation into two independent equations, one for  $x$  and one for  $y$ , namely

$$H_{xx} + \kappa^2 H = 0,$$

and

$$Q_{yy} + \rho^2 Q = 0,$$

with  $\rho^2 = \nu^2 - \kappa^2$ .

# Wave equation : Closed form solution

## Hyperbolic PDEs

The second step is to solve these differential equations, which all have trigonometric functions as solutions, viz.

$$H(x) = A \cos(\kappa x) + B \sin(\kappa x),$$

and

$$Q(y) = C \cos(\rho y) + D \sin(\rho y).$$

The boundary conditions require that  $F(x, y) = H(x)Q(y)$  are zero at the boundaries, meaning that  $H(0) = H(L) = Q(0) = Q(L) = 0$ . This yields the solutions

$$H_m(x) = \sin\left(\frac{m\pi x}{L}\right) \quad Q_n(y) = \sin\left(\frac{n\pi y}{L}\right),$$

or

$$F_{mn}(x, y) = \sin\left(\frac{m\pi x}{L}\right) \sin\left(\frac{n\pi y}{L}\right).$$

# Wave equation : Closed form solution

## Hyperbolic PDEs

With  $\rho^2 = \nu^2 - \kappa^2$  and  $\lambda = c\nu$  we have an eigenspectrum  $\lambda = c\sqrt{\kappa^2 + \rho^2}$  or  $\lambda_{mn} = c\pi/L\sqrt{m^2 + n^2}$ . The solution for  $G$  is

$$G_{mn}(t) = B_{mn} \cos(\lambda_{mn}t) + D_{mn} \sin(\lambda_{mn}t),$$

with the general solution of the form

$$u(x, y, t) = \sum_{mn=1}^{\infty} u_{mn}(x, y, t) = \sum_{mn=1}^{\infty} F_{mn}(x, y) G_{mn}(t).$$

# Wave equation : Closed form solution

## Hyperbolic PDEs

The final step is to determine the coefficients  $B_{mn}$  and  $D_{mn}$  from the Fourier coefficients. The equations for these are determined by the initial conditions  $u(x, y, 0) = f(x, y)$  and  $\partial u / \partial t|_{t=0} = g(x, y)$ .

The final expressions are

$$B_{mn} = \frac{2}{L} \int_0^L \int_0^L dx dy f(x, y) \sin\left(\frac{m\pi x}{L}\right) \sin\left(\frac{n\pi y}{L}\right),$$

and

$$D_{mn} = \frac{2}{L} \int_0^L \int_0^L dx dy g(x, y) \sin\left(\frac{m\pi x}{L}\right) \sin\left(\frac{n\pi y}{L}\right).$$

Inserting the particular functional forms of  $f(x, y)$  and  $g(x, y)$  one obtains the final analytic expressions.



# Two-dimensional wave equation

## Hyperbolic PDEs

We can check our results as function of the number of mesh points and in particular against the stability condition

$$\Delta t \leq \frac{1}{\sqrt{\lambda}} \left( \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}$$

where  $\Delta t$ ,  $\Delta x$  and  $\Delta y$  are the chosen step lengths. In our case  $\Delta x = \Delta y = h$ . How do we find this condition? In one dimension we can proceed as we did for the diffusion equation.

# Two-dimensional wave equation

## Hyperbolic PDEs

The analytic solution of the wave equation in  $2 + 1$  dimensions has a characteristic wave component which reads

$$u(x, y, t) = A \exp(i(k_x x + k_y y - \omega t))$$

Then from

$$u_{xx} \approx \frac{u'_{i+1,j} - 2u'_{i,j} + u'_{i-1,j}}{\Delta x^2},$$

we get, with  $u_j = \exp(ikx_j)$

$$u_{xx} \approx \frac{u_j}{\Delta x^2} (\exp ik\Delta x - 2 + \exp(-ik\Delta x)),$$

or

$$u_{xx} \approx 2 \frac{u_j}{\Delta x^2} (\cos(k\Delta x) - 1) = -4 \frac{u_j}{\Delta x^2} \sin^2(k\Delta x/2)$$

We get similar results for  $t$  and  $y$ .

# Two-dimensional wave equation

## Hyperbolic PDEs

We have

$$\lambda \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \frac{\partial^2 u}{\partial t^2},$$

resulting in

$$\lambda \left[ -4 \frac{u'_{ij}}{\Delta x^2} \sin^2 \left( \frac{k_x \Delta x}{2} \right) - 4 \frac{u'_{ij}}{\Delta y^2} \sin^2 \left( \frac{k_y \Delta y}{2} \right) \right] = -4 \frac{u'_{ij}}{\Delta t^2} \sin^2 \left( \frac{\omega \Delta t}{2} \right),$$

resulting in

$$\sin \left( \frac{\omega \Delta t}{2} \right) = \pm \sqrt{\lambda} \Delta t \left[ \frac{1}{\Delta x^2} \sin^2 \left( \frac{k_x \Delta x}{2} \right) + \frac{1}{\Delta y^2} \sin^2 \left( \frac{k_y \Delta y}{2} \right) \right]^{1/2}.$$

The squared sine functions can at most be unity. The frequency  $\omega$  is real and our wave is neither damped nor amplified.

