Numerical methods in physics

Marc Wagner

Goethe-Universität Frankfurt am Main – summer semester 2021

Version: July 6, 2021

Contents

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)`).
- 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.

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

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

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

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

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

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

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

  - Relative error due to $f(x+h) - f(x)$:

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

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

$\rightarrow$ relative loss of accuracy $\frac{f(x)}{f'(x)h}$ (e.g. $\approx 10^3$ implies that 3 digits are lost)

$$\rightarrow \quad \frac{\delta f'(x)}{f'(x)} = \frac{f(x)}{f'(x)h}\epsilon. \tag{10}$$

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

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

$$\rightarrow \quad h_{\text{opt}} \sim \left(\frac{f(x)}{f''(x)}\epsilon\right)^{1/2} \sim \epsilon^{1/2}$$

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

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

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

  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);
    }

  // *****

  return EXIT_SUCCESS;
}
```

---

[2] Throughout this lecture I use C.

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

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

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

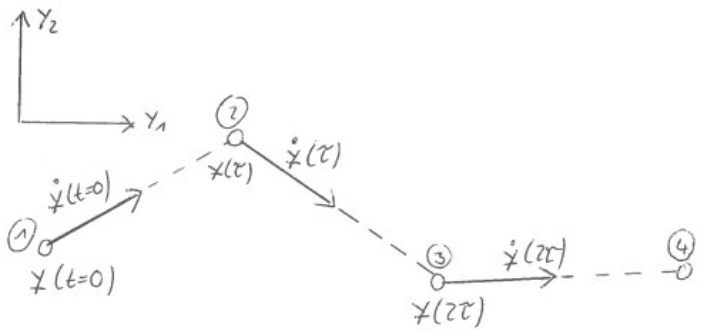
  - Then

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

    (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).

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":

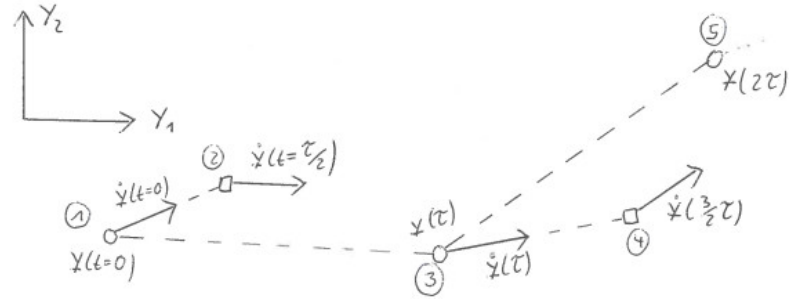
$$\mathbf{k}_1 = \mathbf{f}(\mathbf{y}(t), t)\tau \quad \rightarrow \text{ "full Euler step"} \tag{20}$$

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

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

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

Figure 3.B



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

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

$$\mathbf{y}(t+\tau) = \mathbf{y} + \dot{\mathbf{y}}\tau + \frac{1}{2}\ddot{\mathbf{y}}\tau^2 + \mathcal{O}(\tau^3) = \mathbf{y} + \mathbf{f}\tau + \frac{1}{2}\dot{\mathbf{f}}\tau^2 + \mathcal{O}(\tau^3) = \qquad (24)$$

$$= \mathbf{y} + \mathbf{k}_2 + \mathcal{O}(\tau^3) \qquad (25)$$

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

- Discretization with $\mathcal{O}(\tau^3)$ error per step not unique ($\rightarrow$ tutorials).
- Straightforward to derive discretizations with $\mathcal{O}(\tau^4)$, $\mathcal{O}(\tau^5)$, ... error per step:
  - "3rd-order RK":

$$\mathbf{k}_1 = \mathbf{f}(\mathbf{y}(t), t)\tau \qquad (26)$$

$$\mathbf{k}_2 = \mathbf{f}\Big(\mathbf{y}(t) + \mathbf{k}_1, t + \tau\Big)\tau \qquad (27)$$

$$\mathbf{k}_3 = \mathbf{f}\Big(\mathbf{y}(t) + (1/4)\Big(\mathbf{k}_1 + \mathbf{k}_2\Big), t + (1/2)\tau\Big)\tau \qquad (28)$$

$$\mathbf{y}(t+\tau) = \mathbf{y}(t) + \frac{1}{6}\Big(\mathbf{k}_1 + \mathbf{k}_2 + 4\mathbf{k}_3\Big) + \mathcal{O}(\tau^4). \qquad (29)$$

  - "4th-order RK":

$$\mathbf{k}_1 = \mathbf{f}(\mathbf{y}(t), t)\tau \qquad (30)$$

$$\mathbf{k}_2 = \mathbf{f}\Big(\mathbf{y}(t) + (1/2)\mathbf{k}_1, t + (1/2)\tau\Big)\tau \qquad (31)$$

$$\mathbf{k}_3 = \mathbf{f}\Big(\mathbf{y}(t) + (1/2)\mathbf{k}_2, t + (1/2)\tau\Big)\tau \qquad (32)$$

$$\mathbf{k}_4 = \mathbf{f}\Big(\mathbf{y}(t) + \mathbf{k}_3, t + \tau\Big)\tau \qquad (33)$$

$$\mathbf{y}(t+\tau) = \mathbf{y}(t) + \frac{1}{6}\Big(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\Big) + \mathcal{O}(\tau^5). \qquad (34)$$

– ...

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

$$L = \frac{m}{2}\dot{x}^2 - \frac{m\omega^2}{2}x^2. \tag{35}$$

  - EOMs:

$$m\ddot{x}(t) = -m\omega^2 x(t), \tag{36}$$

    i.e.

$$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t), t) \tag{37}$$

    with

$$\mathbf{y}(t) = (x(t), v(t)) \quad , \quad \mathbf{f}(\mathbf{y}(t), t) = (v(t), -\omega^2 x(t)). \tag{38}$$

  - 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).

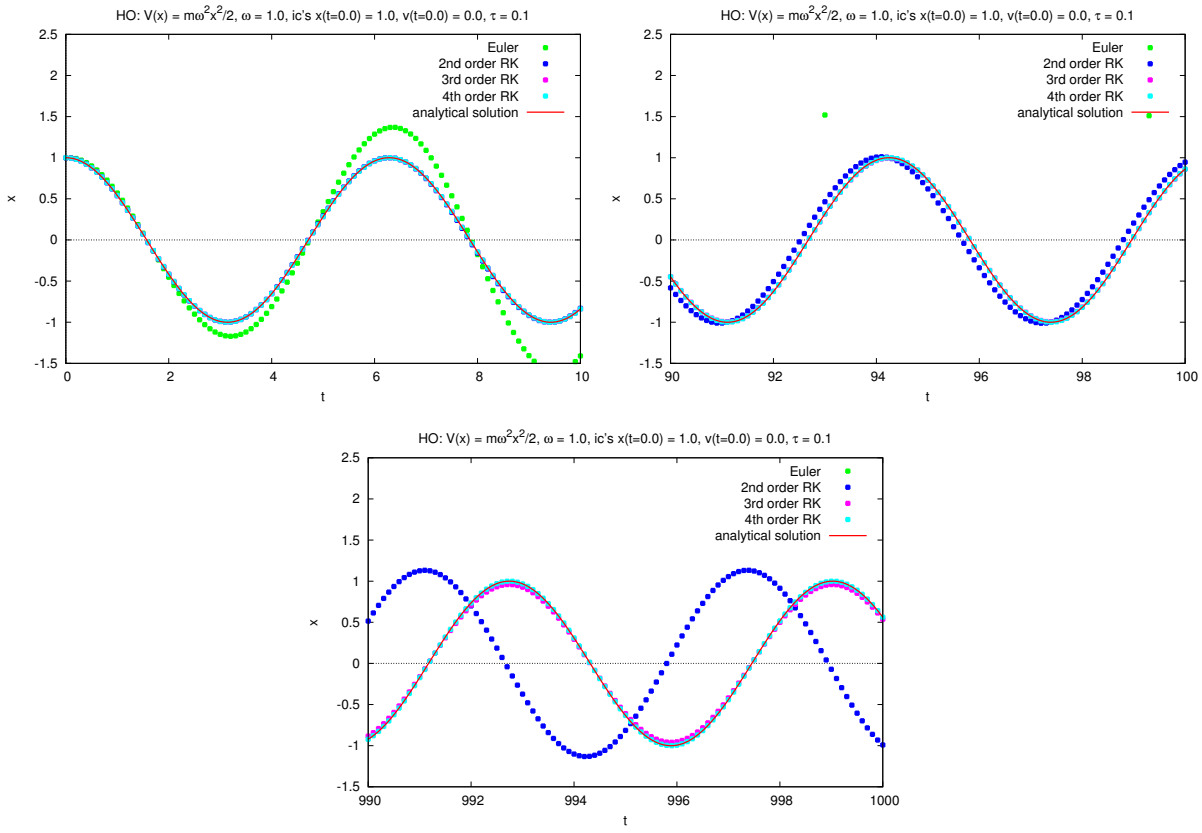
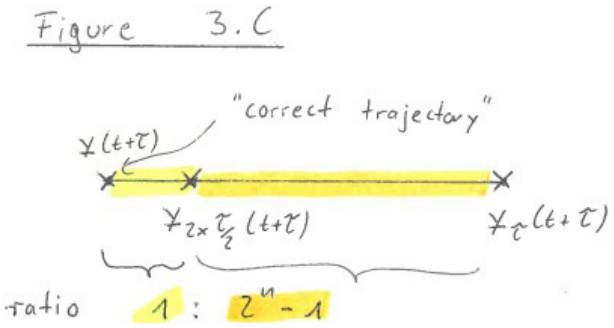


Figure 1: HO, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK.



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

$$\delta_{\text{abs}} = \frac{|\mathbf{y}_{2\times\tau/2}(t+\tau) - \mathbf{y}_{\tau}(t+\tau)|}{2^n - 1}, \tag{39}$$

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

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

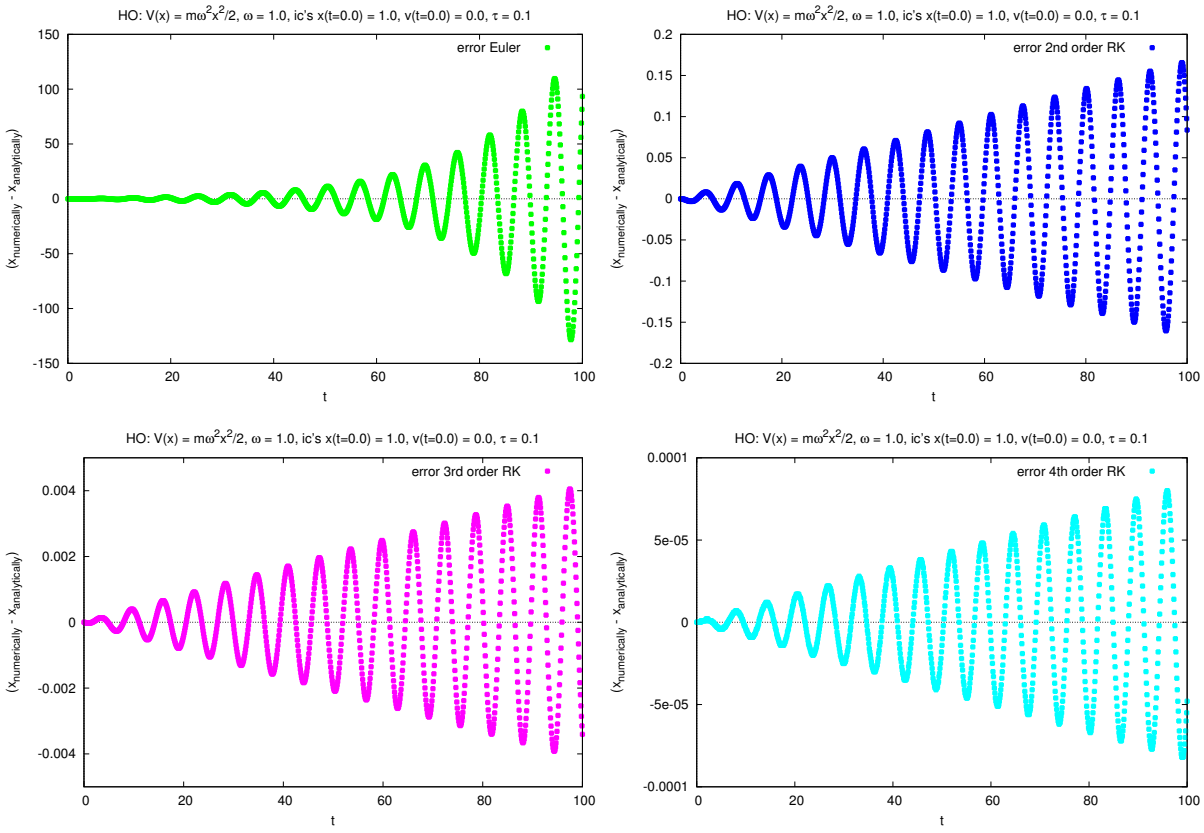


Figure 2: HO, errors of the trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK.

absolute error):

