Numerical methods in physics

Marc Wagner

Goethe-Universität Frankfurt am Main – winter semester 2023/24

Version: February 5, 2024

Contents

****** October 17, 2023 (1st lecture) ******

1 Introduction

- *"Numerical analysis is the study of algorithms that use numerical approximation (as opposed to general symbolic manipulations) for the problems of mathematical analysis (as distinguished from discrete mathematics)."* (Wiki)
- *"Die numerische Mathematik, auch kurz Numerik genannt, beschäftigt sich als Teilgebiet der Mathematik mit der Konstruktion und Analyse von Algorithmen für kontinuierliche mathematische Probleme. Hauptanwendung ist dabei die näherungsweise ... Berechnung von Lösungen mit Hilfe von Computern."* (Wiki)
- Almost no modern physics without computers.
- Even analytical calculations
  - often require computer algebra systems (Mathematica, Maple, ...),
  - are not fully analytical, but "numerically exact calculations" (e.g. mainly analytically, at the end simple 1-dimensional numerical integrations, which can be carried out up to arbitrary precision).
- Goal of this lecture: Learn, how to use computers in an efficient and purposeful way.
  - Implement numerical algorithms, e.g. in C or Fortran, ...
  - ... write program code specifically for your physics problems ...
  - ... use floating point numbers appropriately (understand roundoff errors, why and to what extent accuracy is limited, ...) ...
  - ... quite often computations run several days, weeks or even months, i.e. decide for most efficient algorithms ...
  - ... in practice, parts of your code have to be written from scratch, other parts use existing numerical libraries (e.g. GSL, LAPACK, ARPACK, ...), i.e. learn to use such libraries.
- Typical problems in physics, which can be solved numerically:
  - Linear systems.
  - Eigenvalue and eigenvector problems.
  - Integration in 1 or more dimensions.
  - Differential equations.
  - Root finding (*Nullstellensuche*), optimization (finding minima or maxima).
  - ...
- Computer algebra systems will not be discussed in this lecture:
  - E.g. Mathematica, Maple, ...

- Complement numerical calculations.
- Automated analytical calculations, e.g.
  - solve standard integrals (find the antiderivative [*Stammfunktion*]),
  - simplify lengthy expressions,
  - transform coordinates (e.g. Cartesian coordinates to spherical coordinates),
  - ...

2 Representation of numbers in computers, roundoff errors

2.1 Integers

- Computer memory can store 0's and 1's, so-called bits, $b_j \in \{0, 1\}$.
- Integer: $z = b_{N-1} \dots b_2 b_1 b_0$ (stored in this way in computer memory, i.e. in the binary numeral system),

$$z = \sum_{j=0}^{N-1} b_j 2^j \quad \text{(for positive integers)}. \tag{1}$$

- Typically $N = 32$ (sometimes also $N = 8, 16, 64, 128$)
  $\rightarrow 0 \leq z \leq 2^{32} - 1 = 4\,294\,967\,295$.
- Negative integers: very similar (homework: study Wiki,
  `https://en.wikipedia.org/wiki/Integer_(computer_science)`,
  `https://en.wikipedia.org/wiki/Signed_number_representations`).
- Many arithmetic operations are exact; exceptions:
  - if range is exceeded,
  - division, square root, ... yields another integer obtained by rounding down (*Nachkommastellen abschneiden*), e.g. $7/3 = 2$.

2.2 Real numbers, floating point numbers

- Real numbers are approximated in computers by floating point numbers,

$$z = S \times M \times 2^{E-e}. \tag{2}$$

  - Sign: $S = \pm 1$.
  - Mantissa:

$$M = \sum_{j=0}^{N_M} m_j \left(\frac{1}{2}\right)^j, \tag{3}$$

    $m_0 = 1$ (phantom bit, i.e. $M = 1.???$, "normalized"), $m_j \in \{0, 1\}$ for $j \geq 1$ (representation analogous to representation of integers).
  - Exponent: E is integer, e is integer constant.
- Two frequently used data types: `float` (32 bits), `double` (64 bits) [1].
  - `float`:
    - S: 1 bit.
    - E: 8 bits.

[1] `float` and `double` are C data types. In Fortran `real` and `double precision`.

* M: $N_M = 23$ bits.
* Range:
  $M = 1, 1+\epsilon, 1+2\epsilon, \ldots, 2-2\epsilon, 2-\epsilon$, where $\epsilon = (1/2)^{23} \approx 1.19 \times 10^{-7}$.
  ϵ is relative precision.
  $e = 127$, $E = 1, \ldots, 254$, i.e. $2^{E-e} = 2^{-126} \ldots 2^{+127} \approx 10^{-38} \ldots 10^{+38}$.
  10^{-38} is smallest numbers, 10^{+38} is largest number.

– `double`:

* S: 1 bit.
* E: 11 bits.
* M: $N_M = 52$ bits.
* Range:
  $\epsilon = (1/2)^{52} \approx 2.22 \times 10^{-16}$.
  $2^{E-e} \approx 10^{-308} \ldots 10^{+308}$.

– Homework: Study Ref. [1], section 1.1.1. "Floating-Point Representation".

2.3 Roundoff errors

- Due to the finite number of bits of the mantissa M, real numbers cannot be stored exactly. They are approximated by the closest foating point numbers.
- Equation (2):

$$z \;=\; S \times M \times 2^{E-e}, \tag{4}$$

i.e. relative precision $\epsilon \approx 10^{-7}$ for `float` and $\epsilon \approx 10^{-16}$ for `double`.

2.3.1 Simple examples

- $1 + \tilde{\epsilon} = 1$, if $|\tilde{\epsilon}| < \epsilon$.
- Difference of similar numbers z_1 and z_2 (i.e. the first n decimal digits of z_1 and z_2 are identical, they differ in the $n+1$-th digit):

$$z_1 - z_2 \;=\; \underbrace{\alpha_1}_{\approx 1.???}\, 10^{\beta} - \underbrace{\alpha_2}_{1.???}\, 10^{\beta} \;=\; \underbrace{\Big(\alpha_1 - \alpha_2\Big)}_{\mathcal{O}(10^{-n})}\, 10^{\beta}. \tag{5}$$

– When $\alpha_1 - \alpha_2$ is computed, the first n digits cancel each other
  $\rightarrow$ resulting mantissa has accuracy $10^{-(7-n)}$ (`float`) or $10^{-(16-n)}$ (`double`).
– E.g. difference of two `floats`, which differ relatively by 10^{-6}, is accurate only up to 1 digit.

2.3.2 Another example: numerical derivative via finite difference

- Starting point: function $f(x)$ can be evaluated, $f'(x)$ not (e.g. expression is very long and complicated or can only be calculated numerically).
- Common approach: approximate $f'(x)$ numerically by finite difference, e.g.

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \mathcal{O}(h^3) \tag{6}$$

$$\rightarrow \quad f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \quad \text{(asymmetric)} \tag{7}$$

$$\rightarrow \quad f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \quad \text{(symmetric).} \tag{8}$$

****** October 19, 2023 (2nd lecture) ******

- Problems:
  - If h is large
    $\rightarrow \mathcal{O}(h), \mathcal{O}(h^2)$ large.
  - If h is small
    $\rightarrow f(x+h) - f(x), f(x+h) - f(x-h)$ are differences of similar numbers (see section 2.3.1).
- Optimal choice $h = h_{\text{opt}}$ for asymmetric finite difference (7):
  - Relative error due to $\mathcal{O}(h)$:

$$\delta f'(x) = f'(x) - f'_{\text{finite difference}}(x) = f'(x) - \frac{f(x+h) - f(x)}{h} =$$

$$= -\frac{1}{2}f''(x)h + \mathcal{O}(h^2)$$

$$\rightarrow \quad \frac{\delta f'(x)}{f'(x)} = \frac{-f''(x)h/2}{f'(x)} \sim \frac{f''(x)h}{f'(x)} \tag{9}$$

    ($\sim$ indicates "of the same order as").
  - Relative error due to $f(x+h) - f(x)$:

$$f(x+h) - f(x) \approx f'(x)h$$

$$f(x+h) \approx f(x)$$

$$\rightarrow \quad \text{relative loss of accuracy} = \frac{f(x)}{f(x+h) - f(x)} \approx \frac{f(x)}{f'(x)h}$$

    (e.g. $\approx 10^3$ implies that 3 digits are lost)

$$\rightarrow \quad \frac{\delta f'(x)}{f'(x)} = \text{relative precision} \times \text{relative loss of accuracy} \sim \epsilon\frac{f(x)}{f'(x)h}. \tag{10}$$

  - For $h = h_{\text{opt}}$ both errors are similar:

$$\frac{f''(x)h_{\text{opt}}}{f'(x)} \sim \epsilon\frac{f(x)}{f'(x)h_{\text{opt}}}$$

$$\rightarrow \quad h_{\rm opt} \quad \sim \quad \left(\frac{f(x)}{f''(x)}\epsilon\right)^{1/2} \quad \sim \quad \epsilon^{1/2}$$
$$\rightarrow \quad \left.\frac{\delta f'(x)}{f'(x)}\right|_{\rm opt} \quad \sim \quad \frac{f''(x)h_{\rm opt}}{f'(x)} \quad \sim \quad \epsilon^{1/2}. \tag{11}$$

- Optimal choice $h = h_{\rm opt}$ for symmetric finite difference (8): analogous analysis yields

$$h_{\rm opt} \quad \sim \quad \epsilon^{1/3} \quad , \quad \left.\frac{\delta f'(x)}{f'(x)}\right|_{\rm opt} \quad \sim \quad \epsilon^{2/3}, \tag{12}$$

  i.e. symmetric derivative superior to asymmetric derivative.

- In practice:
  - Estimate errors analytically as sketched above ...
  - ... and test the stability of your results with respect to numerical parameters (h in the derivative example).

- Above estimates are confirmed by the following example program [2].

```
// derivative of sin(x) at x = 1.0 via finite differences

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int j;

  // *****

  printf("h            rel_err_asym   rel_err_sym\n");

  for(j = 1; j <= 15; j++)
    {
      double h = pow(10.0, -(double)j);

      double df_exact = cos(1.0);

      double df_asym = (sin(1.0+h) - sin(1.0)) / h;
      double df_sym = (sin(1.0+h) - sin(1.0-h)) / (2.0 * h);

      double rel_err_asym = fabs((df_exact - df_asym) / df_exact);
      double rel_err_sym = fabs((df_exact - df_sym) / df_exact);

      printf("%.1e      %.3e      %.3e\n", h, rel_err_asym, rel_err_sym);
    }

  // *****
```

[2] Throughout this lecture I use C.

```
  return EXIT_SUCCESS;
}
```

---

h	rel_err_asym	rel_err_sym
1.0e-01	7.947e-02	1.666e-03
1.0e-02	7.804e-03	1.667e-05
1.0e-03	7.789e-04	1.667e-07
1.0e-04	7.787e-05	1.667e-09
1.0e-05	7.787e-06	2.062e-11
1.0e-06	7.787e-07	5.130e-11
1.0e-07	7.742e-08	3.597e-10
1.0e-08	5.497e-09	4.777e-09
1.0e-09	9.724e-08	5.497e-09
1.0e-10	1.082e-07	1.082e-07
1.0e-11	2.163e-06	2.163e-06
1.0e-12	8.003e-05	2.271e-05
1.0e-13	1.358e-03	3.309e-04
1.0e-14	6.861e-03	6.861e-03
1.0e-15	2.741e-02	2.741e-02

---

3 Ordinary differential equations (ODEs), initial value problems

3.1 Physics motivation

- Newton's equations of motion (EOMs), N point masses m_j,

$$m_j \ddot{\mathbf{r}}_j(t) \quad = \quad \mathbf{F}_j(\mathbf{r}_1(t), \ldots, \mathbf{r}_N(t), \dot{\mathbf{r}}_1(t), \ldots, \dot{\mathbf{r}}_N(t), t) \quad , \quad j = 1, \ldots, N, \tag{13}$$

initial conditions

$$\mathbf{r}_j(t=0) \quad = \quad \mathbf{r}_{j,0} \quad , \quad \dot{\mathbf{r}}_j(t=0) \quad = \quad \mathbf{v}_{j,0}. \tag{14}$$

- Calculate trajectories $\mathbf{r}_j(t)$.
- Cannot be done analytically in the majority of cases, e.g. three-body problem "sun and two planets".
- For boundary value problems see section 6 (e.g. quantum mechanics [QM], Schrödinger equation, $\psi(x_1) = 0$, $\psi(x_2) = 0$).

3.2 Euler's method

- Preparatory step: rewrite ODEs to system of first order ODEs.
  - Newton's EOMs equivalent to

$$\dot{\mathbf{r}}_j(t) \quad = \quad \mathbf{v}_j(t) \quad , \quad \dot{\mathbf{v}}_j(t) \quad = \quad \frac{\mathbf{F}_j(\mathbf{r}_1(t), \ldots, \mathbf{r}_N(t), \dot{\mathbf{r}}_1(t), \ldots, \dot{\mathbf{r}}_N(t), t)}{m_j}. \tag{15}$$

  - Define

$$\mathbf{y}(t) \quad = \quad (\mathbf{r}_1(t), \ldots, \mathbf{r}_N(t), \mathbf{v}_1(t), \ldots, \mathbf{v}_N(t)) \tag{16}$$

$$\mathbf{f}(\mathbf{y}(t), t) \quad = \quad \Bigg(\underbrace{\mathbf{v}_1(t), \ldots, \mathbf{v}_N(t)}_{\in \mathbf{y}(t)}, \frac{\mathbf{F}_1(\mathbf{y}(t), t)}{m_1}, \ldots, \frac{\mathbf{F}_N(\mathbf{y}(t), t)}{m_N} \Bigg). \tag{17}$$

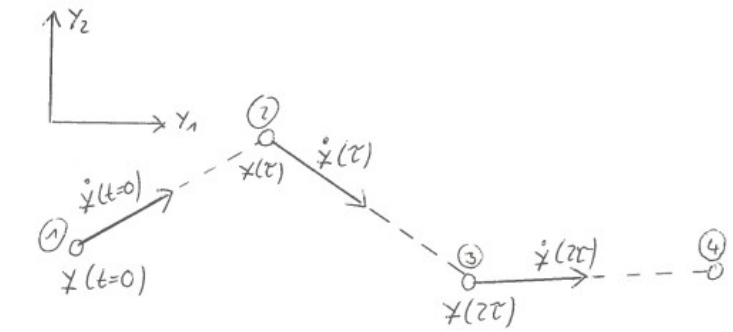
  - Then

$$\dot{\mathbf{y}}(t) \quad = \quad \mathbf{f}(\mathbf{y}(t), t) \tag{18}$$

    (left hand side (lhs) can be evaluated in a straightforward way for given t and $\mathbf{y}(t)$).
  - Always possible to rewrite a system of ODEs according to (18).
- Solve (18) by iteration, i.e. perform many small steps in time, step size τ:

$$\mathbf{y}(t+\tau) \quad = \quad \mathbf{y}(t) + \dot{\mathbf{y}}(t)\tau + \mathcal{O}(\tau^2) \quad = \quad \mathbf{y}(t) + \mathbf{f}(\mathbf{y}(t), t)\tau + \mathcal{O}(\tau^2). \tag{19}$$

Figure 3.4



- τ can be positive ($\rightarrow$ computation of future) or negative ($\rightarrow$ computation of past).
- Problem: method inefficient, because of large discretization errors.
  - $\mathcal{O}(\tau^2)$ error per step.
  - Time evolution from $t = 0$ (initial conditions) to $t = T$
    $\rightarrow T/\tau$ steps
    $\rightarrow \mathcal{O}((T/\tau)\tau^2) = \mathcal{O}(\tau)$ total error (very inefficient).
  - Total error might be underestimated (e.g. chaotic systems are highly sensitive to initial conditions and, thus, to the error per step).

****** October 24, 2023 (3rd lecture) ******

3.3 Runge-Kutta (RK) method

- Same idea as in section 3.2, but improved discretization (stronger suppression of errors with respect to τ).
- "2nd-order RK":

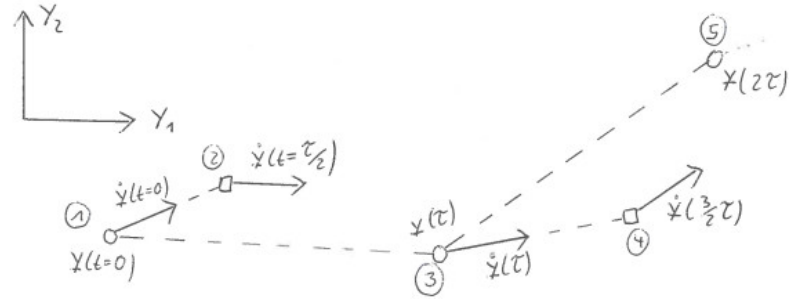
$$\mathbf{k}_1 = \mathbf{f}(\mathbf{y}(t), t)\tau \quad \rightarrow \text{ "full Euler step"} \tag{20}$$

$$\mathbf{k}_2 = \mathbf{f}\Big(\underbrace{\mathbf{y}(t) + (1/2)\mathbf{k}_1, t + (1/2)\tau}_{\rightarrow \text{ "half Euler step"}}\Big)\tau \tag{21}$$

$$\mathbf{y}(t+\tau) = \mathbf{y}(t) + \mathbf{k}_2 + \mathcal{O}(\tau^3). \tag{22}$$

  - $\mathbf{f}(\mathbf{y}(t) + (1/2)\mathbf{k}_1, t + (1/2)\tau)$ in (21): estimated derivative $\dot{\mathbf{y}}(t + \tau/2)$, i.e. after half step.
  - (22): 2nd order RK step (a full "Euler-like" step using the derivative after a half step).

Figure 3.B



- Proof of (22), i.e. that error per step is $\mathcal{O}(\tau^3)$:

$$\begin{aligned}
\mathbf{k}_2 &= \mathbf{f}\Big(\mathbf{y} + (1/2)\mathbf{f}\tau, t + (1/2)\tau\Big)\tau \quad = \\
&= \mathbf{f}\tau + \left(\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\frac{1}{2}\mathbf{f}\tau + \frac{\partial \mathbf{f}}{\partial t}\frac{1}{2}\tau\right)\tau + \mathcal{O}(\tau^3) \quad = \quad \mathbf{f}\tau + \frac{1}{2}\left(\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\dot{\mathbf{y}} + \frac{\partial \mathbf{f}}{\partial t}\right)\tau^2 + \mathcal{O}(\tau^3) \quad = \\
&= \mathbf{f}\tau + \frac{1}{2}\dot{\mathbf{f}}\tau^2 + \mathcal{O}(\tau^3) \qquad (23)
\end{aligned}$$

$$\begin{aligned}
\mathbf{y}(t+\tau) &= \mathbf{y} + \dot{\mathbf{y}}\tau + \frac{1}{2}\ddot{\mathbf{y}}\tau^2 + \mathcal{O}(\tau^3) \quad = \quad \mathbf{y} + \mathbf{f}\tau + \frac{1}{2}\dot{\mathbf{f}}\tau^2 + \mathcal{O}(\tau^3) \quad = \\
&= \mathbf{y} + \mathbf{k}_2 + \mathcal{O}(\tau^3) \qquad (24)
\end{aligned}$$

  (no arguments imply time t, e.g. $\mathbf{y} \equiv \mathbf{y}(t)$, $\mathbf{f} \equiv \mathbf{f}(\mathbf{y}(t), t)$).

- Discretization not unique, e.g. for $\mathcal{O}(\tau^3)$ error per step there are many possible RK expressions (an example is discussed in the tutorials).

- Straightforward to derive discretizations with $\mathcal{O}(\tau^4)$, $\mathcal{O}(\tau^5)$, ... error per step:

  - "3rd-order RK":

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{y}(t), t)\tau \qquad (25)\\
\mathbf{k}_2 &= \mathbf{f}\Big(\mathbf{y}(t) + \mathbf{k}_1, t + \tau\Big)\tau \qquad (26)\\
\mathbf{k}_3 &= \mathbf{f}\Big(\mathbf{y}(t) + (1/4)\Big(\mathbf{k}_1 + \mathbf{k}_2\Big), t + (1/2)\tau\Big)\tau \qquad (27)\\
\mathbf{y}(t+\tau) &= \mathbf{y}(t) + \frac{1}{6}\Big(\mathbf{k}_1 + \mathbf{k}_2 + 4\mathbf{k}_3\Big) + \mathcal{O}(\tau^4). \qquad (28)
\end{aligned}$$

  - "4th-order RK":

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{y}(t), t)\tau \qquad (29)\\
\mathbf{k}_2 &= \mathbf{f}\Big(\mathbf{y}(t) + (1/2)\mathbf{k}_1, t + (1/2)\tau\Big)\tau \qquad (30)\\
\mathbf{k}_3 &= \mathbf{f}\Big(\mathbf{y}(t) + (1/2)\mathbf{k}_2, t + (1/2)\tau\Big)\tau \qquad (31)\\
\mathbf{k}_4 &= \mathbf{f}\Big(\mathbf{y}(t) + \mathbf{k}_3, t + \tau\Big)\tau \qquad (32)\\
\mathbf{y}(t+\tau) &= \mathbf{y}(t) + \frac{1}{6}\Big(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\Big) + \mathcal{O}(\tau^5). \qquad (33)
\end{aligned}$$

– ...

- Common choice is 4th-order RK.
- Even better: numerical tests with different order RKs (higher orders allow larger step sizes τ [which is good], require larger numbers of arithmetic operations per step [which is bad]).
- Example: Compute the trajectory of the 1-dimensional harmonic oszillator (HO).
  - Lagrangian:

$$L = \frac{m}{2}\dot{x}^2 - \frac{m\omega^2}{2}x^2. \tag{34}$$

  - EOMs:

$$m\ddot{x}(t) = -m\omega^2 x(t), \tag{35}$$

    i.e.

$$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t), t) \tag{36}$$

    with

$$\mathbf{y}(t) = (x(t), v(t)) \quad , \quad \mathbf{f}(\mathbf{y}(t), t) = (v(t), -\omega^2 x(t)). \tag{37}$$

  - Initial conditions: $x(t=0) = x_0$, $\dot{x}(t=0) = 0$, i.e. $\mathbf{y}(t=0) = (x_0, 0)$.
  - $\omega = 1.0$, $x_0 = 1.0$, step size $\tau = 0.1$ [3].
  - Resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK are shown in Figure 1.
  - Errors of the trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK are shown in Figure 2.
  - Corresponding C code: see appendix A.

3.3.1 Estimation of errors

- Error per step for n-th order RK can be estimated in the following way:
  - RK step with step size τ
    $\rightarrow \mathbf{y}_\tau(t+\tau)$
    $\rightarrow \vec{\delta}_\tau \approx \mathbf{c}\tau^{n+1}$.
  - 2 RK steps with step size $\tau/2$
    $\rightarrow \mathbf{y}_{2\times\tau/2}(t+\tau)$
    $\rightarrow \vec{\delta}_{2\times\tau/2} \approx 2\mathbf{c}(\tau/2)^{n+1}$.

[3] Assigning dimensionless numbers to dimensionful quantities, e.g. $\omega = 1.0$ or $x_0 = 1.0$, is not always recommended. Usually it is advantageous to define and exclusively use equivalent dimensionless quantities (see section 4).

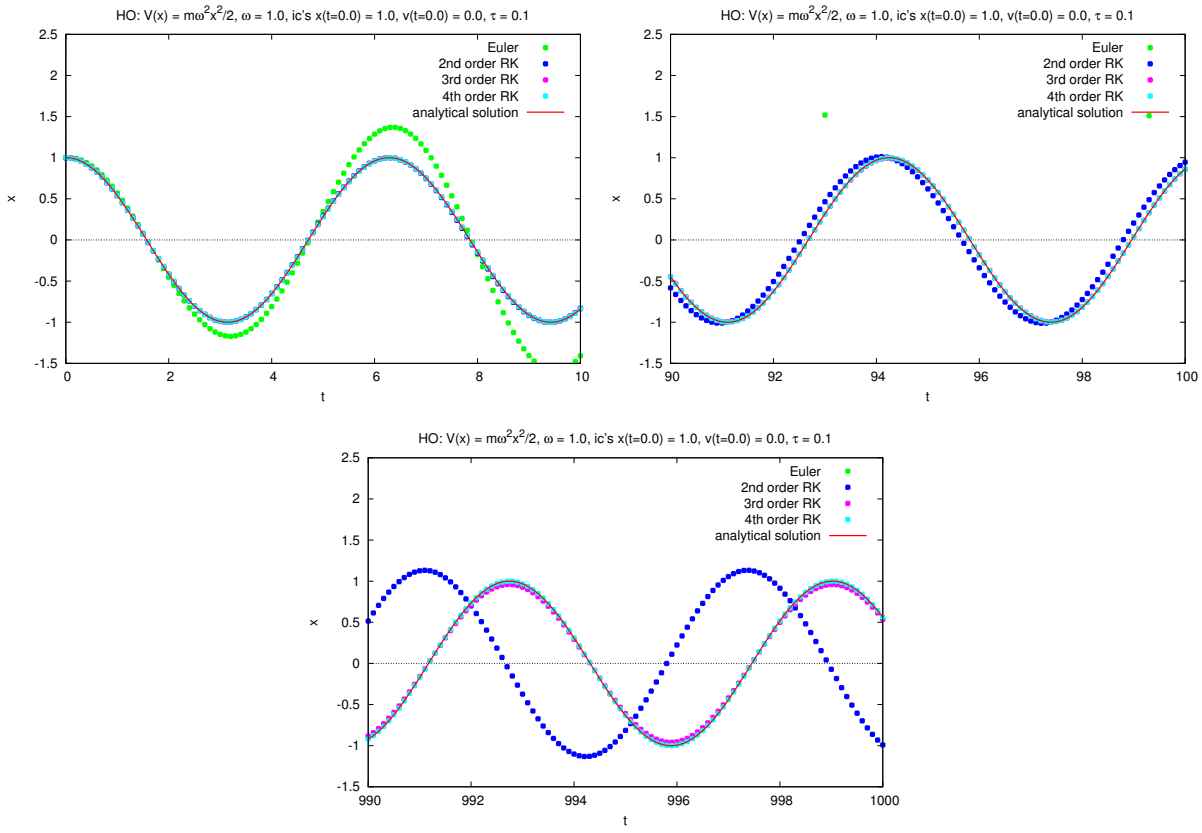
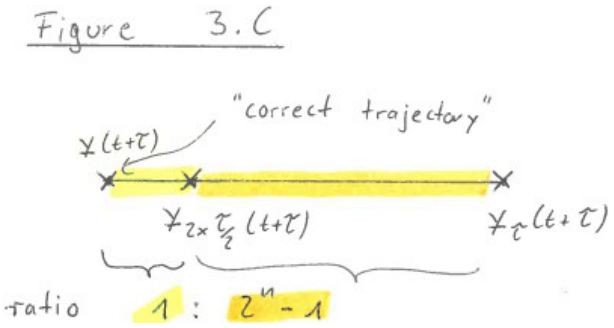


Figure 1: HO, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK.



***** *October 26, 2023 (4th lecture)* *****

– Estimated absolute error for $\mathbf{y}_{2\times\tau/2}(t+\tau)$:

$$\delta_{\text{abs}} = \frac{|\mathbf{y}_{2\times\tau/2}(t+\tau) - \mathbf{y}_{\tau}(t+\tau)|}{2^n - 1}, \tag{38}$$

where $|\ldots|$ can be e.g. Euclidean norm, maximum norm (might be a better choice for many degrees of freedom [dof's]), ...

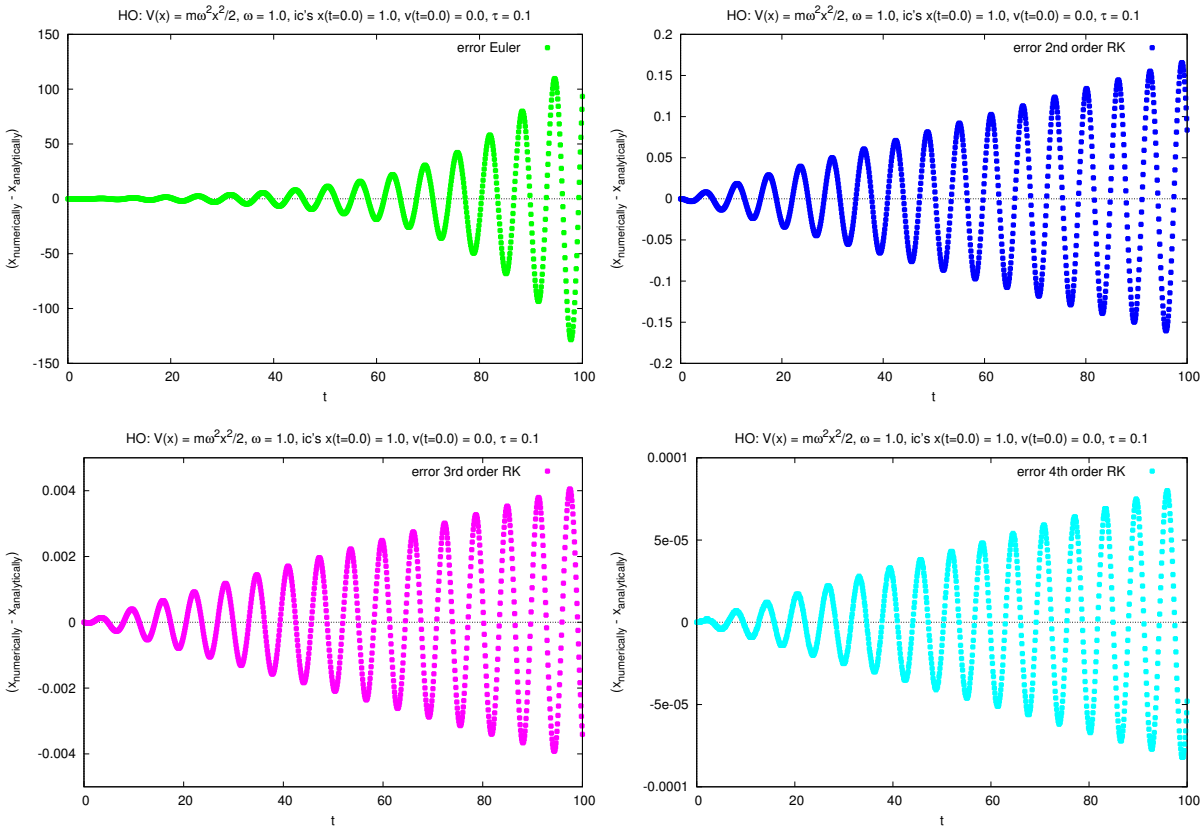


Figure 2: HO, errors of the trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK.

– Estimated relative error for $\mathbf{y}_{2\times\tau/2}(t+\tau)$ (might be more relevant than estimated absolute error):

$$\delta_{\text{rel}} \quad = \quad \frac{\delta_{\text{abs}}}{|\mathbf{y}(t)|}. \tag{39}$$

- Estimated error allows local extrapolation:
  - Correct by estimated error:

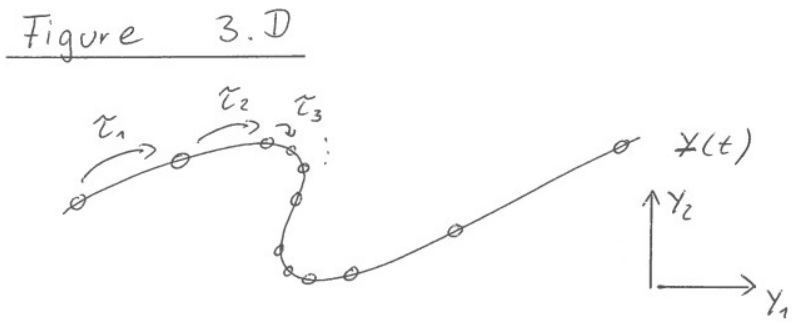
$$\mathbf{y}_{2\times\tau/2}(t+\tau) \quad \rightarrow \quad \mathbf{y}_{2\times\tau/2}(t+\tau) + \frac{\mathbf{y}_{2\times\tau/2}(t+\tau) - \mathbf{y}_{\tau}(t+\tau)}{2^n - 1}. \tag{40}$$

  - However, no estimation of errors, when using (40).

3.3.2 Adaptive step size

- Small step size τ
  $\rightarrow$ small errors, computation slow.
- Large step size τ
  $\rightarrow$ large errors, computation fast.

- Compromise needed: large τ in regions, where $\mathbf{y}(t)$ is smooth, small τ otherwise.



- For given maximum tolerable error $\delta_{\text{abs,max}}$ or $\delta_{\text{rel,max}}$, estimated error allows to estimate corresponding step size $\tau_{\max}$:

$$\frac{\delta_{X,\max}}{\delta_X} = \frac{(\tau_{\max})^{n+1}}{\tau^{n+1}} \quad \rightarrow \quad \tau_{\max} = \tau\left(\frac{\delta_{X,\max}}{\delta_X}\right)^{1/(n+1)} \quad , \quad X \in \{\text{abs}, \text{rel}\}. \tag{41}$$

- Use e.g. the following algorithm to adapt τ in each RK step:
  - *Input:*
    - *Initial conditions* $\mathbf{y}(t=0)$.
    - *Maximum tolerable error* $\delta_{abs,max}$.
    - *Initial step size* τ *(can be coarse).*
  - $t = 0$.

  (1) RK steps:

  $$\mathbf{y}(t) \quad \rightarrow_{\tau} \quad \mathbf{y}_{\tau}(t+\tau) \tag{42}$$
  $$\mathbf{y}(t) \quad \rightarrow_{\tau/2} \rightarrow_{\tau/2} \quad \mathbf{y}_{2\times\tau/2}(t+\tau). \tag{43}$$

  - *Estimated error:*

  $$\delta_{abs} = \frac{|\mathbf{y}_{2\times\tau/2}(t+\tau) - \mathbf{y}_{\tau}(t+\tau)|}{2^n - 1}. \tag{44}$$

  - *Change step size:*

  $$\tau_{new} = 0.9 \times \tau\left(\frac{\delta_{abs,max}}{\delta_{abs}}\right)^{1/(n+1)} \tag{45}$$

  ("0.9" reduces number of RK steps, which have to be repeated with smaller step size).
  - *Clamp* τ_{new} *to* $[0.2\times\tau, 5.0\times\tau]$ *(avoid tiny/huge step size, which might cause breakdown of algorithm).*
  - *If* $\delta_{abs} \leq \delta_{abs,max}$*:*

    → *Accept* $\mathbf{y}_{2\times\tau/2}(t+\tau)$ *(e.g. output to file).*
    $t = t + \tau$ *(i.e. continue at time* $t+\tau$*).*
    $\tau = \tau_{new}$ *(i.e. continue with estimated optimal step size).*
    Go to (1).

Else:

$\rightarrow$ $\tau = \tau_{new}$ *(i.e. reduce step size).*
Go to (1) (i.e. repeat RK steps with smaller step size).

- Modifications possible, e.g. estimate error and τ_{new} by performing RK steps of n-th and $n+1$-th order instead of RK steps with step sizes τ and $\tau/2$.

- Example: 1-dimensional anharmonic oszillator.

  - Lagrangian:

    $$L = \frac{m}{2}\dot{x}^2 - m\alpha x^n \quad , \quad n \in \{2, 20\}. \tag{46}$$

  - EOMs:

    $$m\ddot{x}(t) = -m\alpha n (x(t))^{n-1}, \tag{47}$$

    i.e.

    $$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t), t) \tag{48}$$

    with

    $$\mathbf{y}(t) = (x(t), v(t)) \quad , \quad \mathbf{f}(\mathbf{y}(t), t) = (v(t), -\alpha n (x(t))^{n-1}). \tag{49}$$

  - Initial conditions: $x(t=0) = x_0$, $\dot{x}(t=0) = 0$, i.e. $\mathbf{y}(t=0) = (x_0, 0)$.
  - $\alpha = 0.5\,, 1.0$ for $n = 2\,, 20$, $x_0 = 1.0$, maximum tolerable error $\delta_{\text{abs,max}} = 0.001$, initial step size $\tau = 1.0$.
  - Resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK are shown in Figure 3 (for $n = 2$) and Figure 4 (for $n = 20$).
  - Corresponding C code: see appendix B.

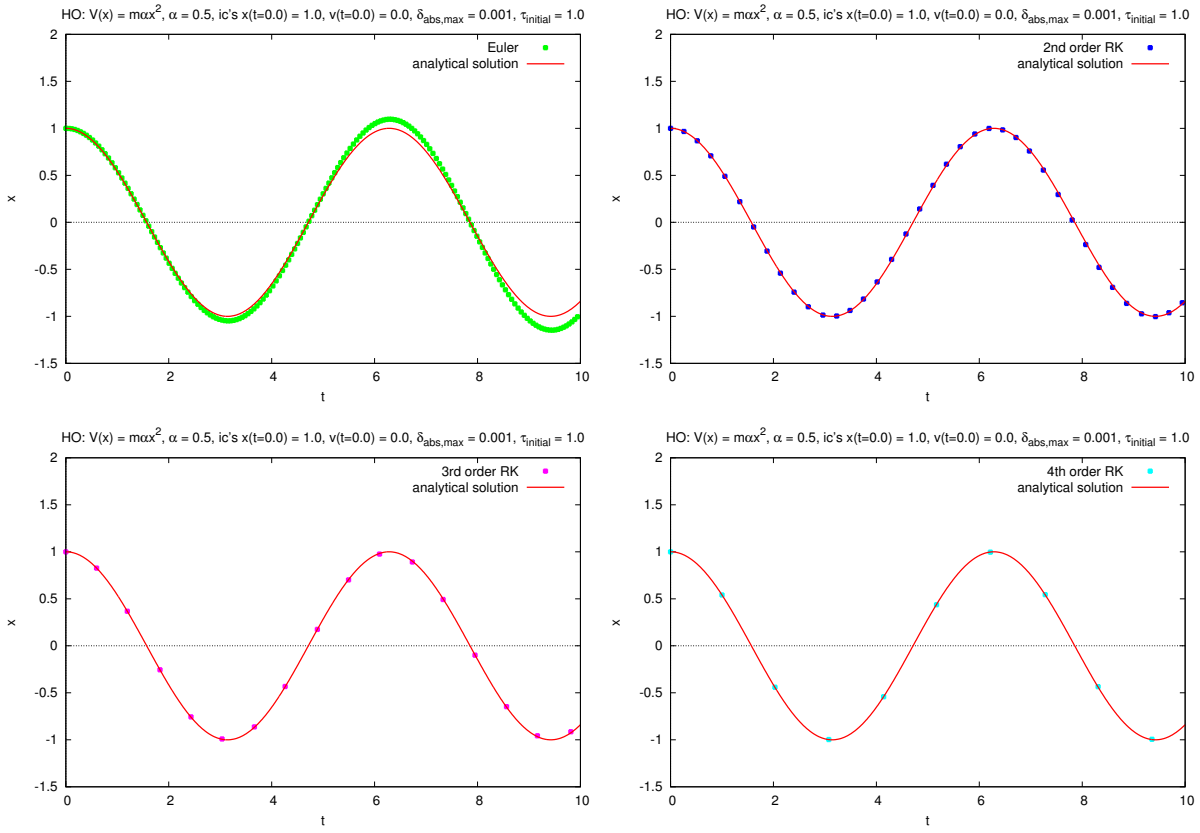


Figure 3: Harmonic oscillator, $V(x) = m\alpha x^2$, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK using adaptive step size.

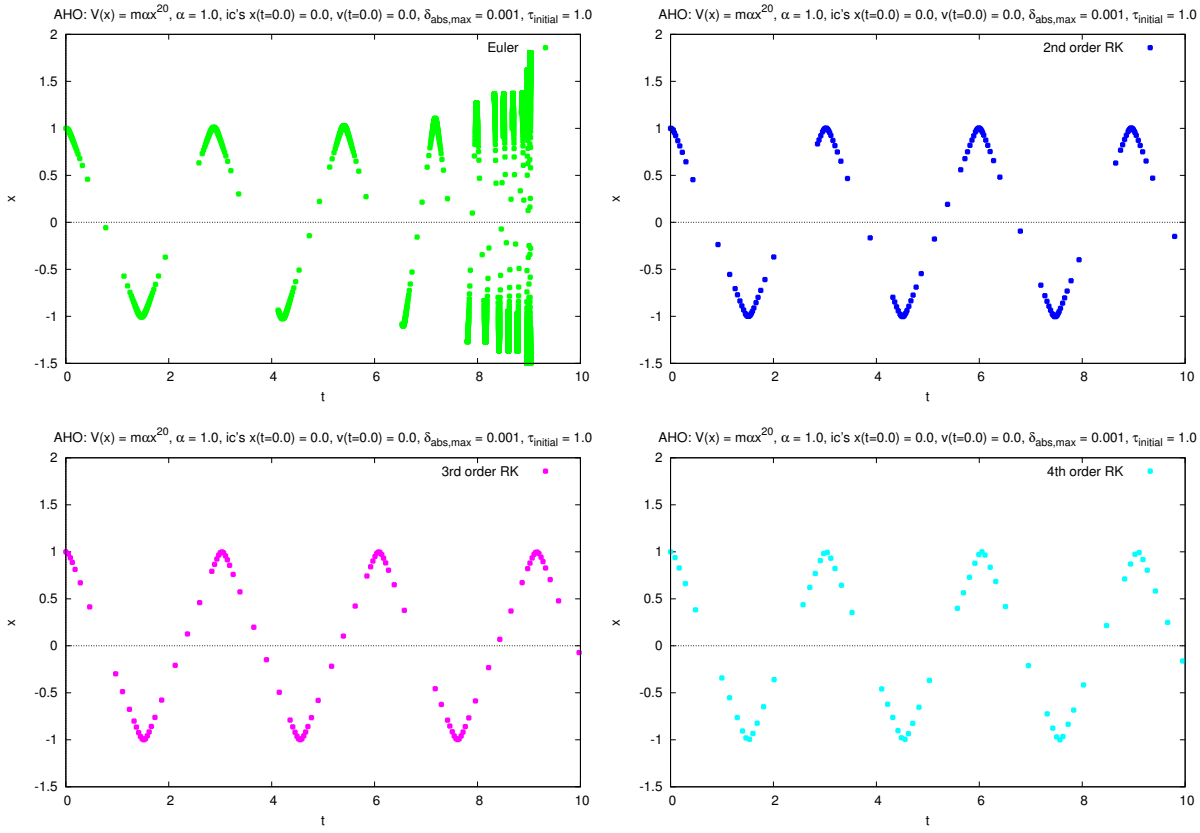


Figure 4: Anharmonic oscillator, $V(x) = m\alpha x^{20}$, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK using adaptive step size.