# Two-dimensional wave equation

## Hyperbolic PDEs

We have

$$\sin\left(\frac{\omega\Delta t}{2}\right) = \pm\sqrt{\lambda}\Delta t \left[ \frac{1}{\Delta x^2} \sin^2\left(\frac{k_x\Delta x}{2}\right) + \frac{1}{\Delta y^2} \sin^2\left(\frac{k_y\Delta y}{2}\right) \right]^{1/2}.$$

The squared sine functions can at most be unity.  $\omega$  is real and our wave is neither damped nor amplified. The numerical  $\omega$  must also be real which is the case when  $\sin(\omega\Delta t/2)$  is less than or equal to unity, meaning that

$$\Delta t \leq \frac{1}{\sqrt{\lambda}} \left( \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}.$$

# Two-dimensional wave equation

## Hyperbolic PDEs

We modify now the wave equation in order to consider a  $2 + 1$  dimensional wave equation with a position dependent velocity, given by

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (\lambda(x, y) \nabla u).$$

If  $\lambda$  is constant, we obtain the standard wave equation discussed in the two previous points. The solution  $u(x, y, t)$  could represent a model for water waves. It represents then the surface elevation from still water. We can model  $\lambda$  as

$$\lambda = gH(x, y),$$

with  $g$  being the acceleration of gravity and  $H(x, y)$  is the still water depth. The function  $H(x, y)$  simulates the water depth using for example measurements of still water depths in say a fjord or the north sea. The boundary conditions are then determined by the coast lines as discussed in point d) below. We have assumed that the vertical motion is negligible and that we deal with long wavelenghts  $\tilde{\lambda}$  compared with the depth of the sea  $H$ , that is  $\tilde{\lambda}/H \gg 1$ . We neglect normally Coriolis effects in such calculations.

# Two-dimensional wave equation

## Hyperbolic PDEs

You can discretize

$$\nabla \cdot (\lambda(x, y) \nabla u) = \frac{\partial}{\partial x} \left( \lambda(x, y) \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y} \left( \lambda(x, y) \frac{\partial u}{\partial y} \right),$$

as follows using again a quadratic domain for  $x$  and  $y$ :

$$\frac{\partial}{\partial x} \left( \lambda(x, y) \frac{\partial u}{\partial x} \right) \approx \frac{1}{\Delta x} \left( \lambda_{i+1/2, j} \left[ \frac{u'_{i+1, j} - u'_{i, j}}{\Delta x} \right] - \lambda_{i-1/2, j} \left[ \frac{u'_{i, j} - u'_{i-1, j}}{\Delta x} \right] \right),$$

and

$$\frac{\partial}{\partial y} \left( \lambda(x, y) \frac{\partial u}{\partial y} \right) \approx \frac{1}{\Delta y} \left( \lambda_{i, j+1/2} \left[ \frac{u'_{i, j+1} - u'_{i, j}}{\Delta y} \right] - \lambda_{i, j-1/2} \left[ \frac{u'_{i, j} - u'_{i, j-1}}{\Delta y} \right] \right).$$

# Two-dimensional wave equation

## Hyperbolic PDEs

How did we do that? Look at the derivative wrt  $x$  only:

First we compute the derivative

$$\frac{d}{dx} \left( \lambda(x) \frac{du}{dx} \right) \Big|_{x=x_i} \approx \frac{1}{\Delta x} \left( \lambda \frac{du}{dx} \Big|_{x=x_{i+1/2}} - \lambda \frac{du}{dx} \Big|_{x=x_{i-1/2}} \right),$$

where we approximated it at the midpoint by going half a step to the right and half a step to the left. Then we approximate

$$\lambda \frac{du}{dx} \Big|_{x=x_{i+1/2}} \approx \lambda_{x_{i+1/2}} \frac{u_{i+1} - u_i}{\Delta x},$$

and similarly for  $x = x_i - 1/2$ .

# Elliptic PDEs



# Laplace's and Poisson's equations

## Elliptic PDEs

Laplace's equation (in 2 spatial dimensions) reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0.$$

with possible boundary conditions  $u(x, y) = g(x, y)$  on the border. There is no time-dependence. Choosing equally many steps in both directions we have a quadratic or rectangular grid, depending on whether we choose equal steps lengths or not in the  $x$  and the  $y$  directions. Here we set  $\Delta x = \Delta y = h$  and obtain a discretized version

$$u_{xx} \approx \frac{u(x+h, y) - 2u(x, y) + u(x-h, y)}{h^2},$$

and

$$u_{yy} \approx \frac{u(x, y+h) - 2u(x, y) + u(x, y-h)}{h^2},$$

# Laplace's and Poisson's equations

Elliptic PDEs

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2},$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2},$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4} [u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}]. \quad (5)$$

This is our final numerical scheme for solving Laplace's equation. Poisson's equation adds only a minor complication to the above equation since in this case we have

$$u_{xx} + u_{yy} = -\rho(\mathbf{x}),$$

and we need only to add a discretized version of  $\rho(\mathbf{x})$  resulting in

$$u_{i,j} = \frac{1}{4} [u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}] + \rho_{i,j}. \quad (6)$$

The way we solve these equations is based on an iterative scheme we discussed in connection with linear algebra, namely the so-called Jacobi, Gauss-Seidel and relaxation methods. The steps are rather simple.

- We start with an initial guess for  $u_{i,j}^{(0)}$  where all values are known.
- To obtain a new solution we solve Eq. (5) or Eq. (6) in order to obtain a new solution  $u_{i,j}^{(1)}$ .
- Most likely this solution will not be a solution to Eq. (5). This solution is in turn used to obtain a new and improved  $u_{i,j}^{(2)}$ .
- We continue this process till we obtain a result which satisfies some specific convergence criterion.