$$\delta_{\text{rel}} \quad = \quad \frac{\delta_{\text{abs}}}{|\mathbf{y}(t)|}. \tag{40}$$

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

$$\mathbf{y}_{2\times\tau/2}(t+\tau) \quad \rightarrow \quad \mathbf{y}_{2\times\tau/2}(t+\tau) + \frac{\mathbf{y}_{2\times\tau/2}(t+\tau) - \mathbf{y}_{\tau}(t+\tau)}{2^n - 1}. \tag{41}$$

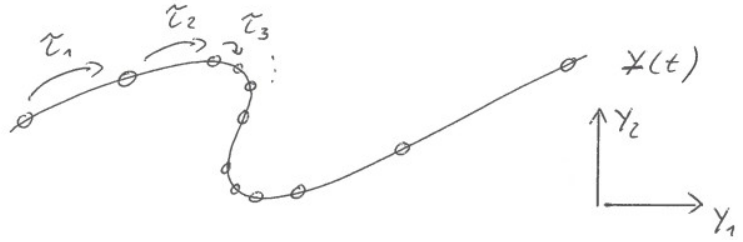
  - However, no estimation of errors, when using (41).

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.

Figure 3.D



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

$$\frac{\delta_{X,\max}}{\delta_X} = \frac{(\tau_{\max})^{n+1}}{\tau^{n+1}} \quad \rightarrow \quad \tau_{\max} = \tau\left(\frac{\delta_{X,\max}}{\delta_X}\right)^{1/(n+1)} \quad , \quad X \in \{\text{abs}, \text{rel}\}. \tag{42}$$

- 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:

  $$\mathbf{y}(t) \quad \rightarrow_{\tau} \quad \mathbf{y}_{\tau}(t+\tau) \tag{43}$$

  $$\mathbf{y}(t) \quad \rightarrow_{\tau/2} \rightarrow_{\tau/2} \quad \mathbf{y}_{2\times\tau/2}(t+\tau). \tag{44}$$

  - *Estimated error:*

  $$\delta_{abs} = \frac{|\mathbf{y}_{2\times\tau/2}(t+\tau) - \mathbf{y}_{\tau}(t+\tau)|}{2^n - 1}. \tag{45}$$

  - *Change step size:*

  $$\tau_{new} = 0.9 \times \tau\left(\frac{\delta_{abs,max}}{\delta_{abs}}\right)^{1/(n+1)} \tag{46}$$

  ("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:

    $$L = \frac{m}{2}\dot{x}^2 - m\alpha x^n \quad , \quad n \in \{2, 20\}. \tag{47}$$

  - EOMs:

    $$m\ddot{x}(t) = -m\alpha n (x(t))^{n-1}, \tag{48}$$

    i.e.

    $$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t), t) \tag{49}$$

    with

    $$\mathbf{y}(t) = (x(t), v(t)) \quad , \quad \mathbf{f}(\mathbf{y}(t), t) = (v(t), -\alpha n (x(t))^{n-1}). \tag{50}$$

  - 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.

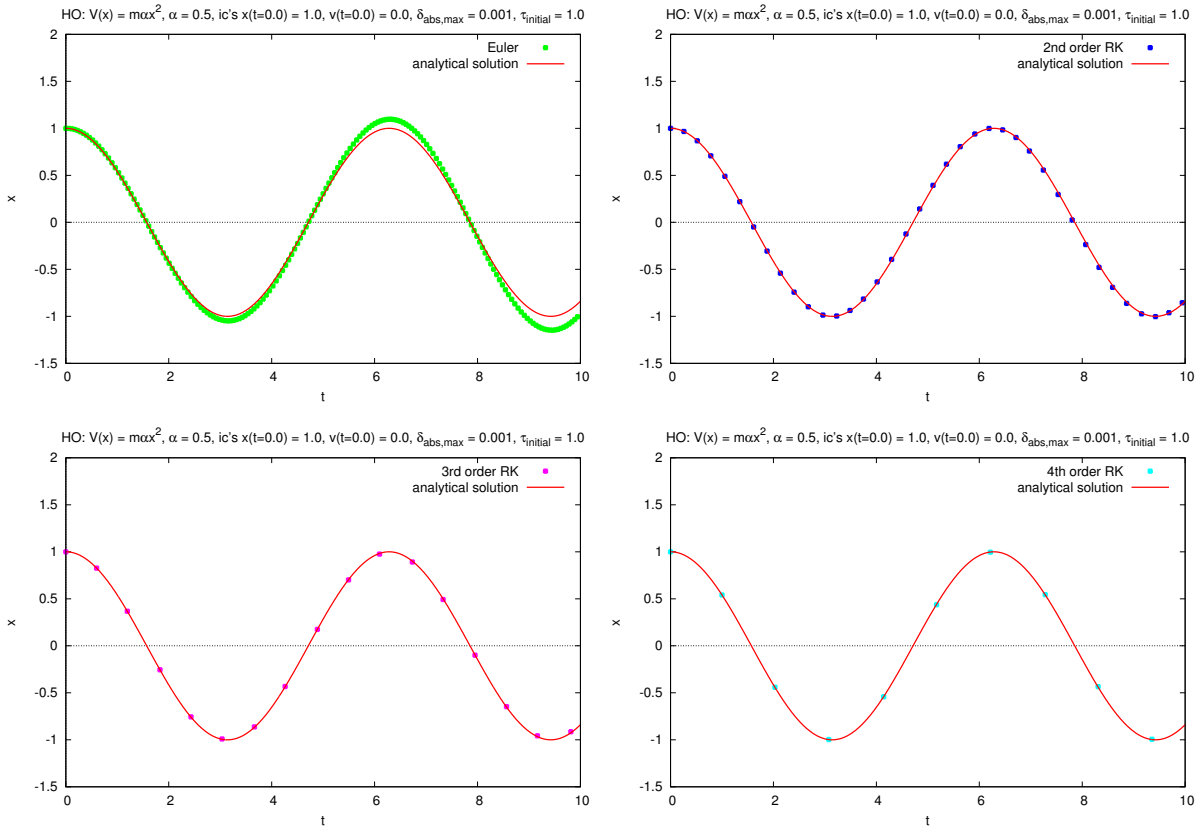


Figure 3: Harmonic oscillator, $V(x) = m\alpha x^2$, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK using adaptive step size.

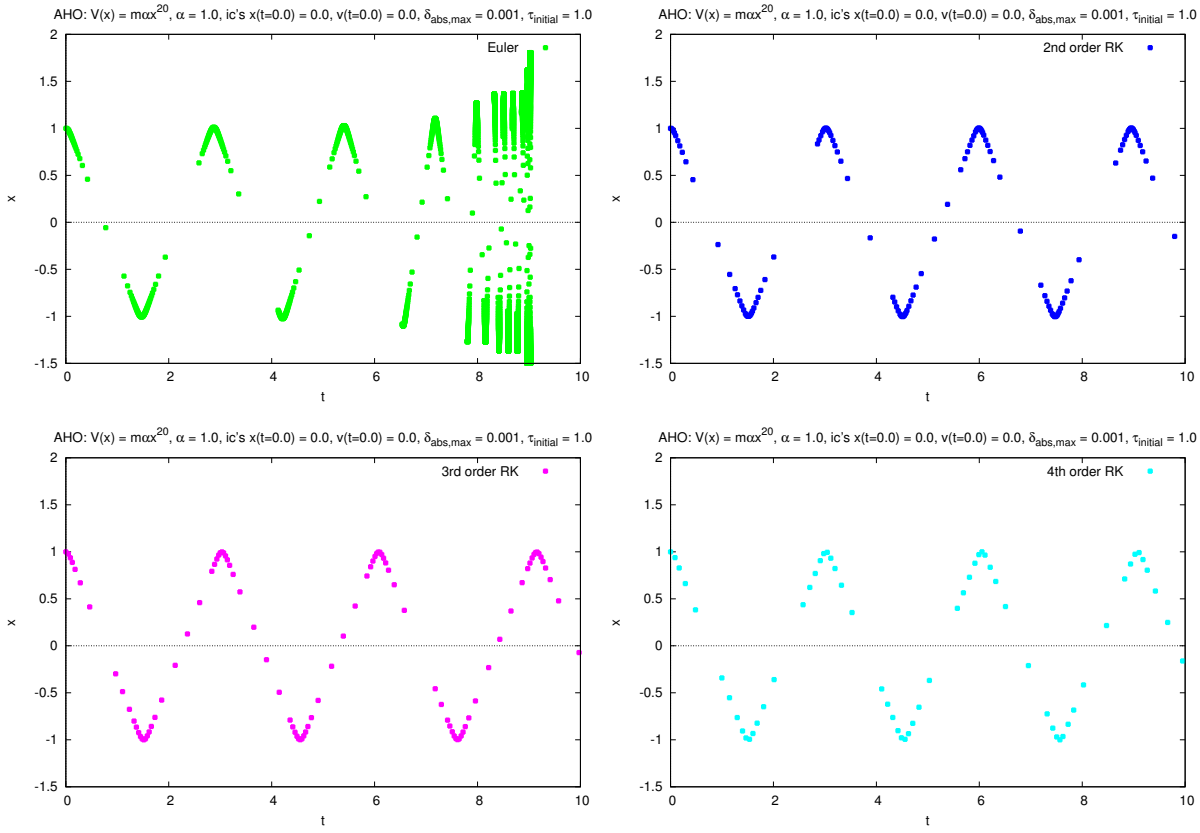


Figure 4: Anharmonic oscillator, $V(x) = m\alpha x^{20}$, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK using adaptive step size.

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:
    $$L = \frac{m}{2}\dot{x}^2 - \frac{m\omega^2}{2}x^2. \tag{51}$$
  - EOMs:
    $$m\ddot{x}(t) = -m\omega^2 x(t) \quad \rightarrow \quad \ddot{x}(t) = -\omega^2 x(t), \tag{52}$$
    i.e. m irrelevant.
  - Measure time in units of $1/\omega$:
    $$\hat{t} = \omega t \quad \rightarrow \quad \frac{d^2}{d\hat{t}^2}x(\hat{t}) = -x(\hat{t}). \tag{53}$$
  - 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:
    $$\hat{x} = \frac{x}{x_0} \quad \rightarrow \quad \frac{d^2}{d\hat{t}^2}\hat{x}(\hat{t}) = -\hat{x}(\hat{t}). \tag{54}$$
  - Now only dimensionless quantities in (54), i.e. straightforward to treat numerically.
  - Figure 5 showing trajectory $\hat{x}(\hat{t})$ is analog of Figure 1 (left top).
- 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).

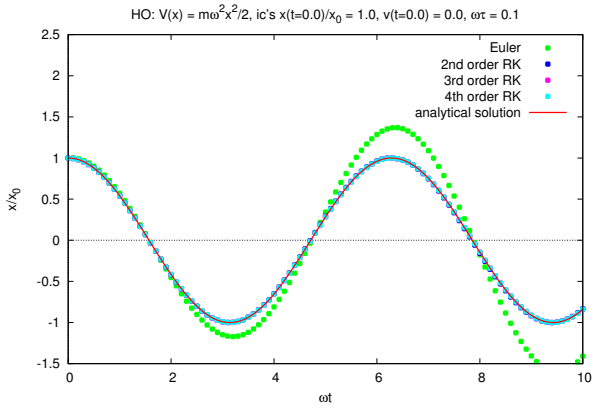


Figure 5: HO, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK (same data as in Figure 1 [left top], but coordinate axes correspond to dimensionless quantities $\hat{t} = \omega t$ and $\hat{x} = x/x_0$).

5 Root finding, solving systems of non-linear equations

5.1 Physics motivation

- N non-linear equations with N unknowns,

$$f_j(x_1, \ldots, x_N) \;=\; 0 \quad , \quad j = 1, \ldots, N \tag{55}$$

or equivalently written in a more compact way

$$\mathbf{f}(\mathbf{x}) \;=\; 0. \tag{56}$$

- Find solutions $\mathbf{x}$ of (56), 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

$$\delta_n \;\leq\; \frac{|x_1 - x_2|}{2^n}, \tag{57}$$

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:

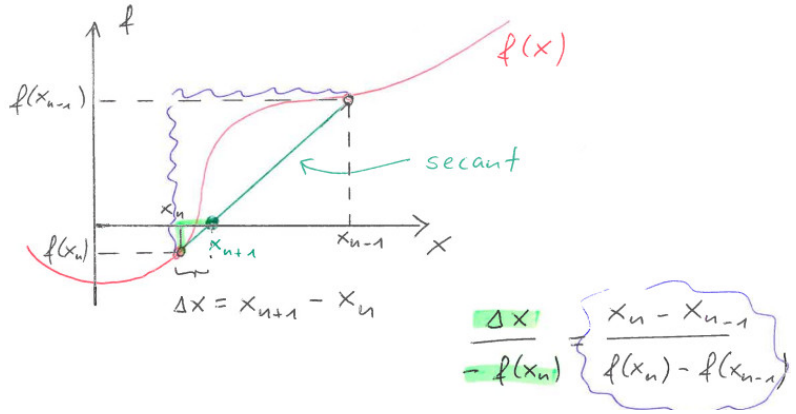
  (+) Always finds a root.

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

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)

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

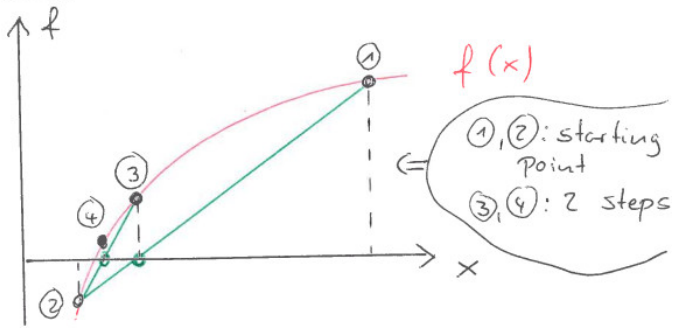
  - *If $|\Delta x|$ sufficiently small:*

    → *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:

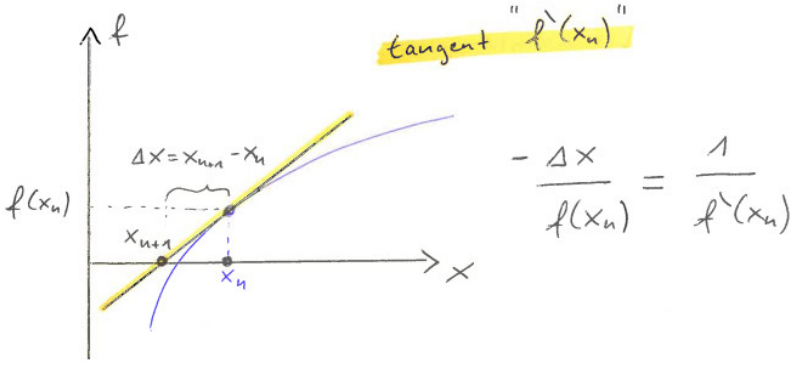
  (+) Converges faster than bisection.

  (−) 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/cheap to evaluate numerically.
  - Each step as sketched below.

Figure 5.C



- Algorithm:

– $n = 1$.

(1)

$$\Delta x \quad = \quad -f(x_n)\frac{1}{f'(x_n)} \quad , \quad x_{n+1} \quad = \quad x_n + \Delta x. \tag{59}$$

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

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

Else:

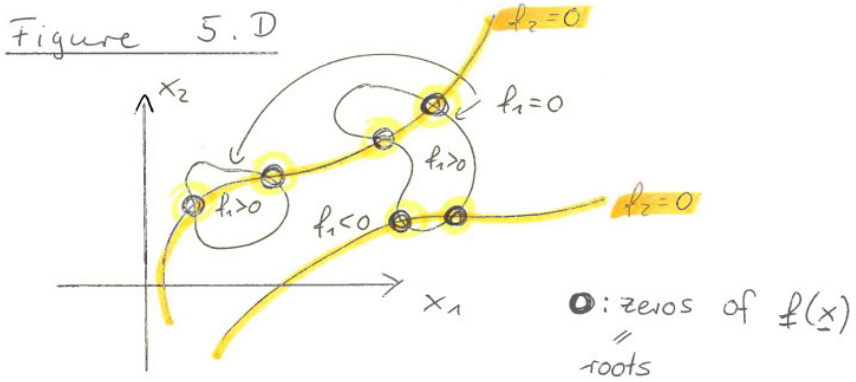
→ $n = n + 1$.
Go to (1).

- 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/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 $f_2(x_1, x_2) = 0$.



  – $N > 2$:
  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:
  - $$0 \;=\; f_j(\mathbf{x}_n + \vec{\delta}) \;=\; f_j(\mathbf{x}_n) + \underbrace{\frac{\partial f_j(\mathbf{x})}{\partial x_k}}_{J_{jk}(\mathbf{x})}\Big|_{\mathbf{x}=\mathbf{x}_n} \delta_k + \mathcal{O}(\delta^2) \quad , \quad j = 1, \ldots, N \tag{60}$$

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

    $$0 \;=\; \mathbf{f}(\mathbf{x}_n) + J(\mathbf{x}_n)\vec{\delta} + \mathcal{O}(\delta^2). \tag{61}$$

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

    $$0 \;=\; \mathbf{f}(\mathbf{x}_n) + J(\mathbf{x}_n)\Delta\mathbf{x} \tag{62}$$

    or equivalently

    $$\Delta\mathbf{x} \;=\; -\Big(J(\mathbf{x}_n)\Big)^{-1}\mathbf{f}(\mathbf{x}_n) \tag{63}$$

    ($\Delta\mathbf{x} \approx \vec{\delta}$, i.e. approximate difference between root and $\mathbf{x}_n$).
  - (62) 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 (63) becomes

    $$\Delta x \;=\; -\frac{1}{f'(x_n)} f(x_n), \tag{64}$$

    which is identical to (59), 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)

  $$\Delta\mathbf{x} \;=\; -\Big(J(\mathbf{x}_n)\Big)^{-1}\mathbf{f}(\mathbf{x}_n) \quad , \quad \mathbf{x}_{n+1} \;=\; \mathbf{x}_n + \Delta\mathbf{x}. \tag{65}$$

  - *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,

$$m_j \ddot{\mathbf{r}}_j(t) \quad = \quad \mathbf{F}_j(\mathbf{r}_1(t), \ldots, \mathbf{r}_N(t), \dot{\mathbf{r}}_1(t), \ldots, \dot{\mathbf{r}}_N(t), t) \quad , \quad j = 1, \ldots, N, \tag{66}$$

boundary conditions

$$\mathbf{r}_j(t_1) \quad = \quad \mathbf{r}_{j,1} \quad , \quad \mathbf{r}_j(t_2) \quad = \quad \mathbf{r}_{j,2} \tag{67}$$

("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,

$$-\frac{\hbar^2}{2m}\psi''(x) + V(x)\psi(x) \quad = \quad E\psi(x), \tag{68}$$

boundary conditions

$$\psi(x_1) \quad = \quad \psi(x_2) \quad = \quad 0 \tag{69}$$

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

  - Example appropriate? E is unknown, i.e. (68) and (69) 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,

$$-\frac{\hbar^2}{2m}\psi''(x) + V(x)\psi(x) \quad = \quad E(x)\psi(x) \quad , \quad E'(x) \quad = \quad 0, \tag{70}$$

    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,

$$\mathbf{y}'(x) \quad = \quad \mathbf{f}(\mathbf{y}(x), x) \tag{71}$$

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

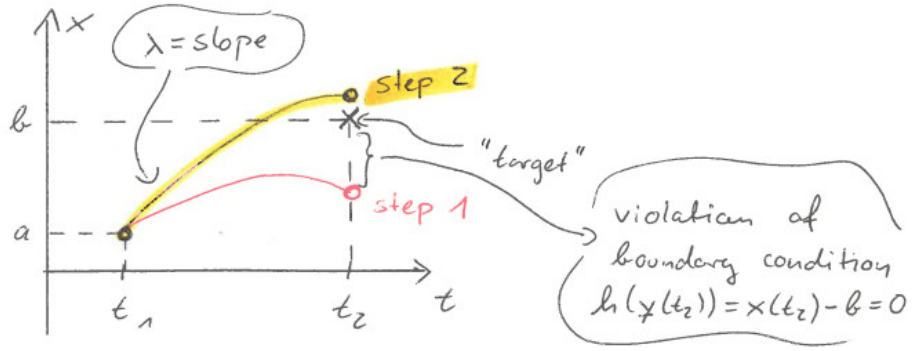
$$g_j(\mathbf{y}(x_1)) \quad = \quad 0 \quad , \quad j = 1, \ldots, n < N \tag{72}$$
$$h_j(\mathbf{y}(x_2)) \quad = \quad 0 \quad , \quad j = 1, \ldots, N - n. \tag{73}$$

- 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 $t = t_2$.

Figure 6.A



  - 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:

$$\lambda \quad \rightarrow \quad \lambda - \frac{h(\mathbf{y}(t_2))}{dh(\mathbf{y}(t_2))/d\lambda}. \tag{74}$$

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

6.2.1 Example: QM, 1 dimension, infinite potential well

- Infinite potential well:

$$V(x) \quad = \quad \begin{cases} 0 & \text{if } 0 \le x \le L \\ \infty & \text{otherwise} \end{cases} . \tag{75}$$

- Schrödinger equation and boundary conditions:

$$-\frac{\hbar^2}{2m}\psi''(x) \quad = \quad E\psi(x) \quad , \quad \psi(x=0) \quad = \quad \psi(x=L) \quad = \quad 0. \tag{76}$$

- Reformulate equations using exclusively dimensionless quantities:

$$\hat{x} \quad = \quad \frac{x}{L} \tag{77}$$

$$\rightarrow \quad \frac{d}{d\hat{x}} \quad = \quad L\frac{d}{dx} \tag{78}$$

$$\rightarrow \quad -\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) \quad = \quad \underbrace{\frac{2mEL^2}{\hbar^2}}_{=\hat{E}}\psi(\hat{x}) \tag{79}$$

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

$$-\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) \quad = \quad \hat{E}\psi(\hat{x}) \quad , \quad \psi(\hat{x}=0) \quad = \quad \psi(\hat{x}=1) \quad = \quad 0. \tag{80}$$

- Analytical solution (to check numerical results):

$$\psi(\hat{x}) \quad = \quad \sqrt{2}\sin(n\pi\hat{x}) \quad , \quad \hat{E} \quad = \quad \pi^2 n^2 \quad , \quad n = 1, 2, \ldots \tag{81}$$

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

$$\psi'(\hat{x}) \quad = \quad \phi(\hat{x}) \quad , \quad \phi'(\hat{x}) \quad = \quad -\hat{E}(\hat{x})\psi(\hat{x}) \quad , \quad \hat{E}'(\hat{x}) \quad = \quad 0, \tag{82}$$

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

$$\mathbf{y}(x) \quad = \quad \Big(\psi(\hat{x}), \phi(\hat{x}), \hat{E}(\hat{x})\Big) \quad , \quad \mathbf{f}(\mathbf{y}(x), x) \quad = \quad \Big(\phi(\hat{x}), -\hat{E}(\hat{x})\psi(\hat{x}), 0\Big). \tag{83}$$

  - 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,

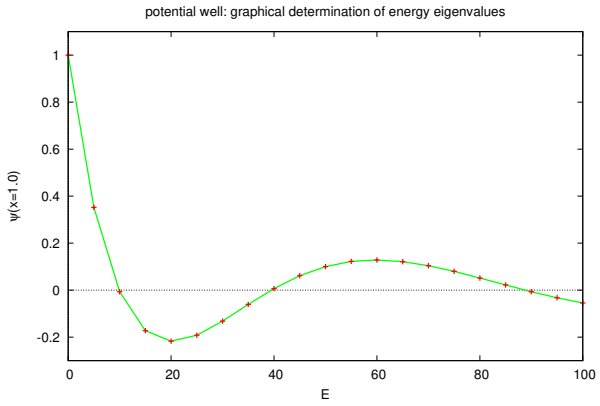


Figure 6: Infinite potential well, crude graphical determination of energy eigenvalues.

  - ∗ 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 wave functions of the three lowest states (4th order RK).
  - ∗ Convergence after three Newton-Raphson steps (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 .

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 .
```