4 Dimensionful quantities on a computer

- Computers work with dimensionless numbers ...
- ... but the majority of quantities in physics is dimensionful (e.g. lengths, time differences, energies) ...?

4.1 Method 1: define units for your computation

- Define units for your computation, e.g. all lengths are measured in meters, i.e. a length 3.77 in computer memory corresponds to 3.77 m.
  - All lengths have to be measured in meters, otherwise results are nonsense.
  - Choose units appropriately (very small and very large numbers should be avoided, e.g. use fm in particle physics and ly in cosmology).
- Advantage: easy to understand.

4.2 Method 2: use exclusively dimensionless quantities

- Reformulate the problem using exclusively dimensionless quantities.
- Example: compute the trajectory of the 1-dimensional harmonic oszillator (same example as in section 3.3).
  - Lagrangian:
    $$L = \frac{m}{2}\dot{x}^2 - \frac{m\omega^2}{2}x^2. \tag{50}$$
  - EOMs:
    $$m\ddot{x}(t) = -m\omega^2 x(t) \quad \rightarrow \quad \ddot{x}(t) = -\omega^2 x(t), \tag{51}$$
    i.e. m irrelevant.
  - Measure time in units of $1/\omega$:
    $$\hat{t} = \omega t \quad \rightarrow \quad \frac{d^2}{d\hat{t}^2}x(\hat{t}) = -x(\hat{t}). \tag{52}$$
  - Moreover, initial conditions introduce length scale, e.g. $x(t=0) = x_0$, $\dot{x}(t=0) = 0$ $\rightarrow$ measure x in units of x_0:
    $$\hat{x} = \frac{x}{x_0} \quad \rightarrow \quad \frac{d^2}{d\hat{t}^2}\hat{x}(\hat{t}) = -\hat{x}(\hat{t}). \tag{53}$$
  - Now only dimensionless quantities in (53), i.e. straightforward to treat numerically.
  - Figure 5 showing trajectory $\hat{x}(\hat{t})$ is analog of Figure 1 (left top).

****** October 31, 2023 (5th lecture) ******

- Advantage: a single computation for different parameter sets (above example: trajectory $\hat{x}(\hat{t})$ shown in Figure 5 valid for arbitrary m, ω and x_0).

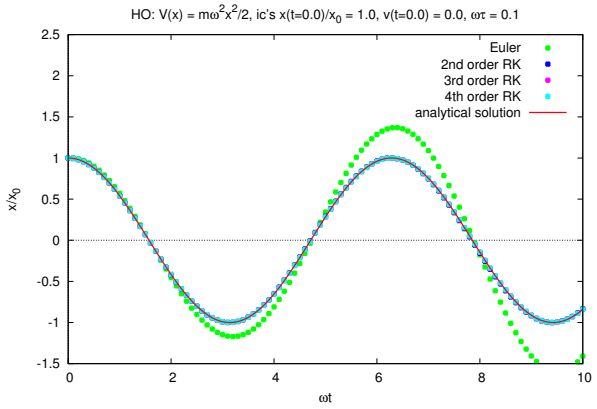


Figure 5: HO, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK (same data as in Figure 1 [left top], but coordinate axes correspond to dimensionless quantities $\hat{t} = \omega t$ and $\hat{x} = x/x_0$).

5 Root finding, solving systems of non-linear equations

5.1 Physics motivation

- N non-linear equations with N unknowns,

$$f_j(x_1, \ldots, x_N) \;=\; 0 \quad , \quad j = 1, \ldots, N, \tag{54}$$

or equivalently written in a more compact way

$$\mathbf{f}(\mathbf{x}) \;=\; 0. \tag{55}$$

- Find solutions $\mathbf{x}$ of (55), i.e. find roots of $\mathbf{f}(\mathbf{x})$.
- Standard problem in physics, e.g. needed to solve the Schrödinger equation (see section 6).
- For systems of linear equations see section 7.

5.2 Bisection (only for $N = 1$)

- Starting point: x_1, x_2 fulfilling $f(x_1) < 0$ and $f(x_2) > 0$ (e.g. plot $f(x)$, then read off appropriate values for x_1 and x_2).
- Bisection always finds a root of $f(x)$, somewhere between x_1 and x_2.
- Algorithm:

  (1) $\bar{x} = (x_1 + x_2)/2$.
  - *If* $f(x_1)f(\overline{x}) < 0$*:*

    $\rightarrow$ $x_2 = \overline{x}$.

    Else:

    $\rightarrow$ $x_1 = \overline{x}$.
  - *If* $|x_1 - x_2|$ *sufficiently small:*

    $\rightarrow$ $x_1 \approx x_2$ *is approximate root.*
    End of algorithm.

    Else:

    $\rightarrow$ *Go to (1).*

- Convergence:
  - Error of approximate root δ defined via $f(x_1 + \delta) = 0$.
  - After n iterations

$$\delta_n \;\leq\; \frac{|x_1 - x_2|}{2^n}, \tag{56}$$

i.e. error decreases exponentially (after 3 to 4 iterations 1 decimal digit more accurate).

– $\delta_{n+1} \approx \delta_n/2$ is called *linear convergence* (δ_{n+1} linear in δ_n).

- Advantages and disadvantages:

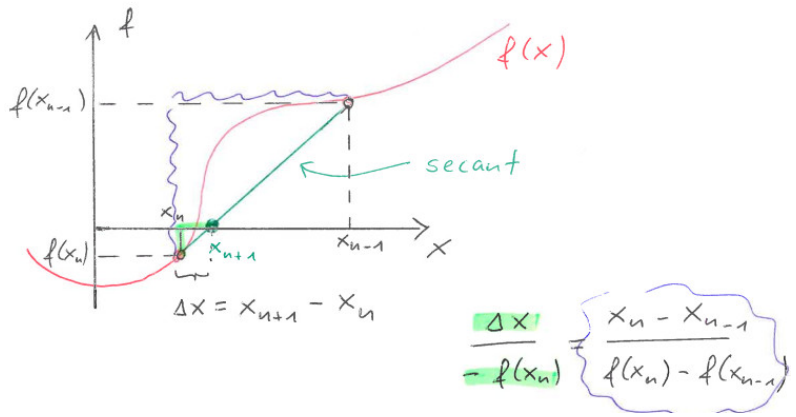
  (+) Always finds a root.

  (−) Linear convergence rather slow (evaluating $f(x)$ might be expensive, can take weeks on HPC systems, when e.g. lattice QCD simulations are necessary).

5.3 Secant method (only for $N = 1$)

- Starting point: x_1, x_2 fulfilling $|f(x_2)| < |f(x_1)|$.
- Secant method might find a root of $f(x)$, not necessarily between x_1 and x_2.
- Basic principle:
  - Iteration.
  - Each step as sketched below.

Figure 5.A



- Algorithm:
  - $n = 2$.

  (1)

$$\Delta x \quad = \quad -f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \quad , \quad x_{n+1} \quad = \quad x_n + \Delta x. \tag{57}$$

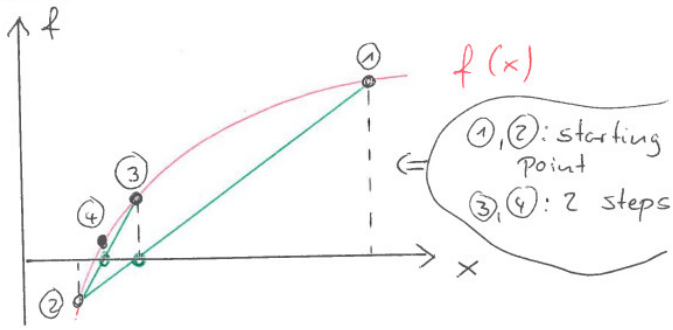
  - *If $|\Delta x|$ sufficiently small:*

    → *x_{n+1} is approximate root.*
    End of algorithm.

    Else:

    → $n = n + 1$.
    Go to (1).

Figure 5.B



- Convergence: $\delta_{n+1} \approx c(\delta_n)^{1.618\ldots}$ (can be shown), i.e. better than linear convergence, better than bisection.
- Advantages and disadvantages:

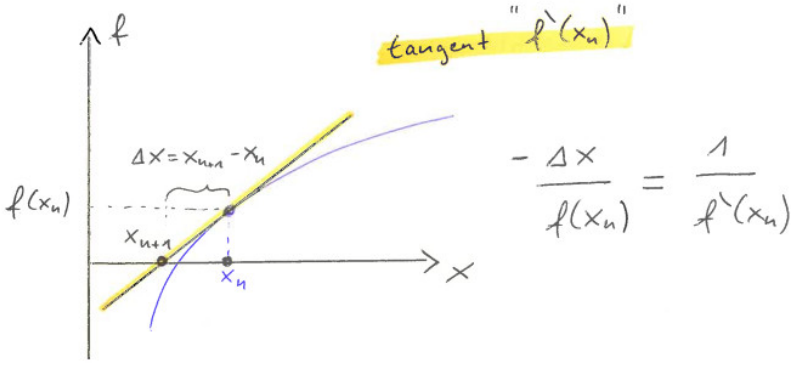
  (+) Converges faster than bisection.

  (−) Does not always find a root.

5.4 Newton-Raphson method (for $N = 1$)

- Starting point: arbitrary x_1.
- Newton-Raphson method might find a root of $f(x)$.
- Basic principle:
  - Similar to secant method (see section 5.3).
  - Use derivative $f'(x_n)$ instead of secant
    $\rightarrow f'$ has to be known analytically/must be cheap to evaluate numerically.
  - Each step as sketched below.

Figure 5.C



- Algorithm:

– $n = 1$.

(1)

$$\Delta x \quad = \quad -\frac{1}{f'(x_n)} f(x_n) \quad , \quad x_{n+1} \quad = \quad x_n + \Delta x. \tag{58}$$

– *If* $|\Delta x|$ *sufficiently small:*

→ x_{n+1} *is approximate root.*
End of algorithm.

Else:

→ $n = n + 1$.
Go to (1).

- Convergence: $\delta_{n+1} \approx (f''(x_n)/2f'(x_n))(\delta_n)^2$ (can be shown), i.e. quadratic convergence, i.e. even better than secant method.
- Advantages and disadvantages:

(+) Converges faster than bisection and secant method.

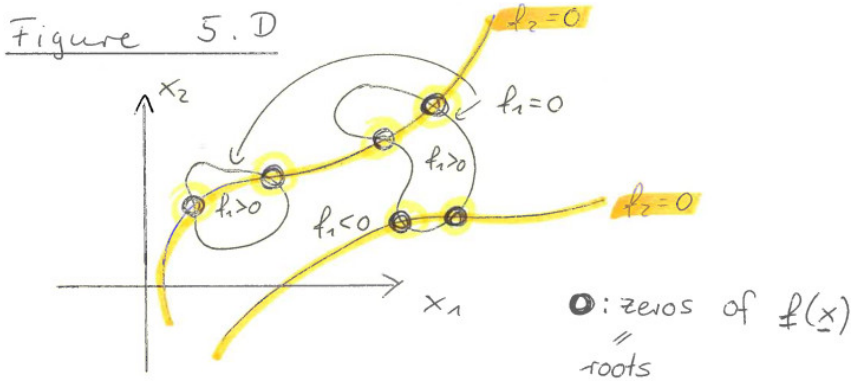
(−) Does not always find a root.

(−) f' has to be known analytically/must be cheap to evaluate numerically.

5.5 Newton-Raphson method (for $N > 1$)

- For $N > 1$ root finding is extremely difficult.

– $N = 2$:
$f_1(x_1, x_2) = 0$, $f_2(x_1, x_2) = 0$.
One has to find intersections of isolines $f_1(x_1, x_2) = 0$ and $f_2(x_1, x_2) = 0$.



****** November 02, 2023 (6th lecture) ******

– $N > 2$:
One has to find intersections of $N - 1$-dimensional isosurfaces $f_j(x_1, \ldots, x_N) = 0$, $j = 1, \ldots, N$.

- Method very successful, if one has a crude estimate of a root (e.g. from a plot, or an approximate analytical calculation).
- Starting point: $\mathbf{x}_1$ (should be close to a root).
- Basic principle:
  - $$0 \;=\; f_j(\mathbf{x}_n + \vec{\delta}) \;=\; f_j(\mathbf{x}_n) + \underbrace{\frac{\partial f_j(\mathbf{x})}{\partial x_k}}_{J_{jk}(\mathbf{x})}\Big|_{\mathbf{x}=\mathbf{x}_n} \delta_k + \mathcal{O}(\delta^2) \quad , \quad j = 1, \ldots, N \tag{59}$$

    ($J_{jk}(\mathbf{x})$: Jacobian matrix) or equivalently

    $$0 \;=\; \mathbf{f}(\mathbf{x}_n) + J(\mathbf{x}_n)\vec{\delta} + \mathcal{O}(\delta^2). \tag{60}$$

  - Neglect $\mathcal{O}(\delta^2)$:

    $$0 \;=\; \mathbf{f}(\mathbf{x}_n) + J(\mathbf{x}_n)\Delta\mathbf{x} \tag{61}$$

    or equivalently

    $$\Delta\mathbf{x} \;=\; -\Big(J(\mathbf{x}_n)\Big)^{-1}\mathbf{f}(\mathbf{x}_n) \tag{62}$$

    ($\Delta\mathbf{x} \approx \vec{\delta}$, i.e. approximate difference between root and $\mathbf{x}_n$).
  - (61) is system of linear equations (solve analytically for $N = 2, 3$ or numerically as discussed in section 7).
  - $N = 1$: $J(x_n) = f'(x_n)$ and (62) becomes

    $$\Delta x \;=\; -\frac{1}{f'(x_n)} f(x_n), \tag{63}$$

    which is identical to (58), left equation, i.e. the $N > 1$ Newton-Raphson method is a generalization of the the $N = 1$ Newton-Raphson method discussed in section 5.4.
- Algorithm:
  - $n = 1$.

  (1)

  $$\Delta\mathbf{x} \;=\; -\Big(J(\mathbf{x}_n)\Big)^{-1}\mathbf{f}(\mathbf{x}_n) \quad , \quad \mathbf{x}_{n+1} \;=\; \mathbf{x}_n + \Delta\mathbf{x}. \tag{64}$$

  - *If* $|\Delta\mathbf{x}|$ *sufficiently small:*

    $\rightarrow$ $\mathbf{x}_{n+1}$ *is approximate root.*
    End of algorithm.

    Else:

    $\rightarrow$ $n = n + 1$.
    Go to (1).

6 Ordinary differential equations, boundary value problems

6.1 Physics motivation

- Newton's EOMs, N point masses m_j,

$$m_j \ddot{\mathbf{r}}_j(t) \quad = \quad \mathbf{F}_j(\mathbf{r}_1(t), \ldots, \mathbf{r}_N(t), \dot{\mathbf{r}}_1(t), \ldots, \dot{\mathbf{r}}_N(t), t) \quad , \quad j = 1, \ldots, N, \tag{65}$$

boundary conditions

$$\mathbf{r}_j(t_1) \quad = \quad \mathbf{r}_{j,1} \quad , \quad \mathbf{r}_j(t_2) \quad = \quad \mathbf{r}_{j,2} \tag{66}$$

("Compute trajectory of a particle, which is at $\mathbf{r}_1$ at time t_1 and at $\mathbf{r}_2$ at time t_2.").

- QM, Schrödinger equation in 1 dimension,

$$-\frac{\hbar^2}{2m}\psi''(x) + V(x)\psi(x) \quad = \quad E\psi(x), \tag{67}$$

boundary conditions

$$\psi(x_1) \quad = \quad \psi(x_2) \quad = \quad 0 \tag{68}$$

(i.e. "$V(x) = \infty$ at $x = x_1, x_2$", e.g. infinite potential well).

  - Example appropriate? E is unknown, i.e. (67) and (68) is rather an eigenvalue problem, not an ordinary boundary value problem ...?
  - Yes, can be reformulated:
    - Consider E as a function of x, i.e. $E = E(x)$.
    - Add another ODE: $E'(x) = 0$.

    → System of ODEs,

$$-\frac{\hbar^2}{2m}\psi''(x) + V(x)\psi(x) \quad = \quad E(x)\psi(x) \quad , \quad E'(x) \quad = \quad 0, \tag{69}$$

    where each solution fulfills $E(x) = \text{const}$.

6.2 Shooting method

- Preparatory step as in section 3.2: rewrite ODEs to system of first order ODEs,

$$\mathbf{y}'(x) \quad = \quad \mathbf{f}(\mathbf{y}(x), x) \tag{70}$$

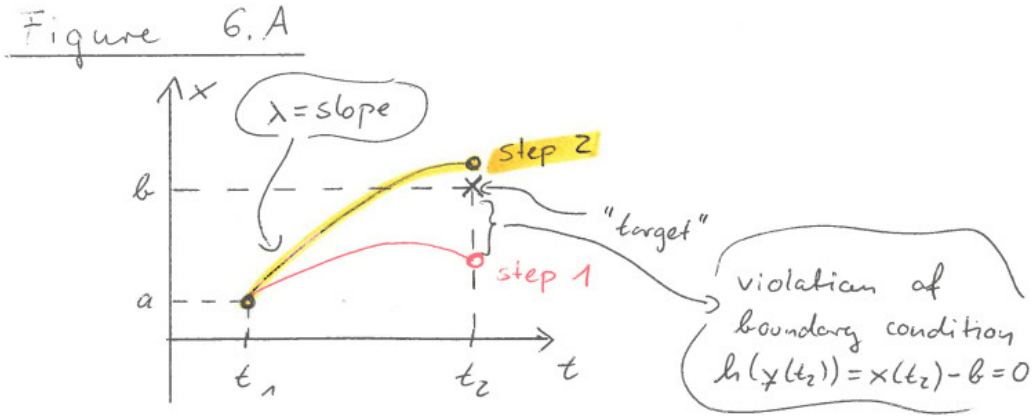
(both $\mathbf{y}$ and $\mathbf{f}$ have N components) and boundary conditions

$$g_j(\mathbf{y}(x_1)) \quad = \quad 0 \quad , \quad j = 1, \ldots, n < N \tag{71}$$
$$h_j(\mathbf{y}(x_2)) \quad = \quad 0 \quad , \quad j = 1, \ldots, N - n. \tag{72}$$

- Basic principle:

- Choose/guess initial conditions $\mathbf{y}(x_1)$ such that
  - boundary conditions $g_j(\mathbf{y}(x_1)) = 0$, $j = 1, \ldots, n < N$ are fulfilled,
  - boundary conditions $h_j(\mathbf{y}(x_2)) = 0$, $j = 1, \ldots, N - n$ are approximately fulfilled ($\mathbf{y}(x_2)$ can be computed using e.g. a RK method from section 3.3).
- Use root finding methods from section 5 (e.g. Newton-Raphson method) to iteratively improve initial conditions $\mathbf{y}(x_1)$, i.e. such that $h_j(\mathbf{y}(x_2)) = 0$.

- Example: mechanics, $m\ddot{x}(t) = F(x(t))$ with $x(t_1) = a$, $x(t_2) = b$.
  - $\mathbf{y}(t) = (x(t), v(t))$, $\mathbf{f}(\mathbf{y}(t), t) = (v(t), F(x(t))/m)$ (as in section 3.2).
  - $g(\mathbf{y}(t_1)) = x(t_1) - a = 0$, $h(\mathbf{y}(t_2)) = x(t_2) - b = 0$.
  - Choose initial conditions $\mathbf{y}(t_1) = (a, \lambda)$.
    - a in 1st component $\rightarrow$ $g(\mathbf{y}(t_1)) = 0$ fulfilled.
    - λ in 2nd component should lead to $h(\mathbf{y}(t_2)) \approx 0$.
  - RK computation of $\mathbf{y}(t)$ from $t = t_1$ to $t = t_2$.



  - Improve initial conditions, i.e. tune λ, using the Newton-Raphson method (see section 5.4):
    - Interpret $h(\mathbf{y}(t_2)) = x(t_2) - b$ as function of λ ($x(t_2)$ depends on initial conditions $\mathbf{y}(t_1)$, i.e. on λ).
    - Compute derivative $dh(\mathbf{y}(t_2))/d\lambda$ (needed by the Newton-Raphson method) numerically (see section 2.3.2).
    - Newton-Raphson step to improve λ:

$$\lambda \quad \rightarrow \quad \lambda - \frac{h(\mathbf{y}(t_2))}{dh(\mathbf{y}(t_2))/d\lambda}. \tag{73}$$

  - Repeat RK computation and Newton-Raphson step, until $h(\mathbf{y}(t_2)) = 0$ (numerically 0, e.g. up to 6 digits).

***** November 07, 2023 (7th lecture) *****

6.2.1 Example: QM, 1 dimension, infinite potential well

- Infinite potential well:

$$V(x) \;=\; \begin{cases} 0 & \text{if } 0 \le x \le L \\ \infty & \text{otherwise} \end{cases} \quad . \tag{74}$$

- Schrödinger equation and boundary conditions:

$$-\frac{\hbar^2}{2m}\psi''(x) \;=\; E\psi(x) \quad , \quad \psi(x=0) \;=\; \psi(x=L) \;=\; 0. \tag{75}$$

- Reformulate equations using exclusively dimensionless quantities:

$$\hat{x} \;=\; \frac{x}{L} \tag{76}$$

$$\rightarrow \quad \frac{d}{d\hat{x}} \;=\; L\frac{d}{dx} \tag{77}$$

$$\rightarrow \quad -\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) \;=\; \underbrace{\frac{2mEL^2}{\hbar^2}}_{=\hat{E}}\psi(\hat{x}) \tag{78}$$

($\hat{E}$ is "dimensionless energy"), i.e.

$$-\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) \;=\; \hat{E}\psi(\hat{x}) \quad , \quad \psi(\hat{x}=0) \;=\; \psi(\hat{x}=1) \;=\; 0. \tag{79}$$

- Analytical solution (to check numerical results):

$$\psi(\hat{x}) \;=\; \sqrt{2}\sin(n\pi\hat{x}) \quad , \quad \hat{E} \;=\; \pi^2 n^2 \quad , \quad n = 1, 2, \ldots \tag{80}$$

- Numerical solution:
  - Rewrite Schrödinger equation to system of first order ODEs:

$$\psi'(\hat{x}) \;=\; \phi(\hat{x}) \quad , \quad \phi'(\hat{x}) \;=\; -\hat{E}(\hat{x})\psi(\hat{x}) \quad , \quad \hat{E}'(\hat{x}) \;=\; 0, \tag{81}$$

($'$ denotes $d/d\hat{x}$) i.e.

$$\mathbf{y}(x) \;=\; \Big(\psi(\hat{x}), \phi(\hat{x}), \hat{E}(\hat{x})\Big) \quad , \quad \mathbf{f}(\mathbf{y}(x), x) \;=\; \Big(\phi(\hat{x}), -\hat{E}(\hat{x})\psi(\hat{x}), 0\Big). \tag{82}$$

  - Initial conditions for RK/shooting method:
    - $\psi(\hat{x} = 0.0) = 0.0$
      (boundary condition at $\hat{x} = 0$),
    - $\phi(\hat{x} = 0.0) = 1.0$
      (must be $\neq 0$, apart from that arbitrary; different choices result in differently normalized wavefunctions),

* $\hat{E}(\hat{x} = 0.0) = \mathcal{E}$
  (will be tuned by Newton-Raphson method such that boundary condition $\psi(\hat{x} = 1) = 0$ is fulfilled).

– C code: see appendix C.

– Crude "graphical determination" of energy eigenvalues (necessary to choose appropriate initial condition for the shooting method):

* Figure 6 shows $\psi(\hat{x} = 1.0)$ as a function of $\mathcal{E}$ computed with 4th order RK; roots indicate energy eigenvalues.

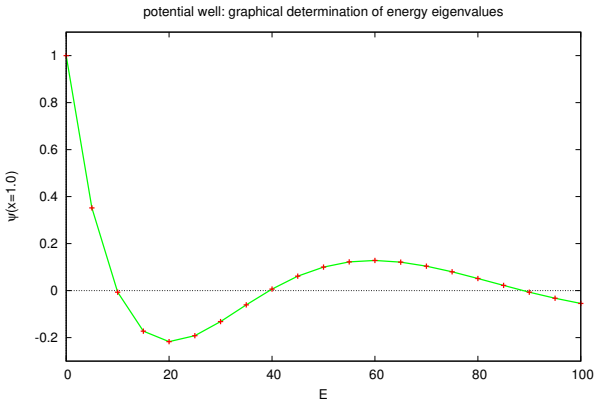


Figure 6: Infinite potential well, crude graphical determination of energy eigenvalues.

* There are 3 eigenvalues in the range $0.0 < \hat{E} < 100.0$:
  $\hat{E}_0 \approx 10.0$, $\hat{E}_1 \approx 40.0$, $\hat{E}_2 \approx 90.0$.

– Shooting method with $\mathcal{E} \in \{10.0, 40.0, 90.0\}$.

* Figure 7 (top) illustrates the first Newton-Raphson step for the second excitation (4th order RK).
* Figure 7 (bottom) shows the resulting non-normalized wave functions of the three lowest states (4th order RK).
* Convergence after three Newton-Raphson steps (4th order RK, 7 digits of accuracy); see program output below.

```
ground state:
E_num = +10.000000 .
E_num =  +9.868296  ,  E_ana =  +9.869604  ,  \psi(x=1) = -0.006541 .
E_num =  +9.869604  ,  E_ana =  +9.869604  ,  \psi(x=1) = +0.000066 .
E_num =  +9.869604  ,  E_ana =  +9.869604  ,  \psi(x=1) = +0.000000 .

1st excitation:
E_num = +40.000000 .
E_num = +39.472958  ,  E_ana = +39.478418  ,  \psi(x=1) = +0.006539 .
E_num = +39.478417  ,  E_ana = +39.478418  ,  \psi(x=1) = -0.000069 .
E_num = +39.478418  ,  E_ana = +39.478418  ,  \psi(x=1) = -0.000000 .
```

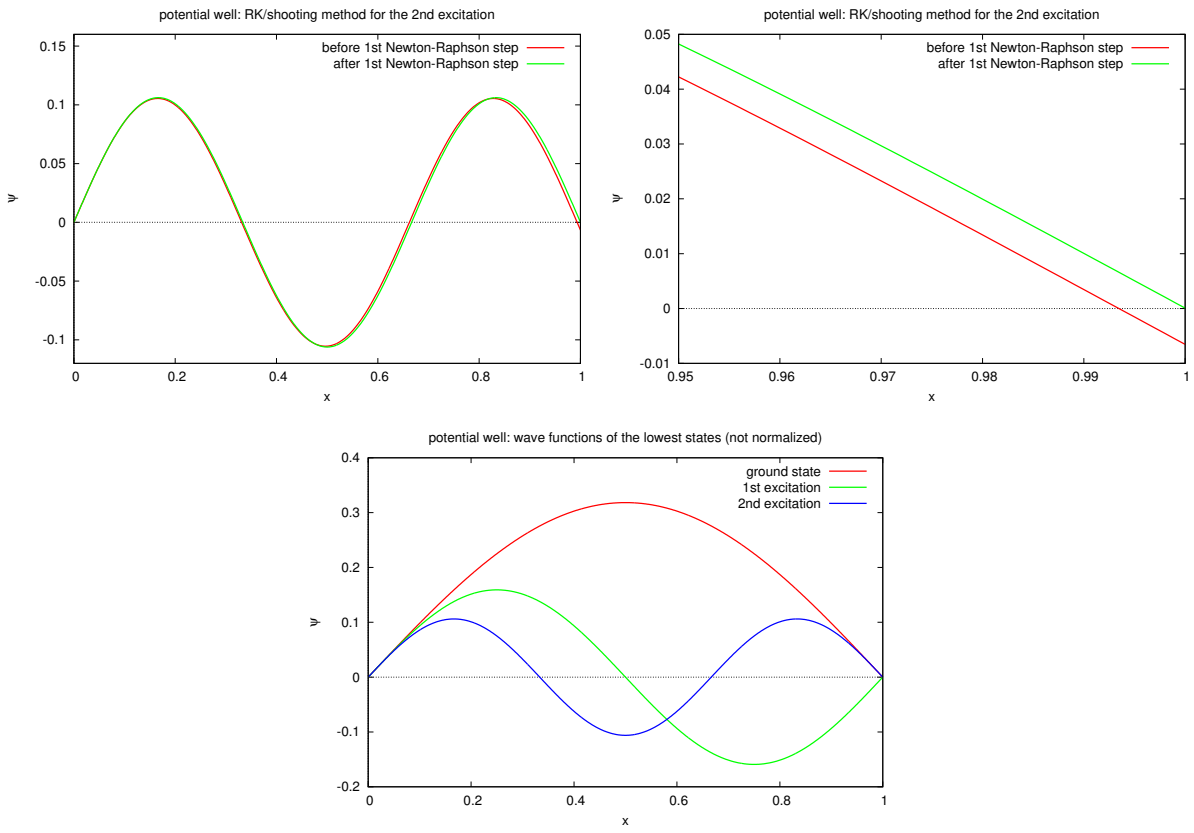


Figure 7: Infinite potential well. **(top)** First Newton-Raphson step for the second excitation. **(bottom)** Non-normalized wave functions of the three lowest states.

```
2nd excitation:
E_num = +90.000000 .
E_num = +88.813303   ,   E_ana = +88.826440   ,   \psi(x=1) = -0.006537 .
E_num = +88.826438   ,   E_ana = +88.826440   ,   \psi(x=1) = +0.000074 .
E_num = +88.826440   ,   E_ana = +88.826440   ,   \psi(x=1) = +0.000000 .
```

6.2.2 Example: QM, 1 dimension, harmonic oscillator

- Schrödinger equation and boundary conditions:

$$-\frac{\hbar^2}{2m}\psi''(x) + \frac{m\omega^2}{2}x^2\psi(x) \quad = \quad E\psi(x) \quad , \quad \psi(x=-\infty) \quad = \quad \psi(x=+\infty) \quad = \quad 0 \qquad (83)$$

  (numerical challenge are boundary conditions at $x = \pm\infty$).

- Reformulate equations using exclusively dimensionless quantities:
  - Length scale from $\hbar$, m, ω:
    $[\hbar] = \mathrm{kg\,m^2/s}$

$[m] = \mathrm{kg}$
$[\omega] = 1/\mathrm{s}$
$\rightarrow$ length scale $a = (\hbar/m\omega)^{1/2}$.

–

$$\hat{x} = \frac{x}{a} \tag{84}$$

$$\rightarrow \quad \frac{d}{d\hat{x}} = a\frac{d}{dx} \tag{85}$$

$$\rightarrow \quad -\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) + \hat{x}^2\psi(\hat{x}) = \underbrace{\frac{2E}{\hbar\omega}}_{=\hat{E}}\psi(\hat{x}) \tag{86}$$

($\hat{E}$ is "dimensionless energy"), i.e.

$$-\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) + \hat{x}^2\psi(\hat{x}) = \hat{E}\psi(\hat{x}) \quad , \quad \psi(\hat{x} = -\infty) = \psi(\hat{x} = +\infty) = 0. \tag{87}$$

- Parity:
  - Parity P: spatial reflection, i.e. $PxP = -x$, $P\psi(+x) = \psi(-x)$.
  - Eigenvalues and eigenfunctions of P:

$$P\psi(x) = \lambda\psi(x) \quad \rightarrow \quad \underbrace{PP}_{=1}\psi(x) = \lambda^2\psi(x) \quad \rightarrow \quad \lambda^2 = 1 \quad \rightarrow \quad \lambda = \pm 1. \tag{88}$$

    * Common notation: $P = \pm$ (instead of $\lambda = \pm$).
    * $P = +$: $P\psi(x) = \psi(-x)$ and $P\psi(x) = +\psi(x) \rightarrow \psi(x) = +\psi(-x)$, i.e. even eigenfunction.
    * $P = -$: $P\psi(x) = \psi(-x)$ and $P\psi(x) = -\psi(x) \rightarrow \psi(x) = -\psi(-x)$, i.e. odd eigenfunction.
  - $[H, P] = 0$, if $V(+x) = V(-x)$, i.e. for symmetric potentials.
    $\rightarrow$ Eigenfunctions $\psi(x)$ of H can be chosen such that they are also eigenfunctions of P.
    $\rightarrow$ $P = +$

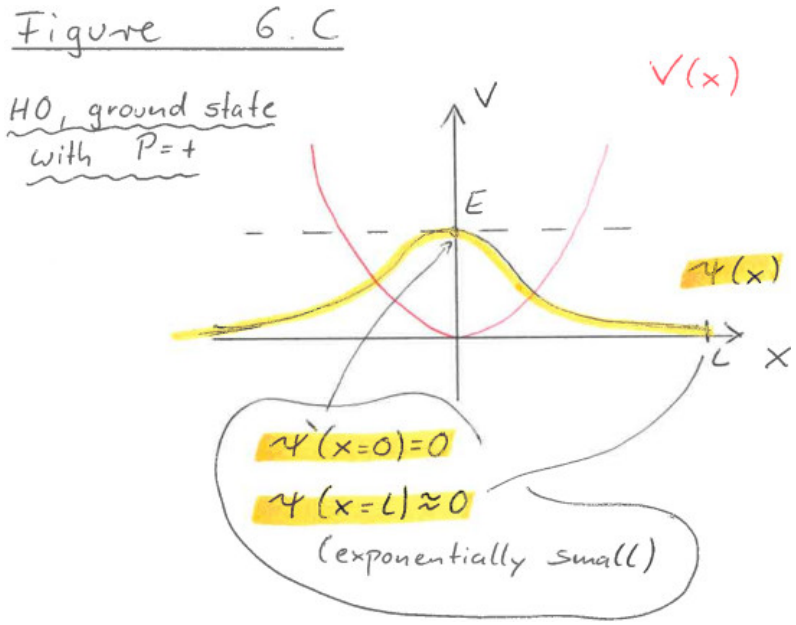
$$\rightarrow \quad \psi(x) = +\psi(-x) \quad \rightarrow \quad \psi'(x = 0) = 0. \tag{89}$$

    $\rightarrow$ $P = -$

$$\rightarrow \quad \psi(x) = -\psi(-x) \quad \rightarrow \quad \psi(x = 0) = 0. \tag{90}$$

****** November 09, 2023 (8th lecture) ******

- Numerical problems with boundary conditions $\psi(\hat{x} = -\infty) = \psi(\hat{x} = +\infty) = 0$ (eq. (87)).
- Numerical solution, first attempt:
  - Use either $\psi'(\hat{x} = 0) = 0$ or $\psi(\hat{x} = 0) = 0$ ((89) or (90)) instead of $\psi(\hat{x} = -\infty) = 0$.
  - Use $\psi(\hat{x} = L/a) = 0$, where $x = L$ is far in the classically forbidden region ($E \ll V(L)$), i.e. where ψ is exponentially suppressed.



– Rewrite Schrödinger equation to system of first order ODEs:

$$\psi'(\hat{x}) \;=\; \phi(\hat{x}) \quad , \quad \phi'(\hat{x}) \;=\; \Big(\hat{x}^2 - \hat{E}(\hat{x})\Big)\psi(\hat{x}) \quad , \quad \hat{E}'(\hat{x}) \;=\; 0, \tag{91}$$

i.e.

$$\mathbf{y}(x) \;=\; \Big(\psi(\hat{x}), \phi(\hat{x}), \hat{E}(\hat{x})\Big) \quad , \quad \mathbf{f}(\mathbf{y}(x), x) \;=\; \Big(\phi(\hat{x}), \Big(\hat{x}^2 - \hat{E}(\hat{x})\Big)\psi(\hat{x}), 0\Big). \tag{92}$$

– Initial conditions for $P = +$ for RK/shooting method:

* $\psi(\hat{x} = 0.0) = 1.0$
  (must be $\neq 0$, apart from that arbitrary; different choices result in differently normalized wavefunctions),
* $\phi(\hat{x} = 0.0) = 0.0$
  (boundary condition at $\hat{x} = 0$),
* $\hat{E}(\hat{x} = 0.0) = \mathcal{E}$
  (will be tuned by Newton-Raphson method such that boundary condition $\psi(\hat{x} = L/a) = 0$ is fulfilled; has to be close to the energy eigenvalue one is interested in [e.g. ground state: $E = \hbar\omega/2$, i.e. $\hat{E} = 1$, i.e. choose $\mathcal{E} \approx 1$]; typically $\mathcal{E}$ is the result of a crude graphical determination of energy eigenvalues [see section 6.2.1]).

(For $P = -$ use $\psi(\hat{x} = 0.0) = 0.0$, $\phi(\hat{x} = 0.0) = 1.0$.)

– Boundary condition $\psi(\hat{x} = L/a) = 0$ numerically hard to implement; a tiny admixture of the exponentially increasing solution will dominate for large $\hat{x}$, as shown in Figure 8 (4th order RK).

- Numerical solution, more practical approach:

  – Use "*... a tiny admixture of the exponentially increasing solution will dominate for large $\hat{x}$ ...*" to your advantage:

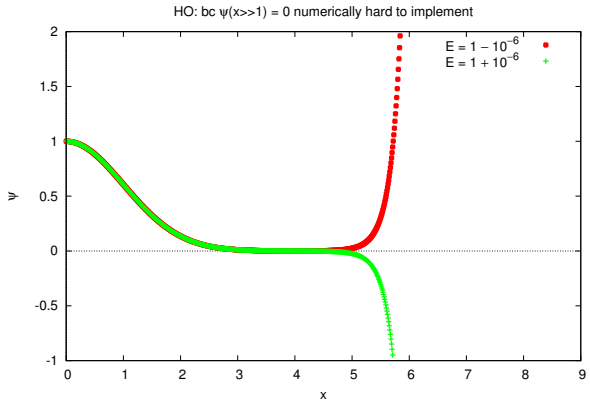


Figure 8: HO, numerical problems with boundary condition $\psi(\hat{x} = L/a) = 0$.

  - Start far in the classically forbidden region using arbitrary initial conditions, e.g.
    $\psi(\hat{x} = L/a) = 1.0$, $\phi(\hat{x} = L/a) = 0.0$, $\hat{E}(\hat{x} = L/a) = \mathcal{E}$
    or
    $\psi(\hat{x} = L/a) = 0.0$, $\phi(\hat{x} = L/a) = 1.0$, $\hat{E}(\hat{x} = L/a) = \mathcal{E}$
    or
    ...
  - Tune $\mathcal{E}$ via the RK/shooting method such that $\psi'(\hat{x} = 0) = 0$ for $P = +$ (or $\psi(\hat{x} = 0) = 0$ for $P = -$).