# Code example for the two-dimensional diffusion equation/Laplace

```
int DiffusionJacobi(int N, double dx, double dt,  
                   double **A, double **q, double abstol){  
    int i,j,k;  
    int maxit = 100000;  
    double sum;  
    double ** Aold = CreateMatrix(N,N);  
  
    double D = dt/(dx*dx);
```

# Code example for the two-dimensional diffusion equation/Laplace

```
for(i=1; i<N-1; i++)
  for(j=1; j<N-1; j++)
    Aold[i][j] = 1.0;
/* Boundary Conditions — all zeros */
for(i=0; i<N; i++){
  A[0][i] = 0.0;
  A[N-1][i] = 0.0;
  A[i][0] = 0.0;
  A[i][N-1] = 0.0;
}
```

# Code example for the two-dimensional diffusion equation/Laplace

```
for(k=0; k<maxit; k++){
    for(i = 1; i<N-1; i++){
        for(j=1; j<N-1; j++){
            A[i][j] = dt*q[i][j] + Aold[i][j] +
                D*(Aold[i+1][j] + Aold[i][j+1] - 4.0*Aold[i][j] +
                    Aold[i-1][j] + Aold[i][j-1]);
        }
    }
    sum = 0.0;
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            sum += (Aold[i][j]-A[i][j])*(Aold[i][j]-A[i][j]);
            Aold[i][j] = A[i][j];
        }
    }
    if (sqrt(sum)<abstol) { DestroyMatrix (Aold ,N,N);
        return k;
    }
}
```

# Other methods

## Other methods

- Finite-element method
- Boundary element methods



# Section 12

## Optimization

# Outline I

- 1 Introduction
- 2 Number representation and numerical precision
- 3 Finite differences and interpolation
- 4 Linear algebra
- 5 How to install libraries on a Linux system
- 6 Eigenvalue problems
- 7 Spectral methods
- 8 Numerical integration

- 9 Random numbers
- 10 Ordinary differential equations
- 11 Partial differential equations

## 12 Optimization

- Optimization in one dimensions
- Bisection method
- Downhill simplex method
- Conjugate gradient methods
- Quasi-Newton method
- $\chi^2$ -minimization
- Genetic algorithms
- Constrained Optimization: Linear programming
- Constrained Optimization: Non-Linear programming
- Simulated annealing

Some initial comments:

- Optimization is a wide field.
- Sometimes there exists a well-established method that can be used in a black-box manner
- but many optimization problems are **true challenges**

Note :

- Maximizing  $f(\mathbf{x})$  is equivalent to minimizing  $-f(\mathbf{x})!$
- It is intimately related to methods of solving non-linear equations (for  $\nabla f(\mathbf{x}) = 0$ ).

# Optimization (minimization/maximization)

## Introduction

To minimize a function  $f(\mathbf{x})$  in  $D$  dimensions,  $\mathbf{x} \in \mathbb{R}^D$ , one has to consider several issues:

- What is the dimensionality  $D$ ?
- Are there constraints on  $\mathbf{x}$ ?
- Is the problem linear or non-linear?
- Is  $f(\mathbf{x})$  such that minimization is a smooth downhill process, or are there traps in the form of local minima?
- Do we want the global minimum of  $f(\mathbf{x})$ , or is it sufficient to make a local minimization?
- Do we have access to derivatives of  $f(\mathbf{x})$ ?

Some classes of optimization problems with examples of related methods

- Global optimization
  - Simulated annealing
- Local optimization without derivatives
  - Bisection method
  - Downhill simplex method
- Local optimization with derivatives
  - Quasi-Newton
  - Conjugate gradient
- Constrained optimization
  - Linear programming
  - Non-linear programming

# Optimization (minimization/maximization)

## Introduction

General remark on precision: Let  $x_m$  be the true minimum of the function  $f(x)$  [we assume  $D = 1$ ], so that

$$f(x) \approx f(x_m) + \frac{1}{2}(x - x_m)^2 f''(x_m); \quad \text{or} \quad f(x) - f(x_m) \approx \frac{1}{2}(x - x_m)^2 f''(x_m)$$

- If  $|x - x_m|$  is so small that  $|f(x) - f(x_m)|$  is of the order the floating-point precision,  $\varepsilon$ , one cannot expect to get closer to  $x_m$  by *any* method
- This gives an estimate of the error in an estimate  $x$  for the minimum  $x_m = x \pm \Delta x$

$$\varepsilon \approx f(x) - f(x_m) \approx \frac{1}{2}(x - x_m)^2 f''(x_m)$$

so that the error in  $x_m$  becomes

$$\Delta x_m \geq \sqrt{\varepsilon} \sqrt{\frac{2f(x_m)}{f''(x_m)}}$$

The error in  $x_m$  ( $\sim \sqrt{\varepsilon}$ ) is typically much larger than the error in  $f(x_m)$  ( $\sim \varepsilon$ )!

For one-dimensional optimization, there exist several methods that are both relatively robust and fast!

Some examples are:

- Bracketing
- Bisection method
- Golden search
- Brent's method (parabolic interpolation)
- Newtons method



# Bisection method

## Optimization

Alex covered this....!

Newtons methods also works well for one-dimensions.

Taylor expansion gives

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(x_n)(x - x_n)^2 + \dots$$

Neglecting terms of second order in  $x$  gives

$$f(x_{n+1}) \approx 0 \qquad f(x_n) + f'(x_n)(x_{n+1} - x_n) \approx 0$$

which results in the the iteration scheme

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

This is Newton's method also known as NewtonRaphson method.

# Downhill simplex method

## Optimization

- The Downhill simplex method is an example of local optimization without derivatives in multi-dimensions!
- does not rely on information on the gradient
- viewed as a generalization of bracketing to multi-dimensions

A simplex in  $D$  dimensions is a geometrical object defined by  $D + 1$  vertices.

- $D = 1$  : a line segment
- $D = 2$  : a triangle
- $D = 3$  : a tetrahedron
- etc.

A simplex is a dynamical object that can grow and shrink; When it reaches the minimum, it gives up and shrinks down around it.

The simplex method is iterative!

The Downhill simplex method consists of the following steps

- 1 Start from a simplex defined by vertices  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{D+1}$  defined so that

$$f(\mathbf{x}_1) \geq f(\mathbf{x}_2) \geq \dots \geq f(\mathbf{x}_{D+1}).$$

- 2 Elementary move: Improve the worst point, *i.e.*  $\mathbf{x}_1$ , by moving in direction of the mean of the better points

$$\mathbf{x}_{\text{mean}} = \frac{1}{D} \sum_{i=2}^{D+1} \mathbf{x}_i$$

by trying

$$\mathbf{x}_a = \mathbf{x}_{\text{mean}} + \lambda(\mathbf{x}_{\text{mean}} - \mathbf{x}_1)$$

for some  $\lambda \in \mathbb{R}$  and  $\mathbf{x}_a \rightarrow \mathbf{x}_1$  if improved.