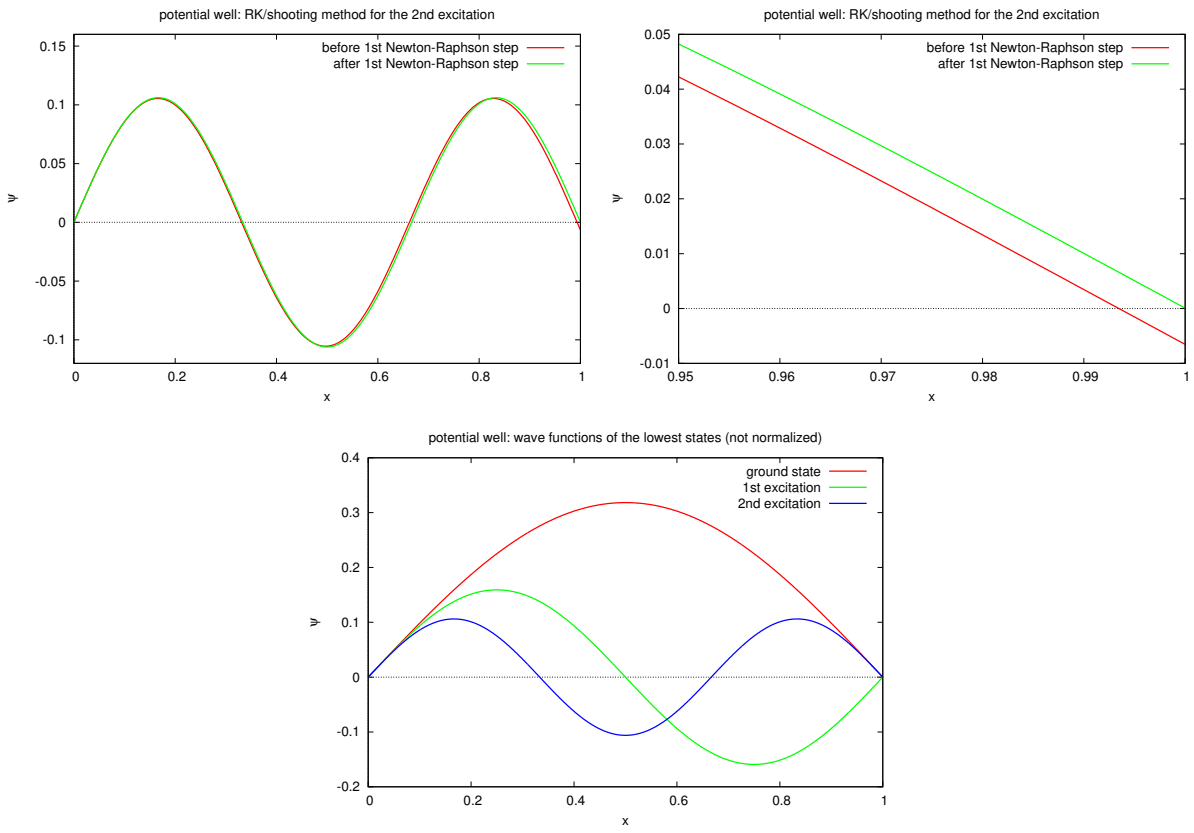


Figure 7: Infinite potential well. **(top)** First Newton-Raphson step for the second excitation. **(bottom)** Wave functions of the three lowest states.

6.2.2 Example: QM, 1 dimension, harmonic oscillator

- Schrödinger equation and boundary conditions:

$$-\frac{\hbar^2}{2m}\psi''(x) + \frac{m\omega^2}{2}x^2\psi(x) \quad = \quad E\psi(x) \quad , \quad \psi(x=-\infty) \quad = \quad \psi(x=+\infty) \quad = \quad 0 \qquad (84)$$

  (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 $a = (\hbar/m\omega)^{1/2}$.

  –

$$\hat{x} \quad = \quad \frac{x}{a} \qquad (85)$$

$$\rightarrow \quad \frac{d}{d\hat{x}} \;=\; a\frac{d}{dx} \tag{86}$$

$$\rightarrow \quad -\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) + \hat{x}^2\psi(\hat{x}) \;=\; \underbrace{\frac{2E}{\hbar\omega}}_{=\hat{E}}\,\psi(\hat{x}) \tag{87}$$

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

$$-\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) + \hat{x}^2\psi(\hat{x}) \;=\; \hat{E}\psi(\hat{x}) \quad , \quad \psi(\hat{x}=-\infty) \;=\; \psi(\hat{x}=+\infty) \;=\; 0. \tag{88}$$

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

$$P\psi(x) = \lambda\psi(x) \quad \rightarrow \quad \underbrace{PP}_{=1}\psi(x) = \lambda^2\psi(x) \quad \rightarrow \quad \lambda^2 \;=\; 1 \quad \rightarrow \quad \lambda \;=\; \pm 1. \tag{89}$$

    - 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.