- From Figure 9 (top) one can read off rough estimates for the energy eigenvalues, which can be used to intialize $\mathcal{E}$ (4th order RK).
- Figure 9 (bottom) shows the resulting wave functions of the four lowest states (4th order RK).
- For initial values $\mathcal{E} \in \{0.9, 2.9, 4.9, 6.9\}$ convergence after three Newton-Raphson steps (7 digits of accuracy); see program output below.

```
ground state:
E_num =  +0.900000 .
E_num =  +0.988598.
E_num =  +0.999834.
E_num =  +1.000000.

1st excitation:
E_num =  +2.900000 .
E_num =  +2.988617.
E_num =  +2.999835.
E_num =  +3.000000.

2nd excitation:
E_num =  +4.900000 .
E_num =  +4.990699.
E_num =  +4.999911.
```

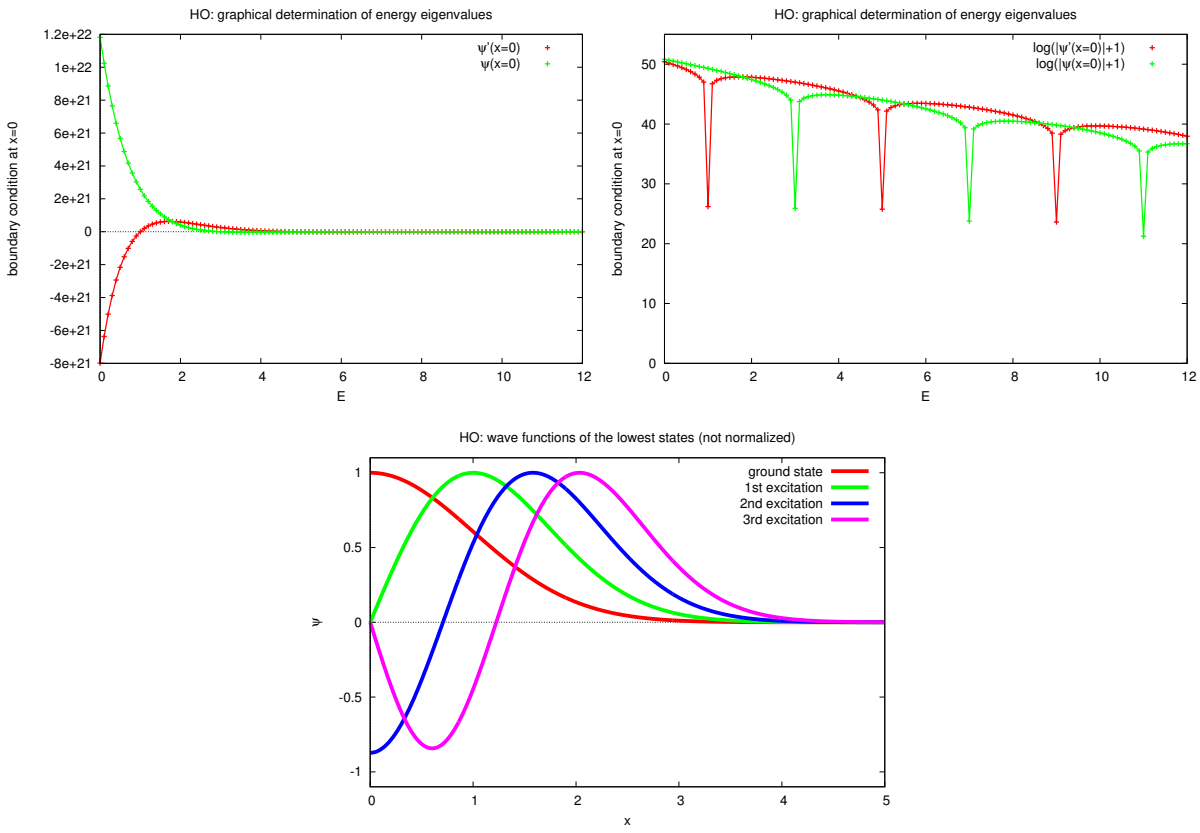


Figure 9: HO. **(top)** Crude graphical determination of energy eigenvalues. **(bottom)** Wave functions of the four lowest states.

```
E_num =  +5.000000.

3rd excitation:
E_num =  +6.900000 .
E_num =  +6.990720.
E_num =  +6.999911.
E_num =  +7.000000.
```

6.2.3 Example: QM, 3 dimensions, spherically symmetric potential

- Spherically symmetric potential: $V(\mathbf{r}) = V(r)$, where $r = |\mathbf{r}|$.
- Rewrite Schrödinger equation in spherical coordinates ...
- ... angular dependence of wavefunctions proportional to spherical harmonics, i.e. $\psi(r, \vartheta, \varphi) \propto Y_{lm}(\vartheta, \varphi)$ ...
- ... remaining radial equation is second order ODE in r, which can be solved using RK/shooting.

- For details see e.g. Ref. [2].
- Solving such radial equations numerically is common in up-to-date research.
  - For example [3]:
    The potential of two heavy $\bar{b}$ quarks in the presence of two light u and/or d quarks can be computed with lattice QCD (a numerical method to study QCD). This potential can be used in a standard non-relativistic Schrödinger equation to check, whether the quarks may form a stable $\bar{b}\bar{b}ud$ tetraquark.
    "C. Numerical solution of Schrödinger's equation
    To investigate the existence of a bound state rigorously, we numerically solve the Schrödinger equation with the Hamiltonian (6). The strongest binding is expected in an s-wave, for which the radial equation is

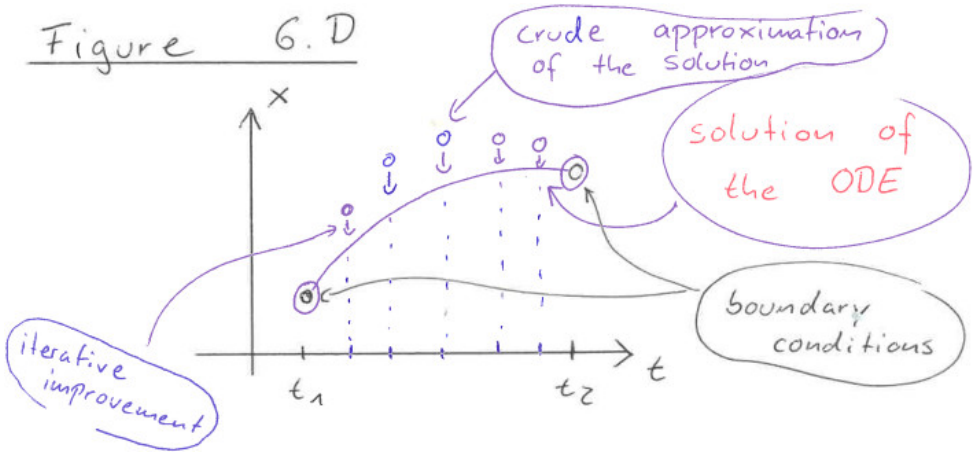
$$\left(-\frac{\hbar^2}{2\mu}\frac{d^2}{dr^2} + 2m_B + V(r) \right) R(r) \quad = \quad ER(r) \tag{93}$$

    with the wave function $\psi(r) = R(r)/r$. We impose Dirichlet boundary conditions $R(r_{max}) = 0$ at sufficiently large r_{max} (we checked that results are stable for $r_{max} \gtrsim 10\,fm$). The radial equation (9) can be solved by standard methods (e.g. 4th order Runge-Kutta shooting) up to arbitrary numerical precision."
  - For example [4]:
    The potential of a $\bar{b}b$ (quark-antiquark) pair and of a $\bar{B}B$ (meson-antimeson) pair (a B meson is composed of a $\bar{b}$ quark and a light u or d quark) can be computed with lattice QCD. These potentials can be used in a coupled-channel non-relativistic Schrödinger equation to predict and explore the spectrum and properties of bottomonium (both stable states and resonances; bottomonium = $\bar{b}b$).
    "A. Numerical methods to solve the coupled channel Schrödinger equation and to determine the poles of the T matrix
    In Section II C we defined the entries of the T matrix (19) as the a priori unknown coefficients $t_{...}$ appearing in the $r \to \infty$ boundary conditions (17) and (18). To determine these coefficients, one has to solve the coupled channel Schrödinger equation (11). To cross check our results, we used two rather different numerical methods. The first method corresponds to discretizing the radial coordinate by a uniform grid and solving the resulting system of linear equations using methods from standard textbooks (for details see Ref. [28]). The second method corresponds to using an ordinary 4th order Runge-Kutta algorithm.
    To find the poles of $T_{\tilde{J}}$ in the complex energy plane, characterized by at least one of its eigenvalues approaching infinity, we applied the Newton-Raphson method to find the roots of $1/det(T_{\tilde{J}})$."

6.3 Relaxation methods

- See e.g. Ref. [1], section 18.0.
- Discretize time, guess solution ...
- ... then iteratively improve the solution, until the ODE is fulfilled.

Figure 6.D


crude approximation of the solution
solution of the ODE
boundary conditions
iterative improvement
x
t
t_1
t_2

7 Solving systems of linear equations

7.1 Problem definition, general remarks

- A: $N \times N$ matrix, i.e. a square matrix, with $\det(A) \neq 0$ (rows and columns are linearly independent).
- $\mathbf{b}_j$, $j = 0, \ldots, M-1$: vectors with N components.
- Typical problems:
  - Solve $A\mathbf{x}_j = \mathbf{b}_j$ (possibly for several vectors $\mathbf{b}_j$).
  - Compute A^{-1}.
    Do not compute A^{-1} and solve $A\mathbf{x}_j = \mathbf{b}_j$ via $\mathbf{x}_j = A^{-1}\mathbf{b}_j$ ... roundoff errors are typically large.
  - Compute $\det(A)$.
- Two types of methods:
  - **Direct methods**:
    - Solution/result after a finite fixed number of arithmetic operations.
    - For large N roundoff errors are typically large.
  - **Iterative methods**:
    - Iterative improvement of approximate solution/result.
    - No problems with roundoff errors.
    - Computationally expensive; therefore, only suited for sparse matrices ("*dünn besetzte Matrizen*").
- How large can N be?
  - Dense matrices ("*dicht besetzte Matrizen*"): $N \gtrsim \mathcal{O}(1000)$.
  - Sparse matrices: $N \gtrsim \mathcal{O}(10^6)$.
- It might be a good idea to check your result, e.g. by computing $A\mathbf{x}_j$ and comparing to $\mathbf{b}_j$, e.g. to exclude large roundoff errors.

****** November 14, 2023 (9th lecture) ******

7.2 Gauss-Jordan elimination (a direct method)

- Goal: solve $A\mathbf{x}_j = \mathbf{b}_j$.
- Problem "compute A^{-1}" included: choose $\mathbf{b}_j = \mathbf{e}_j$, then $A^{-1} = (\mathbf{x}_1\, \mathbf{x}_2\, \ldots\, \mathbf{x}_N)$.
- Basic idea: add/subtract multiples of the linear equations, until solution $\mathbf{x}_j$ is obvious.

- Notation

$$A\mathbf{x}_j \quad = \quad \mathbf{b}_j \quad \rightarrow \quad \left| \begin{array}{cccc|ccc} a_{0,0} & a_{0,1} & a_{0,2} & \dots & b_{0,0} & \dots & b_{0,M-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \dots & b_{1,0} & \dots & b_{1,M-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \dots & b_{2,0} & \dots & b_{2,M-1} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \end{array} \right| . \tag{94}$$

- Step 1: elimination of column 0,

$$a_{0,k}^{(1)} \quad = \quad \frac{a_{0,k}}{a_{0,0}} \quad , \quad k = 0, \dots, N-1 \tag{95}$$

$$b_{0,k}^{(1)} \quad = \quad \frac{b_{0,k}}{a_{0,0}} \quad , \quad k = 0, \dots, M-1 \tag{96}$$

$$a_{j,k}^{(1)} \quad = \quad a_{j,k} - a_{j,0} a_{0,k}^{(1)} \quad , \quad j = 1, \dots, N-1 \quad , \quad k = 0, \dots, N-1 \tag{97}$$

$$b_{j,k}^{(1)} \quad = \quad b_{j,k} - a_{j,0} b_{0,k}^{(1)} \quad , \quad j = 1, \dots, N-1 \quad , \quad k = 0, \dots, M-1 \tag{98}$$

(assumption: $a_{0,0} \neq 0$), then $a_{0,0}^{(1)} = 1$ and $a_{j,0}^{(1)} = 0$, $j = 1, \dots, N-1$, i.e.

$$\left| \begin{array}{cccc|ccc} 1 & a_{0,1}^{(1)} & a_{0,2}^{(1)} & \dots & b_{0,0}^{(1)} & \dots & b_{0,M-1}^{(1)} \\ 0 & a_{1,1}^{(1)} & a_{1,2}^{(1)} & \dots & b_{1,0}^{(1)} & \dots & b_{1,M-1}^{(1)} \\ 0 & a_{2,1}^{(1)} & a_{2,2}^{(1)} & \dots & b_{2,0}^{(1)} & \dots & b_{2,M-1}^{(1)} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \end{array} \right| . \tag{99}$$

- Step n ($n = 2, \dots, N$): elimination of column $n-1$,

$$a_{n-1,k}^{(n)} \quad = \quad \frac{a_{n-1,k}^{(n-1)}}{a_{n-1,n-1}^{(n-1)}} \quad , \quad k = n-1, \dots, N-1 \tag{100}$$

$$b_{n-1,k}^{(n)} \quad = \quad \frac{b_{n-1,k}^{(n-1)}}{a_{n-1,n-1}^{(n-1)}} \quad , \quad k = 0, \dots, M-1 \tag{101}$$

$$a_{j,k}^{(n)} \quad = \quad a_{j,k}^{(n-1)} - a_{j,n-1}^{(n-1)} a_{n-1,k}^{(n)} \quad , \quad j \neq n-1 \quad , \quad k = n-1, \dots, N-1 \tag{102}$$

$$b_{j,k}^{(n)} \quad = \quad b_{j,k}^{(n-1)} - a_{j,n-1}^{(n-1)} b_{n-1,k}^{(n)} \quad , \quad j \neq n-1 \quad , \quad k = 0, \dots, M-1 \tag{103}$$

(assumption: $a_{n-1,n-1}^{(n-1)} \neq 0$), then $a_{n-1,n-1}^{(n)} = 1$ and $a_{j,n-1}^{(n)} = 0$, $j \neq n-1$, i.e. column $n-1$ contains $0 \dots 0\, 1\, 0 \dots 0$.

- (95) to (98) are contained in (100) to (103), when defining $a_{j,k}^{(0)} = a_{j,k}$, $b_{j,k}^{(0)} = b_{j,k}$.

- After step N:

$$\left| \begin{array}{cccc|ccc} 1 & 0 & 0 & \dots & b_{0,0}^{(N)} & \dots & b_{0,M-1}^{(N)} \\ 0 & 1 & 0 & \dots & b_{1,0}^{(N)} & \dots & b_{1,M-1}^{(N)} \\ 0 & 0 & 1 & \dots & b_{2,0}^{(N)} & \dots & b_{2,M-1}^{(N)} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \end{array} \right| \quad \rightarrow \quad \mathbb{1}\mathbf{x}_j \quad = \quad \mathbf{b}_j^{(N)} \tag{104}$$

i.e. "**b** columns" are solutions $\mathbf{x}_j$.

- Advantages and disadvantages:

  (−) Rather slow to solve $A\mathbf{x}_j = \mathbf{b}_j$.

  (−) All vectors $\mathbf{b}_j$ have to be treated at the same time, otherwise even more inefficient.

  (+) Quite o.k. to compute A^{-1}.

7.2.1 Pivoting

- Problems, when using the Gauss-Jordan elimination as presented above:
  - Assumption $a^{(n-1)}_{n-1,n-1} \neq 0$ might not be fulfilled.
  - Large roundoff errors, if $a^{(n-1)}_{n-1,n-1} \approx 0$.
- Solution: reorder linear equations, i.e. rows of A and $\mathbf{b}_j$, in a numerically advantageous way.
- **Partial pivoting**:
  - Before step n swap row $n-1$ and row j, where $n-1 \leq j \leq N-1$ and
    $$|a^{(n-1)}_{j,n-1}| \quad = \quad \max_k |a^{(n-1)}_{k,n-1}|. \tag{105}$$

  → $a^{(n-1)}_{n-1,n-1} \neq 0$, because $\det(A) \neq 0$ (see section 7.1).

  → Significantly smaller roundoff errors.
- **Scaled partial pivoting**:
  - Problem with partial pivoting:
    - E.g., if $a_{0,0} = |a^{(0)}_{0,0}|$ is small, then partial pivoting will swap line 0 with another line before step 1.
    - However, if you multiply line 0 with a huge number, before using the Gauss-Jordan elimination method, you solve an equivalent system of linear equations, which has the same solution; $a_{0,0} = |a^{(0)}_{0,0}|$ is now large and pivoting will not swap line 0 with another line before step 1; roundoff errors might then be rather large.
  - Before step n swap row $n-1$ and row j, where $n-1 \leq j \leq N-1$ and
    $$\frac{|a^{(n-1)}_{j,n-1}|}{\max_l |a^{(0)}_{j,l}|} \quad = \quad \max_k \frac{|a^{(n-1)}_{k,n-1}|}{\max_l |a^{(0)}_{k,l}|} \tag{106}$$
    ("the $n-1$-th element in line j is large compared to the other elements in the line").
- There are even better pivoting strategies, e.g. **full pivoting**, where also columns are swapped.

7.3 Gauss elimination with backward substitution (a direct method)

- Similar to Gauss-Jordan elimination:
  - Step n ($n = 1, \ldots, N-1$):
    - Proceed as defined for Gauss-Jordan elimination (section 7.2) ...
    - ... but modify only rows below row $n-1$, i.e. generate 0's below $a^{(n-1)}_{n-1,n-1}$, but not above.

$$\left|\begin{array}{cccc|ccc} a^{(0)}_{0,0} & a^{(0)}_{0,1} & a^{(0)}_{0,2} & \ldots & b^{(0)}_{0,0} & \ldots & b^{(0)}_{0,M-1} \\ 0 & a^{(1)}_{1,1} & a^{(1)}_{1,2} & \ldots & b^{(1)}_{1,0} & \ldots & b^{(1)}_{1,M-1} \\ 0 & 0 & a^{(2)}_{2,2} & \ldots & b^{(2)}_{2,0} & \ldots & b^{(2)}_{2,M-1} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \end{array}\right| . \tag{107}$$

  - Then backward substitution, i.e. computation of $\mathbf{x}_n$:
    - Start with

$$x_{N-1,n} = \frac{b^{(N-1)}_{N-1,n}}{a^{(N-1)}_{N-1,N-1}} . \tag{108}$$

    - Then

$$x_{N-2,n} = \frac{b^{(N-2)}_{N-2,n} - a^{(N-2)}_{N-2,N-1} x_{N-1,n}}{a^{(N-2)}_{N-2,N-2}} . \tag{109}$$

    - I.e. perform N steps $j = N-1, N-2, \ldots, 0$,

$$x_{j,n} = \frac{b^{(j)}_{j,n} - \sum_{k=j+1}^{N-1} a^{(j)}_{j,k} x_{k,n}}{a^{(j)}_{j,j}} . \tag{110}$$

- Advantages and disadvantages:
  - (−) All vectors $\mathbf{b}_j$ have to be treated at the same time, otherwise inefficient.
  - (+) For a small number of vectors $\mathbf{b}_j$, i.e. for $M \ll N$, Gauss elimination with backward substitution is $\approx 1.5\times$ faster than Gauss-Jordan elimination.
- Numerical experiment:
  - Random matrices and vectors, $N \in \{4, 100\}$, elements $a_{j,k}$ and b_j chosen uniformly in $[-1, +1]$.
  - Gauss elimination with backward substitution using different pivoting strategies.
  - Corresponding C code: see appendix D.

```
N = 4
-----

no pivoting:

+0.68  -0.21  +0.57  +0.60  |  +0.82
```

```
-0.60  -0.33  +0.54  -0.44  |  +0.11
-0.05  +0.26  -0.27  +0.03  |  +0.90
+0.83  +0.27  +0.43  -0.72  |  +0.21

+0.68  -0.21  +0.57  +0.60  |  +0.82
+0.00  -0.52  +1.04  +0.09  |  +0.84
+0.00  +0.24  -0.23  +0.07  |  +0.96
+0.00  +0.53  -0.26  -1.45  |  -0.79

+0.68  -0.21  +0.57  +0.60  |  +0.82
+0.00  -0.52  +1.04  +0.09  |  +0.84
+0.00  +0.00  +0.26  +0.11  |  +1.35
+0.00  +0.00  +0.81  -1.36  |  +0.07

+0.68  -0.21  +0.57  +0.60  |  +0.82
+0.00  -0.52  +1.04  +0.09  |  +0.84
+0.00  +0.00  +0.26  +0.11  |  +1.35
+0.00  +0.00  +0.00  -1.69  |  -4.19

x = ( -2.22  +7.31  +4.24  +2.47 ).

b_check = ( +0.82  +0.11  +0.90  +0.21 ).

b_check - b = ( -2.2e-16  +1.5e-16  -1.1e-16  -1.7e-15 ).

|b_check - b| = +1.74535e-15.

--------------------

partial pivoting:

+0.68  -0.21  +0.57  +0.60  |  +0.82
-0.60  -0.33  +0.54  -0.44  |  +0.11
-0.05  +0.26  -0.27  +0.03  |  +0.90
+0.83  +0.27  +0.43  -0.72  |  +0.21

+0.83  +0.27  +0.43  -0.72  |  +0.21
+0.00  -0.13  +0.85  -0.97  |  +0.26
+0.00  +0.27  -0.25  -0.01  |  +0.92
+0.00  -0.43  +0.21  +1.18  |  +0.65

+0.83  +0.27  +0.43  -0.72  |  +0.21
+0.00  -0.43  +0.21  +1.18  |  +0.65
+0.00  +0.00  -0.11  +0.73  |  +1.32
+0.00  +0.00  +0.79  -1.33  |  +0.07

+0.83  +0.27  +0.43  -0.72  |  +0.21
+0.00  -0.43  +0.21  +1.18  |  +0.65
+0.00  +0.00  +0.79  -1.33  |  +0.07
+0.00  +0.00  +0.00  +0.54  |  +1.33

x = ( -2.22  +7.31  +4.24  +2.47 ).

b_check = ( +0.82  +0.11  +0.90  +0.21 ).

b_check - b = ( -1.1e-16  -3.6e-16  -3.3e-16  +1.1e-16 ).
```

```
|b_check - b| = +5.15537e-16.

--------------------

scaled partial pivoting:

+0.68  -0.21  +0.57  +0.60  |  +0.82
-0.60  -0.33  +0.54  -0.44  |  +0.11
-0.05  +0.26  -0.27  +0.03  |  +0.90
+0.83  +0.27  +0.43  -0.72  |  +0.21

+0.68  -0.21  +0.57  +0.60  |  +0.82
+0.00  -0.52  +1.04  +0.09  |  +0.84
+0.00  +0.24  -0.23  +0.07  |  +0.96
+0.00  +0.53  -0.26  -1.45  |  -0.79

+0.68  -0.21  +0.57  +0.60  |  +0.82
+0.00  +0.24  -0.23  +0.07  |  +0.96
+0.00  +0.00  +0.55  +0.23  |  +2.88
+0.00  +0.00  +0.25  -1.59  |  -2.88

+0.68  -0.21  +0.57  +0.60  |  +0.82
+0.00  +0.24  -0.23  +0.07  |  +0.96
+0.00  +0.00  +0.55  +0.23  |  +2.88
+0.00  +0.00  +0.00  -1.69  |  -4.19

x = ( -2.22  +7.31  +4.24  +2.47 ).

b_check = ( +0.82  +0.11  +0.90  +0.21 ).

b_check - b = ( -2.2e-16  -6.9e-17  +1.1e-16  -1.5e-15 ).

|b_check - b| = +1.52081e-15.

--------------------

N = 100
-------

no pivoting:

|b_check - b| = +2.25693e-11.

partial pivoting:

|b_check - b| = +1.46047e-12.

scaled partial pivoting:

|b_check - b| = +3.28886e-13.
```

****** November 16, 2023 (10th lecture) ******

7.4 LU decomposition (a direct method)

- LU decomposition of A:

$$A = LU \tag{111}$$

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots \\ \alpha_{1,0} & 1 & 0 & 0 & \dots \\ \alpha_{2,0} & \alpha_{2,1} & 1 & 0 & \dots \\ \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix} \tag{112}$$

$$U = \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} & \dots \\ 0 & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \dots \\ 0 & 0 & \beta_{2,2} & \beta_{2,3} & \dots \\ 0 & 0 & 0 & \beta_{3,3} & \dots \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix}. \tag{113}$$

  - L: lower triangular matrix.
  - U: upper triangular matrix.
  - Allows efficient computation of the solution of $A\mathbf{x} = \mathbf{b}$ as well as of $\det(A)$ (see section 7.4.2 and section 7.4.3).

7.4.1 Crout's algorithm

- To compute the LU decomposition of A, one has to solve N^2 equations,

$$a_{j,k} = \sum_{l=0}^{N-1} \alpha_{j,l}\beta_{l,k}, \tag{114}$$

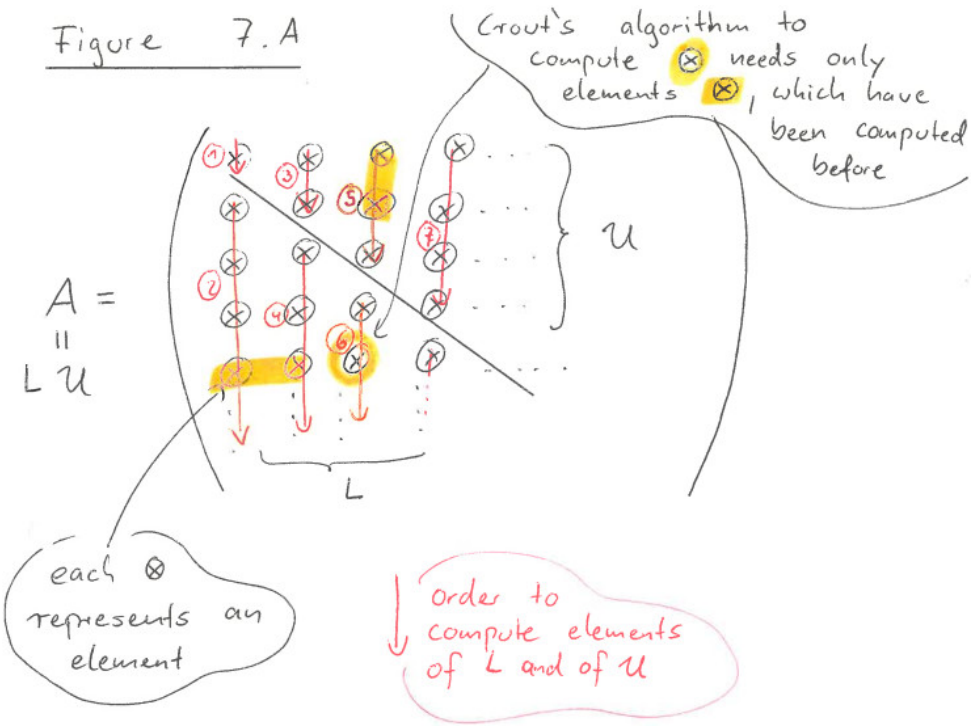
  with respect to $\alpha_{j,k}$ and $\beta_{j,k}$.

- Solving these equations is trivial, when considering them in a particular order:
  - For $k = 0, 1, \dots, N-1$, i.e. for all columns:
    - Step 1:

$$\beta_{j,k} = a_{j,k} - \sum_{l=0}^{j-1} \alpha_{j,l}\beta_{l,k} \quad , \quad j = 0, 1, \dots, k. \tag{115}$$

    - Step 2:

$$\alpha_{j,k} = \frac{1}{\beta_{k,k}} \left(a_{j,k} - \sum_{l=0}^{k-1} \alpha_{j,l}\beta_{l,k} \right) \quad , \quad j = k+1, k+2, \dots, N-1. \tag{116}$$



- Pivoting as important as for Gauss-Jordan elimination and for Gauss elimination with backward substitution.
  - Proceed as discused in section 7.2.1, e.g. use partial pivoting or scaled partial pivoting.
  - For $j = k$ in (115) use "optimal row", i.e. swap row k with one of the rows $k+1, k+2, \ldots, N-1$ (the resulting LU decomposition corresponds to a "row-permuted matrix A").
  - "Optimal" depends on the pivoting strategy, e.g. for partial pivoting the optimal row has the largest $\beta_{k,k}$.
  - Optimal row can be determined rather efficiently, because expressions marked in red in (115) and (116) are identical for $j \geq k$
    $\rightarrow$ first compute all "red expressions"
    $\rightarrow$ then exchange rows according to pivoting strategy.

7.4.2 Computation of the solution of $A\mathbf{x} = \mathbf{b}$

- Proceed in two steps:

  (1) Compute $\mathbf{y}$, defined by

  $$A\mathbf{x} \quad = \quad L \underbrace{U\mathbf{x}}_{=\mathbf{y}} \quad = \quad \mathbf{b}, \tag{117}$$

  via forward substitution,

  $$y_j \quad = \quad b_j - \sum_{k=0}^{j-1} \alpha_{j,k} y_k \quad , \quad j = 0, 1, \ldots, N-1, \tag{118}$$

i.e. solve $L\mathbf{y} = \mathbf{b}$ (note that, when pivoting has been used in the computation of the LU decomposition, the components of $\mathbf{b}$ have to be reordered accordingly, i.e. one has to keep track of and store the permutation of rows, while computing the LU decomposition).

(2) Compute $\mathbf{x}$ via backward substitution (as in section 7.3),

$$x_j = \frac{y_j - \sum_{k=j+1}^{N-1} \beta_{j,k} x_k}{\beta_{j,j}} \quad , \quad j = N-1, N-2, \ldots, 0, \tag{119}$$

i.e. solve $U\mathbf{x} = \mathbf{y}$.

- Advantages and disadvantages:

  (+) LU decomposition independent of vectors $\mathbf{b}_j$, i.e. corresponding solutions $\mathbf{x}_j$ do not have to be computed at the same time.

  (+) Not slower than Gauss-Jordan elimination and Gauss elimination with backward substitution for $A\mathbf{x}_j = \mathbf{b}_j$ and for A^{-1}.

  (+) Allows computation of $\det(A)$ (see section 7.4.3)

7.4.3 Computation of $\det(A)$

- $$\det(A) = \det(LU) = \underbrace{\det(L)}_{=1} \det(U) = \prod_{j=0}^{N-1} \beta_{j,j}. \tag{120}$$

- Pivoting can change the sign of $\det(A)$:

$$\det(A) = (-1)^{\text{sign(row permutation)}} \prod_{j=0}^{N-1} \beta_{j,j}. \tag{121}$$

7.5 QR decomposition (a direct method)

- Due to limited time not discussed.

7.6 Iterative refinement of the solution of $A\mathbf{x} = \mathbf{b}$ (for direct methods)

- Numerically obtained $\mathbf{x}$ (e.g. via LU decomposition) is only approximate solution of $A\mathbf{x} = \mathbf{b}$, because of roundoff errors, i.e. $A\mathbf{x} = \tilde{\mathbf{b}} \neq \mathbf{b}$

- Refine $\mathbf{x}$ as follows:

$$A\Big(\mathbf{x} + \delta\mathbf{x}\Big) = \mathbf{b} \quad \rightarrow \quad A\delta\mathbf{x} = \mathbf{b} - \underbrace{A\mathbf{x}}_{=\tilde{\mathbf{b}}} = \delta\mathbf{b}, \tag{122}$$

i.e. solve

$$A\delta\mathbf{x} \quad = \quad \delta\mathbf{b}; \tag{123}$$

refined solution is $\mathbf{x} + \delta\mathbf{x}$.

- Several iterations possible.
- Highly recommended:
  - Computationally inexpensive, when using the LU decomposition
  - Might improve accuracy significantly.

7.7 Conjugate gradient method (an iterative method)

- Problem: storing $N \times N$ matrices for $N \gg 10000$ typically exceeds memory limit.
  - E.g. a real 10000×10000 matrix requires $(10000)^2 \times 8 \approx 1\,\text{GB}$.
- Sparse matrices of that size can be stored easily (only elements $\neq 0$ need to be stored).
  - E.g. a real tridiagonal 10000×10000 matrix requires $3 \times 10000 \times 8 \ll 1\,\text{MB}$.
- Applying direct methods to large sparse matrices still not practicable, because direct methods "transform sparse matrices into dense matrices".
- Iterative methods do not transform A, i.e. only use the original A.
  $\rightarrow$ Iterative methods particularly suited to solve $A\mathbf{x} = \mathbf{b}$, when A is a large sparse matrix.

****** November 21, 2023 (11th lecture) ******

7.7.1 Symmetric positive definite A

- Goal: solve $A\mathbf{x} = \mathbf{b}$ (a single vector $\mathbf{b}$, no computation of A^{-1} or $\det(A)$).
- Basic idea:
  - Minimize

    $$f(\mathbf{x}) \quad = \quad \frac{1}{2}\mathbf{x}A\mathbf{x} - \mathbf{b}\mathbf{x}, \tag{124}$$

    which describes an N-dimensional paraboloid, with respect to $\mathbf{x}$.
  - The minimum is characterized by

    $$\nabla f(\mathbf{x}) \quad = \quad A\mathbf{x} - \mathbf{b} \quad = \quad 0, \tag{125}$$

    i.e. it is the solution of $A\mathbf{x} = \mathbf{b}$.
- Algorithm:
  - *Guess solution* $\mathbf{x}_0$*, e.g.* $\mathbf{x}_0 = 0$ *(can be far away from the correct solution).*

– $n = 0$.

(1) Select direction $\mathbf{p}_n$ *(details below).*

– *Minimize* $f(\mathbf{x}_n + \alpha_n \mathbf{p}_n)$ *with respect to* α_n.

– $\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha_n \mathbf{p}_n$.

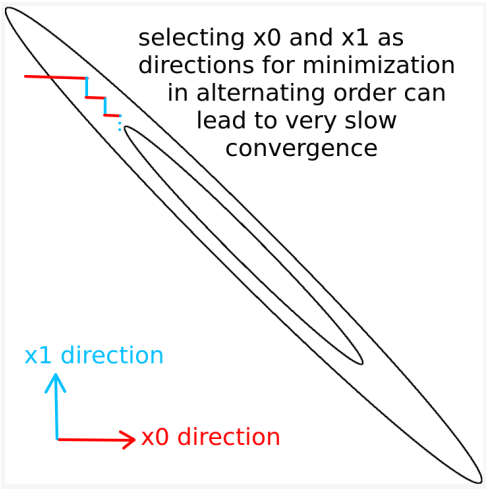
– *If* $|\mathbf{b} - A\mathbf{x}_{n+1}|$ *sufficiently small:*

→ $\mathbf{x}_{n+1}$ *is approximate solution.*
End of algorithm.

Else:

→ $n = n + 1$.
Go to (1).

- It is extremely important to chose the directions $\mathbf{p}_n$ in a clever way. Otherwise the algorithm can be very slow and impractical. A simple example for $N = 2$ is shown in the figure below (black ellipsoids are isolines of the paraboloid $f(\mathbf{x}) = 0$).



- Detailed equations:

– $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ ("residual", gradient of the paraboloid at $\mathbf{x}_0$), $\mathbf{p}_0 = \mathbf{r}_0$.

– During each iteration:

$$\alpha_n = \frac{\mathbf{r}_n \mathbf{r}_n}{\mathbf{p}_n A \mathbf{p}_n} \tag{126}$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha_n A \mathbf{p}_n \tag{127}$$

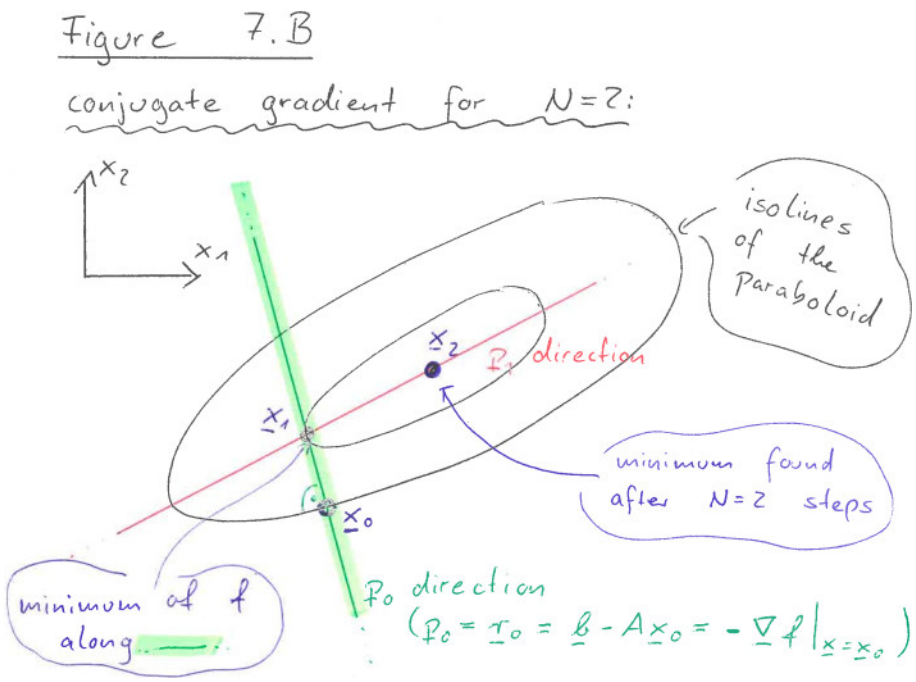
$$\beta_n = \frac{\mathbf{r}_{n+1} \mathbf{r}_{n+1}}{\mathbf{r}_n \mathbf{r}_n} \tag{128}$$

$$\mathbf{p}_{n+1} = \mathbf{r}_{n+1} + \beta_n \mathbf{p}_n \tag{129}$$

(see Ref. [1], section 2.7.6).

- One can show: after n steps $\mathbf{x}_n$ is not just minimum with respect to direction $\mathbf{p}_{n-1}$, but also minimum with respect to all previous directions $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{n-2}$.
→ Solution of $A\mathbf{x} = \mathbf{b}$ after N steps or less.

- Typically solution of $A\mathbf{x} = \mathbf{b}$ obtained after significantly less than N steps.



7.7.2 Example: static electric charge inside a grounded box in 2 dimensions

- Consider a static electric charge (charge q) in 2 dimensions centered inside a quadratic grounded box (box length $2R$). Compute the electrostatic potential $\phi(x, y)$ numerically by solving the Poisson equation

$$\triangle\phi(x,y) \quad = \quad q\delta(x,y) \tag{130}$$

with boundary conditions

$$\phi(x,y) \quad = \quad 0 \quad \text{if } |x| \geq R \text{ or } |y| \geq R. \tag{131}$$

- Discretize the linear partial differential equation (130) by introducing a uniform lattice with $(2n+1) \times (2n+1)$ lattice sites and by replacing derivatives by finite differences[4],

$$(x,y) \quad \rightarrow \quad (x_j, y_k) = \quad (j,k) \times a \quad , \quad j,k = -n, -(n-1), \ldots, +n \quad , \quad a = \frac{2R}{2n} \tag{132}$$

$$\phi(x,y) \quad \rightarrow \quad \phi_{j,k} \quad \equiv \quad \phi(x_j, y_k) \tag{133}$$

$$\triangle\phi(x,y) \quad \rightarrow \quad \frac{\phi_{j+1,k} + \phi_{j-1,k} + \phi_{j,k+1} + \phi_{j,k-1} - 4\phi_{j,k}}{a^2} \quad \equiv \quad \triangle\phi(x,y)\Big|_{(x,y)=(x_j,y_j)} \tag{134}$$

$$\delta(x,y) \quad \rightarrow \quad \frac{1}{a^2}\delta_{j,0}\delta_{k,0} \quad \equiv \quad \delta(x_j, y_j) \tag{135}$$

[4]There might be better ways to treat this partial differential equation numerically (see section 13). The main intention of this example is to demonstrate the conjugate gradient method in the context of a simple physics example.

($\delta_{j,k}$ denotes the Kronecker delta).

- Dimensionless discretized Poisson equation:

$$\triangle\phi(x,y) \;=\; q\delta(x,y) \tag{136}$$

$$\rightarrow \quad \triangle\phi(x,y)\Big|_{(x,y)=(x_j,y_j)} \;=\; q\delta(x_j,y_j) \tag{137}$$

$$\rightarrow \quad \hat{\phi}_{j+1,k} + \hat{\phi}_{j-1,k} + \hat{\phi}_{j,k+1} + \hat{\phi}_{j,k-1} - 4\hat{\phi}_{j,k} \;=\; \delta_{j,0}\delta_{k,0} \tag{138}$$

with $\hat{\phi} = \phi/q$.

- Discretized boundary conditions:

$$\hat{\phi}_{-n,k} \;=\; \hat{\phi}_{+n,k} \;=\; \hat{\phi}_{j,-n} \;=\; \hat{\phi}_{j,+n} \;=\; 0, \tag{139}$$

i.e. the potential on the $8n$ boundary lattice sites is fixed to $\hat{\phi} = 0$.

- The potential $\hat{\phi}_{j,k}$, $j,k = -(n-1), -(n-2), \ldots, +(n-1)$ on the remaining $N = (2n-1)^2$ lattice sites inside the quadratic box has to be determined by solving the linear equations (138), $\hat{\phi}_{j,k}$, $j,k = -(n-1), -(n-2), \ldots, +(n-1)$ (there are N linear equations for N unknowns).

- These linear equations can be written in the standard form

$$A_{(j,k),(j',k')}x_{(j',k')} \;=\; b_{(j,k)} \quad , \quad j,k,j',k' = -(n-1), -(n-2), \ldots, +(n-1) \tag{140}$$

by defining

$$A_{(j,k),(j',k')} \;=\; \delta_{j+1,j'}\delta_{k,k'} + \delta_{j-1,j'}\delta_{k,k'} + \delta_{j,j'}\delta_{k+1,k'} + \delta_{j,j'}\delta_{k-1,k'} - 4\delta_{j,j'}\delta_{k,k'} \tag{141}$$

$$x_{(j,k)} \;=\; \hat{\phi}_{j,k} \tag{142}$$

$$b_{(j,k)} \;=\; \delta_{j,0}\delta_{k,0}. \tag{143}$$

***** November 23, 2023 (12th lecture) *****

- In the program code it is convenient to replace indices (j,k) by superindices r via

$$r \;=\; (2n-1)\Big(k + (n-1)\Big) + \Big(j + (n-1)\Big) \quad , \quad r = 0, 1, \ldots, N-1, \tag{144}$$

which is equivalent to

$$j \;=\; (r \bmod (2n-1)) - (n-1) \quad , \quad k \;=\; \lfloor r/(2n-1) \rfloor - (n-1) \tag{145}$$

(see corresponding C code in appendix E).

- A is a sparse symmetric matrix (entries "-4" on the diagonal, "$+1$" on four off-diagonals, two below and two above the diagonal). One can show that A is negative definite (which is as good as the requirement "positive definite" [see title of section 7.7.1]; one just has to multiply all linear equations by -1, to obtain the positive definite matrix $-A$).