Illustration

<https://notendur.hi.is/jonasson/olbest/sim-anneal1.pdf>

# Multi-dimensional Taylor expansion

## Optimization

Let  $f(\mathbf{x})$  be a function in  $\mathbf{x} \in \mathbb{R}^D$  with continuous 1st and 2nd partial derivatives

Then it follows that

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \mathbf{J}(\mathbf{x}) \cdot \Delta\mathbf{x} + \frac{1}{2} \Delta\mathbf{x}^T \mathbf{H}(\mathbf{x}) \Delta\mathbf{x}$$

where we have introduced the

- Jacobian

$$J_i(f)(\mathbf{x}) = \partial_i f(\mathbf{x})$$

$$\mathbf{J}(f)(\mathbf{x}) = \nabla f(\mathbf{x})$$

- Hessian matrix

$$H_{ij}(f)(\mathbf{x}) = \partial_i \partial_j f(\mathbf{x})$$

$$\mathbf{H}(f)(\mathbf{x}) = \nabla \otimes \nabla f(\mathbf{x})$$

Note that the Hessian and Jacobian are related by  $H_{ij}(f)(\mathbf{x}) = J_i(\partial_j f)(\mathbf{x})$ .

# Multi-dimensional Taylor expansion

## Optimization

The Hessian matrix corresponding to the function  $f(\mathbf{x})$  written out in detail:

$$\mathbf{H}(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

# Multi-dimensional Taylor expansion

## Optimization

Minimizing the function  $f(\mathbf{x})$  in  $D$ -dimensions, can be viewed as solving  $\nabla f(\mathbf{x}) = \mathbf{0}$ , which is a system of  $D$  generally non-linear equations

One has

$$f(\mathbf{x}') = f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot (\mathbf{x}' - \mathbf{x}) + \frac{1}{2}(\mathbf{x}' - \mathbf{x})^T \mathbf{H}(\mathbf{x})(\mathbf{x}' - \mathbf{x}) + \dots$$

Now taking the gradient with respect to primed coordinates gives

$$\nabla' f(\mathbf{x}') = \nabla f(\mathbf{x}) + \mathbf{H}(\mathbf{x})(\mathbf{x}' - \mathbf{x}) + \dots$$

If the true minimum is at  $\mathbf{x}'$ , so that  $\nabla' f(\mathbf{x}') = \mathbf{0}$ , one has after neglecting higher order terms

$$\mathbf{H}(\mathbf{x})(\mathbf{x}' - \mathbf{x}) = -\nabla f(\mathbf{x}),$$

that is, a linear system in  $\delta \mathbf{x} = \mathbf{x}' - \mathbf{x}$  with the Hessian being the system matrix and  $\mathbf{b} = -\nabla f(\mathbf{x})$  being the right-hand-side!

Methods from solving linear systems can be used for minimizing functions!

For instance, if the Hessian is available, we could solve the linear system

$$\mathbf{H}(\mathbf{x})(\mathbf{x}' - \mathbf{x}) = -\nabla f(\mathbf{x})$$

to find the minimum  $\mathbf{x}'$  by the use of the conjugate gradient method that we have seen previously.

Note that the Hessian (for a well-behaved function) is symmetric as required!

We will therefore not discuss this further here!



To calculating the true minimum using  $\nabla' f(\mathbf{x}') = \mathbf{0}$  and

$$f(\mathbf{x}') = f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot (\mathbf{x}' - \mathbf{x}) + \frac{1}{2}(\mathbf{x}' - \mathbf{x})^T \mathbf{H}(\mathbf{x})(\mathbf{x}' - \mathbf{x}) + \dots$$

has some disadvantages:

- 1st and 2nd derivatives are needed
- the inverse of the Hessian matrix must be calculated

These issues makes the above methods in many cases impractical (but used in e.g. the *Levenberg-Marquardt method* for  $\chi^2$ -minimization).

Instead in the *quasi-Newton method*, one performs the following iterative steps starting from some initial guess  $\mathbf{x}_0$

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \lambda_i \mathcal{H}_i(\mathbf{x}) \cdot \nabla f(\mathbf{x}_i)$$

where  $\lambda_i$  is determined by line minimization, where  $\mathcal{H}_i$  is a  $D \times D$  matrix constructed so that  $\mathcal{H}_i \rightarrow \mathbf{H}^{-1}$  when  $i \rightarrow \infty$ .

See Numerical Recipes (or other sources) for details!

Quasi-Newton (or variable metric methods) in multi-dimensions come in two flavors (which only differ in some minor details):

- Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm
  - [http://en.wikipedia.org/wiki/BFGS\\_method](http://en.wikipedia.org/wiki/BFGS_method)
- Davidon-Fletcher-Powell (DFP) formula
  - [http://en.wikipedia.org/wiki/DFP\\_updating\\_formula](http://en.wikipedia.org/wiki/DFP_updating_formula)
- other variants
  - [http://en.wikipedia.org/wiki/Quasi-Newton\\_method](http://en.wikipedia.org/wiki/Quasi-Newton_method)

The main idea is to base iterations on the multi-dimensional Taylor expansion, but using an approximate Hessian constructed “as one goes”!

In quasi-Newton methods the Hessian matrix does not need to be computed. The Hessian is updated by analyzing successive gradient vectors instead.

Comments : In practical applications, the BFGS approach seems to be preferred!

## Some comments

- The quasi-Newton method is often comparable in efficiency to the conjugate-gradient method
- However it requires more memory if  $D$  is large
- Memory requirement scales as  $D^2$  for the quasi-Newton method and as  $D$  for the conjugate-gradient method.
- Unlike the conjugate-gradient method, the quasi-Newton methods do not require the Hessian matrix to be evaluated directly.