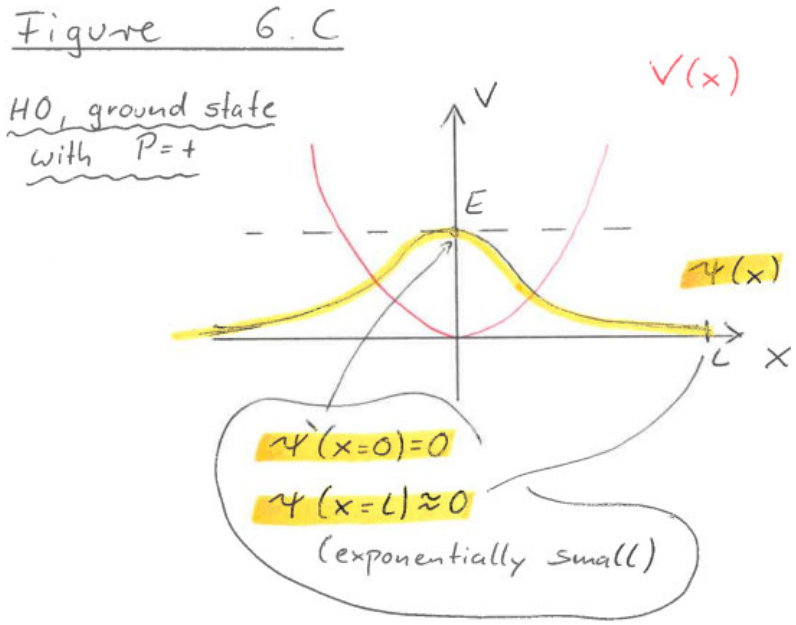
    $\rightarrow$ $P = +$

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

    $\rightarrow$ $P = -$

$$\rightarrow \quad \psi(x) \;=\; -\psi(-x) \quad \rightarrow \quad \psi(x=0) \;=\; 0. \tag{91}$$

- Numerical problems with boundary conditions $\psi(\hat{x} = -\infty) = \psi(\hat{x} = +\infty) = 0$ (eq. (88)).
- Numerical solution, first attempt:
  - Use either $\psi'(\hat{x} = 0) = 0$ or $\psi(\hat{x} = 0) = 0$ ((90) or (91)) 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:

$$\psi'(\hat{x}) = \phi(\hat{x}) \quad , \quad \phi'(\hat{x}) = \left(\hat{x}^2 - \hat{E}(\hat{x})\right)\psi(\hat{x}) \quad , \quad \hat{E}'(\hat{x}) = 0, \tag{92}$$

    i.e.

$$\mathbf{y}(x) = \left(\psi(\hat{x}), \phi(\hat{x}), \hat{E}(\hat{x})\right) \quad , \quad \mathbf{f}(\mathbf{y}(x), x) = \left(\phi(\hat{x}), \left(\hat{x}^2 - \hat{E}(\hat{x})\right)\psi(\hat{x}), 0\right). \tag{93}$$

  - 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:

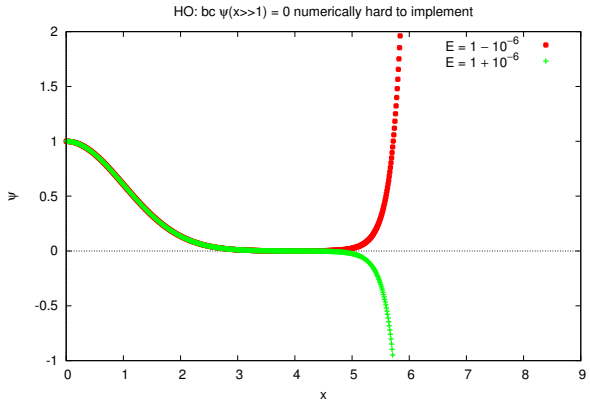


Figure 8: HO, numerical problems with boundary condition $\psi(\hat{x} = L/a) = 0$.

  - 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.
```

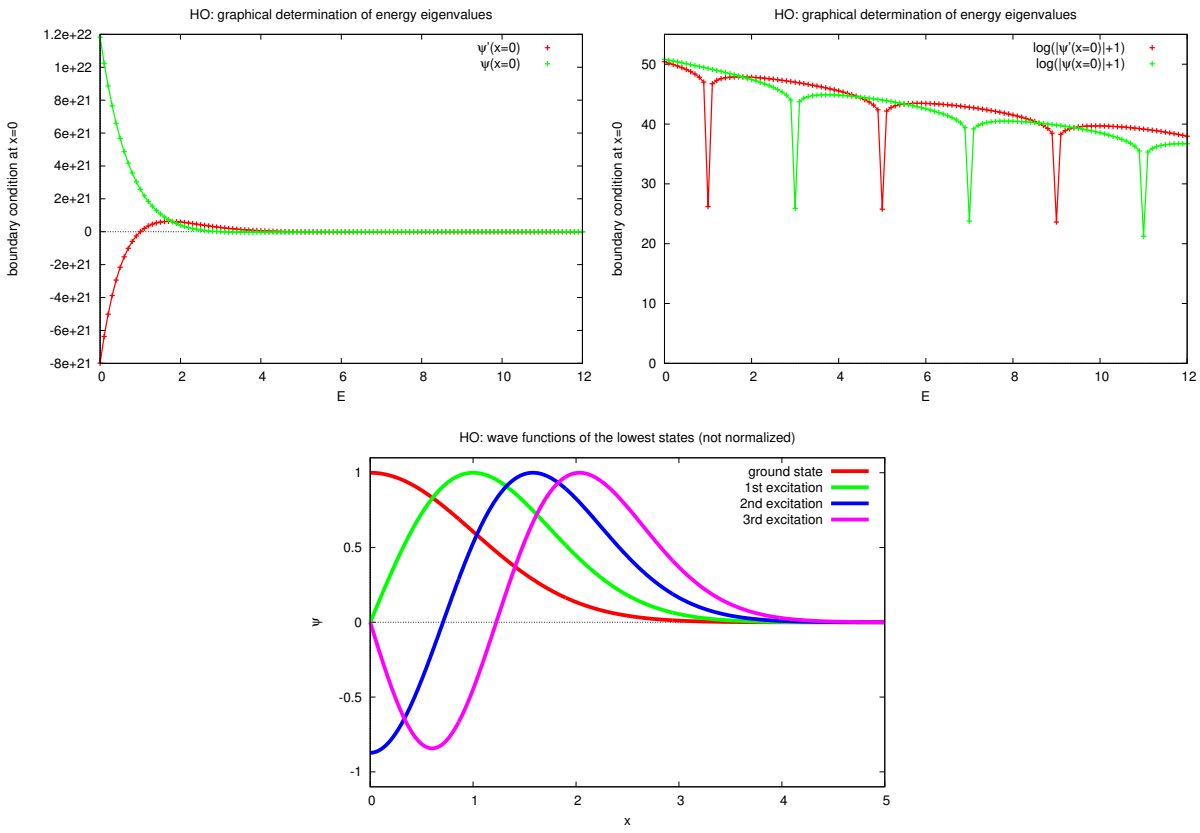


Figure 9: HO. **(top)** Crude graphical determination of energy eigenvalues. **(bottom)** Wave functions of the four lowest states.

```
E_num =  +5.000000.

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

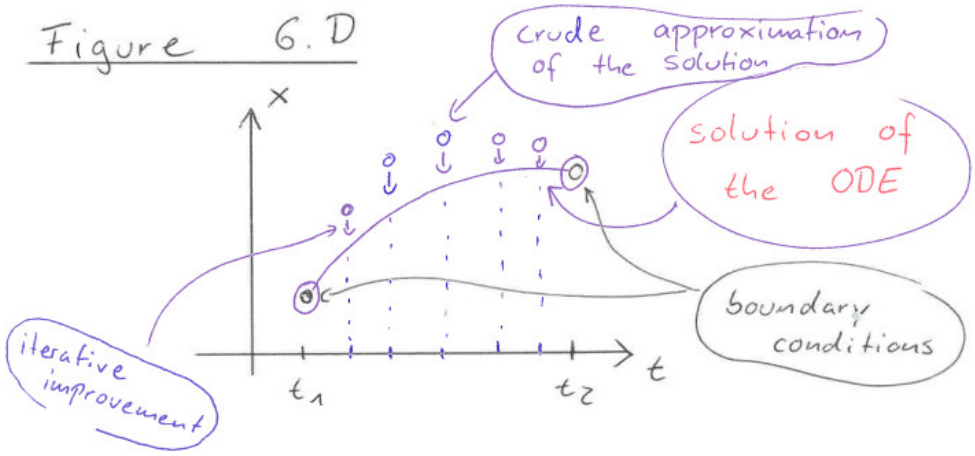
6.2.3 Example: QM, 3 dimensions, spherically symmetric potential

- 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].

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.



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.

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

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

- Step 1: elimination of column 0,

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

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

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

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

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

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

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

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

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

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

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

(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 $0 \ldots 0\, 1\, 0 \ldots 0$.

- (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:

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

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

    $$|a_{j,n-1}^{(n-1)}| \quad = \quad \max_k |a_{k,n-1}^{(n-1)}|. \tag{105}$$

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

  $\rightarrow$ 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

    $$\frac{|a_{j,n-1}^{(n-1)}|}{\max_l |a_{j,l}^{(0)}|} \quad = \quad \max_k \frac{|a_{k,n-1}^{(n-1)}|}{\max_l |a_{k,l}^{(0)}|} \tag{106}$$

    ("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.

$$\left|\begin{array}{cccc|ccc} a_{0,0}^{(0)} & a_{0,1}^{(0)} & a_{0,2}^{(0)} & \dots & b_{0,0}^{(0)} & \dots & b_{0,M-1}^{(0)} \\ 0 & a_{1,1}^{(1)} & a_{1,2}^{(1)} & \dots & b_{1,0}^{(1)} & \dots & b_{1,M-1}^{(1)} \\ 0 & 0 & a_{2,2}^{(2)} & \dots & b_{2,0}^{(2)} & \dots & b_{2,M-1}^{(2)} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \end{array}\right|. \tag{107}$$

  - Then backward substitution, i.e. computation of $\mathbf{x}_n$:
    - Start with
    $$x_{N-1,n} = \frac{b_{N-1,n}^{(N-1)}}{a_{N-1,N-1}^{(N-1)}}. \tag{108}$$
    - Then
    $$x_{N-2,n} = \frac{b_{N-2,n}^{(N-2)} - a_{N-2,N-1}^{(N-2)} x_{N-1,n}}{a_{N-2,N-2}^{(N-2)}}. \tag{109}$$
    - I.e. perform N steps $j = N-1, N-2, \dots, 0$,
    $$x_{j,n} = \frac{b_{j,n}^{(j)} - \sum_{k=j+1}^{N-1} a_{j,k}^{(j)} x_{k,n}}{a_{j,j}^{(j)}}. \tag{110}$$