- In the program code the matrix A should not be stored as an array (this would unnecessarily occupy a huge amount of memory and prevent computations with a large number of lattice sites). A can conveniently be implemented as a function (see corresponding C code in appendix E).
- Comparison of several lattice sizes:
  - $n = 100 \quad \rightarrow \quad$ 420 iterations, 0.7 sec,
  - $n = 400 \quad \rightarrow \quad$ 1598 iterations, 40.5 sec,
  - $n = 1000 \quad \rightarrow \quad$ 3958 iterations, 10 min 14 sec

  (CPU time on a standard laptop, stopping criterion $|A\mathbf{x} - \mathbf{b}| < 10^{-10}$).
- Note that $n = 1000$ corresponds to a matrix A with $N \approx 4 \times 10^6$ rows and columns. Using a direct method would require around $(4 \times 10^6)^2 \times 8\,\text{byte} = 128\,000\,\text{GB}$.
- Figure 10:
  - Shows ϕ/q for $n = 100$ plotted along one of the axes (red dots) and along one of the diagonals (green dots).

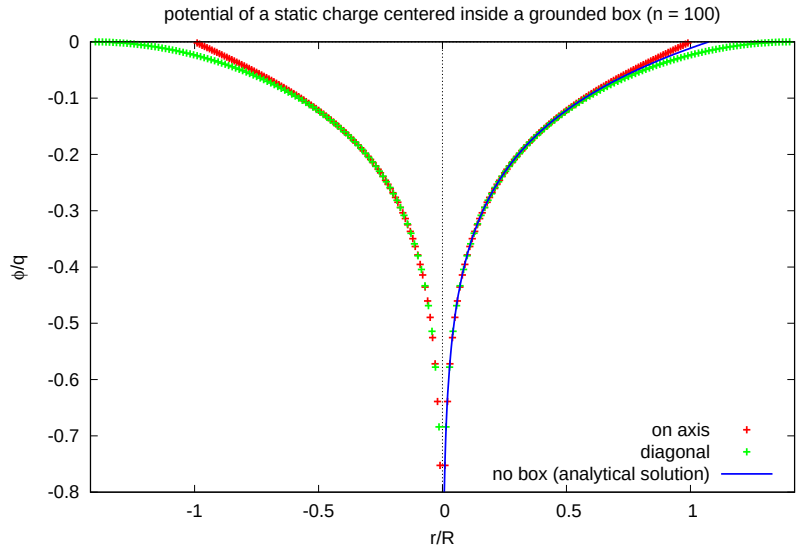


Figure 10: Numerically computed electrostatic potential ϕ for a static charge q centered inside a grounded cubic box as a function of r (red points: along the x axis; green points: along a diagonal). The blue curve represents the potential in absence of the grounded box, which can be calculated analytically.

– The blue curve represents the potential in absence of the grounded box, which can be calculated analytically:

$$\begin{aligned}
&\triangle\phi_{\text{no box}}(x,y) &&= q\delta(x,y)\\
\rightarrow\quad &\int d^2r\,\triangle\phi_{\text{no box}}(x,y) &&= \oint d\mathbf{n}\,\underbrace{\nabla\phi_{\text{no box}}(x,y)}_{=\mathbf{E}(x,y)} = 2\pi r E_r(r) = q\\
\rightarrow\quad &E_r(r) &&= \frac{q}{2\pi r}\\
\rightarrow\quad &\phi_{\text{no box}}(r) &&= \int dr\,E_r(r) = \frac{q}{2\pi}\ln(r/R). \qquad (146)
\end{aligned}$$

A constant has been added such that $\phi_{\text{no box}}$ matches the numerical on-axis result at $r/R = 0.5$.

– The grounded quadratic box breaks rotational symmetry and causes distortions compared to $\phi_{\text{no box}}$, which are rather pronounced close to the boundary.
(this is expected)

– For intermediate r/R the red and green data points and the blue curve are very similar.
(this is also expected and a valuable cross-check of the numerics)

– For very small r/R (if $r/a \gg 1$ is not anymore fulfilled), there are also strong discrepancies between the numerical solutions and $\phi_{\text{no box}}$ (hardly visible in the plot). The reason for these discrepancies are discretization errors. To study small r/R one has to work with large lattices, i.e. use large n.
(this is typical for lattice computations)

7.7.3 Generalizations

- For non-symmetric and/or non-positive definite matrices A:
  – Biconjugate gradient method.
  – Minimum residual method.
  – Generalized minimum residual method.
  – ...

7.7.4 Condition number, preconditioning

- Any $N \times N$ matrix can be decomposed according to

$$A = U\ \text{diag}(\omega_0, \omega_1, \ldots, \omega_{N-1})\ V^T, \qquad (147)$$

where U and V are orthogonal matrices and $\omega_j \geq 0$ are the singular values of A [5].

[5] For symmetric positive definite matrices A, as discussed in section 7.7.1, singular values are identical to eigenvalues and the columns of the matrix $U = V$ contain the normalized eigenvectors (mathematically this is equivalent to the well-known spectral decomposition from quantum mechanics).

- Condition number:

$$\text{cond}(A) \quad = \quad \frac{\max_j(\omega_j)}{\min_j(\omega_j)}. \tag{148}$$

- If $\text{cond}(A)$ is large, the conjugate gradient method is inefficient:
  - Many iterations necessary.
  - Numerical accuracy limited.
- Illustration for symmetric positive definite A:
  - ω_j are eigenvalues of A.
  - Semi-axes of the ellipsoids $f(\mathbf{x}) = (1/2)\mathbf{x}A\mathbf{x} - \mathbf{b}\mathbf{x} = \text{const}$ are proportional to $(\omega_j)^{-1/2}$.
  - Numerically problematic, if ellipsoids have significantly different semi-axes.
  - Numerically ideal, if ellipsoids are spheres (solution obtained after one step; see "Figure 7.B").
- Caution:
  - If A is not symmetric and positive definite, one could solve $A^T A\mathbf{x} = A^T\mathbf{b}$ instead of $A\mathbf{x} = \mathbf{b}$; then one could use the conjugate gradient method, since $A^T A$ is symmetric and positive definite.
  - Do not do that, because $\text{cond}(A^T A) = (\text{cond}(A))^2$, i.e. the conjugate gradient method applied to $A^T A$ is significantly more inefficient than other generalized methods (see section 7.7.3) applied to A.

****** November 28, 2023 (13th lecture) ******

- Quite often preconditioning is advantageous:
  - Select an $N \times N$ matrix $\tilde{A}$ with the following properties:
    - $\tilde{A} \approx A$.
    - $\tilde{A}\mathbf{x} = \mathbf{b}$ can be solved easily (e.g. analytically).
  - Compute $\tilde{A}^{-1}$, solve $\tilde{A}\mathbf{y} = \mathbf{b}$.
  - Solve $\tilde{A}^{-1}A\mathbf{x} = \tilde{A}^{-1}\mathbf{b} = \mathbf{y}$ numerically (advantage: $\text{cond}(\tilde{A}^{-1}A) \approx 1$, because $\tilde{A} \approx A$).

8 Numerical integration

8.1 Numerical integration in 1 dimension

- Goal: Compute the definite integral

$$I = \int_a^b dx\, f(x). \tag{149}$$

- Many applications in physics, e.g. normalizing wave functions in quantum mechanics, solving ODEs by separation of variables, etc.

8.1.1 Newton-Cotes formulas

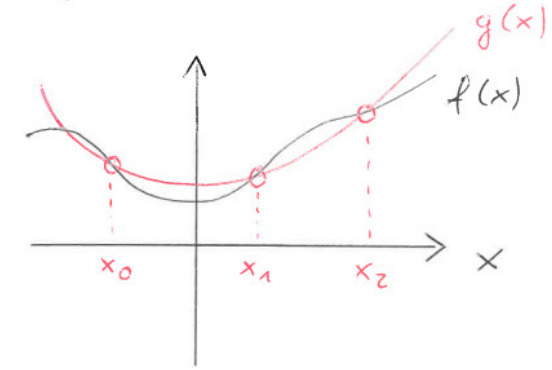
- Basic principle: Approximate $f(x)$ using a polynomial, integrate the polynomial analytically.
- In detail:
  - Approximate $f(x)$ using Lagrange polynomials:
    - Select $n+1$ points x_j, $j = 0, 1, \ldots, n$.
    - Compute/evaluate samples $f_j = f(x_j)$.
    - Langrange polynomials:

$$l_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k}, \tag{150}$$

      i.e. $l_j(x_k) = \delta_{jk}$.
    - Approximation of $f(x)$:

$$f(x) \approx g(x) = \sum_{j=0}^{n} f_j l_j(x). \tag{151}$$

Figure 8.A



– Integrate $g(x)$ instead of $f(x)$ to obtain an approximation of the integral,

$$I = \int_a^b dx\, f(x) \approx \int_a^b dx\, g(x) = \int_a^b dx \sum_{j=0}^n f_j l_j(x) =$$
$$= \sum_{j=0}^n f_j \underbrace{\int_a^b dx\, l_j(x)}_{=w_j} = \sum_{j=0}^n f_j w_j. \tag{152}$$

– Error estimates (valid for $a \le x_j \le b$):

∗ Even n:

$$\Delta I = \left| \int_a^b dx \Big(f(x) - g(x) \Big) \right| = \frac{1}{(n+2)!} \underbrace{\left(\int_a^b dx\, x \prod_{j=0}^n (x - x_j) \right)}_{\sim (b-a)^{n+3}} f^{(n+2)}(\xi) \tag{153}$$

with $a < \xi < b$ ($f^{(n+2)}$ denotes the $n+2$-th derivative, i.e. the integration of degree $n+1$ polynomials is exact).

∗ Odd n:

$$\Delta I = \left| \int_a^b dx \Big(f(x) - g(x) \Big) \right| = \frac{1}{(n+1)!} \underbrace{\left(\int_a^b dx \prod_{j=0}^n (x - x_j) \right)}_{\sim (b-a)^{n+2}} f^{(n+1)}(\xi) \tag{154}$$

with $a < \xi < b$ ($f^{(n+1)}$ denotes the $n+1$-th derivative, i.e. the integration of degree n polynomials is exact).

∗ Error estimates ΔI are only useful, if derivatives $f^{(n+2)}$ and $f^{(n+1)}$ are bounded (not the case, if f has singularities).

∗ For a derivation of these error estimates see e.g. Ref. [5].

- **Trapezoidal rule** ($n = 1$): $x_0 = a$, $x_1 = b$,

$$I = \int_a^b dx\, f(x) \approx \left(\frac{1}{2} f_0 + \frac{1}{2} f_1 \right) h \tag{155}$$

$$\Delta I = \frac{1}{2} \left(\int_a^b dx\, \frac{(x-a)(x-b)}{2} \right) f''(\xi) = -\frac{h^3}{12} f''(\xi) = \mathcal{O}(h^3) \tag{156}$$

($h = (b-a)/n$).

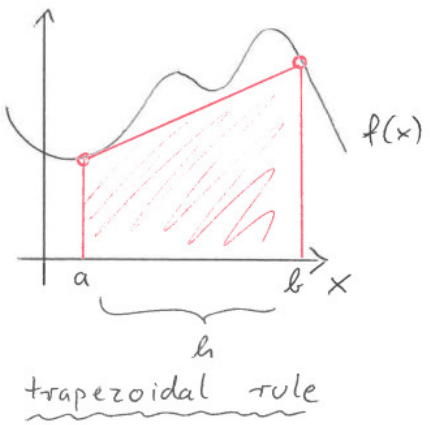
- **Simpson's rule** ($n = 2$): $x_0 = a$, $x_1 = a + h$, $x_2 = a + 2h = b$, i.e. equidistant points,

$$I = \int_a^b dx\, f(x) \approx \left(\frac{1}{3} f_0 + \frac{4}{3} f_1 + \frac{1}{3} f_2 \right) h \tag{157}$$

$$\Delta I = \ldots = -\frac{h^5}{90} f^{(4)}(\xi) = \mathcal{O}(h^5) \tag{158}$$

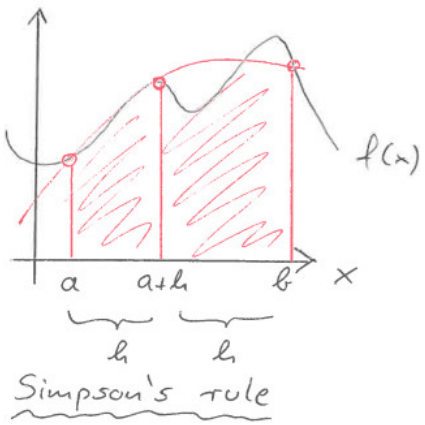
(coefficients 1/3, 4/3 and 1/3 can be obtained in a straightforward way from (152)).

Figure 8.B



trapezoidal rule

Figure 8.C



Simpson's rule

- There are further common "integration rules", e.g. Simpson's 3/8 rule or Boole's rule.
- Examples (see Ref. [5]):
  - $$I_1 = \int_0^1 dx \, \frac{1}{1+x^2} = \frac{\pi}{4}. \tag{159}$$
    - $I_1 = +0.7853\ldots$ (analytically).
    - $I_1 = +0.7500\ldots$, $\Delta I_1 = +0.0353\ldots$ (Trapezoidal rule).
    - $I_1 = +0.7833\ldots$, $\Delta I_1 = +0.0020\ldots$ (Simpson's rule).
  - $$I_2 = \int_0^1 dx \, e^x = e - 1. \tag{160}$$
    - $I_2 = +1.7182\ldots$ (analytically).
    - $I_2 = +1.8591\ldots$, $\Delta I_2 = -0.1408\ldots$ (Trapezoidal rule).
    - $I_2 = +1.7188\ldots$, $\Delta I_2 = -0.0005\ldots$ (Simpson's rule).
- **Iterated trapezoidal rule**:
  - Split the interval $[a, b]$ into N sub-intervals of the same size and apply the trapezoidal rule for each sub-interval:
    $$I = \int_a^b dx \, f(x) \approx \left(\frac{1}{2}f_0 + f_1 + f_2 + \ldots + f_{N-1} + \frac{1}{2}f_N\right)h = T_N \tag{161}$$
    $$\Delta I = N \times \mathcal{O}(h^3) = \mathcal{O}(1/N^2). \tag{162}$$
    $(h = (b-a)/N)$.
  - Iteratively increase the number of sub-intervals by a factor of 2 in each step, i.e. $N \to 2N$, until the approximation of I is sufficiently accurate (error is reduced by a factor of around 4 in each step).

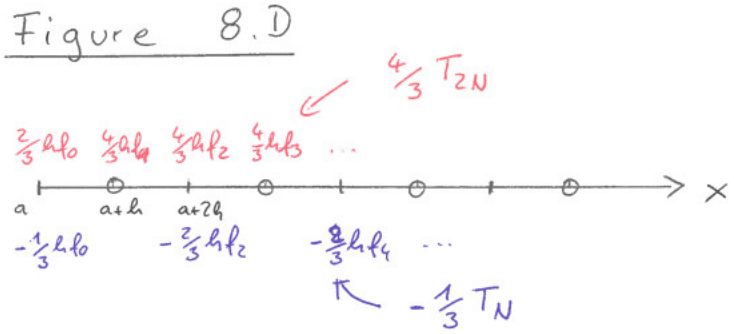
- **Iterated Simpson's rule**:
  - Approximate I according to

$$I = \int_a^b dx\, f(x) \approx \frac{4}{3}T_{2N} - \frac{1}{3}T_N. \tag{163}$$

  - Naive expectation: $\Delta I = \mathcal{O}(1/N^2)$.
  - Closer inspection shows that Δ is much smaller:

$$\frac{4}{3}T_{2N} - \frac{1}{3}T_N = \left(\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \ldots + \frac{2}{3}f_{2N-2} + \frac{4}{3}f_{2N-1} + \frac{1}{3}f_{2N}\right)h \tag{164}$$

    ($h = (b-a)/2N$), which is the iterated Simpson's rule, i.e. $\Delta I = \mathcal{O}(1/N^4)$.



- Algorithm:

  (1) Compute T_N (eq. (161)).

  (2) Compute T_{2N} (eq. (161)), reuse even samples $f_0, f_1, f_2, \ldots, f_N$ from step (1) as "even samples" $f_0, f_2, f_4, \ldots, f_{2N}$ (evaluating $f(x)$ might be expensive).

  - *Approximate I according to $(4/3)T_{2N} - (1/3)T_N$, to reduce the error from $\mathcal{O}(1/N^2)$ to $\mathcal{O}(1/N^4)$.*
  - *If the approximation of I is sufficiently accurate:*

    → End of algorithm.

    Else:

    → $N \to 2N$.
    Go to (1).

8.1.2 Gaussian integration

- Due to limited time not discussed.

8.2 Numerical integration in $D \geq 2$ dimensions

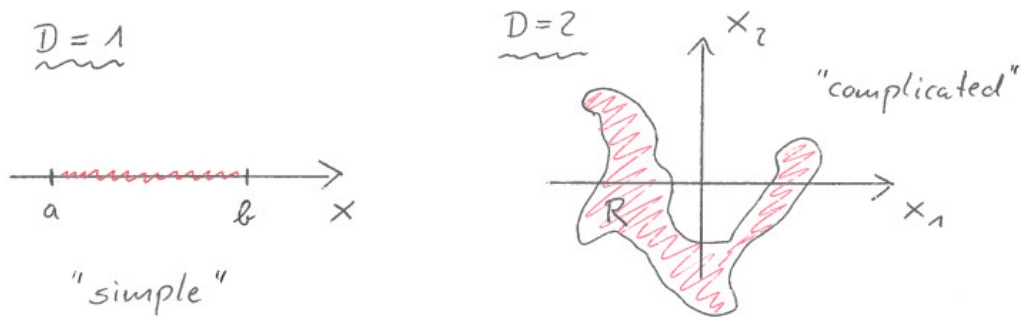
- Goal: Compute the definite integral

$$I = \int_R d^D x \, f(\mathbf{x}), \tag{165}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_D)$ and $R \subset \mathbb{R}^D$ is the domain of integration.

- More difficult than in 1 dimension, because:
  - Number of samples $f_j = f(\mathbf{x}_j)$ can be very large (N samples in 1 dimension $\to N^D$ samples in D dimensions).
  - R might be "complicated".

Figure 8.6



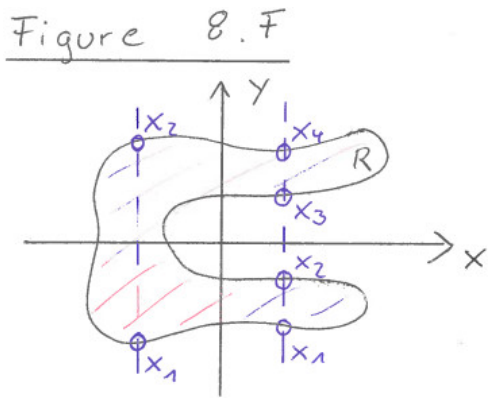
8.2.1 Nested 1-dimensional integration

- $D = 3$ in the following (generalization to arbitrary D obvious).
- Notation: $\mathbf{x} = (x, y, z)$.
- Determine x_1 (minimal x in R) and x_2 (maximal x in R).
- Determine $y_1(x)$ (minimal y in $R \cup S(x)$, where $S(x)$ is a plane parallel to the y-z plane containing x) and $y_2(x)$ (maximal y in $R \cup S(x)$).
- Determine $z_1(x, y)$ (minimal z in $R \cup S(x, y)$, where $S(x, y)$ is a straight line parallel to the z axis containing x and y) and $z_2(x, y)$ (maximal z in $R \cup S(x, y)$).
- I can be written as nested integrals in 1 dimension,

$$I = \int_R d^3 x \, f(x, y, z) = \int_{x_1}^{x_2} dx \underbrace{\int_{y_1(x)}^{y_2(x)} dy \underbrace{\int_{z_1(x,y)}^{z_2(x,y)} dz \, f(x, y, z)}_{=I(x,y)}}_{=I(x)} . \tag{166}$$

- Nested integrals might be more complicated, if R is not convex. e.g.

$$\int_{y_1(x)}^{y_2(x)} dy \ldots \quad \rightarrow \quad \int_{y_1(x)}^{y_2(x)} dy \ldots + \int_{y_3(x)}^{y_4(x)} dy \ldots \tag{167}$$



- Step 1:
  Compute samples of $I(x, y)$ (typically N^2 samples) using e.g. techniques from section 8.1.
- Step 2:
  Compute samples of $I(x)$ (typically N samples) using e.g. techniques from section 8.1.
- Step 3:
  Compute I using e.g. techniques from section 8.1.

8.2.2 Monte Carlo integration

- Statistical approximation of I using random numbers; similar to an experimental measurement the result has an error bar.
- Select N points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N \in R$ randomly and uniformly.
-

$$I \quad = \quad \int_R d^D x \, f(\mathbf{x}) \quad = \quad \underbrace{V(R)\langle f \rangle}_{=\bar{I} \approx I} \pm \underbrace{V(R) \left(\frac{\langle (f - \langle f \rangle)^2 \rangle}{N-1} \right)^{1/2}}_{=\Delta I}, \tag{168}$$

  where $V(R)$ is the volume of R, $\langle (f - \langle f \rangle)^2 \rangle = \langle f^2 \rangle - \langle f \rangle^2$ and

$$\langle f \rangle \quad = \quad \frac{1}{N} \sum_{j=1}^{N} f(\mathbf{x}_j) \tag{169}$$

$$\langle f^2 \rangle \quad = \quad \frac{1}{N} \sum_{j=1}^{N} \Big(f(\mathbf{x}_j) \Big)^2. \tag{170}$$

- Error ΔI: probability for $I \in [\overline{I} - \Delta I, \overline{I} + \Delta I]$ is $\approx 68\%$.
- Major disadvantage: slow convergence, i.e. $\Delta I \propto 1/\sqrt{N}$ ("to reduce the error by a factor of around 2 you need 4 times as many samples").
- Advantages:
  - Very large D possible, e.g. $D = 10^3$ or $D = 10^6$ (quite efficient, if $f(\mathbf{x})$ is "smooth"; very inefficient, if $f(\mathbf{x})$ has strongly localized peaks).

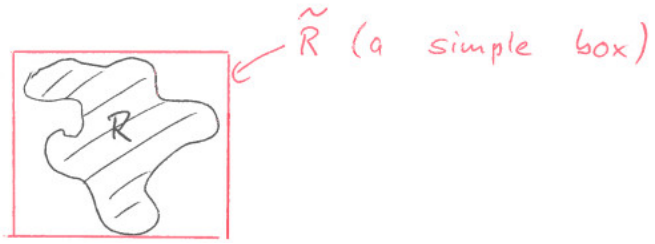
****** December 05, 2023 (15th lecture) ******

  - "Complicated domains" R possible, if R can be defined by a function

$$g(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in R \\ 0 & \text{otherwise} \end{cases} . \tag{171}$$

    * Define "simple domain" $\tilde{R} \supset R$, e.g. a D-dimensional box.

Figure 8.6



    *

$$I = \int_{\tilde{R}} d^D x \, f(\mathbf{x}) g(\mathbf{x}). \tag{172}$$

    * Right hand side of (172) can be evaluated in a straightforward way using Monte Carlo integration.

8.2.3 When to use which method?

- For very precise computations (many digits of accuracy)
  $\rightarrow$ nested 1-dimensional integration
  (convergence of Monte Carlo integration too slow, $\Delta I \propto 1/\sqrt{N}$).
- Complicated domain easy for Monte Carlo integration, more difficult for nested 1-dimensional integration.
- Nested 1-dimensional integration requires smooth integrands, otherwise error estimates useless.
- Monte Carlo integration requires integrands, which are not strongly peaked, otherwise huge statistical errors.

- Complicated domain, integrand, which is not strongly peaked, limited accuracy o.k.
  → Monte Carlo integration.
- Simple domain, integrand smooth
  → nested 1-dimensional integration.
- Strongly oscillating or discontinuous integrand
  → Monte Carlo integration.

9 Eigenvalues and eigenvectors

9.1 Problem definition, general remarks

- **Eigenvalue problem**: Find eigenvalues λ_j and eigenvectors $\mathbf{v}_j \neq 0$, $j = 1, \ldots, N$ fulfilling

$$A\mathbf{v}_j = \lambda_j \mathbf{v}_j \tag{173}$$

(A: $N \times N$ matrix).

- Eigenvalues are roots of the characteristic polynomial $\det(A - \lambda_j)$, i.e. solutions of

$$\det\Big(A - \lambda_j\Big) = 0. \tag{174}$$

- The characteristic polynomial is a degree-N polynomial, i.e.
  - N roots (= eigenvalues) λ_j (might be complex, not necessarily different),
  - N eigenvectors $\mathbf{v}_j$ (might be complex, not necessarily linearly independent).
- Properties of eigenvalues and eigenvectors for specific classes of matrices:
  - A **real**, **symmetric** ($A^T = A$):
    λ_j real, $\mathbf{v}_j$ can be chosen real.
  - A **complex**, **hermitian** ($A^\dagger = A$):
    λ_j real.
  - A **not symmetric/not hermitian**:
    λ_j and $\mathbf{v}_j$ typically complex.
  - A **normal** ($AA^\dagger = A^\dagger A$):
    - λ_j pairwise distinct:
      $\mathbf{v}_j$ orthogonal.
    - λ_j degenerate:
      can be chosen orthogonal (e.g. using Gram-Schmidt orthogonalization).
- **Generalized eigenvalue problem**: find eigenvalues λ_j and eigenvectors $\mathbf{v}_j \neq 0$, $j = 1, \ldots, N$ fulfilling

$$A\mathbf{v}_j = \lambda_j B\mathbf{v}_j \tag{175}$$

(A, B: $N \times N$ matrices).

  - Can be rewritten as a standard eigenvalue problem:

$$B^{-1}A\mathbf{v}_j = \lambda_j \mathbf{v}_j. \tag{176}$$

  - If A symmetric and B symmetric and positive definite, use the following method:
    - Choelesky decomposition ("square root of a matrix"; see e.g. Ref. [1]): $B = LL^T$, where L is a lower triangular matrix.

* Then

$$A\mathbf{v}_j = \lambda_j L L^T \mathbf{v}_j \tag{177}$$

$$\underbrace{L^{-1}A(L^T)^{-1}}_{=A'}\underbrace{L^T\mathbf{v}_j}_{=\mathbf{v}'_j} = \lambda_j \underbrace{L^T\mathbf{v}_j}_{=\mathbf{v}'_j}, \tag{178}$$

which is a standard eigenvalue problem with a symmetric matrix A' ($(L^{-1})^T = (L^T)^{-1}$ can be shown in a straightforward way). Note that $B^{-1}A$ in (176) is typically not symmetric.

* Computing L^{-1} and solving $L^T\mathbf{v}_j = \mathbf{v}'_j$ simple, because L is a lower triangular matrix (see e.g. section 7.3 and section 7.4.2).

9.2 Basic principle of numerical methods for eigenvalue problems

- Iterative procedure: apply similarity transformations

$$\begin{aligned} A &\rightarrow \\ &\rightarrow (P_1)^{-1}AP_1 \rightarrow \\ &\rightarrow (P_2)^{-1}(P_1)^{-1}AP_1P_2 \rightarrow \\ &\dots \\ &\rightarrow \underbrace{(P_n)^{-1}\dots(P_2)^{-1}(P_1)^{-1}}_{=Q^{-1}} A \underbrace{P_1P_2\dots P_n}_{=Q} = Q^{-1}AQ \end{aligned} \tag{179}$$

such that $Q^{-1}AQ$ is diagonal.

- In practice: stop iteration, as soon as $Q^{-1}AQ$ is "almost a diagonal matrix" (e.g. absolute values of off-diagonal elements $< \epsilon = 10^{-6}$).
- Matrix $Q^{-1}AQ = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_N)$:
  - Eigenvalues λ_j.
  - Eigenvectors $\mathbf{e}_j$.

****** December 07, 2023 (16th lecture) ******

- Matrix A:
  - Eigenvalues λ_j, because

$$\begin{aligned} \det(A-\lambda_j) &= \det(Q^{-1})\det\Big(A-\lambda_j\Big)\det(Q) = \det\Big(Q^{-1}\Big(A-\lambda_j\Big)Q\Big) = \\ &= \det\Big(Q^{-1}AQ-\lambda_j\Big), \end{aligned} \tag{180}$$

    i.e. A and $Q^{-1}AQ$ have the same characteristic polynomial and, consequently, the same eigenvalues λ_j.
  - Eigenvectors $Q\mathbf{e}_j$, i.e. the columns of Q are the eigenvectors, because

$$Q^{-1}AQ\mathbf{e}_j = \lambda_j\mathbf{e}_j \tag{181}$$

$$\rightarrow \quad A(Q\mathbf{e}_j) = \lambda_j(Q\mathbf{e}_j). \tag{182}$$

- Summary:

  (1) Iteratively apply similarity transformations, until $Q^{-1}AQ$ is diagonal.

  (2) Eigenvalues are the diagonal elements of $Q^{-1}AQ$.

  (3) If eigenvectors are needed, $Q = P_1 P_2 \dots P_n$ has to be computed; eigenvectors are columns of Q.

9.3 Jacobi method

- A must be real and symmetric:

  $\rightarrow$ A is normal.

  $\rightarrow$ λ_j real, $\mathbf{v}_j$ can be chosen real and orthogonal.

  $\rightarrow$ Eigenvectors form an orthogonal matrix,

$$Q = \left(\mathbf{v}_1\ \mathbf{v}_2\ \dots\ \mathbf{v}_n\right) \quad \rightarrow \quad Q^{-1} = Q^T = \begin{pmatrix} (\mathbf{v}_1)^T \\ (\mathbf{v}_2)^T \\ \dots \\ (\mathbf{v}_n)^T \end{pmatrix}, \tag{183}$$

  which diagonalizes A, i.e.

$$Q^T A Q = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_N). \tag{184}$$

- Advantages and disadvantages:

  (+) Simple.

  (−) Somewhat slower than other methods, e.g. the QR method (see e.g. Ref. [1]).

- P_j: rotation in p-q plane,

$$A \rightarrow A' = (P_j)^T A P_j, \tag{185}$$

  such that $A'_{p,q} = A'_{q,p} = 0$.

  –

$$P_j = \begin{pmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & +c & & & +s & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & -s & & & +c & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{pmatrix}, \tag{186}$$

  where $c = \cos(\varphi)$ and $s = \sin(\varphi)$.

  –

$$A'_{kl} = \underbrace{((P_j)^T)_{k,m}}_{=(P_j)_{m,k}} A_{m,n} (P_j)_{n,l}. \tag{187}$$

* $A'_{p,p} = (P_j)_{m,p} A_{m,n} (P_j)_{n,p} = c^2 A_{p,p} + s^2 A_{q,q} - 2csA_{p,q}$.
* $A'_{q,q} = c^2 A_{q,q} + s^2 A_{p,p} + 2csA_{p,q}$.
* $A'_{p,q} = A'_{q,p} = (c^2 - s^2)A_{p,q} + sc(A_{p,p} - A_{q,q})$.
* $k \neq p, q$: $A'_{k,p} = A'_{p,k} = cA_{k,p} - sA_{k,q}$.
* $k \neq p, q$: $A'_{k,q} = A'_{q,k} = cA_{k,q} + sA_{k,p}$.
* $k, l \neq p, q$: $A'_{k,l} = A_{k,l}$.

– Choose φ such that $A'_{p,q} = A'_{q,p} = 0$, i.e.

$$\frac{c^2 - s^2}{2sc} = \frac{A_{q,q} - A_{p,p}}{2A_{p,q}} \tag{188}$$

and after defining $\theta = (A_{q,q} - A_{p,p})/2A_{p,q}$ and using $(c^2 - s^2)/2sc = (1/t - t)/2$

$$t^2 + 2\theta t - 1 = 0$$
$$\rightarrow \quad t = -\theta \pm \left(\theta^2 + 1\right)^{1/2}, \tag{189}$$

where $t = \tan(\varphi)$; numerical tests have shown that it is advantageous to choose the smaller $|t|$, i.e.

$$t = \frac{\mathrm{sign}(\theta)}{|\theta| + (\theta^2 + 1)^{1/2}}, \tag{190}$$

implying $|\varphi| \leq \pi/4$.

- Using (188) the above equations can be implemented in the following equivalent, more convenient form:
  – $A'_{p,p} = A_{p,p} - tA_{p,q}$.
  – $A'_{q,q} = A_{q,q} + tA_{p,q}$.
  – $A'_{p,q} = A'_{q,p} = 0$.
  – $k \neq p, q$: $A'_{k,p} = A'_{p,k} = A_{k,p} - s(A_{k,q} + \tau A_{k,p})$, where $\tau = \tan(\varphi/2) = s/(1 + c)$.
  – $k \neq p, q$: $A'_{k,q} = A'_{q,k} = A_{k,q} + s(A_{k,p} - \tau A_{k,q})$.
  – $k, l \neq p, q$: $A'_{k,l} = A_{k,l}$.
- Convergence of the Jacobi method:
  – Applying the Jacobi rotation (185) results in $A'_{p,q} = A'_{q,p} = 0$, but other off-diagonal elements might become larger.
  – Is convergence guaranteed?
  – Define "deviation from diagonal matrix": $S = \sum_{k \neq l} (A_{k,l})^2$.
  – One can show: $S' = S - 2(A_{p,q})^2$, i.e. S will approach 0, if "large off-diagonal elements $A_{p,q}$ are rotated to 0".
- How to choose p and q?
  – Jacobi 1846: "rotate the largest off-diagonal elements $|A_{p,q}| = |A_{q,p}|$ to 0".
  – Jacobi's strategy is numerically too expensive (finding the largest off-diagonal elements is $\mathcal{O}(N^2)$, while a Jacobi rotation is only $\mathcal{O}(N)$).

– Nowadays: cyclic Jacobi method, pick off-diagonal elements in a fixed order, e.g. $A_{0,1}$, $A_{0,2}$, ..., $A_{0,N-1}$, $A_{1,2}$, $A_{1,3}$, ..., $A_{1,N-1}$, $A_{2,3}$, ...

- Eigenvectors, if needed, are columns of $Q = P_1 P_2 \dots P_n$:
  - Initialize $Q = 1$.
  - After each Jacobi rotation (185): $Q \to Q' = QP_j$:
    * $Q'_{k,p} = cQ_{k,p} - sQ_{k,q}$.
    * $Q'_{k,q} = cQ_{k,q} + sQ_{k,p}$.
    * $l \neq p, q$: $Q'_{k,l} = Q_{k,l}$.

9.4 Example: molecule oscillations inside a crystal

- N point masses, nearest neighbors coupled by springs (a simple model to study molecule oscillations inside a 1-dimensional crystal):

$$L = \sum_{j=1}^{N} \frac{1}{2} m \dot{x}_j^2 - \sum_{j=1}^{N-1} \frac{1}{2} k (x_j - x_{j+1})^2. \tag{191}$$

Figure 9.A

- Since the Langrangian is quadratic in $\dot{x}_j$ and x_j, the EOMs are linear,

$$m\ddot{x}_1 = +k(x_2 - x_1) \tag{192}$$
$$m\ddot{x}_2 = +k(x_3 - x_2) + k(x_1 - x_2) \tag{193}$$
$$\dots, \tag{194}$$

i.e. can be written in matrix form,

$$M\ddot{\mathbf{x}} = -K\mathbf{x} \tag{195}$$

$$M = \begin{pmatrix} m & & & \\ & m & & \\ & & m & \\ & & & \dots \end{pmatrix} \quad \text{(mass matrix)} \tag{196}$$

$$K = \begin{pmatrix} +k & -k & & \\ -k & +2k & -k & \\ & -k & +2k & \dots \\ & & \dots & \dots \end{pmatrix} \quad \text{(stiffness matrix)}, \tag{197}$$

where $\mathbf{x} = (x_1, x_2, \dots, x_N)$.

- EOMs reformulated using dimensionless quantities:

$$\frac{d^2}{d\hat{t}^2}\hat{\mathbf{x}} = -\hat{K}\hat{\mathbf{x}} \tag{198}$$

$$\hat{K} = \begin{pmatrix} +1 & -1 & & \\ -1 & +2 & -1 & \\ & -1 & +2 & \dots \\ & & \dots & \dots \end{pmatrix}, \tag{199}$$

where $\hat{t} = \sqrt{k/m}t$, $\hat{\mathbf{x}} = (x_1, x_2, \dots, x_N)/L$ and L is a length scale, e.g. from the initial conditions.

- The ansatz

$$\hat{\mathbf{x}} = \mathbf{v}_j e^{i\hat{\omega}_j \hat{t}} \tag{200}$$

reduces the system of second order ODEs (198) to an eigenvalue problem,

$$-\hat{\omega}_j^2 \mathbf{v}_j = -\hat{K}\mathbf{v}_j. \tag{201}$$

****** December 12, 2023 (17th lecture) ******

- Since $\hat{K}$ is real and symmetric, the Jacobi method can be used to solve the eigenvalue problem.
- Computation for $N = 10$:
  - C code to compute eigenvalues and eigenvectors of $\hat{K}$: see appendix F.

```
N = 10
------

initial matrix

+1.00   -1.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00
-1.00   +2.00   -1.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00
+0.00   -1.00   +2.00   -1.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00
+0.00   +0.00   -1.00   +2.00   -1.00   +0.00   +0.00   +0.00   +0.00   +0.00
+0.00   +0.00   +0.00   -1.00   +2.00   -1.00   +0.00   +0.00   +0.00   +0.00
+0.00   +0.00   +0.00   +0.00   -1.00   +2.00   -1.00   +0.00   +0.00   +0.00
+0.00   +0.00   +0.00   +0.00   +0.00   -1.00   +2.00   -1.00   +0.00   +0.00
+0.00   +0.00   +0.00   +0.00   +0.00   +0.00   -1.00   +2.00   -1.00   +0.00
+0.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00   -1.00   +2.00   -1.00
+0.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00   +0.00   -1.00   +1.00

S = 1.80000e+01.

**********

sweep    1 ...
```

```
+0.16   -0.11   -0.22   +0.09   +0.11   -0.05   -0.04   +0.03   +0.01   +0.02
-0.11   +3.63   -0.11   +0.33   +0.09   -0.13   -0.06   +0.04   +0.04   +0.02
-0.22   -0.11   +0.48   -0.08   -0.36   +0.14   +0.15   +0.02   -0.12   +0.01
+0.09   +0.33   -0.08   +3.40   -0.08   +0.40   +0.07   -0.14   -0.03   -0.07
+0.11   +0.09   -0.36   -0.08   +0.63   +0.01   -0.38   +0.31   -0.14   +0.23
-0.05   -0.13   +0.14   +0.40   +0.01   +3.23   -0.05   +0.34   +0.20   +0.03
-0.04   -0.06   +0.15   +0.07   -0.38   -0.05   +0.72   +0.17   -0.50   -0.27
+0.03   +0.04   +0.02   -0.14   +0.31   +0.34   +0.17   +2.86   -0.09   -0.04
+0.01   +0.04   -0.12   -0.03   -0.14   +0.20   -0.50   -0.09   +1.89   +0.00
+0.02   +0.02   +0.01   -0.07   +0.23   +0.03   -0.27   -0.04   +0.00   +0.99

S = 2.91374e+00.

**********

sweep    2 ...

+0.03   -0.04   -0.05   +0.01   -0.00   +0.01   +0.01   -0.02   +0.01   -0.01
-0.04   +3.89   -0.03   +0.03   +0.02   +0.05   +0.01   -0.04   +0.02   -0.02
-0.05   -0.03   +0.13   -0.03   -0.06   -0.04   -0.10   +0.01   -0.01   +0.08
+0.01   +0.03   -0.03   +2.58   -0.01   +0.05   +0.26   -0.01   -0.05   -0.09
-0.00   +0.02   -0.06   -0.01   +1.39   -0.08   -0.01   +0.04   +0.05   +0.00
+0.01   +0.05   -0.04   +0.05   -0.08   +3.62   +0.02   +0.00   -0.00   -0.01
+0.01   +0.01   -0.10   +0.26   -0.01   +0.02   +0.37   +0.01   +0.03   -0.00
-0.02   -0.04   +0.01   -0.01   +0.04   +0.00   +0.01   +3.18   +0.00   -0.00
+0.01   +0.02   -0.01   -0.05   +0.05   -0.00   +0.03   +0.00   +2.00   +0.00
-0.01   -0.02   +0.08   -0.09   +0.00   -0.01   -0.00   -0.00   +0.00   +0.82

S = 2.53839e-01.

**********

sweep    3 ...

...

S = 2.12206e-02.

**********

sweep    4 ...

...

S = 7.26279e-06.

**********

sweep    5 ...

...

S = 2.26242e-10.

**********
```

```
sweep    6 ...

...

S = 1.12777e-32.

**********

lambda_00 =  +0.000000.
v_00 = ( +0.32 , +0.32 , +0.32 , +0.32 , +0.32 , +0.32 , +0.32 , +0.32 , +0.32 , +0.32 ).

lambda_01 =  +3.902113.
v_01 = ( -0.07 , +0.20 , -0.32 , +0.40 , -0.44 , +0.44 , -0.40 , +0.32 , -0.20 , +0.07 ).

lambda_02 =  +0.097887.
v_02 = ( -0.44 , -0.40 , -0.32 , -0.20 , -0.07 , +0.07 , +0.20 , +0.32 , +0.40 , +0.44 ).

lambda_03 =  +2.618034.
v_03 = ( +0.26 , -0.43 , -0.00 , +0.43 , -0.26 , -0.26 , +0.43 , +0.00 , -0.43 , +0.26 ).

lambda_04 =  +1.381966.
v_04 = ( +0.36 , -0.14 , -0.45 , -0.14 , +0.36 , +0.36 , -0.14 , -0.45 , -0.14 , +0.36 ).

lambda_05 =  +3.618034.
v_05 = ( +0.14 , -0.36 , +0.45 , -0.36 , +0.14 , +0.14 , -0.36 , +0.45 , -0.36 , +0.14 ).

lambda_06 =  +0.381966.
v_06 = ( +0.43 , +0.26 , +0.00 , -0.26 , -0.43 , -0.43 , -0.26 , -0.00 , +0.26 , +0.43 ).

lambda_07 =  +3.175571.
v_07 = ( -0.20 , +0.44 , -0.32 , -0.07 , +0.40 , -0.40 , +0.07 , +0.32 , -0.44 , +0.20 ).

lambda_08 =  +2.000000.
v_08 = ( +0.32 , -0.32 , -0.32 , +0.32 , +0.32 , -0.32 , -0.32 , +0.32 , +0.32 , -0.32 ).

lambda_09 =  +0.824429.
v_09 = ( -0.40 , -0.07 , +0.32 , +0.44 , +0.20 , -0.20 , -0.44 , -0.32 , +0.07 , +0.40 ).
```

– General solution:

$$\hat{\mathbf{x}} \;=\; \sum_{j=1}^{N} \underbrace{\mathbf{v}_{j-1}\Big(A_j \cos(\hat{\omega}_j \hat{t}) + B_j \sin(\hat{\omega}_j \hat{t})\Big)}_{\text{normal modes}}, \tag{202}$$

where $\hat{\omega}_j^2 = \lambda_{j-1}$.