- We just saw that we may solve the often non-linear equations  $\nabla f(\mathbf{x}) = \mathbf{0}$  to find the minimum for  $f(\mathbf{x})$ .
- Usually this is an impractical method (as mentioned previously)
- Exception: curve fitting
- **Problem** : Fit the data points  $(x_i, y_i)$  with errors  $\sigma_i$  ( $i = 1, \dots, N$ ) to a functional form  $f(x, \mathbf{a})$ , where  $\mathbf{a} = (a_1, \dots, a_M)$  are parameters ( $M < N$ )
- **Cost function** : This can be done by minimizing the cost function

$$\chi^2(\mathbf{a}) = \sum_{i=1}^N \left( \frac{y_i - f(x_i; \mathbf{a})}{\sigma_i} \right)^2$$

with respect to  $\mathbf{a}$ .

## Two possibilities

- Linear problems :  $f(x; \mathbf{a})$  depends linearly on  $a_i$  for all  $i$ .
- Non-linear problems :  $f(x; \mathbf{a})$  depends non-linearly on  $a_i$ .
  - Levenberg-Marquardt algorithm

The linear problems result in a linear system of equations that is simple to solve; the non-linear problem is harder!

## Linear $\chi^2$ -fitting

Assume that  $f(x, \mathbf{a})$  is a linear combination of some basis functions  $f_k(x)$

$$f(x, \mathbf{a}) = \sum_{k=1}^M a_k f_k(x)$$

The cost function becomes

$$\chi^2(\mathbf{a}) = \sum_{i=1}^N \left( \frac{y_i - f(x_i; \mathbf{a})}{\sigma_i} \right)^2 = \sum_{i=1}^N \left( b_i - \sum_{k=1}^M A_{ik} a_k \right)^2 = |\mathbf{b} - \mathbf{A}\mathbf{a}|$$

where  $b_i = y_i/\sigma_i$  and  $A_{ik} = f_k(x_i)/\sigma_i$ .

Now minimizing  $\chi^2(\mathbf{a})$  gives

$$\nabla \chi^2(\mathbf{a}) = \mathbf{0} \quad \Rightarrow \quad \mathbf{A}^T \mathbf{A} \mathbf{a} = \mathbf{A}^T \mathbf{b}$$

which can be solved, for instance, by the singular value decomposition (why?)!

Non-Linear  $\chi^2$ -fitting: Levenberg-Marquardt method (is one possibility)

The **Levenberg-Marquardt** method can be taught of as a combination of the steepest descent and Newton method!

- Cost function

$$\chi^2(\mathbf{a}) = \sum_{i=1}^N \left( \frac{y_i - f(x_i; \mathbf{a})}{\sigma_i} \right)^2$$

- Gradient (or Jacobian)

$$\frac{\partial \chi^2(\mathbf{a})}{\partial a_k} = -2 \sum_{i=1}^N \frac{y_i - f(x_i; \mathbf{a})}{\sigma_i^2} \frac{\partial f(x_i; \mathbf{a})}{\partial a_k}$$

- Hessian

$$\frac{\partial^2 \chi^2(\mathbf{a})}{\partial a_k \partial a_l} = 2 \sum_{i=1}^N \frac{1}{\sigma_i^2} \left[ \frac{\partial f(x_i; \mathbf{a})}{\partial a_k} \frac{\partial f(x_i; \mathbf{a})}{\partial a_l} - \{y_i - f(x_i; \mathbf{a})\} \frac{\partial^2 f(x_i; \mathbf{a})}{\partial a_k \partial a_l} \right]$$

- Hessian (continued): The last term of the Hessian should be small if we are near the minimum (so that the fit is good). Note that it vanishes exactly if  $f$  depends linearly on  $a_i$ .

For convenience, this term is neglected to give an approximate Hessian

$$\frac{\partial^2 \chi^2(\mathbf{a})}{\partial \mathbf{a}_k \partial \mathbf{a}_l} \approx \mathbf{A}_{kl} = 2 \sum_{i=1}^N \frac{1}{\sigma_i^2} \frac{\partial f(x_i; \mathbf{a})}{\partial \mathbf{a}_k} \frac{\partial f(x_i; \mathbf{a})}{\partial \mathbf{a}_l}$$

- Defining

$$\beta_k = -\frac{1}{2} \frac{\partial \chi^2(\mathbf{a})}{\partial \mathbf{a}_k} \quad \alpha_{kl} = \frac{1}{2} \mathbf{A}_{kl} = \frac{1}{2} \frac{\partial^2 \chi^2(\mathbf{a})}{\partial \mathbf{a}_k \partial \mathbf{a}_l} \quad \delta \mathbf{a}_k = \mathbf{a}'_k - \mathbf{a}_k$$

- A steepest decent step would be  $\delta \mathbf{a}_k = \text{const } \beta_k$ , but the LM-method uses a modified steepest decent step



- Step type I: The “steepest decent like” step used in the LM-algorithm is

$$\delta \mathbf{a}_k = \frac{1}{\lambda \alpha_{kk}} \beta_k$$

where  $\lambda$  is a constant (see below) and  $\alpha_{kk}$  is introduced because the different components  $a_k$  may behave very differently

- Note that  $\alpha_{kk}$  is guaranteed to be positive due to the use of the approximate Hessian
- Step type II: A Newton step becomes:

$$\mathbf{a}' = \mathbf{a} - \mathbf{A}(\mathbf{a})^{-1} \nabla \chi^2(\mathbf{a}) \qquad \sum_I \alpha_{kI} \delta \mathbf{a}_I = \beta_k$$

- The Levenberg-Marquardt method can be seen as an elegant way to interpolate between these two types of steps, by changing the parameter  $\lambda$