- 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.
```

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,

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

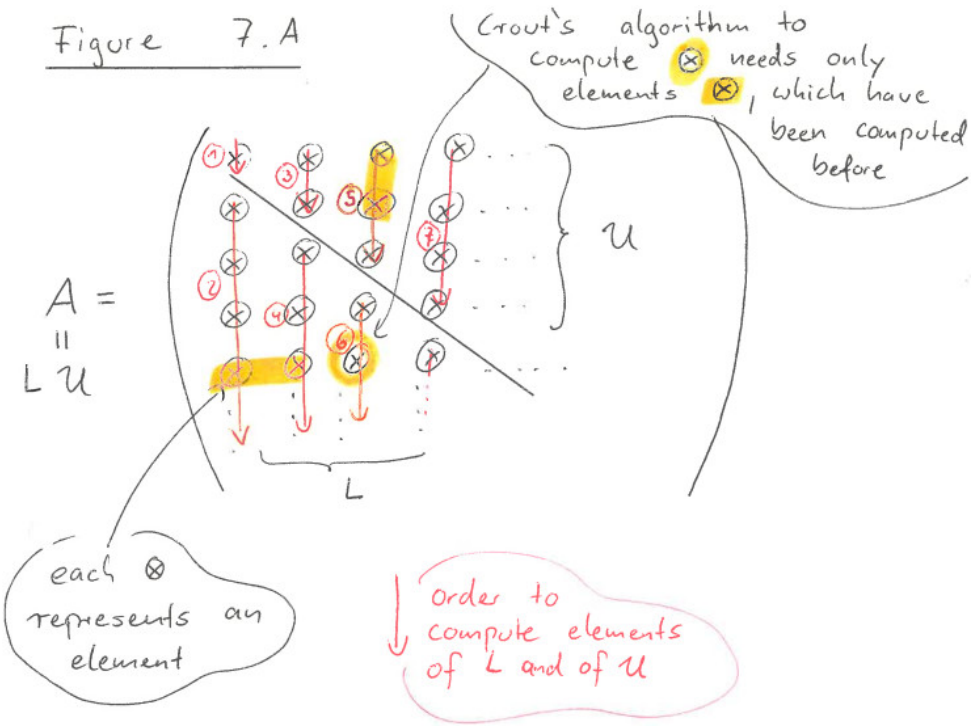
  with respect to $\alpha_{j,k}$ and $\beta_{j,k}$.

- Solving these equations is trivial, when treating 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

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

  via forward substitution,

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

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 $A\mathbf{x}_j = \mathbf{b}_j$ as well as 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.

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).

- Detailed equations:
  - $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ ("residual"), $\mathbf{p}_0 = \mathbf{r}_0$.
  - During each iteration:

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

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

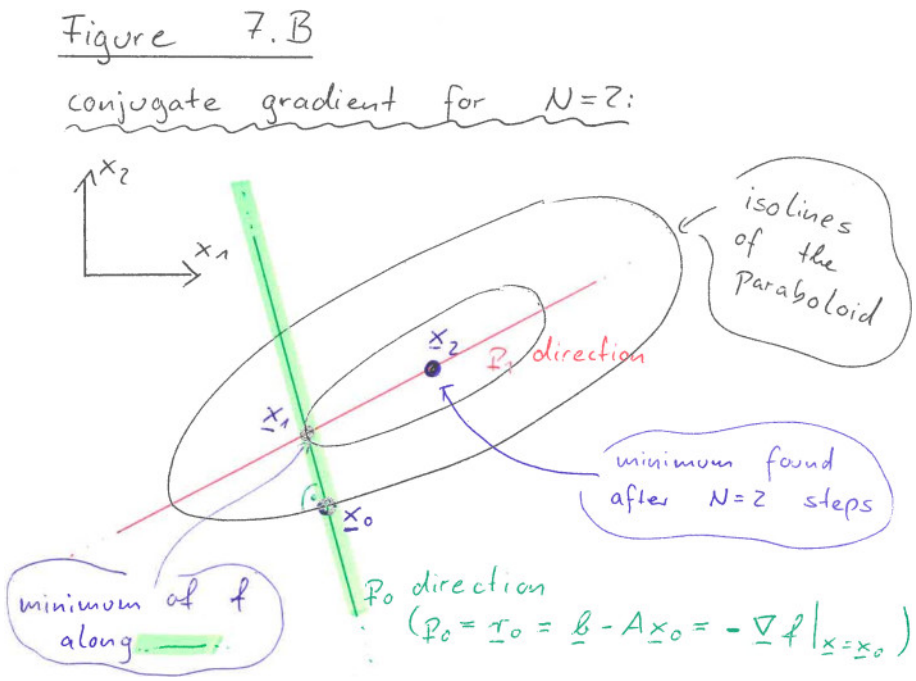
$$\beta_n = \frac{\mathbf{r}_{n+1} \mathbf{r}_{n+1}}{\mathbf{r}_n \mathbf{r}_n} \tag{128}$$

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

  (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 Generalizations

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

7.7.3 Condition number, preconditioning

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

$$A = U \, \mathrm{diag}(\omega_0, \omega_1, \ldots, \omega_{N-1}) \, V^T, \tag{130}$$

  where U and V are orthogonal matrices and $\omega_j \geq 0$ are the singular values of A [4].

- Condition number:

$$\mathrm{cond}(A) = \frac{\max_j(\omega_j)}{\min_j(\omega_j)}. \tag{131}$$

- If $\mathrm{cond}(A)$ 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).

- 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 $\mathrm{cond}(A^T A) = (\mathrm{cond}(A))^2$, i.e. the conjugate gradient method applied to $A^T A$ is significantly more inefficient than other generalized methods (see section 7.7.2) applied to A.

- Quite often preconditioning is advantageous:

[4] For symmetric positive definite matrices A, as discussed in section 7.7.1, singular values are identical to eigenvalues.

- 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

$$I = \int_a^b dx\, f(x). \tag{132}$$

- 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

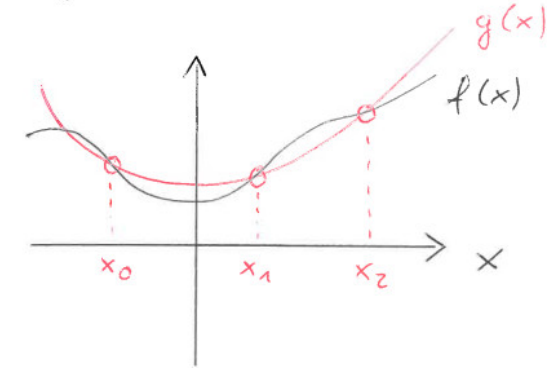
- Basic principle: Approximate $f(x)$ using a polynomial, integrate the polynomial analytically.
- In detail:
  - Approximate $f(x)$ using Lagrange polynomials:
    - Select $n+1$ points x_j, $j = 0, 1, \ldots, n$.
    - Compute/evaluate samples $f_j = f(x_j)$.
    - Langrange polynomials:

$$l_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k}, \tag{133}$$

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

$$f(x) \approx g(x) = \sum_{j=0}^{n} f_j l_j(x). \tag{134}$$

Figure 8.A



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

$$I = \int_a^b dx\, f(x) \approx \int_a^b dx\, g(x) = \int_a^b dx \sum_{j=0}^n f_j l_j(x) =$$
$$= \sum_{j=0}^n f_j \underbrace{\int_a^b dx\, l_j(x)}_{=w_j} = \sum_{j=0}^n f_j w_j. \tag{135}$$

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

∗ Even n:

$$\Delta I = \left| \int_a^b dx \Big(f(x) - g(x) \Big) \right| = \frac{1}{(n+2)!} \underbrace{\left(\int_a^b dx\, x \prod_{j=0}^n (x - x_j) \right)}_{\sim (b-a)^{n+3}} f^{(n+2)}(\xi) \tag{136}$$

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:

$$\Delta I = \left| \int_a^b dx \Big(f(x) - g(x) \Big) \right| = \frac{1}{(n+1)!} \underbrace{\left(\int_a^b dx \prod_{j=0}^n (x - x_j) \right)}_{\sim (b-a)^{n+2}} f^{(n+1)}(\xi) \tag{137}$$

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. [3].

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

$$I = \int_a^b dx\, f(x) \approx \left(\frac{1}{2} f_0 + \frac{1}{2} f_1 \right) h \tag{138}$$

$$\Delta I = \frac{1}{2} \left(\int_a^b dx\, \frac{(x-a)(x-b)}{2} \right) f''(\xi) = -\frac{h^3}{12} f''(\xi) = \mathcal{O}(h^3) \tag{139}$$

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

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

$$I = \int_a^b dx\, f(x) \approx \left(\frac{1}{3} f_0 + \frac{4}{3} f_1 + \frac{1}{3} f_2 \right) h \tag{140}$$

$$\Delta I = \ldots = -\frac{h^5}{90} f^{(4)}(\xi) = \mathcal{O}(h^5) \tag{141}$$

(coefficients 1/3, 4/3 and 1/3 can be obtained in a straightforward way from (135)).

Figure 8.B

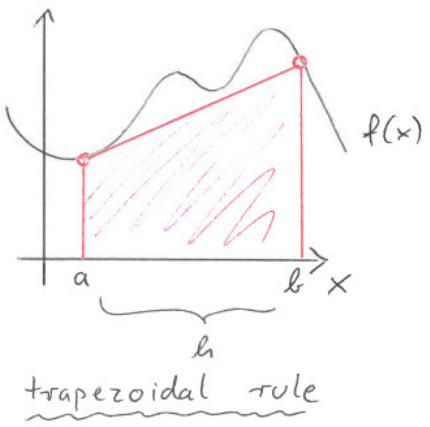
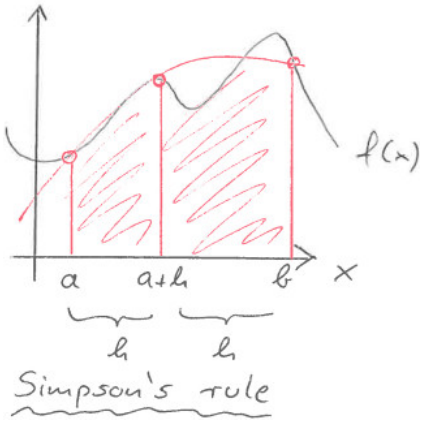


Figure 8.C



- There are further common "integration rules", e.g. Simpson's 3/8 rule or Boole's rule.
- Examples (see Ref. [3]):
  - $$I_1 = \int_0^1 dx \, \frac{1}{1+x^2} = \frac{\pi}{4}. \tag{142}$$
    - $I_1 = +0.7853\ldots$ (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).
  - $$I_2 = \int_0^1 dx \, e^x = e - 1. \tag{143}$$
    - $I_2 = +1.7182\ldots$ (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:
    $$I = \int_a^b dx \, f(x) \approx \left(\frac{1}{2}f_0 + f_1 + f_2 + \ldots + f_{N-1} + \frac{1}{2}f_N\right)h = T_N \tag{144}$$
    $$\Delta I = N \times \mathcal{O}(h^3) = \mathcal{O}(1/N^2). \tag{145}$$
    $(h = (b-a)/N)$.
  - 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).

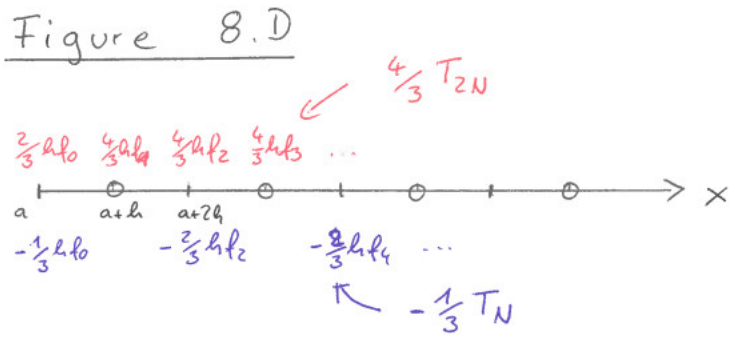
- **Iterated Simpson's rule**:
  - Approximate I according to

$$I = \int_a^b dx\, f(x) \approx \frac{4}{3}T_{2N} - \frac{1}{3}T_N. \tag{146}$$

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

$$\frac{4}{3}T_{2N} - \frac{1}{3}T_N = \left(\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \ldots + \frac{2}{3}f_{2N-2} + \frac{4}{3}f_{2N-1} + \frac{1}{3}f_{2N}\right)h \tag{147}$$

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



- Algorithm:

  (1) Compute T_N (eq. (144)).

  (2) Compute T_{2N} (eq. (144)), 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:*

    $\rightarrow$ End of algorithm.

    Else:

    $\rightarrow$ $N \rightarrow 2N$.
    Go to (1).

8.1.2 Gaussian integration

- Due to limited time not discussed.

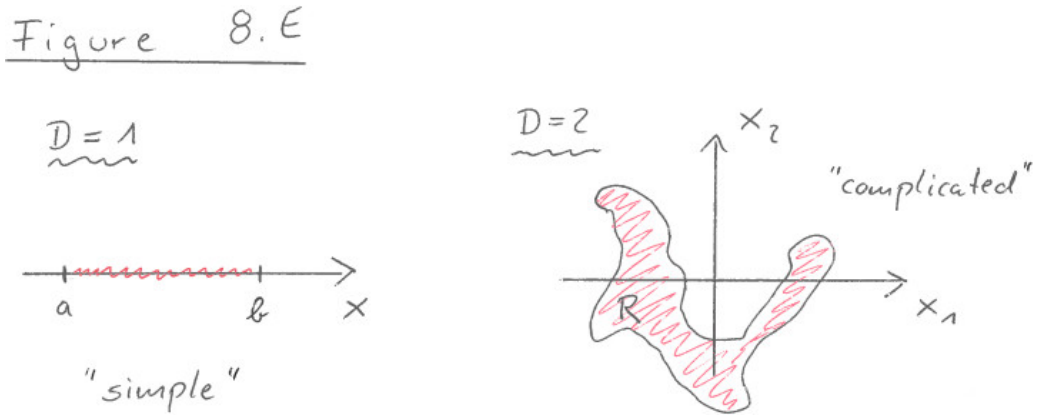
8.2 Numerical integration in $D \geq 2$ dimensions

- Goal: Compute the definite integral

$$I = \int_R d^D x \, f(\mathbf{x}), \tag{148}$$

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".



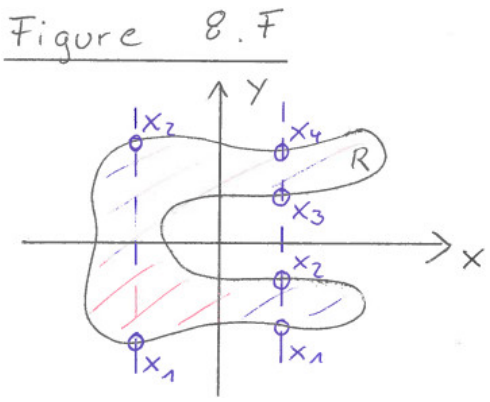
8.2.1 Nested 1-dimensional integration

- $D = 3$ 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,