– Solve EOMs for initial conditions: $x_1(t=0) = L$, $x_j(t=0) = 0$ for $j = 2, \ldots, N$, $\dot{x}(t=0) = 0$ for $j = 1, \ldots, N$.

*

$$\dot{\hat{\mathbf{x}}}(\hat{t}=0) \;=\; \sum_{j=1}^{N} \mathbf{v}_{j-1} B_j \hat{\omega}_j \;=\; 0 \quad\rightarrow\quad B_j \;=\; 0, \tag{203}$$

because eigenvectors $\mathbf{v}_j$ are orthogonal and, thus, linearly independent.

∗

$$\hat{\mathbf{x}}(\hat{t}=0) \quad = \quad \sum_{j=1}^{N} \mathbf{v}_{j-1} A_j \quad = \quad (1,0,\ldots,0) \quad \rightarrow \quad A_j \quad = \quad v_{j-1,1} \tag{204}$$

(first index of $v_{j-1,1}$ is eigenvector index, second index is component index), where $\mathbf{v}_j \mathbf{v}_k = \delta_{j,k}$ has been used.

∗ Solution:

$$\hat{\mathbf{x}} \quad = \quad \sum_{j=1}^{N} \mathbf{v}_{j-1} v_{j-1,1} \cos(\hat{\omega}_j \hat{t}) \tag{205}$$

(see Figure 11, from which e.g. the speed of sound can be read off).

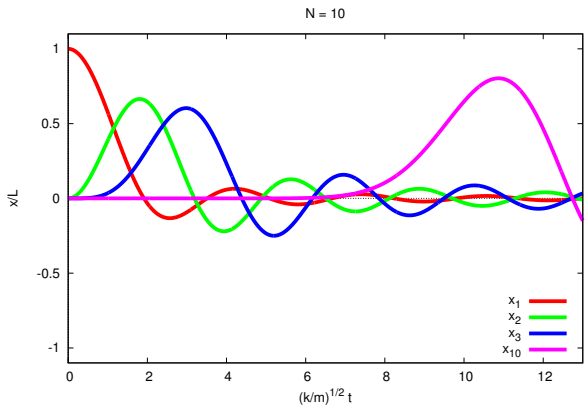


Figure 11: "Molecule oscillations in a 1-dimensional crystal" ($N = 10$ molecules).

- Can be generalized in a straightforward way to study small oscillations of any system of N point masses (after first order Taylor expansion, EOMs are of the form $M\ddot{\mathbf{x}} = -K\mathbf{x}$).

10 Interpolation, extrapolation, approximation

- Problem definition:
  - Starting point: $f_j = f(x_j)$, $j = 0, \ldots, N$ ("data points") for $x_0 < x_1 < \ldots < x_N$, where $f(x)$ is not known.
  - Goals:
    - Determine $g(x) \approx f(x)$ approximately for $x_{\min} \leq x \leq x_{\max}$.
    - Determine $g(y) \approx f(y)$ for fixed $y \neq x_0, x_1, \ldots, x_N$.
  - $x_0 \leq x_{\min} \leq x_{\max} \leq x_N$ or $x_0 < y < x_N$
    $\rightarrow$ **interpolation**,
    otherwise
    $\rightarrow$ **extrapolation**.
  - $g(x_j) = f_j = f(x_j)$, $j = 0, \ldots, N$
    $\rightarrow$ **interpolation**
  - otherwise (i.e. $g(x_j) \approx f_j = f(x_j)$)
    $\rightarrow$ **approximation**.
- Basic principle: approximate $f(x)$ using a specific ansatz for $g(x)$, e.g. simple (typically polynomials) or physically motivated.
- Physics motivation:
  - f_j: experimental measurements (e.g. $f_j \equiv V(r)$ [a potential] or $f_j \equiv (d\sigma/d\Omega)(\Omega)$ [a differential cross section], ...).
  - f_j: are results from a time consuming numerical computation or simulation.

  $\rightarrow$ Approximation $g(x) \approx f(x)$ often needed, e.g. for a subsequent analytical calculation.
  - For example [3]:
    The potential of two heavy $\bar{b}$ quarks in the presence of two light u and/or d quarks can be computed with lattice QCD (a numerical method to solve QCD) for discrete $\bar{b}\bar{b}$ separations $r = na$ ($n = 1, 2, \ldots$; a: lattice spacing). To use this potential in a standard non-relativistic Schrödinger equation (see also section 6.2.3), the lattice data points need to be parameterized by a continuous function. A physically motivated ansatz is

    $$V(r) \quad = \quad -\frac{\alpha}{r} \exp\left(-\left(\frac{r}{d}\right)^2 \right) \tag{206}$$

    ($1/r$, because of 1-gluon-exchange at small r [leading order perturbation theory; see lectures on quantum field theory]; $\exp(-r^2/d^2)$, because of color screening at large r). One has to fit this ansatz to the lattice data points, i.e. one has to determine the optimal values for α and d.

10.1 Polynomial interpolation

- Find a degree-N polynomial $g(x)$, which interpolates f_j, $j = 0, \ldots, N$, i.e. $g(x_j) = f_j$.

- Unique solution (easy to show).
- $g(x)$ can be obtained e.g. using Lagrange polynomials (see section 8.1.1):

$$l_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k} \tag{207}$$

$$g(x) = \sum_{j=0}^{n} f_j l_j(x). \tag{208}$$

- Polynomial interpolation for $N \gtrsim 4$ not recommended:
  - For large N polynomials exhibit strong oscillations.
  - Even though $g(x_j) = f_j$, $g(x)$ and $f(x)$ are most likely quite different.
  - Examples for $N = 3$ and $N = 9$ are shown in Figure 12.

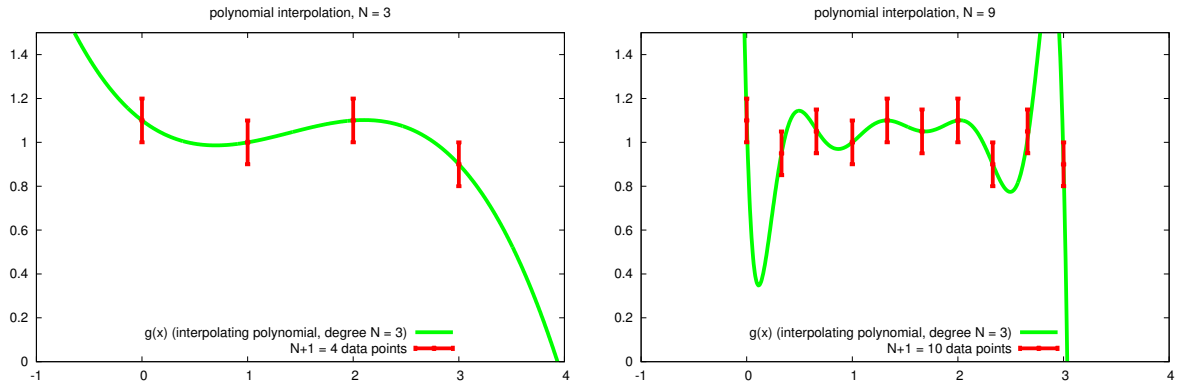


Figure 12: Polynomial interpolation of $N + 1$ data points f_j for $N = 3$ and $N = 9$. While the data points are consistent with a constant, i.e. $f(x) = \text{const}$, the interpolating degree-N polynomials $g(x)$ are oscillating, in particular for the larger $N = 9$.

****** December 14, 2023 (18th lecture) ******

10.2 Cubic spline interpolation

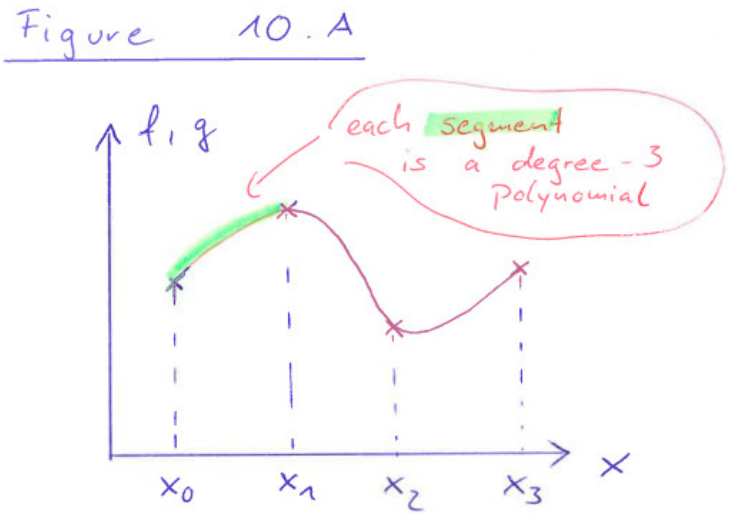
- Connect N degree-3 polynomials $y_j(x)$, $j = 0, \ldots, N - 1$,

$$g(x) = y_j(x) \quad \text{for } x_j \leq x \leq x_{j+1}, \tag{209}$$

  such that
  - $y_j(x_j) = f_j$ and $y_j(x_{j+1}) = f_{j+1}$ (two polynomials $y_j(x)$ and $y_{j+1}(x)$ are connected at data point (x_{j+1}, f_{j+1})),

– $y_j'(x_{j+1}) = y_{j+1}'(x_{j+1})$ and $y_j''(x_{j+1}) = y_{j+1}''(x_{j+1})$ (the piecewise defined function $g(x)$ is C^2 continuous).

Such a piecewise defined polynomial is called "cubic spline".



- Advantage (compared to polynomial interpolation discussed in section 10.1): only degree-3 polynomials, i.e. polynomial degree is small, even though the number of data points $(N+1)$ might be large; thus, no unnecessary oscillations.
- Construction of a cubic spline:
  - To interpolate data points f_j, $j = 0, \ldots, N$, degree-1 polynomials are sufficient:

$$y_j(x) = f_j A(x) + f_{j+1} B(x), \tag{210}$$

    where

$$A(x) = \frac{x_{j+1} - x}{x_{j+1} - x_j} \tag{211}$$

$$B(x) = \frac{x - x_j}{x_{j+1} - x_j} \tag{212}$$

    are the Lagrange polynomials (207) for $N = 1$.
  - If the second derivatives $f_j'' = f''(x_j)$, $j = 0, \ldots, N$ are given (in addition to the data points f_j),

$$y_j(x) = f_j A(x) + f_{j+1} B(x) + f_j'' C(x) + f_{j+1}'' D(x), \tag{213}$$

    where

$$C(x) = \frac{1}{6}\Big(A(x)^3 - A(x)\Big)(x_{j+1} - x_j)^2 \tag{214}$$

$$D(x) = \frac{1}{6}\Big(B(x)^3 - B(x)\Big)(x_{j+1} - x_j)^2. \tag{215}$$

  - Determine f_j'', $j = 0, \ldots, N$ such that the resulting spline $g(x)$ is C^2 continuous:
    * Impose $y_{j-1}'(x_j) = y_j'(x_j)$, $j = 1, \ldots, N-1$.

* Insert (213):

$$\frac{x_j - x_{j-1}}{6} f''_{j-1} + \frac{x_{j+1} - x_{j-1}}{3} f''_j + \frac{x_{j+1} - x_j}{6} f''_{j+1} = \\ = \frac{f_{j+1} - f_j}{x_{j+1} - x_j} - \frac{f_j - f_{j-1}}{x_j - x_{j-1}}. \tag{216}$$

* To determine f''_j, $j = 1, \ldots, N-1$, one has to solve this system of $N-1$ linear equations (use one of the methods discussed in section 7).
* f''_0 and f''_N can be set to arbitrary values (a common choice is $f''_0 = f''_N = 0$, the so-called "natural spline").

- Figure 13 shows a cubic spline interpolating the data points already used in Figure 12, right ($N = 9$ example). In contrast to the degree-9 polynomial from Figure 12, the cubic spline does not exhibit any unnecessary oscillations.

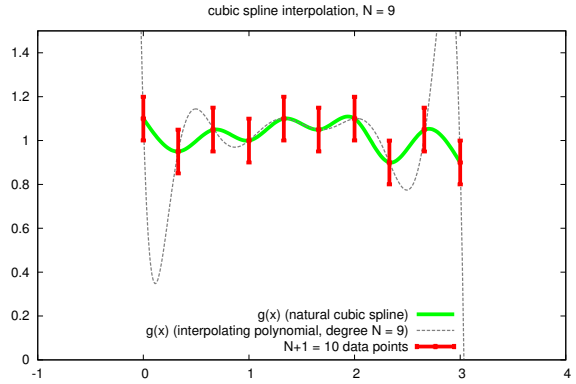


Figure 13: Cubic spline interpolation of $N + 1 = 10$ data points f_j.

- Splines and related topics form a huge field of research (CAGD = Computer Aided Geometric Design):
  - Goal: Describe and parameterize curves and surfaces in a mathematical way.
  - Useful e.g. in engineering [ships, cars, etc.], scientific or medical visualization, animated movies, computer games, ...

10.3 Method of least squares

- Data points f_j often exhibit statistical fluctuations (e.g. f_j can be experimental measurements, results of Monte Carlo integrations or simulations, ...).
- $g(x)$ should not reflect these statistical fluctuations, i.e. in such cases approximation more suited than interpolation.
- Select an ansatz $g(x; \mathbf{a})$ ($\mathbf{a} = (a_0, \ldots, a_M)$ are parameters, which will be determined such that $g(x; \mathbf{a})$ approximates the data points in an optimal way).

– E.g. a low degree polynomial, $g(x;\mathbf{a}) = a_0 + a_1 x + a_2 x^2$ ...

– ... or $g(x;\mathbf{a}) = a_0/x$ (if f_j describe a Coulomb-like potential) ...

– ... or $g(x;\mathbf{a}) = (a_0/x)\exp(-a_1 x)$ (if f_j describe a potential with a limited range) ...

– ...

- Determine $\mathbf{a}$ by minimizing

$$G(\mathbf{a}) \;=\; \sum_{j=0}^{N} \Big(g(x_j;\mathbf{a}) - f_j \Big)^2 \tag{217}$$

with respect to $\mathbf{a}$.

– $(g(x_j;\mathbf{a}) - f_j)^2$: squared difference of approximating function and data points ($\rightarrow$ "method of least squares").

– Minimization equivalent to solving

$$\nabla^{(\mathbf{a})} G(\mathbf{a}) \;=\; 0. \tag{218}$$

- $g(x;\mathbf{a})$ linear in a_j,

$$g(x;\mathbf{a}) \;=\; \sum_{j=0}^{M} a_j g_j(x) \tag{219}$$

(e.g. $g_j(x) = x^j$, if $g(x;\mathbf{a})$ is a degree-M polynomial):

– Insert (219) in (217):

$$\begin{aligned} G(\mathbf{a}) \;&=\; \sum_{j=0}^{N} \Big(\sum_{k=0}^{M} a_k g_k(x_j) - f_j \Big) \Big(\sum_{l=0}^{M} a_l g_l(x_j) - f_j \Big) \;= \\ &=\; \sum_{j=0}^{N} \Big(\sum_{k=0}^{M} A_{j,k} a_k - f_j \Big) \Big(\sum_{l=0}^{M} A_{j,l} a_l - f_j \Big), \end{aligned} \tag{220}$$

where

$$A \;=\; \begin{pmatrix} g_0(x_0) & g_1(x_0) & \dots & g_M(x_0) \\ g_0(x_1) & g_1(x_1) & \dots & g_M(x_1) \\ \vdots & \vdots & & \vdots \\ g_0(x_N) & g_1(x_N) & \dots & g_M(x_N) \end{pmatrix}. \tag{221}$$

– (218):

$$\begin{aligned} &\frac{\partial}{\partial a_m} G(\mathbf{a}) \;=\; 2 \sum_{j=0}^{N} \Big(\sum_{k=0}^{M} A_{j,k} a_k - f_j \Big) A_{j,m} \;=\; 0 \\ &\rightarrow \quad A^T A \mathbf{a} \;=\; A^T \mathbf{f} \end{aligned} \tag{222}$$

i.e. one has to solve a system of linear equations to determine the parameters $\mathbf{a}$ (e.g. by using methods from section 7).

- $g(x;\mathbf{a})$ not linear in a_j:
  - (218) is system of non-linear equations.
  - Solving such systems is difficult (see section 5.5).
  - Typically a good estimate of the parameters $\mathbf{a}$ is needed to solve such systems of non-linear equations, e.g. by using the Newton-Raphson method.
- Figure 14 shows the least squares approximation of data points already used in Figure 12, right ($N = 9$ example) and Figure 13 using degree-0, degree-1 and degree-2 polynomials; in contrast to Figure 12 and Figure 13, there are no oscillations.

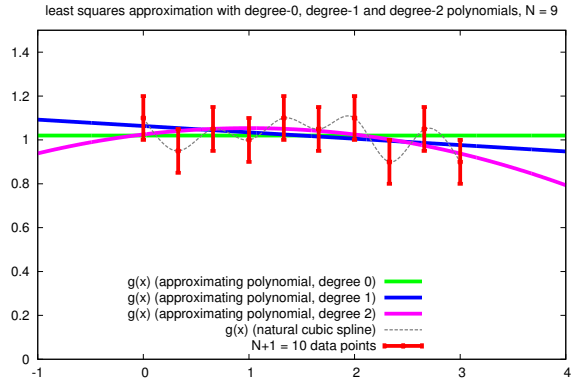


Figure 14: Least squares approximation with degree-0, degree-1 and degree-2 polynomials of $N + 1 = 10$ data points f_j.

****** December 19, 2023 (19th lecture) ******

10.4 χ^2 minimizing fits

- Quite often, data points have errors, which have been ignored so far.
- Notation: σ_j is the error of data point f_j (i.e. "value $f_j \pm \sigma_j$ at x_j").
- When approximating data points using an ansatz $g(x;\mathbf{a})$ (as e.g. in section 10.3), data points with small errors should have a stronger influence on the parameters $\mathbf{a}$ than data points with large errors.
- Replace (217) by

$$\chi^2 \quad = \quad \sum_{j=0}^{N} \left(\frac{g(x_j;\mathbf{a}) - f_j}{\sigma_j} \right)^2 \tag{223}$$

to fit $g(x_j;\mathbf{a})$ to the data points f_j with errors σ_j in an optimal way ("if σ_j is small, $g(x_j;\mathbf{a})$ must be close to f_j ... otherwise χ^2 would be large").

- Resulting, i.e. minimal χ^2 indicates the quality of the fit:
  - "Good fit"
    $\rightarrow$ each term in (223) should be of order 1
    $\rightarrow$ "reduced χ^2" $= \chi^2/\text{dof} = \chi^2/(N-M) \approx 1$.
  - $\chi^2/\text{dof} \gg 1$
    $\rightarrow$ ansatz $g(x_j;\mathbf{a})$ not consistent with data points.
  - $\chi^2/\text{dof} \ll 1$
    $\rightarrow$ errors are either overestimated or data points are correlated.
- For details see textbooks on data analysis.
- Simple and common example: χ^2 minimizing fit of a constant a.
  - Ansatz: $g(x;\mathbf{a}) = a$.
  - Minimizing

$$\chi^2 = \sum_{j=0}^{N} \left(\frac{a - f_j}{\sigma_j} \right)^2 \tag{224}$$

    is equivalent to solving

$$0 = \frac{d}{da}\chi^2 = 2\sum_{j=0}^{N} \frac{a - f_j}{(\sigma_j)^2}, \tag{225}$$

    i.e.

$$a = \sum_{j=0}^{N} \underbrace{\frac{1/(\sigma_j)^2}{\sum_{k=0}^{N} 1/(\sigma_k)^2}}_{=w_j} f_j = \sum_{j=0}^{N} w_j f_j, \tag{226}$$

    where w_j is the "weight of data point f_j" ($0 \le w_j \le 1$, $\sum_{j=0}^{N} w_j = 1$).
  - An example is shown in Figure 15.

10.4.1 Error estimates for fit parameters a_j ("basics of data analysis")

- Since the data points f_j have errors σ_j, the fit parameters a_j should have errors as well (denoted in the following by Δa_j).
- One has to propagate the errors of f_j to obtain meaningful errors for a_j, i.e. the errors Δa_j depend on the errors σ_j.
- In the following two simple methods of error propagation: resampling, jackknife method.
- In general, data analysis (e.g. the computation and propagation of errors) can be very complex (see advanced/specialized articles and textbooks on data analysis, e.g. [6]).

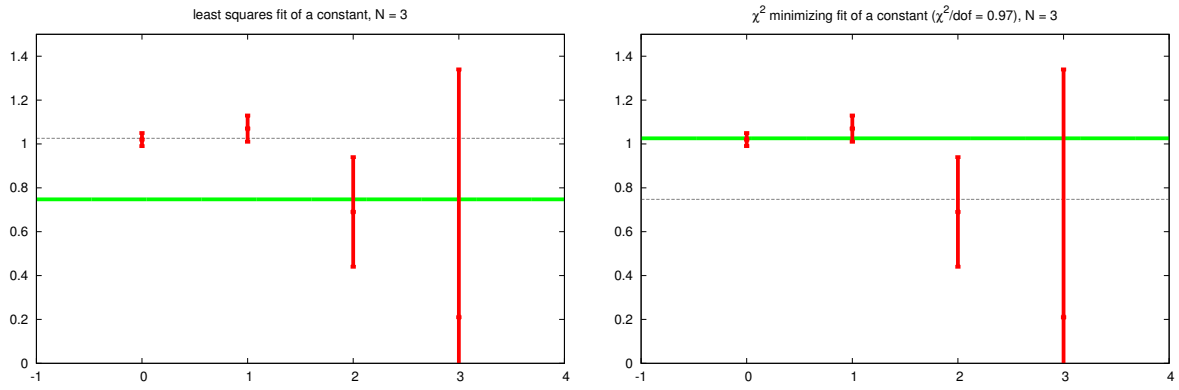


Figure 15: Comparison of a least squares fit (**left**) and a χ^2 minimizing fit (**right**) of a constant to $N+1=4$ data points f_j.

Preliminary considerations: data points and samples

- A data point f_j and its error σ_j are typically based on samples $f_j^{(1)}, \ldots, f_j^{(S)}$.
- Samples can e.g. be identical experimental measurements or computations based on statistically fluctuating initial conditions/input.
- Assuming these samples are Gaussian distributed, realistic and mathematically justified estimates are

$$f_j = \frac{1}{S}\sum_{s=1}^{S} f_j^{(s)} \tag{227}$$

$$\sigma_j = \left(\frac{1}{S(S-1)}\sum_{s=1}^{S}\left(f_j^{(s)} - f_j\right)^2\right)^{1/2}. \tag{228}$$

  The probability that the "true unknown value" of f_j (the mean of the Gaussian distribution or, equivalently, f_j according to (227) for $S \to \infty$) is inside the interval $[f_j - \sigma_j, f_j + \sigma_j]$ is $\approx 68\%$ (see also section 8.2.2).
- Assuming a Gaussian distribution is quite common and often reasonable. For example, when defining the samples not as results of single measurements, but as averages of many measurements, the samples will be (almost) Gaussian distributed (*central limit theorem*: "infinite sums of random variables are Gaussian distributed").

Resampling

- Use resampling, if you have the f_j and σ_j, but not the samples $f_j^{(s)}$ (not ideal, but often the case, e.g. when using results from the literature).

- Basic idea of resampling: Generate artificial data/samples $f_j'^{(1)}, \ldots, f_j'^{(S')}$ ($S' \neq S$ in general) according to the Gaussian distribution

$$\frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(- \frac{(f_j'^{(s)} - f_j)^2}{2\sigma_j^2} \right). \tag{229}$$

  Note that these artificial samples have the same mean, but a smaller variance than the original samples (smaller by the factor $1/\sqrt{S}$).

- A conceptually simple (not very efficient) method to generate such a sample $f_j'^{(s)}$ is the following:

(step 1) Select a random number $f_j'^{(s)}$ inside the interval $[f_j - \lambda\sigma_j, f_j + \lambda\sigma_j]$ according to a uniform distribution.

(step 2) Select a random number x inside the interval $[0, 1]$ according to a uniform distribution. If $x \leq e^{-(f_j'^{(s)} - f_j)^2/2\sigma_j^2}$, use $f_j'^{(s)}$ as sample, otherwise go back to (step 1).

  $\lambda \approx 6$ might be a good choice (larger values of λ lead to samples $f_j^{(s)}$ with a distribution closer to (229); however, for larger values of λ the method is less efficient, i.e. (step 1) needs to be repeated more often).

- Once the samples $f_j'^{(s)}$, $j = 1, \ldots, N$, $s = 1, \ldots, S'$ are generated, one has to carry out a χ^2 minimizing fit for each s to the "data points" $f_j'^{(s)} \pm \sigma_j$ (as discussed above). The resulting fit parameters are denoted as $a_j^{(s)}$.

- The errors for the fit parameters are then estimated according to

$$\Delta a_j \;\; = \;\; \left(\frac{1}{S'} \sum_{s=1}^{S'} \left(a_j^{(s)} - a_j \right)^2 \right)^{1/2}. \tag{230}$$

  Note that there is an important difference, when comparing this equation e.g. to (228), the prefactor $1/S'$ in (230) compared to $1/S(S-1)$ in (228):

  - $1/S(S-1)$ in (228):
    σ_j should not correspond to the standard deviation, i.e. the fluctuations of the original samples. It is an uncertainty, which should decrease $\propto 1/\sqrt{S}$, when the number of samples is increased.
  - $1/S'$ in (230):
    Δa_j should represent the fluctuations of the generated samples, which reflect the errors σ_j. Thus, (230) is an ordinary standard deviation, which approaches a constant for large S'. S' should be chosen sufficiently large such that the resulting Δa_j are close to these constants.

- Possible problem: original samples $f_j^{(s)}$ might be correlated.

– For example $f_1^{(s)}$ and $f_2^{(s)}$ might be measurements taken at the same time (indicated by the identical index (s)) of related quantities, e.g. the temperature at two nearby spatial points, $T(x)$ and $T(x+\epsilon)$. If $f_1^{(s)}$ is larger than than its mean, there is a high probability that also $f_2^{(s)}$ is larger than its mean.

– Such correlations are not considered, because artificial samples $f_j'^{(s)}$ are generated independently, i.e. are uncorrelated.

****** December 21, 2023 (20th lecture) ******

Jackknife method

- Literature: [6, 7].
- Use the jackknife method, if you have the f_j and the samples $f_j^{(s)}$ (and, consequently, also σ_j via (228)).
- Compute S *reduced samples* (also called *jackknife samples*)

$$f_j^{(s),\mathrm{red}} = \frac{1}{S-1}\sum_{r\neq s} f_j^{(r)}. \tag{231}$$

- Carry out a χ^2 minimizing fit for each s to the "data points" $f_j^{(s),\mathrm{red}} \pm \sigma_j$ (as discussed above). The resulting fit parameters are denoted as $a_j^{(s),\mathrm{red}}$.
- The errors for the fit parameters are then estimated according to

$$\Delta a_j = \left(\frac{S-1}{S}\sum_{s=1}^{S}\left(a_j^{(s),\mathrm{red}} - a_j\right)^2\right)^{1/2}. \tag{232}$$

  Note that the prefactor is different from both (228) and (230) ($(S-1)/S$ versus $1/S(S-1)$ and $1/S'$).

  – Since the variance of the reduced samples (231) is small compared to the original samples, there must be a larger prefactor for a realistic error estimate.

- The jackknife method takes correlations between the original samples $f_j^{(s)}$ into account. The reason is that correlations are also present in the reduced samples. This is in contrast e.g. to resampling, where correlations are unknown and, thus, not present in the generated artificial samples.

Generalization of resampling and the jackknife method

- Resampling and the jackknife method are not limited to fitting. Both methods can be used to propagate errors for "arbitrary complicated mathematical or numerical operations" on data points with errors.

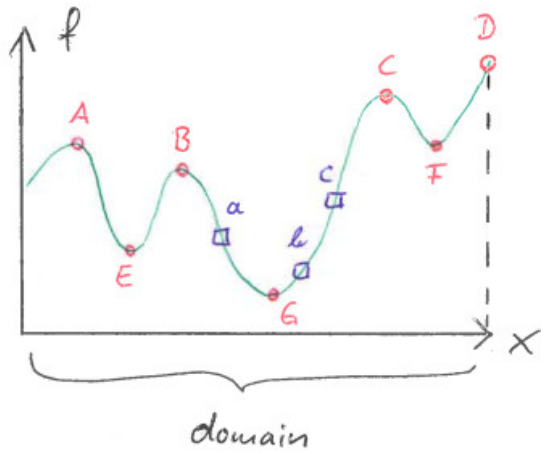
- Just replace the χ square minimizing fit by any operation $\mathcal{O}$ leading to results $\mathbf{a}$, i.e. an operation $\mathbf{a} = \mathcal{O}(\mathbf{f})$.
- For example one can compute the error of the logarithm of a quantity f with error σ or samples $f^{(s)}$ using $a = \mathcal{O}(f) = \ln(f)$.
- Homework and consistency check: Show that computing the error of a quantity f with samples $f^{(s)}$ via the jackknife method (i.e. $a = \mathcal{O}(f) = f$) results in the well-known expression (228), i.e. the jackknife error Δa_j in that particular case is identical to the error σ_j.

11 Function minimization, optimization

11.1 Problem definition, general remarks

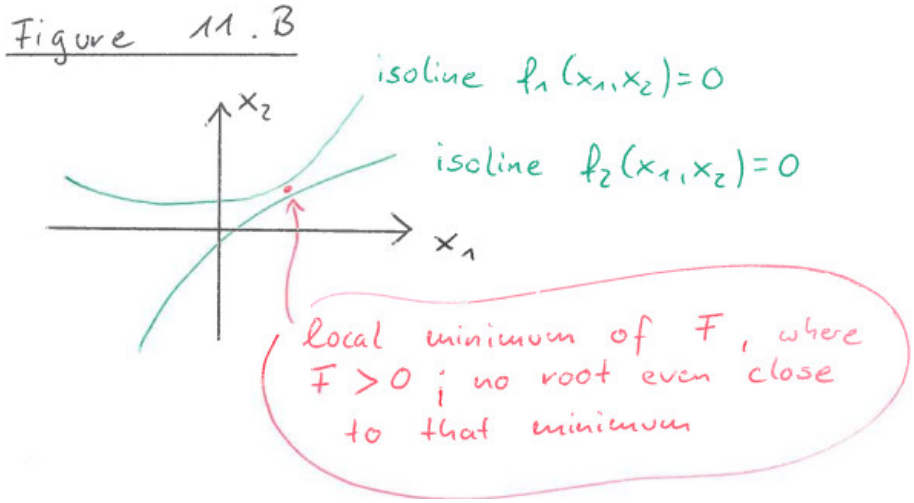
- $f(\mathbf{x})$: real valued function ($\mathbf{x} = (x_1, \ldots, x_D)$).
- Problem: find a local or the global minimum of $f(\mathbf{x})$, i.e. a value $\mathbf{x}$ minimizing $f(\mathbf{x})$ locally or globally.
- Function maximization is equivalent to function minimization, because maximum of $f(\mathbf{x})$ is minimum of $-f(\mathbf{x})$.
- Algorithms for function minimization should be
  - fast (i.e. the number of evaluations of $f(\mathbf{x})$ should be reduced to a minimum),
  - able to find the global minimum.
- Finding the global minimum is extremely difficult, in particular in $D \geq 2$ dimensions; typical strategies are
  - repeated function minimization starting at different $\mathbf{x}$,
  - minimize function, add a random perturbation to the (possibly local) minimum, minimize again, add another random perturbation, minimize again, ...
- Notation, classification of function values x of $f(x)$ in $D = 1$ dimension:

Figure 11.A



  - A, B, C: local maxima.
  - D: global maximum ($f'(x) \neq 0$ possible on the boundary of the domain).
  - E, F: local minima.
  - G: global minimum ($f'(x) = 0$ inside the domain).
  - a, b, c: enclose minimum (see section 11.2).

- Root finding versus function minimization:
  - At first glance problems seem to be very similar: $\mathbf{f}(\mathbf{x}) = 0$ versus $\nabla f(\mathbf{x}) = 0$.
  - There are, however, significant differences:
    * Root finding:
      $f_1(\mathbf{x})$, $f_2(\mathbf{x})$, ... are independent functions.
      Function minimization:
      $(\nabla f)_1(\mathbf{x})$, $(\nabla f)_2(\mathbf{x})$, ... are related via $f(\mathbf{x})$.
    * Root finding:
      Not obvious, "which direction one has to follow", to find $f_j(\mathbf{x}) = 0$ for all j.
      Function minimization:
      "Simply follow the negative gradient" to find a minimum, i.e. $\nabla f(\mathbf{x}) = 0$.
    * Root finding:
      Very hard, if number of dimensions D is large.
      Function minimization:
      Comparatively simple, even if number of dimensions D is large.
- Since function minimization is easier than root finding, one could be tempted to reformulate root finding $\mathbf{f}(\mathbf{x}) = 0$ as function minimization:
  - Roots of $\mathbf{f}(\mathbf{x})$ are global minima of $F(\mathbf{x}) = \sum_{j=1}^{D} (f_j(\mathbf{x}))^2$.
  - Do not do that, i.e. do not try to find roots of $\mathbf{f}(\mathbf{x})$ by searching minima of $F(\mathbf{x})$.
  - Function minimization algorithms typically find local minima of $F(\mathbf{x})$, which do not correspond to roots of $\mathbf{f}(\mathbf{x})$.

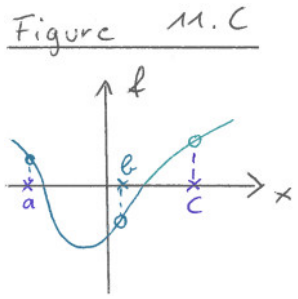


****** January 09, 2024 (21th lecture) ******

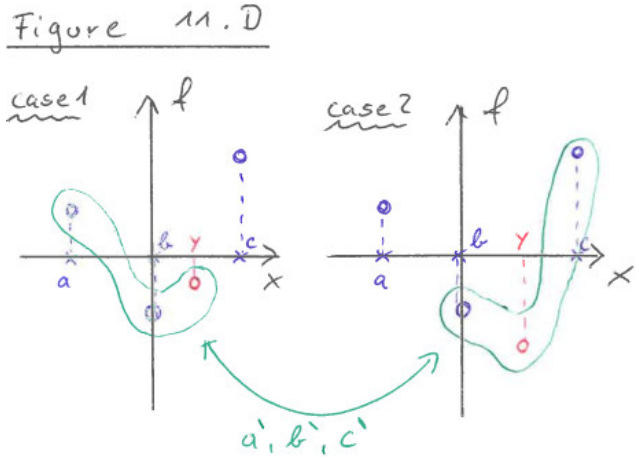
11.2 Golden section search in $D = 1$ dimension

- Similar to bisection for root finding.

- Starting point: minimum localized inside interval $[a, c]$ via $f(a) > f(b) < f(c)$, where $a < b < c$ (minimum can be left or right of b).



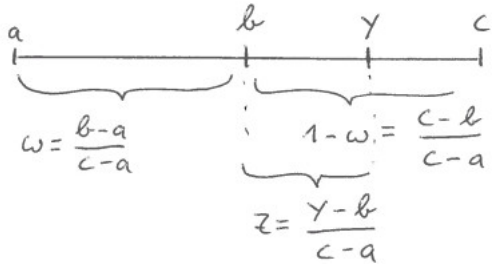
- Evaluate $f(y)$:
  - $b < y < c$:
    - If $f(y) > f(b)$ (case 1):
      → Replace (a, b, c) by $(a', b', c') = (a, b, y)$, i.e. minimum now localized inside interval $[a, y]$.
    - If $f(y) < f(b)$ (case 2):
      → Replace (a, b, c) by $(a', b', c') = (b, y, c)$, i.e. minimum now localized inside interval $[b, c]$.



  - $a < y < b$:
    - Analogous to $b < y < c$.
  - Iterate this step, until $c - a$ is sufficiently small.
- How to choose y, to reduce the size of the interval $[a, c]$ as quickly as possible?
  - Define relative sizes of subintervals:
    - $w = (b - a)/(c - a)$.
    - $1 - w = (c - b)/(c - a)$.

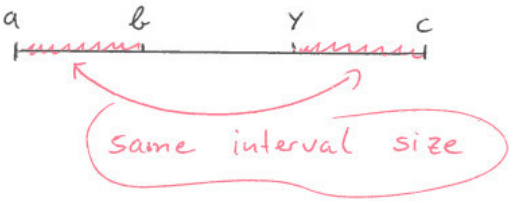
* $z = (y - b)/(c - a)$.

Figure 11.E



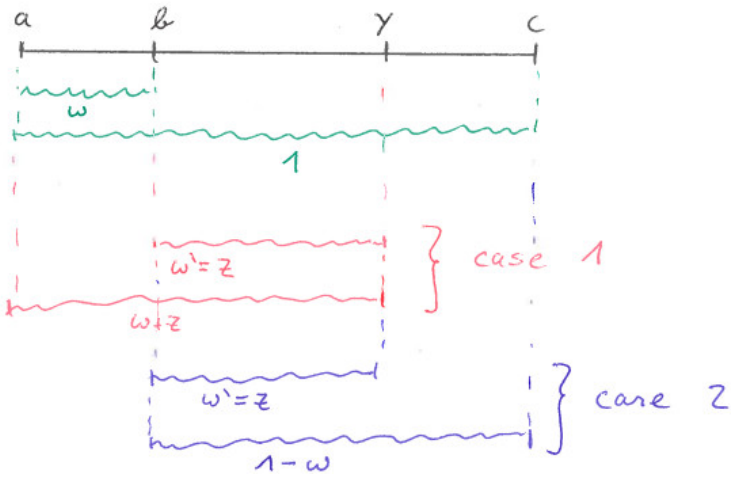
- Relative size of "new interval" $[a', c']$:
  * Case 1: $w + z$.
  * Case 2: $1 - w$.
- Determine z (and thereby y) via $w + z = 1 - w$
  $\rightarrow z = 1 - 2w = (1 - w) - w$, i.e. z is chosen such that the reduction of the size of $[a, b]$ is in both cases the same ("no bad case").
  ("It does not matter, whether we get rid of the left interval $[a, b]$ or the right interval $[y, c]$, because both have relative size w.")

Figure 11.F



- Rate of covergence:
  - How to choose w in the first place such that it stays constant throughout the algorithm?

Figure 11.6



* Case 1: $w' = z$, $c' - a' = w + z$
  $w/1 = w'/(c' - a') = z/(w + z)$.
* Case 2: $w' = z$, $c' - a' = 1 - w$
  $w/1 = w'/(c' - a') = z/(1 - w)$.
* Inserting $z = 1 - 2w$ yields $w^2 - 3w + 1 = 0$ in both cases.
* Solutions:
  - $w = (3 + \sqrt{5})/2 = 2.618\ldots$ (excluded, because > 1).
  - $w = (3 - \sqrt{5})/2 = 0.381\ldots$ (allowed),
  - "Golden section": $(1 + \sqrt{5})/2$.

("If we have $w = (a - b)/(a - c) = 0.381\ldots$ and carry out one step, we obtain three points with the same ratio, either a, b and y with $(y - b)/(y - a) = 0.381\ldots$ [case 1] or b, y and c with $(y - b)/(c - b) = 0.381\ldots$ The interval $[b, y]$ is in both cases the new short interval, the new y' has to be chosen in the new long interval, either in $[a, b]$ [case 1] or in $[y, c]$ [case 2]).")

– $w = (3 - \sqrt{5})/2$ is a stable fixed point (can be shown), i.e. even when starting with $w \neq (3 - \sqrt{5})/2$, w will quickly approach $(3 - \sqrt{5})/2$, when choosing z according to $z/(1 - w) = 0.381\ldots$ ("when dividing the larger interval according to the Golden section").

– Consequently,

$$c' - a' = \underbrace{\left(1 - \frac{3 - \sqrt{5}}{2}\right)}_{0.618\ldots}(c - a), \tag{233}$$

i.e. linear convergence (slightly slower than root finding with bisection [see section 5.2]).

- Accuracy:

– For $x \approx x_{\min}$

$$f(x) \approx f_{\min} + \frac{f''(x_{\min})}{2}(x - x_{\min})^2. \tag{234}$$

($(x_{\min}, f_{\min})$ is the minimum).

– When represented as floating point numbers $f(x)$ and $f_{\min}$ are different, only if

$$\frac{(f''(x_{\min})/2)(x - x_{\min})^2}{f_{\min}} > \epsilon, \tag{235}$$

where $\epsilon = \mathcal{O}(10^{-7})$ for `float` and $\epsilon = \mathcal{O}(10^{-16})$ for `double` (ϵ is the relative precision discussed in section 2.2).