- This is achieved by considering the following “LM-equation system”

$$\sum_l \tilde{\alpha}_{kl} \delta a_l = \beta_l \quad \tilde{\alpha}_{kl} = \begin{cases} \alpha_{kk}(1 + \lambda) & k = l \\ \alpha_{kl} & k \neq l \end{cases}$$

- Observations

- $\lambda \rightarrow 0$  : a Newton step because  $\tilde{\alpha}_{kl}$  and  $\alpha_{kl}$  are the same in this limit
- $\lambda \gg 1$  (or  $\lambda \rightarrow \infty$ ): modified Steepest decent step because the diagonal elements dominates the matrix  $\tilde{\alpha}_{kl}$

## The steps of the Levenberg-Marquardt method

- 1 Pick some initial  $\mathbf{a}$  (“guess parameters”) and a large value for  $\lambda$
- 2 Solve the “LM-equation system”

$$\sum_l \tilde{\alpha}_{kl} \delta \mathbf{a}_l = \beta_k$$

for  $\delta \mathbf{a}$  and calculate  $\chi^2(\mathbf{a} + \delta \mathbf{a})$

- 3 Update
  - If  $\chi^2(\mathbf{a} + \delta \mathbf{a}) < \chi^2(\mathbf{a})$ , update  $\mathbf{a} \rightarrow \mathbf{a}_{new} = \mathbf{a} + \delta \mathbf{a}$ ; decrease  $\lambda$  and continue to step 2
  - If  $\chi^2(\mathbf{a} + \delta \mathbf{a}) \geq \chi^2(\mathbf{a})$ , increase  $\lambda$  (decreasing the step size) and go back to step 2 without changing  $\mathbf{a}$ .

Iterate till some stopping criterion is reached!

## The steps of the Levenberg-Marquardt method

- 1 Pick some initial  $\mathbf{a}$  (“guess parameters”) and a large value for  $\lambda$
- 2 Solve the “LM-equation system”

$$\sum_l \tilde{\alpha}_{kl} \delta \mathbf{a}_l = \beta_k$$

for  $\delta \mathbf{a}$  and calculate  $\chi^2(\mathbf{a} + \delta \mathbf{a})$

- 3 Update
  - If  $\chi^2(\mathbf{a} + \delta \mathbf{a}) < \chi^2(\mathbf{a})$ , update  $\mathbf{a} \rightarrow \mathbf{a}_{new} = \mathbf{a} + \delta \mathbf{a}$ ; decrease  $\lambda$  and continue to step 2
  - If  $\chi^2(\mathbf{a} + \delta \mathbf{a}) \geq \chi^2(\mathbf{a})$ , increase  $\lambda$  (decreasing the step size) and go back to step 2 without changing  $\mathbf{a}$ .

Iterate till some stopping criterion is reached!

## Software implementing the Levenberg-Marquardt method

- **MinPack** (contained in Slatec) is a Fortran library for the solving of systems of nonlinear equations, or the least squares minimization of the residual of a set of linear or nonlinear equations.
  - [www.netlib.org/minpack/](http://www.netlib.org/minpack/)
  - various C/C++ wrappers exist for this library
- GNU Scientific Library

Practical issues: What to do when we cannot calculate the (approximate) Hessian analytically?

Numerical examples

To appear later....

To appear later....

To appear later....



To appear later....

## Simulated annealing

Simulated annealing (1983) is a method for doing global optimization!

- Global minimization is generally very hard, because the system tends to get trapped in local minima
- Simulated annealing is an attempt to circumvent this problem by using the so-called [Metropolis algorithm](#)
- The function to minimize is thought of as an energy  $E$
- In the Metropolis algorithm, steps [upward](#) in  $E$  do occur [with probability  $\exp(-\Delta E/k_B T)$ ], as is needed to escape from local minima.
- The “temperature”  $T$  serves as a control parameter
- One typically starts at high  $T$  (high mobility) and reduce  $T$  till the system freezes
- The hope is that the final frozen state is the [global energy minimum](#)

# The Metropolis algorithm (1953)

# Recall from Statistical Physics

## Optimization

Consider a system that can be found in many different states  $\mathbf{x}$  (e.g. a system of  $N$  spins  $\{s_i\}_{[1,N]}$ ),  $\mathbf{x} = (s_1, \dots, s_N)$  is the state vector of spins.

We want to compute a thermal average of some quantity  $A$  (e.g. the average spin).

We know from statistical physics that in thermal equilibrium this average is given by:

$$\langle A \rangle = \frac{1}{Z} \sum_{\mathbf{x}} A(\mathbf{x}) e^{-H(\mathbf{x})/T} = \sum_{\mathbf{x}} A(\mathbf{x}) P_B(\mathbf{x})$$

- $T$ : temperature;
- $H(\mathbf{x})$ : Hamiltonian of the system in state  $\mathbf{x}$ ;
- $Z = \sum_{\mathbf{x}} e^{-H(\mathbf{x})/k_B T}$  is the *partition function*;
- $P_B(\mathbf{x}) = [e^{-H(\mathbf{x})/k_B T}] / Z$  is the Boltzmann distribution;
- $k_B$  is the Boltzmann constant.

We can also define the *free energy*  $F$  such that

$$Z = e^{-F/k_B T}, \quad \text{i.e.,} \quad F = -k_B T \ln Z$$

In the following we will use the notation  $P_B(\mathbf{x}) = e^{-S(\mathbf{x})}$ .

How would you compute  $\langle A \rangle$ ?

## NOT TO DO!

- **Brute force:** compute  $A$  for all configurations and average them.  
**Problem:** the number of configurations explodes, usually exponentially, with the number of particles.  
**Example:** # states =  $2^N$  for  $N$  spin systems, so even for a rather small system  $N = 100$  this yields # states  $\approx 1.3 \times 10^{30}$  which is quite a lot.
- **Direct sampling:** generate configurations at random and average those.  
**Problems:**  $e^{-H/k_B T}$  varies exponentially and most of the generated states give negligible contribution to the average.  $Z$  has to be calculated also and is a source of error.