$$I = \int_R d^3 x \, f(x, y, z) = \int_{x_1}^{x_2} dx \underbrace{\int_{y_1(x)}^{y_2(x)} dy \underbrace{\int_{z_1(x,y)}^{z_2(x,y)} dz \, f(x, y, z)}_{=I(x,y)}}_{=I(x)} . \tag{149}$$

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

$$\int_{y_1(x)}^{y_2(x)} dy \ldots \quad \rightarrow \quad \int_{y_1(x)}^{y_2(x)} dy \ldots + \int_{y_3(x)}^{y_4(x)} dy \ldots \tag{150}$$



- 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.
-

$$I \quad = \quad \int_R d^D x \, f(\mathbf{x}) \quad = \quad \underbrace{V(R)\langle f \rangle}_{=\bar{I} \approx I} \pm \underbrace{V(R) \left(\frac{\langle (f - \langle f \rangle)^2 \rangle}{N-1} \right)^{1/2}}_{=\Delta I}, \tag{151}$$

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

$$\langle f \rangle \quad = \quad \frac{1}{N} \sum_{j=1}^{N} f(\mathbf{x}_j) \tag{152}$$

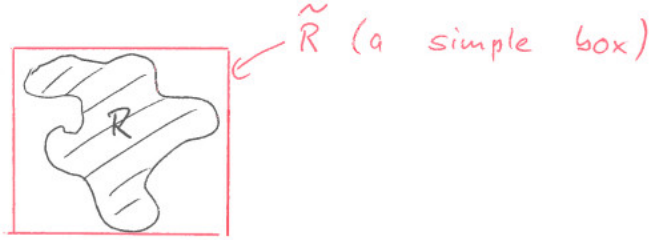
$$\langle f^2 \rangle \quad = \quad \frac{1}{N} \sum_{j=1}^{N} \Big(f(\mathbf{x}_j) \Big)^2. \tag{153}$$

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

$$g(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in R \\ 0 & \text{otherwise} \end{cases} . \tag{154}$$

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

Figure 8.6



*

$$I = \int_{\tilde{R}} d^D x \, f(\mathbf{x}) g(\mathbf{x}). \tag{155}$$

    * Right hand side of (155) 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.
  $\rightarrow$ Monte Carlo integration.

- Simple domain, integrand smooth
  $\rightarrow$ nested 1-dimensional integration.
- Strongly oscillating or discontinuous integrand
  $\rightarrow$ 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

$$A\mathbf{v}_j = \lambda_j \mathbf{v}_j \tag{156}$$

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

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

$$\det\Big(A - \lambda_j\Big) = 0. \tag{157}$$

- 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

$$A\mathbf{v}_j = \lambda_j B\mathbf{v}_j \tag{158}$$

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

  - Can be rewritten into a standard eigenvalue problem:

$$B^{-1}A\mathbf{v}_j = \lambda_j \mathbf{v}_j. \tag{159}$$

  - 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

$$A\mathbf{v}_j = \lambda_j L L^T \mathbf{v}_j \tag{160}$$

$$\underbrace{L^{-1}A(L^T)^{-1}}_{=A'}\underbrace{L^T\mathbf{v}_j}_{=\mathbf{v}'_j} = \lambda_j \underbrace{L^T\mathbf{v}_j}_{=\mathbf{v}'_j}, \tag{161}$$

which is a standard eigenvalue problem with a symmetric matrix A' ($B^{-1}A$ in (159) 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

$$\begin{aligned} A &\rightarrow \\ &\rightarrow (P_1)^{-1}AP_1 \rightarrow \\ &\rightarrow (P_2)^{-1}(P_1)^{-1}AP_1P_2 \rightarrow \\ &\dots \\ &\rightarrow \underbrace{(P_n)^{-1}\dots(P_2)^{-1}(P_1)^{-1}}_{=Q^{-1}} A \underbrace{P_1P_2\dots P_n}_{=Q} = Q^{-1}AQ \end{aligned} \tag{162}$$

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 $\mathbf{e}_j$.

- Matrix A:
  - Eigenvalues λ_j, because

$$\begin{aligned} \det(A-\lambda_j) &= \det(Q^{-1})\det\Big(A-\lambda_j\Big)\det(Q) = \det\Big(Q^{-1}\Big(A-\lambda_j\Big)Q\Big) = \\ &= \det\Big(Q^{-1}AQ-\lambda_j\Big), \end{aligned} \tag{163}$$

    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

$$Q^{-1}AQ\mathbf{e}_j = \lambda_j\mathbf{e}_j \tag{164}$$

$$\rightarrow \quad A(Q\mathbf{e}_j) = \lambda_j(Q\mathbf{e}_j). \tag{165}$$

- 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,

$$Q = \left(\mathbf{v}_1\ \mathbf{v}_2\ \dots\ \mathbf{v}_n\right) \quad \rightarrow \quad Q^{-1} = Q^T = \begin{pmatrix} (\mathbf{v}_1)^T \\ (\mathbf{v}_2)^T \\ \dots \\ (\mathbf{v}_n)^T \end{pmatrix}, \tag{166}$$

  which diagonalizes A, i.e.

$$Q^T A Q = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_N). \tag{167}$$

- 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,

$$A \rightarrow A' = (P_j)^T A P_j, \tag{168}$$

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

  –

$$P_j = \begin{pmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & +c & & & +s & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & -s & & & +c & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{pmatrix}, \tag{169}$$

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

  –

$$A'_{kl} = \underbrace{((P_j)^T)_{k,m}}_{=(P_j)_{m,k}} A_{m,n} (P_j)_{n,l}. \tag{170}$$

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

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

$$\frac{c^2 - s^2}{2sc} = \frac{A_{q,q} - A_{p,p}}{2A_{p,q}} \tag{171}$$

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

$$t^2 + 2\theta t - 1 = 0$$
$$\rightarrow \quad t = -\theta \pm \left(\theta^2 + 1\right)^{1/2}, \tag{172}$$

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

$$t = \frac{\mathrm{sign}(\theta)}{|\theta| + (\theta^2 + 1)^{1/2}}, \tag{173}$$

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

- Using (171) 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 (168) 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 (168): $Q \to Q' = QP_j$:
    * $Q'_{k,p} = cQ_{k,p} - sQ_{k,q}$.
    * $Q'_{k,q} = cQ_{k,q} + sQ_{k,p}$.
    * $l \neq p, q$: $Q'_{k,l} = Q_{k,l}$.

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):

$$L = \sum_{j=1}^{N} \frac{1}{2} m \dot{x}_j^2 - \sum_{j=1}^{N-1} \frac{1}{2} k (x_j - x_{j+1})^2. \tag{174}$$

Figure 9.A

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

$$m\ddot{x}_1 = +k(x_2 - x_1) \tag{175}$$
$$m\ddot{x}_2 = +k(x_3 - x_2) + k(x_1 - x_2) \tag{176}$$
$$\dots, \tag{177}$$

i.e. can be written in matrix form,

$$M\ddot{\mathbf{x}} = -K\mathbf{x} \tag{178}$$

$$M = \begin{pmatrix} m & & & \\ & m & & \\ & & m & \\ & & & \dots \end{pmatrix} \quad \text{(mass matrix)} \tag{179}$$

$$K = \begin{pmatrix} +k & -k & & \\ -k & +2k & -k & \\ & -k & +2k & \dots \\ & & \dots & \dots \end{pmatrix} \quad \text{(stiffness matrix)}, \tag{180}$$

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

- EOMs reformulated using dimensionless quantities:

$$\frac{d^2}{d\hat{t}^2}\hat{\mathbf{x}} = -\hat{K}\hat{\mathbf{x}} \tag{181}$$

$$\hat{K} = \begin{pmatrix} +1 & -1 & & \\ -1 & +2 & -1 & \\ & -1 & +2 & \dots \\ & & \dots & \dots \end{pmatrix}, \tag{182}$$

  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

$$\hat{\mathbf{x}} = \mathbf{v}_j e^{i\hat{\omega}_j \hat{t}} \tag{183}$$

  reduces the system of second order ODEs (181) to an eigenvalue problem,

$$-\hat{\omega}_j^2 \mathbf{v}_j = -\hat{K}\mathbf{v}_j. \tag{184}$$

- 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 E.