– Consequently, the accuracy of golden section search (and many other minimization algorithms), characterized by $x - x_{\min}$ in (235), is limited,

$$x_{\min,\text{numerically}} - x_{\min} \gtrsim \left(\frac{2f_{\min}}{f''(x_{\min})}\right)^{1/2} \sqrt{\epsilon} \sim \sqrt{\epsilon}. \tag{236}$$

11.3 Function minimization using quadratic interpolation in $D = 1$ dimension

- Due to limited time not discussed.

11.4 Function minimization using derivatives in $D = 1$ dimension

- Due to limited time not discussed.

****** January 11, 2024 (22th lecture) ******

11.5 Function minimization in $D \geq 2$ dimensions by repeated minimization in 1 dimension

- Problem in $D \geq 2$ dimensions: localizing a minimum not as easy as in $D = 1$ dimension, where three points a, b, c with $f(a) > f(b) < f(c)$ are sufficient.
- Algorithm for function minimization in $D \geq 2$ dimensions by repeated minimization in 1 dimension (has already been used for the conjugate gradient method; see section 7.7.1):
  - *Guess minimum* $\mathbf{x}_0$*, e.g.* $\mathbf{x}_0 = 0$ *(can be far away from any minimum).*
  - $n = 0$.

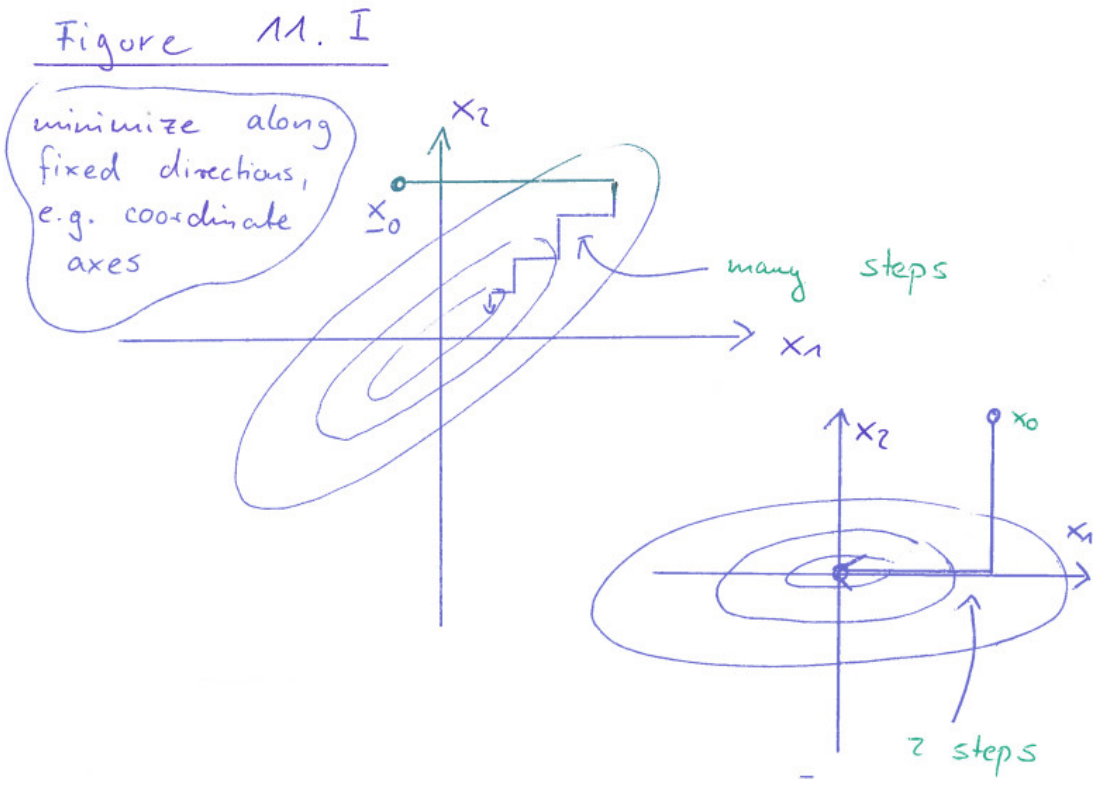
  (1) Select direction $\mathbf{p}_n$ *(details below).*
  - *Minimize* $f(\mathbf{x}_n + \alpha_n \mathbf{p}_n)$ *with respect to* α_n.
  - $\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha_n \mathbf{p}_n$.
  - *If* $\mathbf{x}_{n+1}$ *is sufficiently close to a minimum (e.g.* $|\mathbf{x}_{n+1} - \mathbf{x}_n| < \epsilon$*):*

    $\rightarrow$ $\mathbf{x}_{n+1}$ *is approximate minimum.*
    End of algorithm.

Else:

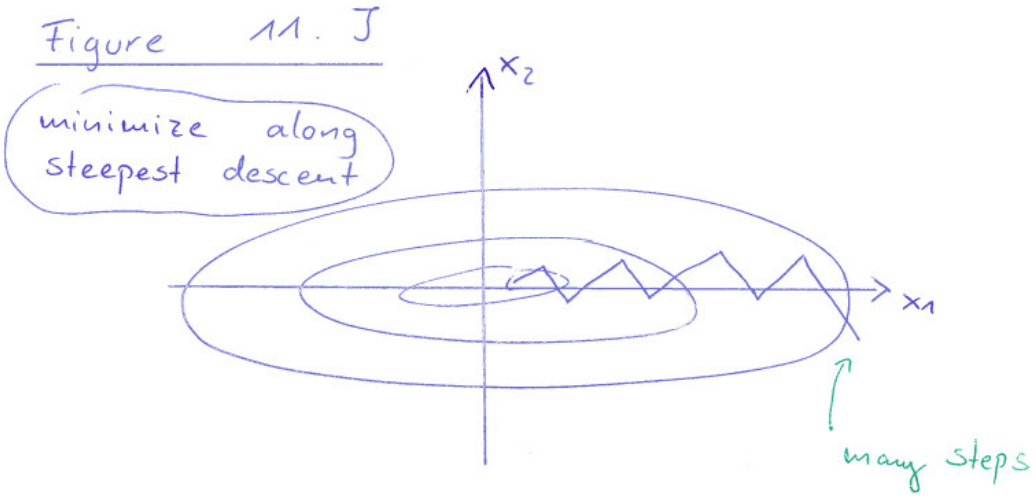
→ $n = n + 1$.
Go to (1).

- Efficiency of the algorithm strongly depends, on how the directions $\mathbf{p}_n$ are chosen.
  - Simple example:
    - $D = 2$ dimensions.
    - $f(x_1, x_2)$: a paraboloid, which is "wide in $(+1, +1)/\sqrt{2}$ direction and narrow in $(-1, +1)/\sqrt{2}$ direction",
      $$f(x_1, x_2) = \frac{x^2 + y^2 + 2xy}{2a^2} + \frac{x^2 + y^2 - 2xy}{2b^2} \tag{237}$$
      with $a \gg b$.
    - Directions for minimization $\mathbf{p}_n$: $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_1$, $\mathbf{e}_2$, ...
      → Many minimizations in 1 dimension required, algorithm quite inefficient.
    - Directions for minimization $\mathbf{p}_n$: $(+1, +1)/\sqrt{2}$, $(-1, +1)/\sqrt{2}$.
      → Only two minimizations in 1 dimension required, algorithm very efficient.
    - See also Ref. [1], Figure 10.7.1 and section 7.7.1.



    - It might seem to be a good strategy to select $\mathbf{p}_n = \nabla f(\mathbf{x}_n)$, i.e. to minimize along the direction of steepest descent.
      → Many minimizations in 1 dimension required, almost as inefficient as selecting $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_1$, $\mathbf{e}_2$, ...

* See also Ref. [1], Figure 10.8.1.



- Efficient way to select directions $\mathbf{p}_n$ are "conjugate directions":
  - Basic idea:
    * Select $\mathbf{p}_n$ such that minimization with respect to previous directions $\mathbf{p}_j$, $j = 0, \ldots, n-1$ is preserved (then $\mathbf{p}_n$ and $\mathbf{p}_j$, $j = 0, \ldots, n-1$ are called conjugate directions).
    * Minimum found within D minimizations in 1 dimension.
  - Mathematical details:
    * $\mathbf{x}_1$ is minimum of $f(\mathbf{x})$ along direction $\mathbf{p}_0$, i.e. $\mathbf{p}_0 \nabla f(\mathbf{x}_1) = 0$.
    * Select a conjugate direction $\mathbf{p}_1$ such that the gradient of $f(\mathbf{x})$ in $\mathbf{p}_0$ direction vanishes along direction $\mathbf{p}_1$, i.e. $\mathbf{p}_0 \nabla f(\mathbf{x}_1 + \lambda \mathbf{p}_1) = 0$ for all λ.
      · In general not possible.
      · It is possible, if $f(\mathbf{x})$ is quadratic, i.e. describes a paraboloid,
      $$f(\mathbf{x}) \quad = \quad c - \mathbf{b}\mathbf{x} + \frac{1}{2}\mathbf{x}A\mathbf{x}. \tag{238}$$
      · Then
      $$\nabla f(\mathbf{x}) \quad = \quad -\mathbf{b} + A\mathbf{x}. \tag{239}$$
      · $\mathbf{p}_0 \nabla f(\mathbf{x}_1) = 0$ becomes
      $$\mathbf{p}_0 \Big(-\mathbf{b} + A\mathbf{x}_1 \Big) \quad = \quad 0. \tag{240}$$
      · The condition for conjugate directions $\mathbf{p}_0 \nabla f(\mathbf{x}_1 + \lambda \mathbf{p}_1) = 0$ becomes
      $$\mathbf{p}_0 \Big(-\mathbf{b} + A(\mathbf{x}_1 + \lambda \mathbf{p}_1) \Big) \quad = \quad \mathbf{p}_0 A \lambda \mathbf{p}_1 \quad = \quad 0, \tag{241}$$
      i.e. $\mathbf{p}_0$ and $\mathbf{p}_1$ are conjugate, if
      $$\mathbf{p}_0 A \mathbf{p}_1 \quad = \quad 0. \tag{242}$$
      · If $f(\mathbf{x})$ is not quadratic, select "approximate conjugate direction" via Taylor expansion of $f(\mathbf{x})$,
      $$f(\mathbf{x}) \quad = \quad \underbrace{f(\mathbf{x}_1)}_{=c} + \sum_{j=1}^{D} \underbrace{\partial_j f(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_1}}_{=b_j} (x_j - x_{1,j})$$

$$+\frac{1}{2}\underbrace{\sum_{j,k=1}^{D}\partial_j\partial_k f(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_1}}_{=A_{j,k}}(x_j-x_{1,j})(x_k-x_{1,k})+\ldots, \tag{243}$$

i.e. use A from (243) in (242).

- Then minimum found approximately within D minimizations in 1 dimension.
- In practice: Repeat minimization in 1 dimension, until $\mathbf{x}_n$ is approximate minimum; converges typically fast, in particular, if $f(\mathbf{x})$ is similar to a paraboloid.
- If $f(\mathbf{x})$ is quite different from a paraboloid, other methods might be more efficient.

– For more details, e.g. concrete methods to construct conjugate directions, see Ref. [1], section 10.7 and section 10.8.

****** January 16, 2024 (23th lecture), January 17, 2024 (24th lecture) ******

11.6 Downhill simplex method ($D \geq 2$ dimensions)

- Downhill simplex method: simple algorithm for function minimization in $D \geq 2$ dimensions, however, not very efficient.
  → Suited for rather simple optimization problems.

- Simplex:
  - Defined by $D+1$ points $\mathbf{p}_j$.
    - $D = 2$: triangle.
    - $D = 3$: tetrahedron.
    - ...
  - Exclusively consider non-degenerate simplexes, i.e. $\mathbf{p}_1 - \mathbf{p}_0$, $\mathbf{p}_2 - \mathbf{p}_1$, ..., $\mathbf{p}_D - \mathbf{p}_0$ are linearly independent.

- Basic principle: simplex moves "downhill", deforms according to the "terrain" defined by $f(\mathbf{x})$, stops at a local minimum.

- Initial simplex:
  - $\mathbf{p}_0$: estimate of minimum, i.e. an input parameter.
  - $\mathbf{p}_j = \mathbf{p}_0 + \lambda_j \mathbf{e}_j$, where λ_j is a typical length scale in j direction, i.e. an input parameter.

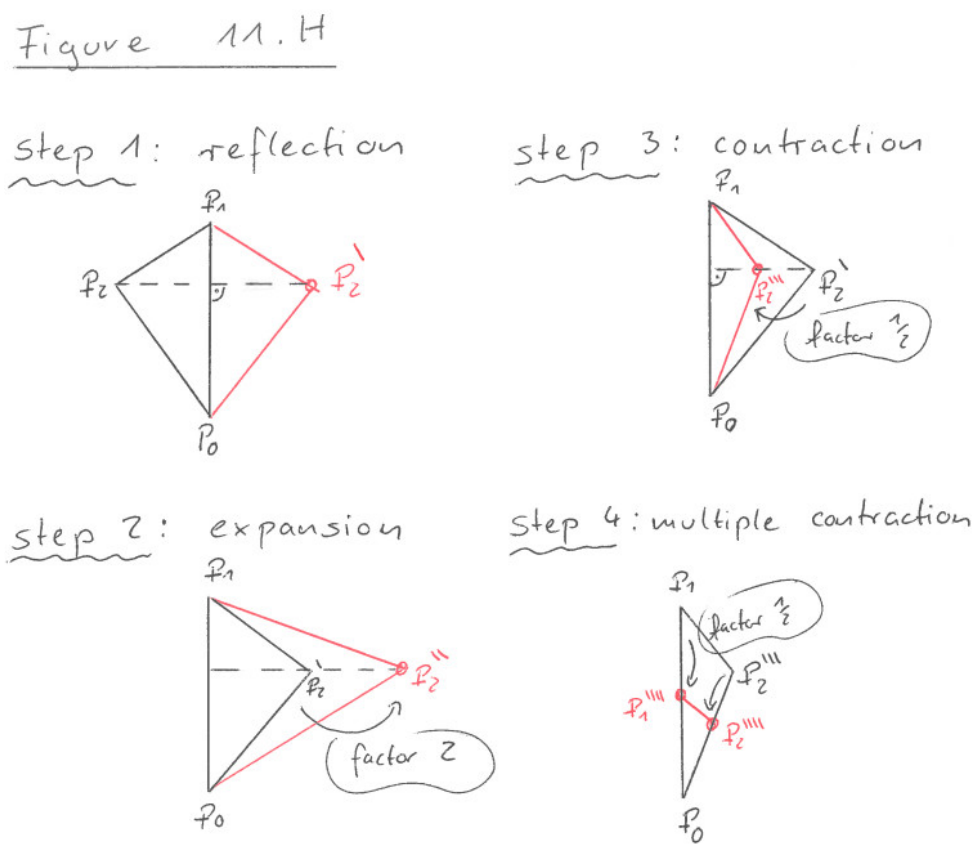
- Sketch of deformation steps (for $D = 2$):

(1) Relabel points $\mathbf{p}_j$ such that $f(\mathbf{p}_0) \leq f(\mathbf{p}_1) \leq \ldots \leq f(\mathbf{p}_D)$.

– Step 1: reflection ("move the point, where f is largest").
$\mathbf{p}_2 \to \mathbf{p}_2'$.

– If $f(\mathbf{p}_2') < f(\mathbf{p}_0)$:

    - Step 2: expansion ("expand the simplex to take larger steps").
      $\mathbf{p}_2' \to \mathbf{p}_2''$.
    - Replace $\mathbf{p}_2$ by the better of the two points $\mathbf{p}_2'$ and $\mathbf{p}_2''$.
    - Goto (1).
  - If $f(\mathbf{p}_2') < f(\mathbf{p}_1)$:
    - Replace $\mathbf{p}_2$ by $\mathbf{p}_2'$.
    - Goto (1).
  - Step 3: contraction ("contract the simplex in a valley floor to ooze down the valley").
    $\tilde{\mathbf{p}} \to \mathbf{p}_2'''$, where $\tilde{\mathbf{p}}$ is the better of the two points $\mathbf{p}_2$ and $\mathbf{p}_2'$.
  - If $f(\mathbf{p}_2''') < f(\mathbf{p}_2)$:
    - Replace $\mathbf{p}_2$ by $\mathbf{p}_2'''$.
    - Goto (1).
  - Step 4: multiple contraction ("contract the simplex in all directions to pass through the eye of a needle").
    $\mathbf{p}_1 \to \mathbf{p}_1''''$ and $\mathbf{p}_2''' \to \mathbf{p}_2''''$
  - Goto (1).
  - For details see Ref. [1], section 10.5.



- Stopping criteria:

- Less obvious than in $D = 1$ dimension.
- E.g. if $|\mathbf{p}_D - \mathbf{p}_0| < \epsilon$ (where $f(\mathbf{p}_D) > f(\mathbf{p}_j)$, $j = 0, \ldots, D-1$ and $f(\mathbf{p}_0) < f(\mathbf{p}_j)$, $j = 1, \ldots, D$) ...
- ... or if deformation of simplex is almost negligible.
- As a cross-check one should start the downhill simplex method again at the found minimum with a large initial simplex.

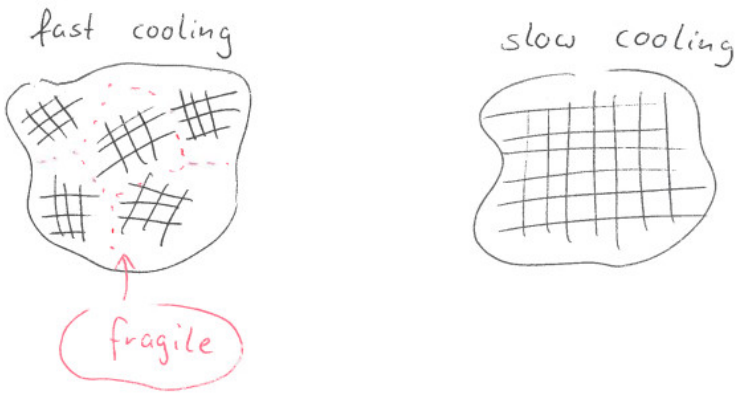
11.7 Simulated annealing

- Previously discussed methods typically find a local minimum inside a given interval (golden section search) or close to, where the minimization is started ("conjugate directions", downhill simplex method).
- How to find the global minimum of a function $f(\mathbf{x})$?
  - Repeated function minimization starting at different $\mathbf{x}$.
    - If only a small number of different minima is found, the global minimum might be among them.
    - Quite often, however, there is a very large number of minima and $f(\mathbf{x})$ is "complicated" (i.e. no idea, where the global minimum is).
  - A promising method to find the global minimum is simulated annealing.

11.7.1 Discrete minimization

- Instead of a continuous domain parameterized by $\mathbf{x}$ consider an extremely large number of discrete configurations $S = \{s_1, s_2, \ldots, s_N\}$ and a function $f(s)$, where $s \in S$.
- E.g. $N \approx 10^{100}$, i.e. impossible to evaluate $f(s)$ for all $s \in S$.
- Goal: Find that s_j minimizing $f(s)$.
- Basic idea is realized in nature:
  - Cooling of a liquid, e.g. steel (annealing = Ausglühen).
  - Fast cooling: not enough time for atoms/molecules to form a uniform crystalline structure corresponding to the energetic minimum; the resulting steel is fragile.
  - Slow cooling: atoms/molecules will form a uniform crystal, which is very stable.

Figure 11.K



- Application of this idea to the minimization of $f(s)$:
  - Move step by step through the space of configurations, $s^{(0)} \to s^{(1)} \to s^{(2)} \to \ldots$, where $s^{(n)}$ and $s^{(n+1)}$ are similar.
  - Steps $s^{(n)} \to s^{(n+1)}$, where $f(s^{(n)}) > f(s^{(n+1)})$ ("downhill"), are preferred ...
  - ... but also steps $s^{(n)} \to s^{(n+1)}$, where $f(s^{(n)}) \leq f(s^{(n+1)})$ ("uphill"), are allowed ("quite often one has to overcome a couple of mountains to reach the global minimum").
  - Slowly decrease the probability to perform uphill steps during the simulated annealing.
    - Similar to annealing of steel.
    - At the beginning, when the steel is hot, the structure of atoms/molecules readily changes, because the energy of a configuration is less important.
    - Later, when the temperature is getting lower, atoms/molecules freeze in a crystalline configuration with small energy.

- Simulated annealing algorithm:
  - *Start at arbitrary $s^{(0)} \in S$ and large temperature T.*
  - $n = 0$.

  (1) Randomly select $s^{(n+1)} \in S$, where "$s^{(n+1)} \approx s^{(n)}$", i.e. a similar configuration (see example below).
  - *If $f(s^{(n+1)}) \leq f(s^{(n)})$:*

    $\to$ Do nothing ("accept $s^{(n+1)}$").

    Else:

    $\to$ Either accept $s^{(n+1)}$ with probability $e^{-(f(s^{(n+1)})-f(s^{(n)}))/T}$ ...
    ...or replace $s^{(n+1)}$ by $s^{(n)}$ (i.e. "keep previous configuration") with inverse probability $1 - e^{-(f(s^{(n+1)})-f(s^{(n)}))/T}$.
  - *Slowly reduce T (typically not after every step, but after a fixed number of steps).*
  - *Go to (1).*

- Example: traveling salesperson problem.
  - N cities on a 2 dimensional map, i.e. (x_j, y_j), $j = 1, \ldots, N$.
  - Problem: find the shortest loop connecting all cities.
  - Configurations: S is the set of all permutations of $(1, 2, \ldots, N)$ (i.e. $N!$ different configurations, typically a huge number).
  - The function $f(s)$ is the length of the loop:

$$f(s_j) \quad = \quad \sum_{k=1}^{N} \left(\left(x_{p_j(k)} - x_{p_j(k-1)} \right)^2 + \left(y_{p_j(k)} - y_{p_j(k-1)} \right)^2 \right)^{1/2} \tag{244}$$

    $((x_0, y_0) \equiv (x_N, y_N)$; configurations s_j are permutations $p_j)$.
  - Simulated annealing:
    - Selecting a similar next configuration:
      - Reverse a randomly chosen subsequence, e.g.
        $s^{(n)} = \ldots - 17 - 3 - 9 - 6 - 14 - \ldots$
        $\rightarrow \quad s^{(n+1)} = \ldots - 17 - 6 - 9 - 3 - 14 - \ldots$
      - Randomly move a randomly chosen subsequence, e.g.
        $s^{(n)} = \ldots - 5 - 3 - 9 - 1 - 16 - 14 - 21 - \ldots$
        $\rightarrow \quad s^{(n+1)} = \ldots - 5 - 16 - 14 - 3 - 9 - 1 - 21 - \ldots$
      - There are many other possibilities.
    - Reducing the temperature:
      - Select initial T larger than typical $|f(s^{(n+1)}) - f(s^{(n)})|$.
      - Reduce T by 10% after $100 \times N$ iterations or after $10 \times N$ "successful updates".
      - Stop simulated annealing, when the algorithm freezes, i.e. the configuration does not change anymore.
  - Possible variant of the traveling salesperson problem:

$$f(s_j) \quad \rightarrow \quad f(s_j) + \lambda \sum_{k=1}^{N} \left(\mu_{p(k)} - \mu_{p(k-1)} \right)^2, \tag{245}$$

    where $\mu_j = +1$, if city j is "east of the river", and $\mu_j = -1$, if city j is "west of the river".
    - $\lambda > 0$: crossing the river is expensive/takes times/etc., i.e. the salesperson wants to avoid it.
    - $\lambda < 0$: the salesperson likes to cross the river.
    - See Ref. [1], Figure 10.12.1.

11.7.2 Continuous minimization

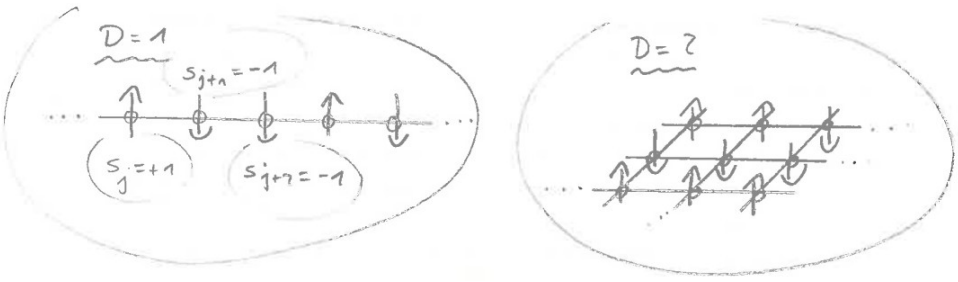
- Simulated annealing can also be applied to minimize functions $f(\mathbf{x})$ with a continuous domain.
- Selecting a "similar next configuration" $\mathbf{x}^{(n+1)}$ might be more difficult than for a discrete set of configurations:

– Typical problem: too few downhill steps $\mathbf{x}^{(n)} \to \mathbf{x}^{(n+1)}$.
– See Ref. [1], section 10.12.2.

12 Monte Carlo simulations of partition functions

12.1 Ising model

- A commonly used simple model to understand magnetism.
- Spins $s_j = \pm 1$ on the sites of a D-dimensional cubic lattice.



- A particular state (= spin configuration) can be described via $C = (s_1, s_2, \ldots)$.
  - N spins $\rightarrow 2^N$ states.
  - Example: a tiny magnet with 100 spins in each spatial direction in $D = 3$ dimensions.
    $\rightarrow N = 100^3 = 10^6$ spins.
    $\rightarrow 2^N = 2^{(10^6)} \approx 10^{3 \times 10^5}$ possible states.
- Hamilton operator:

$$H = -\lambda \sum_{\langle j,k \rangle} s_j s_k - B \sum_j s_j. \tag{246}$$

  - $\sum_{\langle j,k \rangle}$: sum over neighboring spins, i.e. over $2D$ spins.
  - Ferromagnetism for $\lambda > 0$.
  - B: external magnetic field.
- Partition function:

$$Z = \sum_C e^{-\beta H(C)} \tag{247}$$

  ($\beta = 1/T$: inverse temperature [$k_B = 1$]).
- Expectation value (= long-term average) of an observable $O(C)$:

$$\left\langle O \right\rangle = \frac{1}{Z} \sum_C e^{-\beta H(C)} O(C) \tag{248}$$

  ($e^{-\beta H(C)}/Z$ is the probability to find the system in state C).
- In principle a computer can calculate the sums in (248).

- In practice, however, this is not possible for interesting/realistic system sizes.
  - Consider the above example with $N = 10^6$ spins.
  - Typical time a computer needs to add two numbers: $t = 10^{-9}\,\mathrm{s}$.
  - Total estimated time to compute the partition function Z:

    $$2^N \times t \;\approx\; 10^{3\times 10^5} \times 10^{-9}\,\mathrm{s} \;\approx\; 10^{3\times 10^5} \times 3 \times 10^{-17}\,\mathrm{y} \;\approx\; 10^{3\times 10^5}\,\mathrm{y} \tag{249}$$

    $(1\,\mathrm{y} \approx 365 \times 24 \times 60 \times 60\,\mathrm{s} \approx 3 \times 10^7\,\mathrm{s})$.
- Solution: Do not compute partition functions exactly but compute unbiased statistical estimates with error bars (similar to experimental results) via **Monte Carlo simulations**.
- Techniques discussed in this section can also be used to simulated path integrals in quantum field theory (lattice field theory).
  → Monte Carlo simulations are useful in many areas of physics, e.g. solid state physics (Ising-Model, Magnetism, ...) and particle physics (in particular QCD = quantum chromodynamics).

12.2 Basic principle of Monte Carlo simulations

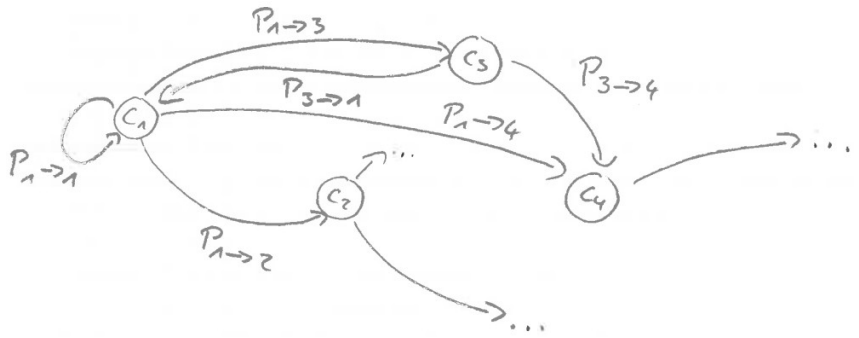
- Monte Carlo simulations are based on **Markov chains**:
  - In our context, a Markov chain is a random generator for states $\tilde{C}_1, \tilde{C}_2, \ldots, \tilde{C}_n$ with probability

    $$P(\tilde{C}_j) \;=\; e^{-\beta H(\tilde{C}_j)}/Z. \tag{250}$$

  - Then

    $$\left\langle O \right\rangle \;\approx\; \frac{1}{n}\sum_{j=1}^{n} O(\tilde{C}_j). \tag{251}$$

  - The number of generated states n can be significantly smaller than the total number of states $m = 2^N$, e.g. $n = 10^3 \ldots 10^6$ compared to $m = 10^{3\times 10^5}$.
  - A Markov chain generates almost exclusively likely/important/representative states, which have typically small energies.
- A Markov chain is composed of
  - states $C_1, C_2, \ldots, C_m$, which are identical to the states of the system, e.g. in the case of the Ising model one state for each possible spin configuration,
  - transition probabilities $P_{j\to k}$, $j, k = 1, 2, \ldots, m$.



- Go randomly from one state to the next according to the transition probabilities $P_{j\to k}$.
  - If you are in state C_j go to state C_k with probability $P_{j\to k}$.
  - This implies

    $$\sum_k P_{j\to k} \quad = \quad 1 \quad , \quad j = 1, 2, \ldots, m \tag{252}$$

    (a necessary condition for probabilities).

- $P_l(C; C_i)$: probability to find the Markov chain after l steps in state C, when starting in state C_i.

- Define the transition probabilities $P_{j\to k}$ in such a way that the probability to find the Markov chain after many steps in state C is $P(C)$ from (250), independent of the initial state C_i, i.e.

  $$\lim_{l\to\infty} P_l(C, C_i) \quad = \quad P(C) \tag{253}$$

  (this is what we need; only then we can use (251)). $P(C)$ is then called the **stationary distribution** of the Markov chain.

- A stationary distribution implies

  $$\sum_j P(C_j) P_{j\to k} = P(C_k) \quad , \quad k = 1, 2, \ldots, m, \tag{254}$$

  i.e. the transition probabilities $P_{j\to k}$ have to fulfill (254).

- If a set of probabilities $P(C_j)$ fulfills (254), it is not guaranteed that also (253) is fulfilled. It is, however, often the case.
  In the following we assume for simplicity that (254) implies (253).

- Quite often one defines the transition probabilities $P_{j\to k}$ in such a way that **detailed balance** is fulfilled,

  $$P(C_j) P_{j\to k} = P(C_k) P_{k\to j} \quad , \quad j, k = 1, 2, \ldots, m. \tag{255}$$

  - Detailed balance (255) is a stronger condition than (254), i.e. (255) implies (254).

– Proof:

$$\sum_j \text{eq. (255)}$$

$$\rightarrow \quad \sum_j P(C_j)P_{j\to k} \quad = \quad P(C_k)\underbrace{\sum_j P_{k\to j}}_{=1} \quad = \quad P(C_k). \tag{256}$$

– Typically, it is simpler to find and define transition probabilities $P_{j\to k}$ fulfilling (255) than just (254).

- In the following we will discuss two typical Monte Carlo algorithms, i.e. suitable ways to define $P_{j\to k}$ such that (255) and, consequently, (253) are fulfilled.

12.3 Examples of common Monte Carlo algorithms

12.3.1 Metropolis algorithm

- In state $\tilde{C}_l = C_j$ randomly propose a candidate for the next state C_k with probability $W_{j\to k}$.

  – The probabilities $W_{j\to k}$ must be defined in such a way that $W_{j\to k} = W_{k\to j}$. Usually this is not difficult.

- Accept the proposed state with probability $\min(1, e^{-\beta(H(C_k)-H(C_j))})$, i.e. $\tilde{C}_{l+1} = C_k$, otherwise keep the current state, i.e. $\tilde{C}_{l+1} = C_j$ (the current state appears then again in the sum (251) used to compute $\langle O\rangle$; it is not discarded).

- Proof that detailed balance (255) is fulfilled:

$$\begin{aligned}
&\frac{e^{-\beta H(C_j)}}{Z}P_{j\to k} \quad = \quad \frac{e^{-\beta H(C_k)}}{Z}P_{k\to j} \\
\rightarrow \quad &\frac{e^{-\beta H(C_j)}}{Z}W_{j\to k}\min(1, e^{-\beta(H(C_k)-H(C_j))}) \quad = \\
&\qquad = \quad \frac{e^{-\beta H(C_k)}}{Z}W_{k\to j}\min(1, e^{-\beta(H(C_j)-H(C_k))}) \\
\rightarrow \quad &\min(e^{-\beta H(C_j)}, e^{-\beta H(C_j)}) \quad = \quad \min(e^{-\beta H(C_k)}, e^{-\beta H(C_j)}).
\end{aligned} \tag{257}$$

- In practice it is important that already a rather small number of generated states $\tilde{C}_1, \tilde{C}_2, \ldots, \tilde{C}_n$ forms a representative set of states, i.e. a set such that (251) is a good approximation (small depends on the system, could e.g. be $n = 10^3$). This requires:

  (A) The candidate for next state C_k proposed according to the probabilities $W_{j\to k}$ should be significantly different from the current state $\tilde{C}_l = C_j$.

  (B) C_k should not be rejected too often.
  $\rightarrow$ In practice, acceptance rates of $\approx 50\%$ often turned out to be a good choice.

  Note that (A) and (B) counteract each other and it is important to find a compromise.

- Advantages/disadvantages of the Metropolis algorithm:

  (+) Straightforward to implement.

  (+) Very flexible, i.e. can be applied to almost any system/partition function.

  (−) Rather slow, i.e. $\tilde{C}_l$ and $\tilde{C}_{l+1}$ tend to be similar. As a consequence, n must be large, which implies long computation times.

  (−) The efficiency strongly depends on how the probabilities $W_{j\to k}$ are chosen (numerical experiments and optimizations might be necessary).

- Application of the Metropolis algorithm to the Ising model: To propose a candidate for the next state, randomly and uniformly select $n \ll N$ spins, then randomly and uniformly select their orientation, i.e. 50% $s_j = +1$ and 50% $s_j = -1$ (this guarantees $W_{j\to k} = W_{k\to j}$).

****** January 23, 2024 (25th lecture) ******

12.3.2 Heatbath algorithm

- Randomly and uniformly select one of the degrees of freedom (in case of the Ising model one of the spins), then randomly set its value according to the probability distribution $P(C) = e^{-\beta H(C)}/Z$ (all other degrees of freedom are kept fixed):

$$P_{j\to k} \;=\; \frac{P(C_k)}{\sigma} \quad , \quad \sigma \;=\; \sum_{l'} P(C_l) \tag{258}$$

  ($\sum_{l'}$: sum over all states, which can be reached by changing the selected degree of freedom [in the case of the Ising model just two states]).

  – The probabilities $P_{j\to k}$ in (258) correspond to thermal equilibrium for this particular degree of freedom, while all others are kept fixed.

  – In other words, the selected degree of freedom is connected to a heatbath.

- Proof that detailed balance (255) is fulfilled:

$$\begin{aligned} &\frac{e^{-\beta H(C_j)}}{Z} P_{j\to k} \;=\; \frac{e^{-\beta H(C_k)}}{Z} P_{k\to j} \\ \to\quad &\frac{e^{-\beta H(C_j)}}{Z} \frac{1}{N} \frac{e^{-\beta H(C_k)}}{Z\sigma} \;=\; \frac{e^{-\beta H(C_k)}}{Z} \frac{1}{N} \frac{e^{-\beta H(C_j)}}{Z\sigma}. \end{aligned}$$

- Advantages/disadvantages of the heatbath algorithm:

  (+) Faster than the Metropolis algorithm.

  (−) The transition probabilities $P_{j\to k}$ are quite often difficult to determine/compute.
  → The heatbath algorithm is only applicable for specific, rather simple systems.

- Application of the heatbath algorithm to the Ising model: Randomly and uniformly select one of the spins, then randomly select its orientation according to the transition probabilities (258) (just two probabilities, which depend on the orientations of the $2D$ neighboring spins).

12.4 Monte Carlo simulation of the Ising model

- In this subsection:
  - $D = 2$ dimensions, $N = 20 \times 20\,\text{spins} = 400\,\text{spins}$, periodic boundary conditions.
  - $B = 0$, i.e. no external magnetic field.
  - Different temperatures T, numerically realized by different dimensionless temperatures T/λ (see section 4).
  - Heatbath algorithm.
- **Critical temperature** T_c:
  - Spontaneous magnetization for $T < T_c$, i.e. $\langle s_j \rangle \neq 0$.
  - No magnetization for $T > T_c$, i.e. $\langle s_j \rangle = 0$.
  - The temperature T_c (= "Curie temperature") separating the two phases can be calculated analytically,
    $$\frac{T_c}{\lambda} \quad = \quad \frac{2}{\ln(1+\sqrt{2})} \quad = \quad 2.269\ldots \tag{259}$$
  - Requires infinitely many spins (no phase transition for finite systems).
- **Thermalization**:
  - Run the Monte Carlo simulation for quite some time (= thermalization), such that the probabilities for the generated states are independent of the initial state (see (253) and the corresponding text).
  - To check, whether the number of thermalization steps is sufficiently large, compare independent runs with different initial states, typically
    - a **hot start** (select all spins s_j randomly),
    - a **cold start** (align all spins, i.e. set $s_j = +1$, $j = 1, \ldots, N$).

    If there is no clearly visible systematic difference in the **Monte Carlo histories** of selected "important observables" (a typical choice of such observables includes the dimensionless energy), the number of thermalization steps might be sufficiently large and one can start to use the now generated states to compute expectation values via (251).
  - Figure 16 shows Monte Carlo histories for $H/\lambda N$ for temperatures $T/\lambda = \{T_c/\lambda - 1.0\,,\, T_c/\lambda\,,\, T_c/\lambda + 1.0\}$ and both hot and cold starts.
    - Cold starts need less thermalization steps and are, thus, more efficient.
    - The system seems to thermalize particularly slow for $T \approx T_c$ (Monte Carlo simulations are typically slow, when simulating close to a phase boundary).
- **Crude determination of the critical temperature** T_c:
  - $\langle s_j \rangle$ is not ideally suited for a simple and straightforward numerical determination of the critical temperature T_c.

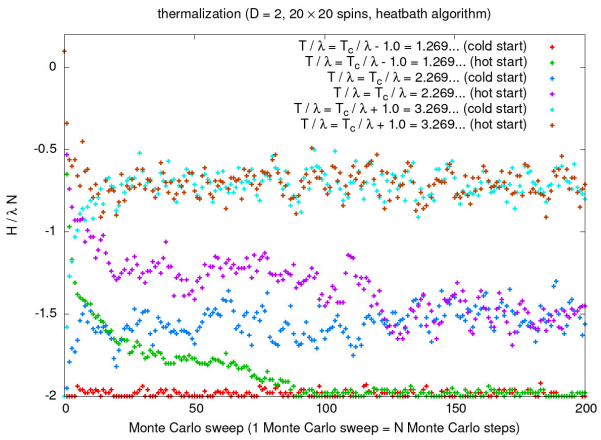


Figure 16: Monte Carlo histories for $H/\lambda N$.

– Other observables like

$$\left\langle (s_j - \overline{s})^2 \right\rangle \quad , \quad \overline{s} = \frac{1}{N} \sum_j s_j \tag{260}$$

(expectation value of the square of the deviation of a single spin from the spin average on the current configuration) also indicate the phase transition.
→ Compute $\langle (s_j - \overline{s})^2 \rangle$ as function of the temperature.

– Strong increase of $\langle (s_j - \overline{s})^2 \rangle$ in the region $2.0 \lesssim T/\lambda \lesssim 2.5$ (see Figure 17). Confirmation of the analytical result (259).

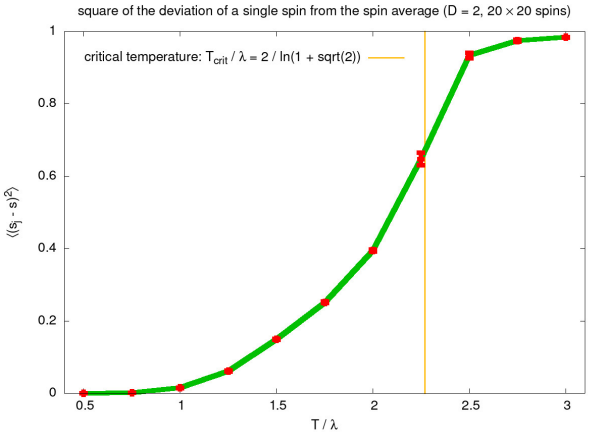


Figure 17: $\langle (s_j - \overline{s})^2 \rangle$ as function of the temperature T/λ.

– The statistical errors in Figure 17 were computed using the simple equation (228) (as samples $f^{(s)}$ one can use $f^{(s)} = (s_j - \overline{s})^2$ computed on $S = n$ spin configurations $\tilde{C}_s$ [notation not ideal: upper index $^{(s)}$ is a sample/spin configuration index; s_j and $\overline{s}$ denote spins/the spin average]; due to translational invariance one can as well use

$f^{(s)} = (1/N)\sum_j (s_j - \overline{s})^2$, which might be more efficient, i.e. smaller errors for the same number of spin configurations).