## Recall from Monte Carlo integration

One can approximate

$$\int_{\Omega} d^n \mathbf{x} f(\mathbf{x}) \approx \frac{V}{N} \sum_{i=1}^N f(\mathbf{x}_i)$$

with an error estimate given by  $\sigma_N$  where

$$\sigma_N^2 = \frac{1}{N-1} \sum_{i=1}^N [f(\mathbf{x}_i) - \langle f \rangle]^2$$

The error scales as  $1/\sqrt{N}$  but if  $f$  has quick variations the scaling factor may become large. In order to reduce the variance, one should make a smart choice for  $\mathbf{x}_i$ . Instead of sampling  $\mathbf{x}$  uniformly on  $V$ , one can instead rewrite the integral as

$$\int_{\Omega} f(\mathbf{x}) d^n \mathbf{x} = \int_{\Omega} \frac{f(\mathbf{x})}{p(\mathbf{x})} p(\mathbf{x}) d^n \mathbf{x} = \int_{\mathbf{y}(\Omega)} \frac{f(\mathbf{x}(\mathbf{y}))}{p(\mathbf{x}(\mathbf{y}))} d^n \mathbf{y}$$

where  $p$  is interpreted as a probability density. If  $\mathbf{y}$  is sampled uniformly then  $\mathbf{x}$  will be sampled following  $p$ .

This yields

$$\langle f \rangle = \frac{1}{N} \sum_{i=1}^N \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)}$$

with the error estimated by

$$\sigma_N^2 = \frac{1}{N-1} \sum_{i=1}^N \left[ \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)} - \left\langle \frac{f}{p} \right\rangle \right]^2$$

Then the variance is reduced if  $f/p$  is slowly varying.

**The idea here:** generate configuration according to the Boltzmann distribution. The difficult part is then to draw states following the Boltzmann distribution. A method to do this is the Metropolis algorithm.



# Metropolis Algorithm

## Optimization

Consider a Markov chain for the sequence of states  $x_n$ . The probability for going from state  $x_n$  to  $x_{n+1}$  is given by the *transition probability*  $W(x_{n+1}|x_n)$ .

- Master equation

$$P_{n+1}(x) - P_n(x) = \sum_y [W(x|y)P_n(y) - W(y|x)P_n(x)]$$

- What to choose for  $W$  such that  $P$  converges to the Boltzmann distribution?

Sufficient condition on  $W$  for  $P \xrightarrow[n \rightarrow \infty]{} P_B$

- **Ergodicity:** any state can be reached after a finite number of steps.
- **Detailed balance:**  $e^{-S(x)} W(y|x) = e^{-S(y)} W(x|y)$

- 1 Show that  $P_B(x) = e^{-S(x)}$  is indeed an equilibrium solution of the Master equation.

Direct from the Master equation.

Another way to see it is that at equilibrium going from state  $x$  to  $y$  does not change  $P$ . Indeed:

$$\begin{aligned} P(y) &= \int dx W(y|x)P(x) \\ &= \int dx e^{-S(x)} W(y|x) \\ &= \int dx e^{-S(y)} W(x|y) \\ &= e^{-S(y)} \int dx W(y|x) \\ &= e^{-S(y)} \end{aligned}$$

2 Show that  $P_n$  converges to  $P_B$ . For this we need a measure for the convergence. Consider  $\|P_n - P_B\| = \int dx |P_n(x) - e^{-S(x)}|$ .

$$\begin{aligned}
 \|P_{n+1} - P_B\| &= \int dy |P_{n+1}(y) - e^{-S(y)}| \\
 &= \int dy \left| \int dx P_n(x) W(y|x) - e^{-S(y)} \right| \\
 &= \int dy \left| \int dx \left[ P_n(x) W(y|x) - e^{-S(y)} W(x|y) \right] \right| \\
 &= \int dy \left| \int dx \left[ P_n(x) W(y|x) - e^{-S(x)} W(y|x) \right] \right| \\
 &= \int dy \left| \int dx \left[ P_n(x) - e^{-S(x)} \right] W(y|x) \right| \\
 &\leq \int dy \int dx |P_n(x) - e^{-S(x)}| W(y|x) \\
 &\leq \int dx |P_n(x) - e^{-S(x)}| = \|P_n - P_B\|
 \end{aligned}$$

Trial-acceptance decomposition:

$$W(y|x) = t(y|x)a(y|x)$$

where  $t(y|x)$  is the trial probability to pick up a new state  $y$  knowing that the current state is  $x$ , and  $a(y|x)$  is the probability to accept this new state.

Symmetric trial probability are often used then the detailed balance reads

$$e^{-S(x)}a(y|x) = e^{-S(y)}a(x|y)$$

Common choices for  $a$  are

- Metropolis acceptance probability

$$a(y|x) = \min \left( 1, \frac{e^{-S(y)}}{e^{-S(x)}} \right)$$

- Smooth acceptance probability

$$a(y|x) = \frac{1}{1 + \frac{e^{S(y)}}{e^{S(x)}}}$$

**Note:** the sequence of states is *correlated*. Care must be taken when performing the average.

## Monte Carlo using Metropolis algorithm

- 1 Start from an initial state  $x = x_0$ .
- 2 Draw a new state  $\tilde{x}$  according to the trial probability  $t(\tilde{x}|x_n)$ .
- 3 Accept the new state with probability  $a(\tilde{x}|x_n)$  satisfying the detailed balance:
  - if accepted set:  $x_{n+1} \leftarrow \tilde{x}$
  - if rejected set:  $x_{n+1} \leftarrow x_n$
- 4 Repeat steps 2–3 until equilibrium is reached and enough data have been generated. Discard non-equilibrium steps for the averaging.