- For more complicated quantities one could e.g. use the jackknife method (see section 10.4.1).
- For a precise determination more effort is needed. In particular one needs to study several system sizes and extrapolate to $N \to \infty$.

- Individual spin configurations also exhibit characteristic features of spontaneous magnetization and the corresponding phase transition:
  - $T > T_c$, hot start, $200 \times N$ Monte Carlo steps: no spontaneous magnetization (see Figure 18, upper left plot).
  - $T < T_c$, hot start, $200 \times N$ Monte Carlo steps: spontaneous magnetization (see Figure 18, upper right plot).
  - $T < T_c$, hot start, $50 \times N$ Monte Carlo steps: a metastable state, "Weiß domains" (see Figure 18, lower plot).

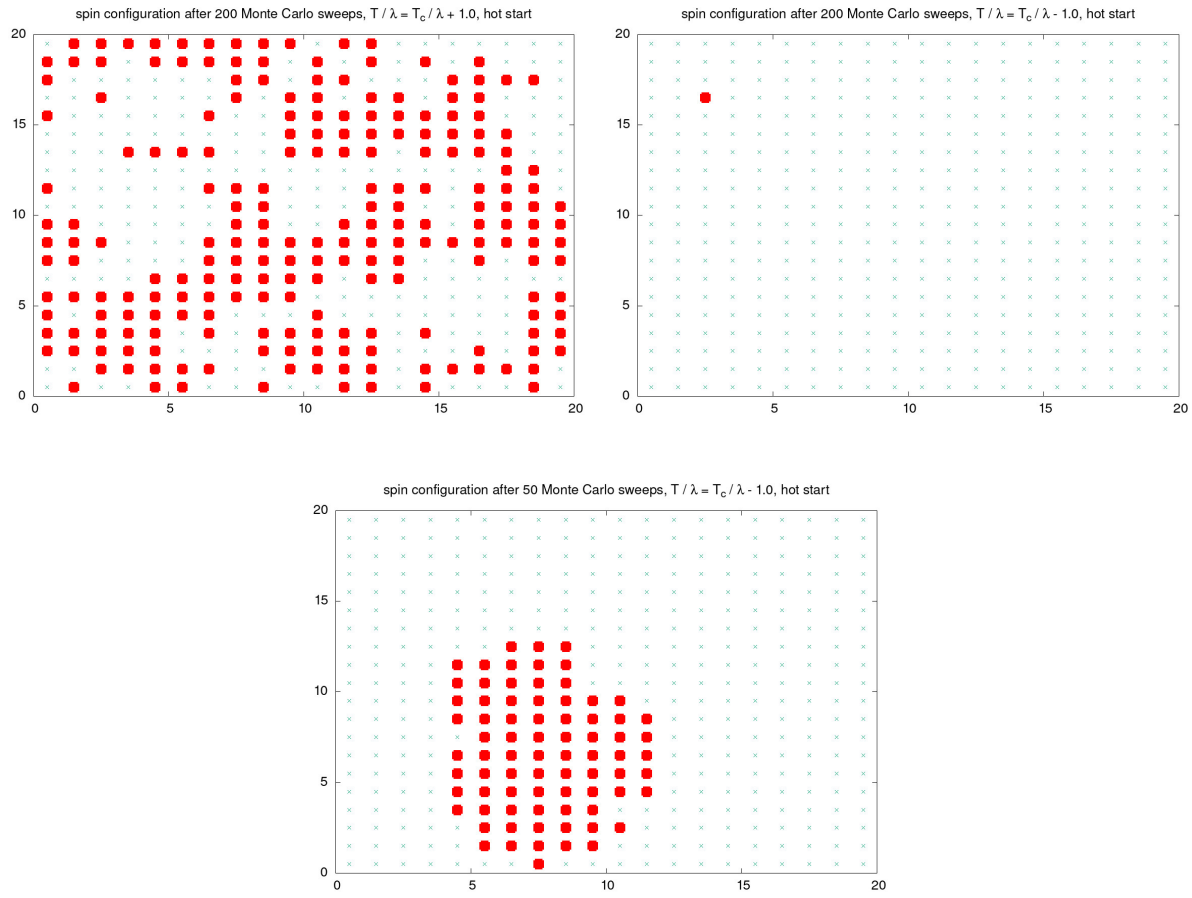


Figure 18: Individual spin configurations (red dot $\to s_j = +1$; green cross $\to s_j = -1$).

13 Partial differential equations (PDEs)

- Literature: [1, 8, 9].

13.1 Introduction

- Solving PDEs numerically is a vast subject. In this lecture only a brief introduction and discussion of selected topics.
- In physics one is often interested in linear second order PDEs,

$$\sum_{j,k} a_{jk} \frac{\partial}{\partial x_j} \frac{\partial}{\partial x_k} u + \sum_j b_j \frac{\partial}{\partial x_j} u + cu \quad = \quad f. \tag{261}$$

  - a_{jk}, b_j, c, f may depend on $x_1, x_2, \ldots$
  - a_{jk} is a symmetric matrix.
  - The goal of this section is to compute $u = u(x_1, x_2, \ldots)$ numerically.
- There are linear second order PDEs of particular interest, **hyperbolic**, **parabolic** and **elliptic** PDEs.
  - Classification according to the real eigenvalues λ_j of a_{jk}.
  - $\lambda_1 < 0$, $\lambda_j > 0$ for all $j \neq 1$ (or $\lambda_1 > 0$, $\lambda_j < 0$ for all $j \neq 1$)
    $\rightarrow$ **hyperbolic** PDEs.
    Example:
    * Wave equation,
      $$\frac{\partial^2}{\partial t^2} u(x,t) \quad = \quad v^2 \frac{\partial^2}{\partial x^2} u(x,t) \tag{262}$$
      ($v = \text{const}$: propagation speed).
  - $\lambda_1 = 0$, $\lambda_j > 0$ for all $j \neq 1$ (or $\lambda_j < 0$ for all $j \neq 1$)
    $\rightarrow$ **parabolic** PDEs.
    Examples:
    * Heat equation,
      $$\frac{\partial u(x,t)}{\partial t} \quad = \quad \alpha \frac{\partial^2 u(x,t)}{\partial x^2} \tag{263}$$
      ($\alpha > 0$; u represents the temperature).
    * Time dependent Schrödinger equation
      $$i\hbar \frac{\partial u(x,t)}{\partial t} \quad = \quad -\frac{\hbar^2}{2m} \frac{\partial^2 u(x,t)}{\partial x^2} + V(x,t) u(x,t) \tag{264}$$
      (u represents the complex wave function).
  - $\lambda_j > 0$ for all j (or $\lambda_j < 0$ for all j)
    $\rightarrow$ **elliptic** PDEs.
    Example:

* Poisson equation,
$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)u(x,y) \quad = \quad \rho(x,y) \tag{265}$$
(for example in electrostatics u represents the electric potential).

– There are different sets of methods to solve these types of PDEs. Identifying the type of a linear second order PDE is a first step to select an appropriate method.

- As in the case of ordinary differential equations, one can distinguish the following two types of problems:

  – **Initial value problems**:

    * Information is given at some initial time t_0, i.e. $u(x_1, x_2, \ldots, t = t_0)$ and possibly $\dot{u}(x_1, x_2, \ldots, t = t_0)$, together with boundary conditions with respect to $x_1, x_2, \ldots$.
    $\rightarrow$ Compute the time evolution starting at time t_0.
    * Important/problematic: stability of the method (see section 13.2.1).
    * For the wave equation (262), the heat equation (263) and the time dependent Schrödinger equation (264) one typically has to solve initial value problems.
    * In some aspects similar to initial value problems for ODEs (see section 3).

  – **Boundary value problems**:

    * Information given at the boundary of some x_1-x_2-... region.
    $\rightarrow$ Solve the PDE consistent with the boundary conditions.
    * Stability of the method is typically not a problem. Efficiency (both CPU time and storage) is the main problem. One cannot just start somewhere and compute the evolution of the function step by step. One must consider the whole x_1-x_2-... region at the same time, which leads to a large system of coupled equations.
    * For the Poisson equation (265) one typically has to solve boundary value problems (already discussed in section 7.7.2).
    * In some aspects similar to boundary value problems for ODEs (see section 6).

13.2 Initial value problems

13.2.1 Stability analysis in the context of a simple example

- Stability of the method is important/problematic for initial value problems.

- To understand the problem and the basic principle of how to analyze stability for a given method and PDE, consider the following simple example:

$$\frac{\partial u(x,t)}{\partial t} \quad = \quad -v\frac{\partial u(x,t)}{\partial x} \tag{266}$$

[6] with periodic boundary conditions in space, i.e. $u(x + L, t) = u(x, t)$. The solution is known and can be derived easily, $u(x, t) = f(x - vt)$ (see e.g. standard lectures on mechanics and electrodynamics and below).

[6] This PDE belongs to the class of flux-conservative PDEs. If possible, it is usually advantageous to cast a PDE in flux-conservative form. See e.g. [1, 8] for an extensive discussion.

- **FTCS (forward time centered space) scheme**:
  - Discretize spacetime by a uniform $(N_x+1)\times(N_t+1)$ lattice: $x \to n_x \Delta x$ (with $\Delta x = L/N_x$), $n_x = 0, 1, \ldots, N_x$, and $t \to n_t \Delta t$, $n_t = 0, 1, \ldots, N_t$.
  - Thus, $u(x,t) \to u(n_x, n_t)$.
  - Approximate the derivatives in (266) by finite differences (see section 2.3.2):
    - Use an asymmetric forward difference for the time derivative:
      $$\frac{\partial u(x,t)}{\partial t} \quad \to \quad \frac{u(n_x, n_t+1) - u(n_x, n_t)}{\Delta t} + \mathcal{O}(\Delta t). \tag{267}$$
    - Use a symmetric difference for the space derivative:
      $$\frac{\partial u(x,t)}{\partial x} \quad \to \quad \frac{u(n_x+1, n_t) - u(n_x-1, n_t)}{2\Delta x} + \mathcal{O}(\Delta x^2). \tag{268}$$
  - The discretized version of (266) is then
    $$\frac{u(n_x, n_t+1) - u(n_x, n_t)}{\Delta t} \quad = \quad -v \frac{u(n_x+1, n_t) - u(n_x-1, n_t)}{2\Delta x}. \tag{269}$$
    This equation can be rearranged in such a way that each $u(n_x, n_t+1)$ (u at time $t+\Delta t$) is expressed in terms of a few $u(n_x, n_t)$ (u at time t),
    $$u(n_x, n_t+1) \quad = \quad u(n_x, n_t) - \Delta t v \frac{u(n_x+1, n_t) - u(n_x-1, n_t)}{2\Delta x}. \tag{270}$$
    This allows to compute the time evolution step by step.
  - Even though this FTCS method might seem reasonable, it is not stable and does not work properly (see below).
- **Lax method**:
  - Very similar to the FTCS method.
  - Just replace $u(n_x, n_t)$ in (270) (the first term on the right-hand-side, i.e. one of the two terms of the asymmetric forward difference for the time derivative) by
    $$\frac{u(n_x-1, n_t) + u(n_x+1, n_t)}{2}, \tag{271}$$
    the average over the two neighboring lattice sites in space.
  - Thus,
    $$u(n_x, n_t+1) \quad = \quad \frac{u(n_x-1, n_t) + u(n_x+1, n_t)}{2} - \Delta t v \frac{u(n_x+1, n_t) - u(n_x-1, n_t)}{2\Delta x}. \tag{272}$$
  - This method is stable and works properly (for suitably chosen Δt and Δx; see below).
- **Numerical test of the FTCS scheme and the Lax method**:
  - $L = 10.0$, $N_x = 100$, i.e. $\Delta x = 0.1$.
  - $\Delta t = 0.1$.
  - $v = 1.0$.

– For each method two numerical tests with initial conditions

(A) A box-like shape,

$$u(x, t = 0) = \begin{cases} 1 & \text{if } -1.0 \leq x \leq +1.0 \\ 0 & \text{otherwise} \end{cases} . \tag{273}$$

(B) A Gaussian shape,

$$u(x, t = 0) \approx \exp\left(-\frac{x^2}{2\sigma^2} \right) \tag{274}$$

with $\sigma = 0.5$.

– Results are collected in Figure 19. The FTCS scheme fails, while the Lax method leads to results consistent with the analytical result $u(x, t) = f(x - vt)$ (the excitation at $t = 0.0$ should move in positive x direction with velocity 1.0 and keep its shape).

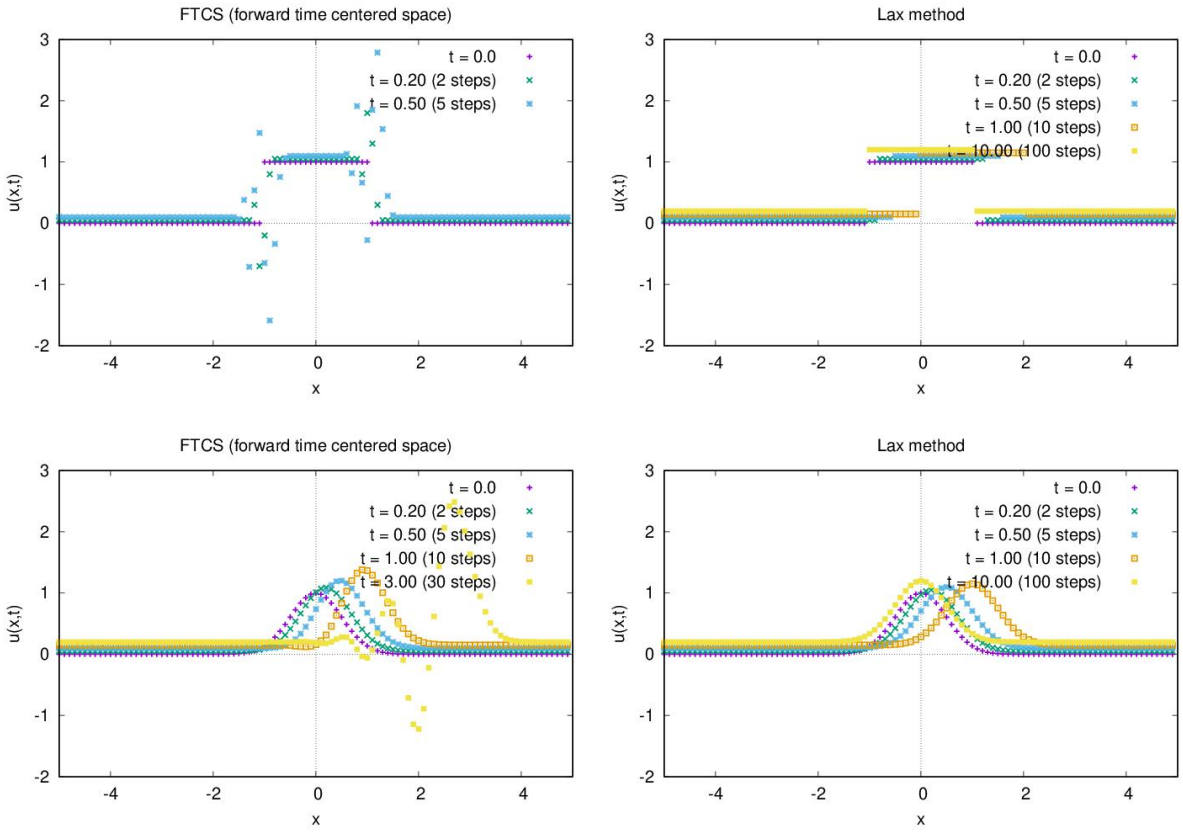


Figure 19: Comparison of the FTCS scheme and the Lax method using a box-like excitation (**top**) and a Gaussian excitation (**bottom**) as initial conditions ($L = 10.0$, $v = 1.0$, $\Delta t = \Delta x = 0.1$). The curves are slightly shifted along the vertical axis. See text for details.

- **Stability analysis**:
  - Why does the FTCS scheme fail and the Lax method work?
  - Solution of the PDE (266) in the continuum:

* Find all independent solutions of the PDE (266) using separation of variables, $u(x,t) = X(x)T(t)$:

$$\frac{\partial u(x,t)}{\partial t} = -v\frac{\partial u(x,t)}{\partial x}$$

$$\rightarrow \quad -\frac{1}{v}\frac{\dot{T}(t)}{T(t)} = \frac{X'(x)}{X(x)} = ik = \text{const}$$

$$\rightarrow \quad X(x) = e^{+ikx} \quad , \quad T(t) = e^{-ikvt}. \tag{275}$$

* General solution via linear superposition:

$$u(x,t) = \int dk\, a(k) e^{ik(x-vt)} \tag{276}$$

(the Fourier representation of the previously mentioned solution $f(x-vt)$).

* Determine $a(k)$ such that initial conditions and boundary conditions are fulfilled.
* Note that, in general, all/many independent solutions $e^{ik(x-vt)}$ contribute to (276), i.e. have non-vanishing coefficients $a(k)$.

– One can carry out similar steps for the FTCS discretized version of the PDE, i.e. for (269):

* Independent solutions:

$$u(n_x, n_t) = \xi^{n_t} e^{ikn_x\Delta x} \quad , \quad \xi = 1 - i\frac{v\Delta t \sin(k\Delta x)}{\Delta x} \tag{277}$$

(in the limit $\Delta x \rightarrow 0$, $\Delta t \rightarrow 0$ [corresponding to $n_x \rightarrow \infty$, $n_t \rightarrow \infty$] the continuum result is recovered, i.e. (277) becomes $e^{ik(x-vt)}$ [to show that, $\lim_{n\rightarrow\infty}(1+x/n)^n = e^x$ is useful]).

* General solution via linear superposition:

$$u(n_x, n_t) = \sum_k a(k)\xi^{n_t} e^{ikn_x\Delta x}. \tag{278}$$

* Determine $a(k)$ such that initial conditions and boundary conditions are fulfilled.
* Again, all/many independent solutions $\xi^{n_t} e^{ikn_x\Delta x}$ contribute to (278), i.e. have non-vanishing coefficients $a(k)$.

– The reason, why the FTCS scheme is not suited to compute the time evolution is the term ξ^{n_t}, because in the FTCS scheme $|\xi| > 1$. Thus, each of the independent solutions appearing in (278) is amplified exponentially with increasing time resulting in the weird and unphysical time evolution shown in the plots in the left column of Figure 19.

– Note that there is no such exponential amplification in the continuum result (276), because the the term corresponding to ξ^{n_t} is $e^{-ikvt} = (e^{-ikv})^t$ and $|e^{-ikv}| = 1$.

****** February 01, 2024 (28th lecture) ******

– From

$$\xi^{n_t} = \left(1 - i\frac{v\Delta t \sin(k\Delta x)}{\Delta x}\right)^{n_t} = \left(1 - ikv\Delta t + \mathcal{O}(\Delta x^2)\right)^{n_t} =$$

$$= \left(e^{-ikv\Delta t} + \mathcal{O}(\Delta x^2) + \mathcal{O}(\Delta t^2)\right)^{n_t} \tag{279}$$

one can see that in the FTCS scheme the small discretization errors in ξ, which are $\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t^2)$, cause the exponential amplifications. Thus, at large n_t numerical FTCS solutions of the PDE (266) are completely dominated by discretization errors.

– This type of stability analysis is called **von Neumann stability analysis** and ξ is called **amplification factor**.

– One can do the same von Neumann stability analysis for the Lax method. The corresponding amplification factor is

$$\xi \quad = \quad \cos(k\Delta x) - i\frac{v\Delta t \sin(k\Delta x)}{\Delta x}. \tag{280}$$

The important difference is that $|\xi|$ can be less or greater than 1 depending on Δx and Δt:

$$|\xi| \quad \begin{cases} \leq & 1 \quad \text{if } v\Delta t/\Delta x \leq 1 \\ > & 1 \quad \text{otherwise} \end{cases} . \tag{281}$$

Thus, for $v\Delta t/\Delta x \leq 1$ one expects the Lax method to be stable, otherwise not.

– For the above example (Figure 19, right column) we used $v = 1.0$, $\Delta t = \Delta x = 0.1$ corresponding to $v\Delta t/\Delta x = 1.0$, i.e. we operated just inside the stable region.

– Numerical experiment:

  * Repeat the Lax computation with $L = 10.0$, $v = 1.0$, $\Delta x = 0.1$ (as before), but $\Delta t = 0.11$ (i.e. a slightly larger discretization step in t direction). Then $|\xi| > 1$.
  * Numerical results are shown in Figure 20. They are similar to those obtained with the FTCS scheme, i.e. not useful. This confirms our expectation from the stability analysis and highlights the importance of having an amplification factor $\xi \leq 1$.

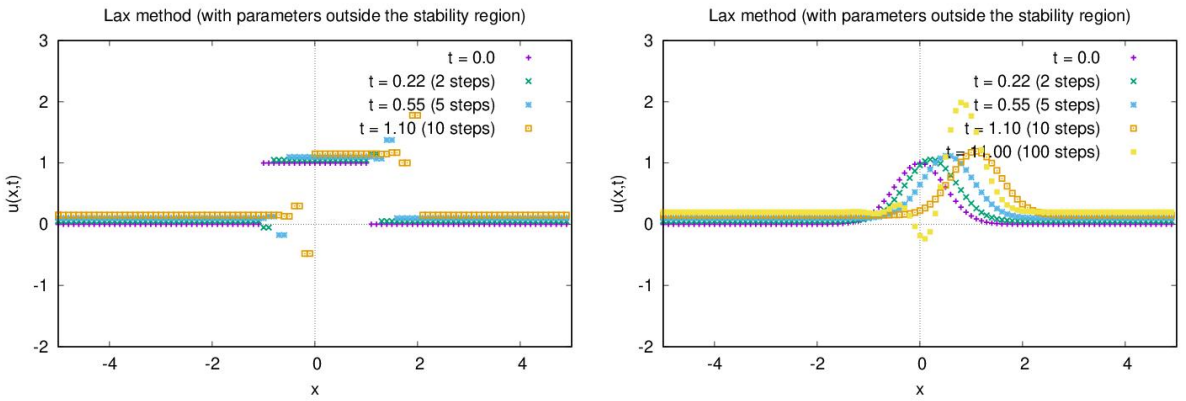


Figure 20: Lax method with parameters outside the stability region using a box-like excitation (**top**) and a Gaussian excitation (**bottom**) as initial conditions ($L = 10.0$, $v = 1.0$, $\Delta t = 0.11$, $\Delta x = 0.1$). See text for details.

– What about amplification factors < 1? Isn't there an exponential suppression and the numerical solution will quickly approach 0?

* We are interested in scales $d \gg \Delta x$, e.g. the intial conditions can exhibit spatial variations of extent $\gg \Delta x$, but not of extent $\lesssim \Delta x$ (for a given physics problem, one has to choose a sufficiently small Δx).
* Consequently, the relevant independent solutions (see e.g. (277)), i.e. those with large prefactors in (276) or (278), have $1/k \gtrsim d \gg \Delta x$, i.e. $k\Delta x \ll 1$. Independent solutions with $k\Delta x \gtrsim 1$ are not important, i.e. have small prefactors in (276) or (278).
* Amplification factors for the Lax method:

$$|\xi|^2 \;=\; 1 - \left(1 - \left(\frac{kv\Delta t}{k\Delta x}\right)^2\right) \sin^2(k\Delta x). \tag{282}$$

  · Relevant independent solutions, which have $k\Delta x \ll 1$
  → $|\xi|$ rather close to 1 (because $\sin^2(k\Delta x) \approx 0$).
  · Irrelevant independent solutions, which have $k\Delta x \gtrsim 1$
  → $|\xi|$ not close to 1.
  · Lax method inside the stability region, i.e. for $v\Delta t/\Delta x < 1$
  → irrelevant independent solutions are strongly suppressed (which is irrelevant)
  → relevant independent solutions suffer from a much weaker suppression, i.e. computing the time evolution over a reasonably large period of time is possible.
  · Lax method outside the stability region, i.e. for $v\Delta t/\Delta x > 1$
  → irrelevant independent solutions are strongly enhanced and quickly spoil the result.

  → $|\xi| < 1$ is a mild problem, compared to $|\xi| > 1$.
* From (282) one can see that in addition to $k\Delta x \ll 1$ also $v\Delta t \approx \Delta x$ favors amplification factors close to 1.
  · One should choose Δt and Δx accordingly.
  · If not, the solution might exhibit sizable errors due to exponential suppression already at small t. An example with $L = 10.0$, $v = 1.0$ and $\Delta x = 0.1$ (as before) and $\Delta t = 0.01$, i.e. $v\Delta t/\Delta x = 0.1$ is shown in Figure 21.

- **Final remarks and summary**:
  - The von Neumann stability analysis can be generalized to other more complicated PDEs. There are also more rigorous methods to carry out stability analyses. See [1].
  - The aim of this section is not to provide a comprehensive discussion of the topic, but rather
    * to demonstrate that stability of the method is crucial, when solving PDEs numerically,
    * to show that *"differencing PDEs is an art as much as a science"* [1] (stability depends both on the method and on the PDE).
  - [1], section 20.1.4: *"In summary, our recommendation for initial value problems that can be cast in flux-conservative form, and especially problems related to the wave equation, is to use the staggered leapfrog method [not discussed in this lecture] when*

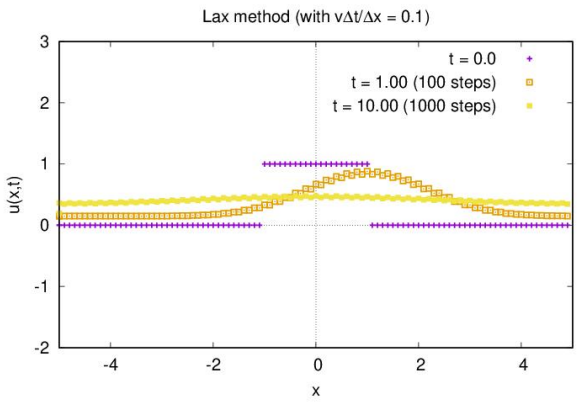


Figure 21: Lax method with parameters inside the stability region but $v\Delta t/\Delta x = 0.1$ using a box-like excitation as initial conditions ($L = 10.0$, $v = 1.0$, $\Delta t = 0.01$, $\Delta x = 0.1$). See text for details.

possible. We have personally had better success with it than with the two-step Lax-Wendroff method [not discussed in this lecture] ...”

→ Even for numerics experts choosing an appropriate method does not seem to be obvious, but might be based on experience, numerical experiments and to some extent guesswork.

13.2.2 Parabolic PDEs: heat equation and Schrödinger equation

- Consider the PDE

$$\frac{\partial u(x,t)}{\partial t} = \alpha \frac{\partial^2 u(x,t)}{\partial x^2} + v(x,t)u(x,t) \tag{283}$$

with initial conditions $u(x, t = 0) = u_0(x)$ and boundary conditions $u(x = 0, t) = \tilde{u}_0(x)$ and $u(x = L, t) = \tilde{u}_L(t)$. The goal is to compute the time evolution of $u(x, t)$ for $t > 0$ and $0 < x < L$.

- (283) contains both the heat equation (263) and the time dependent Schrödinger equation (264).

- A generalization of (283) and this subsection to more than one spatial coordinate, i.e. $x \to x_1, x_2, \ldots$, is straightforward.

****** February 06, 2024 (29th lecture) ******

Discretization via an asymmetric forward difference for the time derivative

- Conceptually similar to the FTCS method from section 13.2.1.

- The discretized version of (283), when using an asymmetric forward difference for the time derivative (and a standard symmetric difference for the second space derivative) is

$$\begin{aligned}\frac{u(n_x, n_t+1) - u(n_x, n_t)}{\Delta t} &= \\ = \alpha \frac{u(n_x+1, n_t) - 2u(n_x, n_t) + u(n_x-1, n_t)}{\Delta x^2} &+ v(n_x, n_t)u(n_x, n_t). \end{aligned} \tag{284}$$

- Left hand side:

$$\frac{u(n_x, n_t+1) - u(n_x, n_t)}{\Delta t} = \frac{\partial u(x,t)}{\partial t} + \frac{1}{2}\frac{\partial^2 u(x,t)}{\partial t^2}\Delta t + \mathcal{O}(\Delta t^2). \tag{285}$$

- Right hand side:

$$\begin{aligned}\alpha \frac{u(n_x+1, n_t) - 2u(n_x, n_t) + u(n_x-1, n_t)}{\Delta x^2} + v(n_x, n_t)u(n_x, n_t) &= \\ = \alpha \frac{\partial^2 u(x,t)}{\partial x^2} + v(x,t)u(x,t) + \mathcal{O}(\Delta x^2). \end{aligned} \tag{286}$$

- Consequently, the discretization (284) has errors of order Δt and of order Δx^2.

- (284) can be rearranged in such a way that each $u(n_x, n_t+1)$ (u at time $t+\Delta t$) is expressed in terms of a few $u(n_x, n_t)$ (u at time t),

$$\begin{aligned}u(n_x, n_t+1) &= \\ = \left(1 - \frac{2\alpha\Delta t}{\Delta x^2} + \Delta t v(n_x, n_t)\right) u(n_x, n_t) &+ \frac{\alpha \Delta t}{\Delta x^2}\Big(u(n_x+1, n_t) + u(n_x-1, n_t)\Big). \end{aligned} \tag{287}$$

  This allows to compute the time evolution step by step. Such schemes are called *fully explicit*.

- Stability analysis (for the case $v(x,t) = 0$): amplification factor

$$\xi = 1 - \frac{4\alpha\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right), \tag{288}$$

  i.e. the method is stable for $|\xi| \leq 1$ corresponding (for real $\alpha > 0$, as it is the case for the heat equation) to $2\alpha\Delta t/\Delta x^2 \leq 1$.

(−) Errors of order Δt (because of asymmetric forward difference).
(Errors of order Δx^2 appear only quadratically and are, thus, less problematic.)

(−) Stable only for very small $\Delta t \lesssim \Delta x^2$.

Discretization via an asymmetric backward difference for the time derivative

- The discretized version of (283), when using an asymmetric backward difference for the time derivative (and a standard symmetric difference for the second space derivative) is

$$\begin{aligned}\frac{u(n_x,n_t+1)-u(n_x,n_t)}{\Delta t} &= \\ = \alpha\frac{u(n_x+1,n_t+1)-2u(n_x,n_t+1)+u(n_x-1,n_t+1)}{\Delta x^2} &+ \\ +v(n_x,n_t+1)u(n_x,n_t+1). &\end{aligned} \tag{289}$$

  (in contrast to (283) the time argument is not t, but $t+\Delta t=(n_t+1)\Delta t$; this is irrelevant in this subsection, but helpful in the context of the Crank-Nicolson method discussed in the next subsection).

- Left hand side:

$$\frac{u(n_x,n_t+1)-u(n_x,n_t)}{\Delta t} = \frac{\partial u(x,t+\Delta t)}{\partial t}-\frac{1}{2}\frac{\partial^2 u(x,t+\Delta t)}{\partial t^2}\Delta t+\mathcal{O}(\Delta t^2). \tag{290}$$

- Right hand side:

$$\begin{aligned}\alpha\frac{u(n_x+1,n_t+1)-2u(n_x,n_t+1)+u(n_x-1,n_t+1)}{\Delta x^2}+v(n_x,n_t+1)u(n_x,n_t+1) &= \\ = \alpha\frac{\partial^2 u(x,t+\Delta t)}{\partial x^2}+v(x,t+\Delta t)u(x,t+\Delta t)+\mathcal{O}(\Delta x^2). &\end{aligned} \tag{291}$$

- Consequently, the discretization (289) has errors of order Δt and of order Δx^2.

- Note that (289) cannot be rearranged in a simple way such that each $u(n_x,n_t+1)$ (u at time $t+\Delta t$) is expressed in terms of a few $u(n_x,n_t)$ (u at time t). Such schemes are called *fully implicit* or *backward time*.

- (289) can easily be solved with respect to $u(n_x,n_t)$,

$$\begin{aligned}u(n_x,n_t) &= \Big(1+\frac{2\alpha\Delta t}{\Delta x^2}-\Delta t v(n_x,n_t+1)\Big)u(n_x,n_t+1)- \\ &-\frac{\alpha\Delta t}{\Delta x^2}\Big(u(n_x+1,n_t+1)+u(n_x-1,n_t+1)\Big).\end{aligned} \tag{292}$$

- This equation can be expressed in matrix-vector form,

$$u(n_x,n_t) = \sum_{n_x'}\mathcal{A}(n_x,n_x')u(n_x',n_t+1) \tag{293}$$

  ($u(n_x,n_t)$ and $u(n_x,n_t+1)$ are vectors with $N_x-1=L/\Delta x-1$ components, $\mathcal{A}(n_x,n_x')$ is an $(N_x-1)\times(N_x-1)$ matrix). The entries of $\mathcal{A}$ can be read off from (292).

- To compute $u(n_x,n_t+1)$, one has to solve the system of linear equations (293), e.g. using the LU decomposition or a suitable iterative method as discussed in section 7.

- Stability analysis (for the case $v(x,t) = 0$): amplification factor

$$\xi \quad = \quad \left(1 + 4\alpha \sin^2\left(\frac{k\Delta x}{2}\right)\right)^{-1}, \tag{294}$$

i.e. the method is unconditionally stable (for $\mathrm{Re}(\alpha) > 0$, as it is the case for the heat equation), because $|\xi| \leq 1$ for any stepsize Δt.

(−) Errors of order Δt (because of asymmetric forward difference).
(Errors of order Δx^2 appear only quadratically and are, thus, less problematic.)

(+) Unconditionally stable.

Crank-Nicolson method

- Add the PDE (283) at time t and at time $t + \Delta t$,

$$\begin{aligned}\frac{\partial u(x,t)}{\partial t} + \frac{\partial u(x,t+\Delta t)}{\partial t} \quad &= \\ = \quad \alpha\frac{\partial^2 u(x,t)}{\partial x^2} + v(x,t)u(x,t) + \alpha\frac{\partial^2 u(x,t+\Delta t)}{\partial x^2} + v(x,t+\Delta t)u(x,t+\Delta t).\end{aligned} \tag{295}$$

- Discretize (295) using (284) (forward difference for the time derivative) and (289) (backward difference for the time derivative). The left hand side has errors of order Δt^2, because errors proportional to Δt cancel (as can be seen from (285) and (290)), while the right hand side has errors of order Δx^2.

- This discretization can be also be obtained (in a more covenient form) by adding (287) and (292) (which are equivalent to (284) and (289)):

$$\begin{aligned}&\left(2 + \frac{2\alpha\Delta t}{\Delta x^2} - \Delta t v(n_x, n_t+1)\right)u(n_x, n_t+1) - \\ &\qquad -\frac{\alpha\Delta t}{\Delta x^2}\Big(u(n_x+1, n_t+1) + u(n_x-1, n_t+1)\Big) \quad = \\ &= \quad \underbrace{\left(2 - \frac{2\alpha\Delta t}{\Delta x^2} + \Delta t v(n_x, n_t)\right)u(n_x, n_t) + \frac{\alpha\Delta t}{\Delta x^2}\Big(u(n_x+1, n_t) + u(n_x-1, n_t)\Big)}_{=b(n_x) \text{ for } n_x>0 \text{ and } n_x<N_x \text{ (for } n_x=1 \text{ and } n_x=N_x-1 \text{ additional contributions from the lhs)}}\end{aligned} \tag{296}$$

$(n_x = 1, 2, \ldots, N_x - 1)$.

- This equation can be expressed in matrix-vector form,

$$\sum_{n'_x} \mathcal{B}(n_x, n'_x)u(n'_x, n_t+1) \quad = \quad b(n_x) \tag{297}$$

($u(n_x, n_t+1)$ and $b(n_x)$ are vectors with $N_x - 1 = L/\Delta x - 1$ components, $\mathcal{B}(n_x, n'_x)$ is an $(N_x - 1) \times (N_x - 1)$ matrix). The entries of $\mathcal{B}$ and b can be read off from (296).

- To compute $u(n_x, n_t + 1)$, one has to solve the system of linear equations (297), e.g. using the LU decomposition or a suitable iterative method as discussed in section 7.
- Stability analysis (for the case $v(x,t) = 0$): amplification factor

$$\xi = \left(1 - 2\alpha \sin^2\left(\frac{k\Delta x}{2}\right)\right)\left(1 + 2\alpha \sin^2\left(\frac{k\Delta x}{2}\right)\right)^{-1}, \tag{298}$$

  i.e. the method is unconditionally stable (for $\mathrm{Re}(\alpha) > 0$, as it is the case for the heat equation), because $|\xi| \leq 1$ for any stepsize Δt.

(+) Errors only of order Δt^2.
(And errors of order Δx^2, as before, which are also quadratically suppressed.)

(+) Unconditionally stable.

Example: solving the heat equation with the Crank-Nicolson method

- Consider the heat equation (263),

$$\frac{\partial u(x,t)}{\partial t} = \alpha \frac{\partial^2 u(x,t)}{\partial x^2}, \tag{299}$$

  with initial conditions $u(x, t = 0) = u_0(x)$ and boundary conditions $u(x = 0, t) = 0$ and $u(x = L, t) = T$.
- Dimensionless heat equation:
  - Dimensionless temperature: $\hat{u} = u/T$ (in the following $\hat{u}$ is denoted as u).
  - Dimensionless space coordinate: $\hat{x} = x/L$.
    $\rightarrow \partial/\partial x = (1/L)\partial/\partial\hat{x}$.
  - Dimensionless time coordinate: $\hat{t} = \alpha t/L^2$.
    $\rightarrow \partial/\partial t = (\alpha/L^2)\partial/\partial\hat{t}$.
  - Inserting these definitions in (299) leads to

$$\frac{\partial u(\hat{x},\hat{t})}{\partial \hat{t}} = \frac{\partial^2 u(\hat{x},\hat{t})}{\partial \hat{x}^2} \tag{300}$$

    and boundary conditions $u(\hat{x} = 0, \hat{t}) = 0$ and $u(\hat{x} = 1, \hat{t}) = 1$.
- (300) can be solved analytically using the ansatz $u(\hat{x},\hat{t}) = X(\hat{x})T(\hat{t})$,

$$u(\hat{x},\hat{t}) = \hat{x} + \sum_{n=1}^{\infty} a_n \sin(n\pi\hat{x}) e^{-n^2\pi^2\hat{t}}, \tag{301}$$

  where the coefficients a_n have to be determined such that the initial conditions $u(\hat{x}, \hat{t} = 0) = u_0(\hat{x})$ are fulfilled (the boundary conditions are already implemented).

- Numerically we consider two concrete examples for the initial conditions $u_0(\hat{x})$:
  - Example 1:
    $u_0(\hat{x}) = \hat{x} - (1/2\pi)\sin(2\pi\hat{x})$, i.e. $a_2 = -1/2\pi$ and $a_n = 0$ for $n \neq 2$
    (see Figure 22, left plot).
  - Example 2:
    $u_0(\hat{x}) = \hat{x} + 2\sin(3\pi\hat{x})$, i.e. $a_3 = 2$ and $a_n = 0$ for $n \neq 3$
    (see Figure 22, right plot).

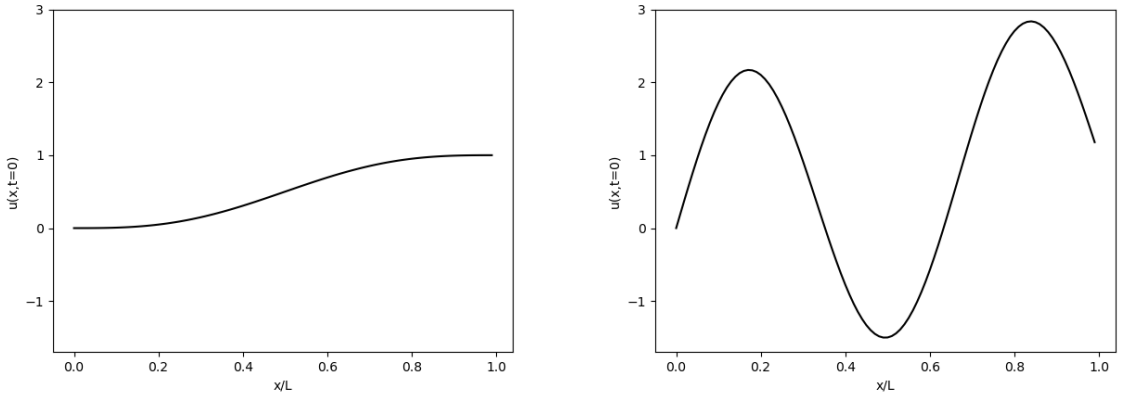


Figure 22: Temperature $u_0(\hat{x})$ at time $\hat{t} = 0$. **(left)** Example 1. **(right)** Example 2.