## Questions

- How do we know when equilibrium is reached?
- How much data must be collected?

## The Ising model

Consider a system of  $N$  spins, where each spin  $s_i$  is located at position  $i$  on a  $d$ -dimensional cubic lattice. Each spin can take the value  $\pm 1$ .

The Hamiltonian of the system is

$$H = -J \sum_{\langle i,j \rangle} s_i s_j$$

where  $J$  is the coupling constant and  $\sum_{\langle i,j \rangle}$  denotes a sum over all nearest-neighbour pair of sites.

We are interested in the magnetization. At thermal equilibrium the magnetization  $M$  is given by

$$M = \frac{\sum_{\text{states}} \sum_i s_i e^{-H/T}}{\sum_{\text{states}} e^{-H/T}}$$

# Example: Ising model

## Optimization

What to choose for  $W$ ? Consider the change in energy  $\Delta H$  due to a switch of a spin  $s_i$ .

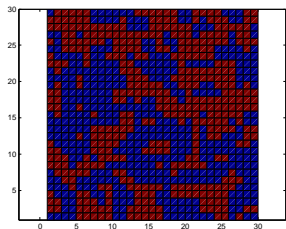
$$\Delta H = -J\Delta s_i \sum_j s_j$$

where the sum is over the neighbour of  $i$ . We then try with  $W = e^{-\Delta H/T}$ . The algorithm reads:

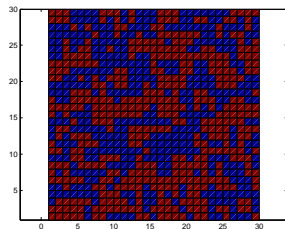
- Start from the initial state  $\forall i \in \llbracket 1, N \rrbracket, s_i = 1$ .
- Draw a spin  $s_i$  at random on the lattice. The considered trial is  $s_i \rightarrow -s_i$ .
- Compute  $W = e^{-\Delta H/T}$ .
- Draw  $r \in [0, 1]$  from a uniform distribution.
  - if  $W > r$  then  $s_i \leftarrow -s_i$ .
  - if  $W < r$  then do not switch.  
**Note:** if  $\Delta H < 0$  then  $W > 1$  which make the switch accepted for sure.
- increment time  $n \rightarrow n + 1$ .
- Repeat, until enough data have been collected for the average.

# Example: Ising model

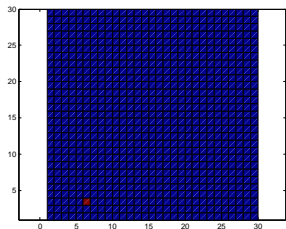
## Optimization



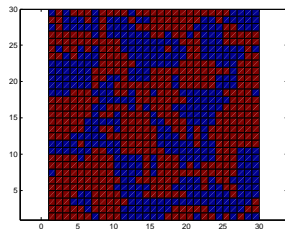
(a) Initial config.  $T = T_c/2$



(b) Initial config.  $T = 3T_c/2$



(c) Equilibrium.  $T = T_c/2$



(d) Equilibrium.  $T = 3T_c/2$

Figure: Spin lattice. Lattice size  $L = 30$ , 400 sweeps.



# Example: Ising model

## Optimization

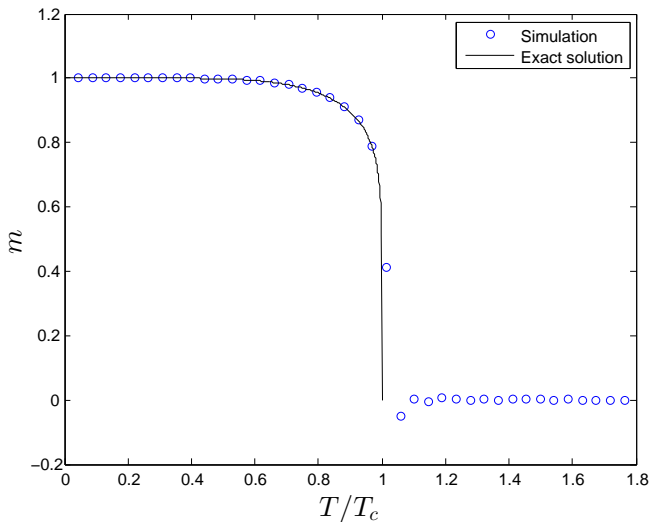


Figure: Normalized magnetization as a function of temperature. Lattice size  $L = 100$ .

How to average in a proper way?

- Eliminate boundary effects by using *periodic boundary condition*.
- When selecting a spin, do it randomly, not sequentially!
- The Monte Carlo time step **should not** be taken as a one switch of spin. An appropriate MC time that should be considered is one under which, on average, all spins have been attempted. This is also called a MC sweep.
- Discard a sufficient number of MC sweeps before collecting data for averages. This avoid the average to be biased by initial configurations that are far away from the equilibrium.
- It is also a good idea to only average on every other sweep or so, in order to reduce correlation between consecutive steps. But do not throw away most of your data for that purpose.
- Assuming a run gives  $N_{\text{sweep}}$  uncorrelated sweep to average over, make  $N_{\text{run}}$  runs to estimate error as

$$\Delta A = \frac{\sigma}{\sqrt{N_{\text{run}}}}$$

How to speed it up?

- Make a table of possible  $w = e^{-\Delta H/T}$  and look it up instead of computing

# Importance sampling for the energy

## Optimization

In statistical physics, one is often interested in the partition function  $Z$ , because many quantities can be deduced from it. Example:

$$\langle E \rangle = \frac{1}{Z} \sum_x H(x) e^{-H(x)/T} = -T^2 \frac{d}{dT} \ln Z$$

Rather than summing over all states, one may sum over all energies:

$$Z = \sum_x e^{-H(x)/T} = \sum_E g(E) e^{-E/T}$$

where  $g$  is the density of states.