- Compute the time evolution with the Crank-Nicolson method using $N_x = 40$, $N_t = 40$ and $\Delta t/\Delta x = 0.1$.
  - Python code to solve the heat equation with the Crank-Nicolson method: see appendix G.
  - The numerical solutions are rather close to the analytical solutions (301).

```
example 1, Nx = 40, Nt = 40, Dt_over_Dx = 0.1
Crank-Nicolson step   0, t =  0.000000 --> err_max = 0.00000e+00
Crank-Nicolson step   1, t =  0.002500 --> err_max = 1.77416e-05
Crank-Nicolson step   2, t =  0.005000 --> err_max = 3.21505e-05
Crank-Nicolson step   3, t =  0.007500 --> err_max = 4.36960e-05
...
Crank-Nicolson step  38, t =  0.095000 --> err_max = 1.75319e-05
Crank-Nicolson step  39, t =  0.097500 --> err_max = 1.63033e-05
Crank-Nicolson step  40, t =  0.100000 --> err_max = 1.51507e-05

example 2, Nx = 40, Nt = 40, Dt_over_Dx = 0.1
Crank-Nicolson step   0, t =  0.000000 --> err_max = 0.00000e+00
Crank-Nicolson step   1, t =  0.002500 --> err_max = 1.90334e-04
Crank-Nicolson step   2, t =  0.005000 --> err_max = 3.04881e-04
Crank-Nicolson step   3, t =  0.007500 --> err_max = 3.66273e-04
...
```

```
Crank-Nicolson step  38, t =  0.095000 --> err_max = 1.95838e-06
Crank-Nicolson step  39, t =  0.097500 --> err_max = 1.60977e-06
Crank-Nicolson step  40, t =  0.100000 --> err_max = 1.32234e-06
```

– The numerical solutions are shown as heatmaps in Figure 23

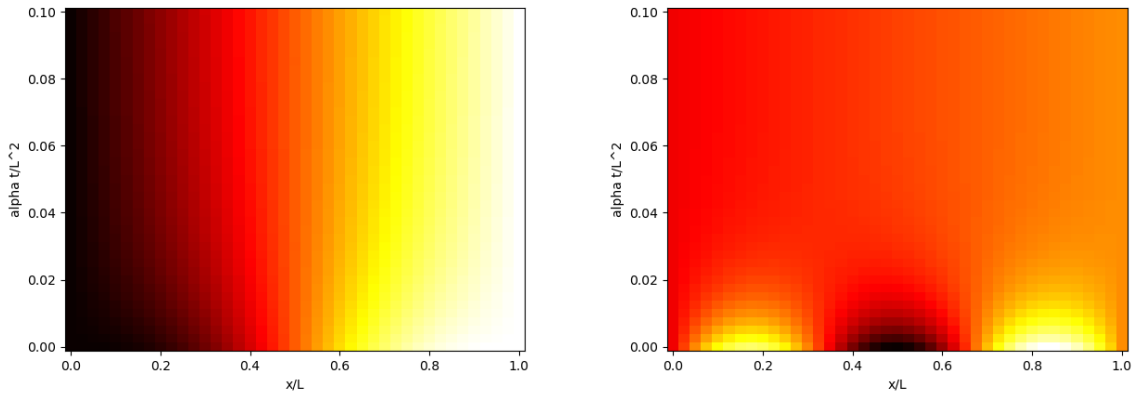


Figure 23: Temperature $u(x,t)$ shown as normalized heatmap (minimal value: black; maximal value: white; colors on the left and right boundaries correspond to $u(x=0,t)=0$ and $u(x=L,t)=1$, respectively). $N_x = 40$, $N_t = N_x$ and $\Delta t/\Delta x = 0.1$. **(left)** Example 1. **(right)** Example 2.

- Vary Δx and Δt: $N_x \in \{20, 40, 80\}$, $N_t = N_x$ and $\Delta t/\Delta x = 0.1$.
  - The discretization errors (i.e. the differences between the numerical solutions and the analytical solution (301)) become smaller with decreasing Δx and Δt as expected. The suppression of errors seems to be even stronger than quadratically in Δx and Δt.

```
example 1, Nx = 20, Nt = 20, Dt_over_Dx = 0.1
Crank-Nicolson step   0, t =  0.000000 --> err_max = 0.00000e+00
Crank-Nicolson step   1, t =  0.005000 --> err_max = 1.29304e-04
Crank-Nicolson step   2, t =  0.010000 --> err_max = 2.12388e-04
Crank-Nicolson step   3, t =  0.015000 --> err_max = 2.61643e-04
...
Crank-Nicolson step  18, t =  0.090000 --> err_max = 8.18830e-05
Crank-Nicolson step  19, t =  0.095000 --> err_max = 7.09846e-05
Crank-Nicolson step  20, t =  0.100000 --> err_max = 6.13662e-05

example 1, Nx = 40, Nt = 40, Dt_over_Dx = 0.1
Crank-Nicolson step   0, t =  0.000000 --> err_max = 0.00000e+00
Crank-Nicolson step   1, t =  0.002500 --> err_max = 1.77416e-05
Crank-Nicolson step   2, t =  0.005000 --> err_max = 3.21505e-05
Crank-Nicolson step   3, t =  0.007500 --> err_max = 4.36960e-05
```

```
...
Crank-Nicolson step  38, t =  0.095000 --> err_max = 1.75319e-05
Crank-Nicolson step  39, t =  0.097500 --> err_max = 1.63033e-05
Crank-Nicolson step  40, t =  0.100000 --> err_max = 1.51507e-05

example 1, Nx = 80, Nt = 80, Dt_over_Dx = 0.1
Crank-Nicolson step   0, t =  0.000000 --> err_max = 0.00000e+00
Crank-Nicolson step   1, t =  0.001250 --> err_max = 2.32678e-06
Crank-Nicolson step   2, t =  0.002500 --> err_max = 4.42953e-06
Crank-Nicolson step   3, t =  0.003750 --> err_max = 6.32442e-06
...
Crank-Nicolson step  78, t =  0.097500 --> err_max = 4.06328e-06
Crank-Nicolson step  79, t =  0.098750 --> err_max = 3.91724e-06
Crank-Nicolson step  80, t =  0.100000 --> err_max = 3.77586e-06
```

– The numerical solutions are shown as heatmaps in Figure 24

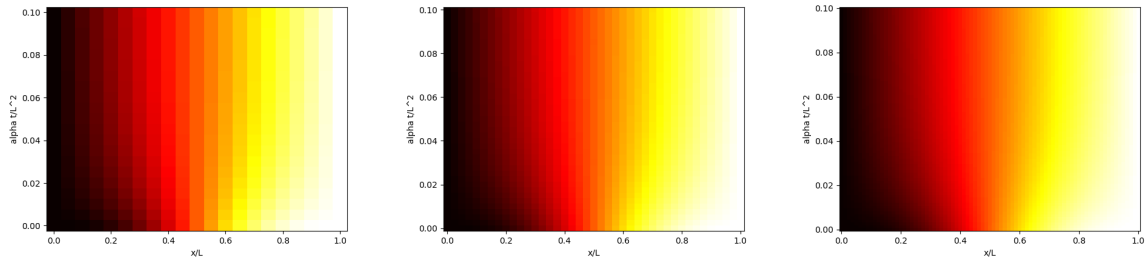


Figure 24: Temperature $u(x,t)$ shown as normalized heatmap (minimal value: black; maximal value: white; colors on the left and right boundaries correspond to $u(x=0,t)=0$ and $u(x=L,t)=1$, respectively) for example 1. $N_t = N_x$ and $\Delta t/\Delta x = 0.1$. **(left)** $N_x = 20$. **(center)** $N_x = 40$. **(right)** $N_x = 80$.

- Optional homework:
  - Explore $\Delta x \neq \Delta t$.
  - Implement the discretization via an asymmetric forward difference for the time derivative. Confirm that it is not stable, if Δx and Δt are not chosen appropriately. Use stable parameters Δx and Δt and compare the accuracy and necessary CPU time to the Crank-Nicolson method.
  - Solve the time dependent Schrödinger equation (264) with the Crank-Nicolson method.

13.3 Boundary value problems

- A possible strategy to solve the Poisson equation numerically (a typical boundary value problem for an elliptic PDE) has already been discussed in section 7.7.2.
- Due to limited time no further discussion of boundary value problems.

A C Code: trajectories for the HO with the RK method

```
// solve system of ODEs
// \vec{\dot{y}}(t) = \vec{f}(\vec{y}(t),t) ,
// initial conditions
// \vec{y}(t=0) = \vec{y}_0 ,
// HO, potential
// V(x) = m \omega^2 x^2 / 2

// **********

#define __EULER__
// #define __RK_2ND__
// #define __RK_3RD__
// #define __RK_4TH__

// **********

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// **********

const int N = 2;  // number of components of \vec{y} and \vec{f}

const double omega = 1.0;  // frequency

const int num_steps = 10000;  // number of steps
const double tau = 0.1;  // step size

// **********

double y[N][num_steps+1];  // discretized trajectories

double y_0[N] = { 1.0 , 0.0 };  // initial conditions

// **********

int main(int argc, char **argv)
{
  int i1, i2;

  // *****

  // initialize trajectories with initial conditions

  for(i1 = 0; i1 < N; i1++)
    y[i1][0] = y_0[i1];

  // *****

  // Euler/RK steps

  for(i1 = 1; i1 <= num_steps; i1++)
```

```
    {
      // 1D HO:
      // y(t) = (x(t) , \dot{x}(t)) ,
      // \dot{y}(t) = f(y(t),t) = (\dot{x}(t) , F/m) ,
      //   where force F = -m \omega^2 x(t)

#ifdef __EULER__

      // k1 = f(y(t),t) * tau

      double k1[N];

      k1[0] = y[1][i1-1]   * tau;
      k1[1] = -pow(omega, 2.0) * y[0][i1-1]   * tau;

      // *****

      for(i2 = 0; i2 < N; i2++)
        y[i2][i1] = y[i2][i1-1] + k1[i2];

#endif

#ifdef __RK_2ND__

      // k1 = f(y(t),t) * tau

      double k1[N];

      k1[0] = y[1][i1-1]   * tau;
      k1[1] = -pow(omega, 2.0) * y[0][i1-1]   * tau;

      // *****

      // k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau

      double k2[N];

      k2[0] = (y[1][i1-1] + 0.5*k1[1])   * tau;
      k2[1] = -pow(omega, 2.0) * (y[0][i1-1] + 0.5*k1[0])   * tau;

      // *****

      for(i2 = 0; i2 < N; i2++)
        y[i2][i1] = y[i2][i1-1] + k2[i2];

#endif

#ifdef __RK_3RD__

...

#endif

#ifdef __RK_4TH__

...
```

```
#endif
    }

  // *****

  // output

  for(i1 = 0; i1 <= num_steps; i1++)
    {
      double t = i1 * tau;
      printf("%9.6lf %9.6lf %9.6lf\n", t, y[0][i1], y[0][i1]-cos(t));
    }

  // *****

  return EXIT_SUCCESS;
}
```

B C Code: trajectories for the anharmonic oscillator with the RK method with adaptive step size

---

```
// solve system of ODEs
// \vec{\dot{y}}(t) = \vec{f}(\vec{y}(t),t) ,
// initial conditions
// \vec{y}(t=0) = \vec{y}_0 ,
// anharmonic oscillator, potential
// V(x) = m \alpha x^n ,
// adaptive stepsize

// **********

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// **********
// physics parameters and functions
// **********

// anharmonic oscillator, V(x) = m \alpha x^n,
//   y = (x , v)
//   f = (v , -\alpha n x^n-1)

const int N = 2;  // number of components of \vec{y} and \vec{f}

// const int n = 2;
// const double alpha = 0.5;
const int n = 20;
const double alpha = 1.0;

double y_0[N] = { 1.0 , 0.0 };  // initial conditions

// function computing f(y(t),t) * tau

void f_times_tau(double *y_t, double t, double *f_times_tau_, double tau)
{
  if(N != 2)
    {
      fprintf(stderr, "Error: N != 2!\n");
      exit(EXIT_FAILURE);
    }

  f_times_tau_[0] = y_t[1] * tau;
  f_times_tau_[1] = -alpha * ((double)n) * pow(y_t[0], ((double)(n-1))) * tau;
}

// **********
// RK parameters
// **********

// #define __EULER__
#define __RK_2ND__
```

```
// #define __RK_3RD__
// #define __RK_4TH__

#ifdef __EULER__
const int order = 1;
#endif

#ifdef __RK_2ND__
const int order = 2;
#endif

#ifdef __RK_3RD__
const int order = 3;
#endif

#ifdef __RK_4TH__
const int order = 4;
#endif

// maximum number of steps
const int num_steps_max = 10000;

// compute trajectory for 0 <= t <= t_max
const double t_max = 10.0;

// maximum tolerable error
const double delta_abs_max = 0.001;

double tau = 1.0;  // initial step size

// **********

double t[num_steps_max+1];  // discretized time
double y[num_steps_max+1][N];  // discretized trajectories

// **********

#ifdef __EULER__

...

#endif

#ifdef __RK_2ND__

// RK step (2nd order), step size tau

void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
{
  int i1;

  // *****

  // k1 = f(y(t),t) * tau

  double k1[N];
  f_times_tau(y_t, t, k1, tau);
```

```
  // *****

  // k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau

  double y_[N];

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + 0.5*k1[i1];

  double k2[N];
  f_times_tau(y_, t + 0.5*tau, k2, tau);

  // *****

  for(i1 = 0; i1 < N; i1++)
    y_t_plus_tau[i1] = y_t[i1] + k2[i1];
}

#endif

#ifdef __RK_3RD__

...

#endif

#ifdef __RK_4TH__

...

#endif

// **********

int main(int argc, char **argv)
{
  double d1;
  int i1, i2;

  // *****

  // initialize trajectories with initial conditions

  t[0] = 0.0;

  for(i1 = 0; i1 < N; i1++)
    y[0][i1] = y_0[i1];

  // *****

  // RK steps

  for(i1 = 0; i1 < num_steps_max; i1++)
    {
      if(t[i1] >= t_max)
       break;
```

```
// *****

double y_tau[N], y_tmp[N], y_2_x_tau_over_2[N];

// y(t)   --> \tau   y_{\tau}(t+\tau)
RK_step(y[i1], t[i1], y_tau, tau);

// y(t)   --> \tau/2   --> \tau_2   y_{2 * \tau / 2}(t+\tau)
RK_step(y[i1], t[i1], y_tmp, 0.5*tau);
RK_step(y_tmp, t[i1]+0.5*tau, y_2_x_tau_over_2, 0.5*tau);

// *****

// estimate error

double delta_abs = fabs(y_2_x_tau_over_2[0] - y_tau[0]);

for(i2 = 1; i2 < N; i2++)
{
  d1 = fabs(y_2_x_tau_over_2[i2] - y_tau[i2]);

  if(d1 > delta_abs)
    delta_abs = d1;
}

delta_abs /= pow(2.0, (double)order) - 1.0;

// *****

// adjust step size (do not change by more than factor 5.0).

d1 = 0.9 * pow(delta_abs_max / delta_abs, 1.0 / (((double)order)+1.0));

if(d1 < 0.2)
d1 = 0.2;

if(d1 > 5.0)
d1 = 5.0;

double tau_new = d1 * tau;

// *****

if(delta_abs <= delta_abs_max)
{
  // accept RK step

   for(i2 = 0; i2 < N; i2++)
    y[i1+1][i2] = y_2_x_tau_over_2[i2];

   t[i1+1] = t[i1] + tau;

  tau = tau_new;
  }
else
{
```

```
        // repeat RK step with smaller step size
       tau = tau_new;

         i1--;
       }
     }

  int num_steps = i1;

  // *****

  // output

  for(i1 = 0; i1 <= num_steps; i1++)
    {
      printf("%9.6lf %9.6lf\n", t[i1], y[i1][0]);
    }

  // *****

  return EXIT_SUCCESS;
}
```

C C Code: energy eigenvalues and wave functions of the infinite potential well with the shooting method

---

```
// compute energy eigenvalues and wave functions of the infinite potential well,
// -\psi'' = E \psi ,
// with boundary conditions \psi(x=0) = \psi(x=1) = 0

// **********

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// **********
// physics parameters and functions
// **********

//   y = (\psi , \phi , E)
//   f = (\phi , -E \psi , 0)

const int N = 3;  // number of components of \vec{y} and \vec{f}

double y_0[N] = { 0.0 , 1.0 , 0.0 };  // Anfangsbedingungen y(t=0).

// function computing f(y(t),t) * tau

void f_times_tau(double *y_t, double t, double *f_times_tau_, double tau)
{
  if(N != 3)
    {
      fprintf(stderr, "Error: N != 3!\n");
      exit(EXIT_FAILURE);
    }

  f_times_tau_[0] = y_t[1] * tau;
  f_times_tau_[1] = -y_t[2] * y_t[0] * tau;
  f_times_tau_[2] = 0.0;
}

// **********
// RK parameters
// **********

// #define __EULER__
// #define __RK_2ND__
// #define __RK_3RD__
#define __RK_4TH__

#ifdef __EULER__
const int order = 1;
#endif

#ifdef __RK_2ND__
const int order = 2;
```

```
#endif

#ifdef __RK_3RD__
const int order = 3;
#endif

#ifdef __RK_4TH__
const int order = 4;
#endif

// number of steps
const int num_steps = 1000;

// compute trajectory (= wave function) from t = t_0 to T = t_1
const double t_0 = 0.0;
const double t_1 = 1.0;

double tau = (t_1 - t_0) / (double)num_steps;  // step size

double h = 0.000001;  // finite difference for numerical derivative

double dE_min = 0.0000001;  // Newton-Raphson accuracy

// **********

#ifdef __EULER__

...

#endif

#ifdef __RK_2ND__

// RK step (2nd order), step size tau

void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
{
  int i1;

  // *****

  // k1 = f(y(t),t) * tau

  double k1[N];
  f_times_tau(y_t, t, k1, tau);

  // *****

  // k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau

  double y_[N];

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + 0.5*k1[i1];

  double k2[N];
  f_times_tau(y_, t + 0.5*tau, k2, tau);
```

```
  // *****

  for(i1 = 0; i1 < N; i1++)
    y_t_plus_tau[i1] = y_t[i1] + k2[i1];
}

#endif

#ifdef __RK_3RD__

...

#endif

#ifdef __RK_4TH__

...

#endif

// **********

// RK computation of the trajectory (= wave function)

double t[num_steps+1];  // discretized time
double y[num_steps+1][N];  // discretized trajectories

double RK(bool output = false)
{
  double d1;
  int i1, i2;

  // *****

  // RK steps

  for(i1 = 0; i1 < num_steps; i1++)
    {
      // y(t)   -->   y(t+\tau)
      RK_step(y[i1], t[i1], y[i1+1], tau);

      t[i1+1] = t[i1] + tau;
    }

  // *****

  if(output == true)
    {
      // output

      for(i1 = 0; i1 <= num_steps; i1++)
        {
          printf("%9.6lf %9.6lf %9.6lf %9.6lf\n", t[i1], y[i1][0], y[i1][1], y[i1][2]);
        }
    }
```

```
  // *****

  return y[num_steps][0];
}

// **********

int main(int argc, char **argv)
{
  int i1;

  // *****

  // initialize trajectories with initial conditions

  t[0] = t_0;

  for(i1 = 0; i1 < N; i1++)
    y[0][i1] = y_0[i1];

  // **********
  // **********
  // **********

  // crude graphical determination of energy eigenvalues

  // **********
  // **********
  // **********

  /*
  double E_min = 0.0;
  double E_max = 100.0;

  double E_step = 5.0;

  for(double E = E_min; E <= E_max; E += E_step)
    {
      // set intial condition (energy)
      y[0][N-1] = E;

      // RK computation of the trajectory (= wave function)
      double psi_1 = RK(false);

      printf("%+.5e %+.5e.\n", E, psi_1);
    }
  */

  // **********
  // **********
  // **********

  // **********
  // **********
  // **********

  // shooting method
```

```
// **********
// **********
// **********

// /*
// intial condition (energy)
// double E = 10.0;
// double E = 40.0;
double E = 90.0;

fprintf(stderr, "E_num = %+10.6lf .\n", E);

while(1)
  {
    // change initial condition (energy)
    y[0][N-1] = E;

    // RK computation of the trajectory (= wave function)
    double psi_1_E = RK(false);

    // *****

    // numerical derivative (d/dh) psi(x=1)
    y[0][N-1] = E-h;
    double psi_1_E_mi_h = RK(false);
    y[0][N-1] = E+h;
    double psi_1_E_pl_h = RK(false);
    double dpsi_1_E = (psi_1_E_pl_h - psi_1_E_mi_h) / (2.0 * h);

    // *****

    // Newton-Raphson step

    double dE = psi_1_E / dpsi_1_E;

    if(fabs(dE) < dE_min)
      break;

    E = E - dE;

    // *****

    // fprintf(stderr, "E_num = %+10.6lf  ,  E_ana = %+10.6lf  ,  \\psi(x=1) = %+.6lf .\n",
    //   E, M_PI*M_PI, psi_1_E);
    // fprintf(stderr, "E_num = %+10.6lf  ,  E_ana = %+10.6lf  ,  \\psi(x=1) = %+.6lf .\n",
    //   E, 4.0*M_PI*M_PI, psi_1_E);
    fprintf(stderr, "E_num = %+10.6lf  ,  E_ana = %+10.6lf  ,  \\psi(x=1) = %+.6lf .\n",
      E, 9.0*M_PI*M_PI, psi_1_E);
  }

// output
RK(true);
// */

// **********
// **********
```

```
  // **********

  return EXIT_SUCCESS;
}
```

D C Code: Gauss elimination with backward substitution, different pivoting strategies

---

```
// solve
// A x = b
// using Gauss elimination with backward substitution and different pivoting strategies

// **********

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// **********

// #define __PARTIAL_PIVOTING__
#define __SCALED_PARTIAL_PIVOTING__

// **********

// size of A, b and x
// const int N = 4;
const int N = 100;

// matrix A (elements will be modified during computation)
double A[N][N];

// vector (elements will be modified during computation)
double b[N];

// solution
double x[N];

// permutation of rows due to pivoting
int p[N];

// **********

// generates a uniformly distributed random number in [min,max]

double DRand(double min, double max)
{
  return min + (max-min) * ( (rand() + 0.5) / (RAND_MAX + 1.0) );
}

// **********

// print A | b

void Print()
{
  int i1, i2;
```

```
  for(i1 = 0; i1 < N; i1++)
    {
      for(i2 = 0; i2 < N; i2++)
        fprintf(stdout, "%+5.2lf  ", A[p[i1]][i2]);

      fprintf(stdout, "|  %+5.2lf\n", b[p[i1]]);
    }

  fprintf(stdout, "\n");
}

// **********

int main(int argc, char **argv)
{
  double d1, d2, d3;
  int i1, i2, i3;

  srand(0);
  // srand((unsigned int)time(NULL));

  // *****

  // generate random matrix A and vector b, elements in [-1.0,+1.0]

  for(i1 = 0; i1 < N; i1++)
    {
      for(i2 = 0; i2 < N; i2++)
        A[i1][i2] = DRand(-1.0, +1.0);

      b[i1] = DRand(-1.0, +1.0);
    }

  // initialize permutation of rows

  for(i1 = 0; i1 < N; i1++)
    p[i1] = i1;

  Print();

  // *****

  // copy matrix A und vektor b (needed at the end to investigate roundoff errors)

  double A_org[N][N];
  double b_org[N];

  for(i1 = 0; i1 < N; i1++)
    {
      for(i2 = 0; i2 < N; i2++)
        A_org[i1][i2] = A[i1][i2];

      b_org[i1] = b[i1];
    }

  // *****
```

```
  // elimination

#ifdef __SCALED_PARTIAL_PIVOTING__

  // store maximum of each row of A, before elements are modified

  double A_ij_max[N];

  for(i1 = 0; i1 < N; i1++)
    {
      A_ij_max[i1] = fabs(A[i1][0]);

      for(i2 = 1; i2 < N; i2++)
        {
          if(fabs(A[i1][i2]) > A_ij_max[i1])
            A_ij_max[i1] = fabs(A[i1][i2]);
        }
    }

#endif

  for(i1 = 0; i1 < N-1; i1++)
    // N-1 elimination steps
    {
      // determine "optimal row" according to pivoting strategy

      int index = i1;

#ifdef __PARTIAL_PIVOTING__

      for(i2 = i1+1; i2 < N; i2++)
        {
          if(fabs(A[p[i2]][i1]) > fabs(A[p[i1]][i1]))
            index = i2;
        }

#endif

#ifdef __SCALED_PARTIAL_PIVOTING__

      d1 = fabs(A[p[i1]][i1]) / A_ij_max[p[i1]];

      for(i2 = i1+1; i2 < N; i2++)
        {
          d2 = fabs(A[p[i2]][i1]) / A_ij_max[p[i2]];

          if(d2 > d1)
            index = i2;
        }

#endif

      i2 = p[i1];
      p[i1] = p[index];
      p[index] = i2;

      // ***
```

```
    for(i2 = i1+1; i2 < N; i2++)
      // for all remaining rows ...
      {
        d1 = A[p[i2]][i1] / A[p[i1]][i1];
        A[p[i2]][i1] = 0.0;

        for(i3 = i1+1; i3 < N; i3++)
          A[p[i2]][i3] -= d1 * A[p[i1]][i3];

        b[p[i2]] -= d1 * b[p[i1]];
      }

    Print();
  }

// *****

// backward substitution

for(i1 = N-1; i1 >= 0; i1--)
  // Für alle Komponenten von x ...
  {
    x[i1] = b[p[i1]];

    for(i2 = i1+1; i2 < N; i2++)
      x[i1] -= A[p[i1]][i2] * x[i2];

    x[i1] /= A[p[i1]][i1];
  }

fprintf(stdout, "x = ( ");

for(i1 = 0; i1 < N-1; i1++)
  {
    fprintf(stdout, "%+5.2lf  ", x[i1]);
  }

fprintf(stdout, "%+5.2lf ).\n\n", x[N-1]);

// *****

// check solution, investigate roundoff errors

double b_check[N];

for(i1 = 0; i1 < N; i1++)
  {
    b_check[i1] = 0.0;

    for(i2 = 0; i2 < N; i2++)
      b_check[i1] += A_org[i1][i2] * x[i2];
  }

fprintf(stdout, "b_check = ( ");

for(i1 = 0; i1 < N-1; i1++)
```

```
    fprintf(stdout, "%+5.2lf  ", b_check[i1]);

  fprintf(stdout, "%+5.2lf ).\n\n", b_check[N-1]);

  fprintf(stdout, "b_check - b = ( ");

  // discrepancy between original b and reconstructed b for each element

  for(i1 = 0; i1 < N-1; i1++)
    fprintf(stdout, "%+.1e  ", b_check[i1] - b_org[i1]);

  fprintf(stdout, "%+.1e ).\n\n", b_check[N-1] - b_org[N-1]);

  // norm of the discrepancy

  double norm = 0.0;

  for(i1 = 0; i1 < N; i1++)
    norm += pow(b_check[i1] - b_org[i1], 2.0);

  norm = sqrt(norm);

  fprintf(stdout, "|b_check - b| = %+.5e.\n", norm);

  // *****

  return EXIT_SUCCESS;
}
```

E C Code: solving the discretized Poisson equation with the conjugate gradient method

---

```
// solve the discretized Poisson equation with the conjugate gradient method

// **********

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// **********

// number of lattice site in each direction is 2*N+1 (boundary included)
const int n = 100;

// dimension of the matrix A and the vectors x and b
const int N = (2*n - 1) * (2*n - 1);

// stop conjugate gradient method as soon as residual is smaller than eps
double eps_eps = 0.0000000001 * 0.0000000001;

// **********

// allocate vector

double *Alloc_Vector()
{
  double *v = (double *)malloc(N * sizeof(double));

  if(v == NULL)
  {
    fprintf(stderr, "Error: double *Alloc_Vector(...\n");
    exit(EXIT_FAILURE);
  }

return v;
}

// **********

// superindex

int Index(int ix, int iy)
{
  if(ix <= -n || ix >= +n || iy <= -n || iy >= +n)
    return -1;

  return (2*n-1)*(iy+(n-1)) + (ix+(n-1));
}

// compute y = A x

void A(double *y, const double *x)
```

```
{
  int ix, iy;

  for(ix = -(n-1); ix <= +(n-1); ix++)
  {
    for(iy = -(n-1); iy <= +(n-1); iy++)
    {
    y[Index(ix, iy)] = -4.0 * x[Index(ix, iy)];

      if(Index(ix-1, iy) != -1)
        y[Index(ix, iy)] += x[Index(ix-1, iy)];

      if(Index(ix+1, iy) != -1)
        y[Index(ix, iy)] += x[Index(ix+1, iy)];

      if(Index(ix, iy-1) != -1)
        y[Index(ix, iy)] += x[Index(ix, iy-1)];

      if(Index(ix, iy+1) != -1)
        y[Index(ix, iy)] += x[Index(ix, iy+1)];
    }
  }
}

// **********

int main(int argc, char **argv)
{
  int i1;
  double *v1 = Alloc_Vector();

  double *x = Alloc_Vector();
  double *b = Alloc_Vector();

  double *r = Alloc_Vector();
  double *p = Alloc_Vector();

  // *****

  // intitialize right hand side b

  for(i1 = 0; i1 < N; i1++)
    b[i1] = 0.0;

  b[Index(0, 0)] = 1.0;

  // *****

  // initialize solution x with 0.0

  for(i1 = 0; i1 < N; i1++)
    x[i1] = 0.0;

  // *****

  // r = b - A x
```

```
A(v1, x);

for(i1 = 0; i1 < N; i1++)
  r[i1] = b[i1] - v1[i1];

// p = r

for(i1 = 0; i1 < N; i1++)
  p[i1] = r[i1];

// *****

// conjugate gradient iteration

int ctr = 0;

double r_r = 0.0;

for(i1 = 0; i1 < N; i1++)
  r_r += r[i1] * r[i1];

while(1)
{
  ctr++;
  fprintf(stderr, "ctr = %4d.\n", ctr);

  if(ctr == 1000000)
    break;

  // ***

  // alpha = r^2 / (p A p)

  A(v1, p);

  double p_A_p = 0.0;

  for(i1 = 0; i1 < N; i1++)
    p_A_p += p[i1] * v1[i1];

  double alpha = r_r / p_A_p;

  // x = x + alpha p

  for(i1 = 0; i1 < N; i1++)
    x[i1] += alpha * p[i1];

  // r = r - alpha A p

  for(i1 = 0; i1 < N; i1++)
    r[i1] -= alpha * v1[i1];

  // beta = (r_new)^2 / (r_old)^2

  double r_r_old = r_r;
  r_r = 0.0;
```

```
    for(i1 = 0; i1 < N; i1++)
      r_r += r[i1] * r[i1];

    fprintf(stderr, "  r_r = %+.5e (%.5e).\n", r_r, r_r_old);

    if(r_r < eps_eps)
     // Hinreichend genaues x erreicht.
      break;

    double beta = r_r / r_r_old;

    // p = r + beta p

    for(i1 = 0; i1 < N; i1++)
      p[i1] = r[i1] + beta * p[i1];
  }

  // *****

  // check result by comparing A x to b

  double *b_check = Alloc_Vector();
  A(b_check, x);
  double norm = 0.0;

  for(i1 = 0; i1 < N; i1++)
    norm += pow(b_check[i1] - b[i1], 2.0);

  norm = sqrt(norm);
  fprintf(stderr, "\n|b_check - b| = %+.5e.\n", norm);

  // *****

  // output of electrostatic potential phi = x

  for(i1 = -(n-1); i1 <= +(n-1); i1++)
    fprintf(stdout, "%+.5e %+.5e\n", ((double)i1) / ((double)n), x[Index(i1, 0)]);  // on axis
    // fprintf(stdout, "%+.5e %+.5e\n", ((double)i1) / ((double)n) * sqrt(2.0), x[Index(i1, i1)]);  // diagonal

  // *****

  free(v1);
  free(b);
  free(x);
  free(r);
  free(p);
  free(b_check);

  // *****

  return EXIT_SUCCESS;
}
```

F C Code: eigenvalues and eigenvectors of a 10×10 stiffness matrix with the Jacobi method

---

```
// compute all eigenvalues lambda and eigenvectors v of a real symmetrix matrix A,
// A v = lambda v ,
// using the Jacobi method

// **********

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// **********

const int N = 10;  // size of A

// real symmetric matrix; will be overwritten; diagonal elements will correspond to eigenvalues
double A[N][N];

// matrix of eigenvectors (product of Jacobi rotations); columns will correspond to eigenvectors
double V[N][N];

const double epsilon = 1.0e-20;  // stop iterations, if S < epsilion

// **********

int main(int argc, char **argv)
{
  FILE *file1;
  int i1, i2, i3;
  char string1[1000];

  // *****

  // initialize matrix A

  for(i1 = 0; i1 < N; i1++)
  {
    for(i2 = 0; i2 < N; i2++)
      A[i1][i2] = 0.0;
  }

  for(i1 = 0; i1 < N-1; i1++)
  {
    A[i1  ][i1  ] += 1.0;
    A[i1  ][i1+1] -= 1.0;
    A[i1+1][i1  ] -= 1.0;
    A[i1+1][i1+1] += 1.0;
  }

  // /*
  for(i1 = 0; i1 < N; i1++)
  {
```

```
  for(i2 = 0; i2 < N; i2++)
    fprintf(stderr, "%+4.2lf   ", A[i1][i2]);

  fprintf(stderr, "\n");
}
// */

// initialize eigenvector matrix

for(i1 = 0; i1 < N; i1++)
{
  for(i2 = 0; i2 < N; i2++)
  {
    if(i1 == i2)
      V[i1][i2] = 1.0;
    else
      V[i1][i2] = 0.0;
  }
}

// *****

// Jacobi method

int ctr = 0;

while(1)
{
  // deviation from diagonal matrix

  double S = 0.0;

  for(i1 = 0; i1 < N; i1++)
  {
    for(i2 = 0; i2 < i1; i2++)
      S += pow(A[i1][i2], 2.0);
  }

  S *= 2.0;
  fprintf(stderr, "S = %.5e.\n", S);

  if(S <= epsilon)
    break;

  // *****

  ctr++;
  fprintf(stderr, "sweep %4d ...\n", ctr);

  // sweep over all off-diagonal elements ...

  for(i1 = 0; i1 < N; i1++)
  {
    for(i2 = 0; i2 < i1; i2++)
    {
      if(fabs(A[i1][i2]) < epsilon / (double)(N*N))
        // avoid divison by "almost 0.0"
```

```
  continue;

// theta
double theta = 0.5 * (A[i2][i2] - A[i1][i1]) / A[i1][i2];

// t
double t = 1.0 / (fabs(theta) + sqrt(pow(theta, 2.0) + 1.0));

if(theta < 0.0)
  t = -t;

// c, s
double c = 1.0 / sqrt(pow(t, 2.0) + 1.0);
double s = t * c;

// tau
double tau = s / (1.0 + c);

// Jacobi rotation

// matrix A

double A_pp = A[i1][i1] - t * A[i1][i2];
double A_qq = A[i2][i2] + t * A[i1][i2];
double A_rp[N], A_rq[N];

for(i3 = 0; i3 < N; i3++)
{
  if(i3 != i1 && i3 != i2)
  {
    A_rp[i3] = A[i3][i1] - s * (A[i3][i2] + tau * A[i3][i1]);
    A_rq[i3] = A[i3][i2] + s * (A[i3][i1] - tau * A[i3][i2]);
  }
}

A[i1][i2] = 0.0;
A[i2][i1] = 0.0;
A[i1][i1] = A_pp;
A[i2][i2] = A_qq;

for(i3 = 0; i3 < N; i3++)
{
  if(i3 != i1 && i3 != i2)
  {
    A[i3][i1] = A_rp[i3];
    A[i1][i3] = A_rp[i3];
    A[i3][i2] = A_rq[i3];
    A[i2][i3] = A_rq[i3];
  }
}

// eigenvector matrix

double V_rp[N], V_rq[N];

for(i3 = 0; i3 < N; i3++)
{
```

```
          V_rp[i3] = V[i3][i1] - s * (V[i3][i2] + tau * V[i3][i1]);
          V_rq[i3] = V[i3][i2] + s * (V[i3][i1] - tau * V[i3][i2]);
        }

        for(i3 = 0; i3 < N; i3++)
        {
          V[i3][i1] = V_rp[i3];
          V[i3][i2] = V_rq[i3];
        }
      }
    }

    // /*
    for(i1 = 0; i1 < N; i1++)
    {
      for(i2 = 0; i2 < N; i2++)
        fprintf(stderr, "%+4.2lf   ", A[i1][i2]);

      fprintf(stderr, "\n");
    }
    // */
  }

  // *****


  for(i1 = 0; i1 < N; i1++)
  {
    fprintf(stderr, "\nlambda_%02d = %+10.6lf.\n", i1, A[i1][i1]);

    fprintf(stderr, "v_%02d = ( ", i1);

    for(i2 = 0; i2 < N; i2++)
    {
      fprintf(stderr, "%+5.2lf", V[i2][i1]);

      if(i2 < N-1)
        fprintf(stderr, " , ");
      else
        fprintf(stderr, " ).\n");
    }
  }

  // *****

  return EXIT_SUCCESS;
}
```

G Python Code: solving the heat equation with the Crank-Nicolson method

---

```
# heat_eq_CN.py

# **********

import math
import numpy as np
import matplotlib.pyplot as plt

# **********

# input parameters

Nx = 40
Nt = 40
Dt_over_Dx = 0.1  # compute the time evolution from t = 0 to t = Nt * Dt_over_Dx * Dx
u_x_eq_0 = 0.0  # boundary condition at x = 0
u_x_eq_L = 1.0  # boundary condition at x = L

# initial conditions u_0(x) = x + \mu sin(n \pi x)
# example 1
n = 2
mu = -1.0 / (2.0 * math.pi)
# example 2
# n = 3
# mu = 2.0

# *****

Dx = 1.0 / Nx
Dt = Dt_over_Dx * Dx

# **********

# intialize temperature at t = 0

u = np.zeros((Nt + 1, Nx - 1))
for i1 in range(0, Nx - 1):
  x = (i1 + 1) * Dx
  u[0][i1] = x + mu * math.sin(n * math.pi * x)

# **********

# Crank-Nicolson method

alpha_Dt_over_Dx_square = Dt_over_Dx / Dx

# initialize matrix B (see lecture notes)
B = np.zeros((Nx - 1, Nx - 1))
for i2 in range(0, Nx - 1):
  B[i2][i2] = 2.0 * (1.0 + alpha_Dt_over_Dx_square)
  if i2 > 0:
```

```
    B[i2][i2-1] = -alpha_Dt_over_Dx_square
  if i2 < Nx-2:
    B[i2][i2+1] = -alpha_Dt_over_Dx_square

# Nt Crank-Nicolson steps
for i1 in range(0, Nt):
  print('Crank-Nicolson step {} ...'.format(i1+1))

  # initialize vector b (see lecture notes)
  b = np.zeros(Nx - 1)
  b[0] = alpha_Dt_over_Dx_square * u_x_eq_0  # "contributions from the lhs", see lecture notes
  b[Nx-2] = alpha_Dt_over_Dx_square * u_x_eq_L  # "contributions from the lhs", see lecture notes
  for i2 in range(0, Nx - 1):
    d1 = 0.0
    if i2 == 0:
      d1 += u_x_eq_0  # boundary condition at x = 0 corresponds to i2 = -1
    else:
      d1 += u[i1][i2-1]
    if i2 == Nx-2:
      d1 += u_x_eq_L  # boundary condition at x = L corresponds to i2 = Nx-1
    else:
      d1 += u[i1][i2+1]
    b[i2] += 2.0 * (1.0 - alpha_Dt_over_Dx_square) * u[i1][i2] + alpha_Dt_over_Dx_square * d1

  # solve B u = b
  u[i1+1] = np.linalg.solve(B, b)

# **********

# plot heatmap

u_with_boundary = np.c_[ [u_x_eq_0] * (Nt+1), u, [u_x_eq_L] * (Nt+1) ]
plt.imshow(u_with_boundary, cmap = 'hot', interpolation = 'nearest', origin = 'lower', extent = (0.0 - 0.5*Dx, 1.0
plt.xlabel('x/L')
plt.ylabel('alpha t/L^2')
plt.show()

# **********

# compare with analytical solution

for i1 in range(0, Nt+1):
  t = i1 * Dt
  err_max = 0.0
  for i2 in range(0, Nx-1):
    x = (i2 + 1) * Dx
    u_analytical = x + mu * math.sin(n * math.pi * x) * math.exp(-n**2 * math.pi**2 * t)
    d1 = abs(u[i1][i2] - u_analytical)
    if d1 > err_max:
      err_max = d1
  print('Crank-Nicolson step {:3d}, t = {:9.6f} --> err_max = {:.5e}'.format(i1, t, err_max))
```

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical recipes 3rd edition: the art of scientific computing," Cambridge University Press (2007).

[2] P. C. Chow, "Computer solutions to the Schrödinger equation," American Journal of Physics **40**, 730 (1972).

[3] P. Bicudo and M. Wagner, "Lattice QCD signal for a bottom-bottom tetraquark," Phys. Rev. D **87**, no. 11, 114511 (2013) [arXiv:1209.6274 [hep-ph]], `https://arxiv.org/abs/1209.6274`.

[4] P. Bicudo, N. Cardoso, L. Müller and M. Wagner, "Study of $I = 0$ bottomonium bound states and resonances in S, P, D, and F waves with lattice QCD static-static-light-light potentials," Phys. Rev. D **107**, no. 9, 094515 (2023) [arXiv:2205.11475 [hep-lat]], `https://arxiv.org/abs/2205.11475`.

[5] H. Grabmüller, "Numerik II (für Ingenieure)," lecture notes, Friedrich-Alexander-Universität Erlangen-Nürnberg (2001).

[6] P. Young, "Everything you wanted to know about data analysis and fitting but were afraid to ask," (2012) `https://arxiv.org/abs/1210.3781`.

[7] `https://web.physics.utah.edu/~detar/phycs6730/handouts/jackknife/jackknife`.

[8] L. Rezzolla, "Numerical Methods for Physics," lecture notes, Goethe-Universität Frankfurt am Main (WiSe 2022/23).

[9] E. Engel, "Einführung in die Programmierung für Physiker – Teil III: Numerische Mathematik," lecture notes, Goethe-Universität Frankfurt am Main (SoSe 2023).