```
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:

$$\hat{\mathbf{x}} \;=\; \sum_{j=1}^{N} \underbrace{\mathbf{v}_{j-1}\Big(A_j \cos(\hat{\omega}_j \hat{t}) + B_j \sin(\hat{\omega}_j \hat{t})\Big)}_{\text{normal modes}}, \tag{185}$$

  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 $j = 1, \ldots, N$.

  *

$$\dot{\hat{\mathbf{x}}}(\hat{t}=0) \;=\; \sum_{j=1}^{N} \mathbf{v}_{j-1} B_j \hat{\omega}_j \;=\; 0 \quad\rightarrow\quad B_j \;=\; 0, \tag{186}$$

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

  *

$$\hat{\mathbf{x}}(\hat{t}=0) \;=\; \sum_{j=1}^{N} \mathbf{v}_{j-1} A_j \;=\; (1, 0, \ldots, 0) \quad\rightarrow\quad A_j \;=\; v_{j-1,1} \tag{187}$$

(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:

$$\hat{\mathbf{x}} \quad = \quad \sum_{j=1}^{N} \mathbf{v}_{j-1} v_{j-1,1} \cos(\hat{\omega}_j \hat{t}) \tag{188}$$

(see Figure 10, from which e.g. the speed of sound can be read off).

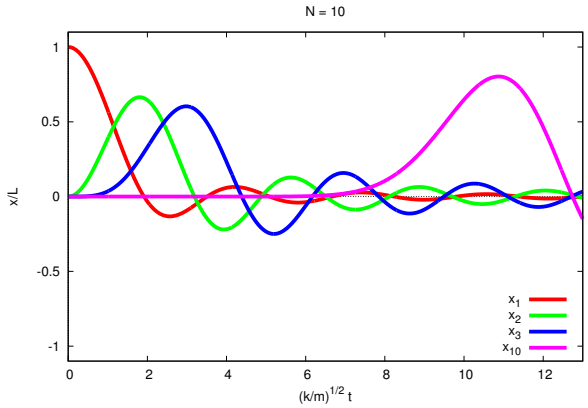


Figure 10: "Molecule oscillations in a 1-dimensional crystal" ($N = 10$ molecules).

- 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.

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):

$$l_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k} \tag{189}$$

$$g(x) = \sum_{j=0}^{n} f_j l_j(x). \tag{190}$$

- 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 quite different.
  - Examples for $N = 3$ and $N = 9$ are shown in Figure 11.

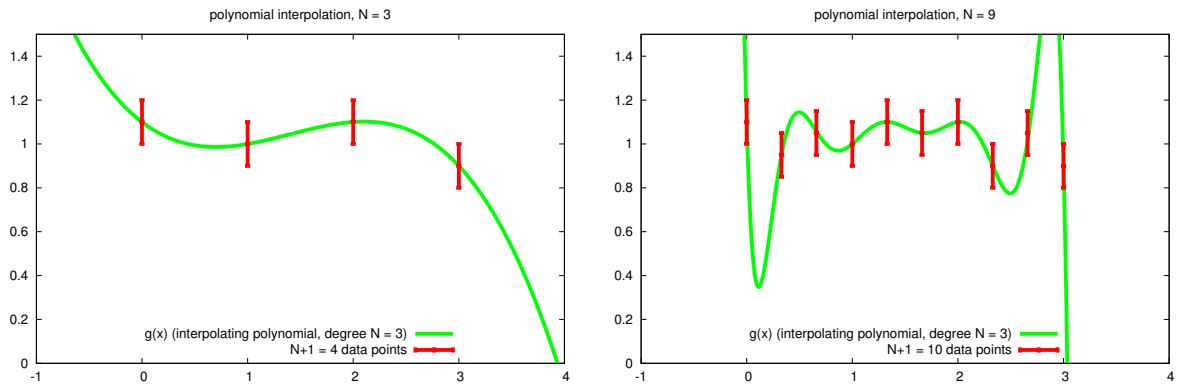


Figure 11: Polynomial interpolation of $N+1$ data points f_j for $N = 3$ and $N = 9$. While the data points are consistent with a constant, i.e. $f(x) = \text{const}$, the interpolating degree-N polynomials $g(x)$ are oscillating.

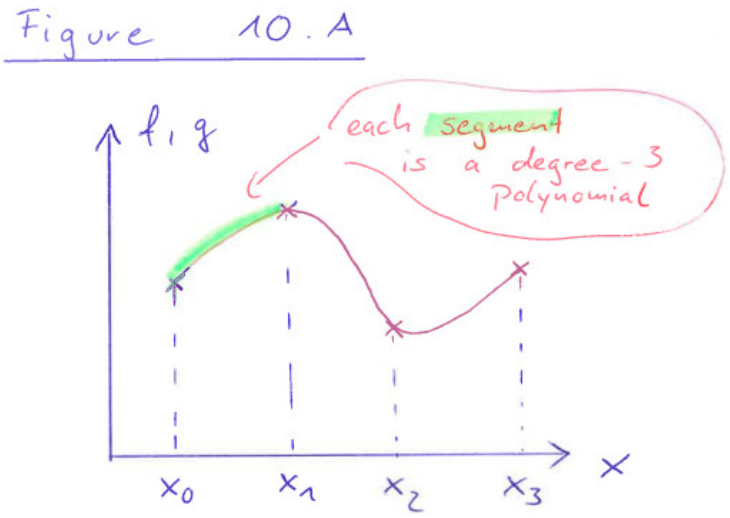
10.2 Cubic spline interpolation

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

$$g(x) \quad = \quad y_j(x) \quad \text{for } x_j \le x \le x_{j+1}, \tag{191}$$

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).



- 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 spline:

  - To interpolate data points f_j, $j = 0, \ldots, N$, degree-1 polynomials are sufficient:

    $$y_j(x) = f_j A(x) + f_{j+1} B(x), \tag{192}$$

    where

    $$A(x) = \frac{x_{j+1} - x}{x_{j+1} - x_j} \tag{193}$$

    $$B(x) = \frac{x - x_j}{x_{j+1} - x_j} \tag{194}$$

    are the Lagrange polynomials (189) 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),

    $$y_j(x) = f_j A(x) + f_{j+1} B(x) + f''_j C(x) + f''_{j+1} D(x), \tag{195}$$

    where

    $$C(x) = \frac{1}{6}\Big(A(x)^3 - A(x)\Big)(x_{j+1} - x_j)^2 \tag{196}$$

    $$D(x) = \frac{1}{6}\Big(B(x)^3 - B(x)\Big)(x_{j+1} - x_j)^2. \tag{197}$$

  - 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 (195):

      $$\frac{x_j - x_{j-1}}{6} f''_{j-1} + \frac{x_{j+1} - x_{j-1}}{3} f''_j + \frac{x_{j+1} - x_j}{6} f''_{j+1} = \\ = \frac{f_{j+1} - f_j}{x_{j+1} - x_j} - \frac{f_j - f_{j-1}}{x_j - x_{j-1}}. \tag{198}$$

    * 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 12 shows a cubic spline interpolating the data points already used in Figure 11, right ($N = 9$ example); in contrast to the degree-9 polynomial from Figure 11, the cubic spline does not exhibit any unnecessary oscillations.

- 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, ...

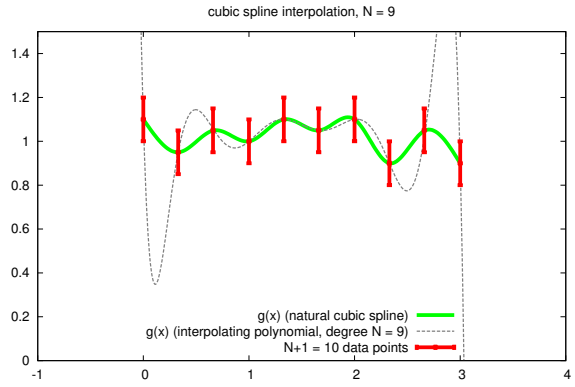


Figure 12: Cubic spline interpolation of $N+1=10$ data points f_j.

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, $g(x;\mathbf{a})=a_0+a_1x+a_2x^2$ ...
  - ... or $g(x;\mathbf{a})=a_0/x$ (if f_j describe a Coulomb-like potential) ...
  - ... or $g(x;\mathbf{a})=(a_0/x)\exp(-a_1x)$ (if f_j describe a potential with a limited range) ...
  - ...
- Determine $\mathbf{a}$ by minimizing

$$G(\mathbf{a}) \quad = \quad \sum_{j=0}^{N}\Big(g(x_j;\mathbf{a})-f_j\Big)^2 \tag{199}$$

  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

$$\nabla^{(\mathbf{a})}G(\mathbf{a}) \quad = \quad 0. \tag{200}$$

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

$$g(x;\mathbf{a}) \quad = \quad \sum_{j=0}^{M} a_j g_j(x) \tag{201}$$

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

- Insert (201) in (199):

$$G(\mathbf{a}) = \sum_{j=0}^{N}\Big(\sum_{k=0}^{M} a_k g_k(x_j) - f_j\Big)\Big(\sum_{l=0}^{M} a_l g_l(x_j) - f_j\Big) = \\ = \sum_{j=0}^{N}\Big(\sum_{k=0}^{M} A_{j,k}a_k - f_j\Big)\Big(\sum_{l=0}^{M} A_{j,l}a_l - f_j\Big), \tag{202}$$

  where

$$A = \begin{pmatrix} g_0(x_0) & g_1(x_0) & \dots & g_M(x_0) \\ g_0(x_1) & g_1(x_1) & \dots & g_M(x_1) \\ \vdots & \vdots & & \vdots \\ g_0(x_N) & g_1(x_N) & \dots & g_M(x_N) \end{pmatrix}. \tag{203}$$

- (200):

$$\frac{\partial}{\partial a_m}G(\mathbf{a}) = 2\sum_{j=0}^{N}\Big(\sum_{k=0}^{M} A_{j,k}a_k - f_j\Big)A_{j,m} = 0$$

$$\rightarrow \quad A^T A\mathbf{a} = A^T\mathbf{f} \tag{204}$$

  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:
  - (200) 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 the system of non-linear equations, e.g. by using the Newton-Raphson method.

- Figure 13 shows the least squares approximation of data points already used in Figure 11, right ($N = 9$ example) and Figure 12 using degree-0, degree-1 and degree-2 polynomials; in contrast to Figure 11 and Figure 12, there are no oscillations.

10.4 χ^2 minimizing fits

- Quite often, data points have errors, which have been ignored so far.
- Notation: σ_j is 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 impose stronger constraints on the parameters $\mathbf{a}$ than data points with large errors.

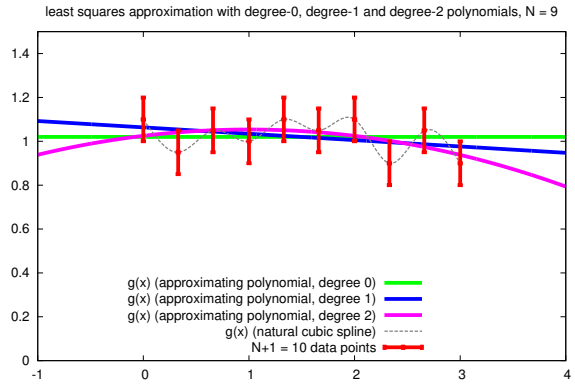


Figure 13: Least squares approximation with degree-0, degree-1 and degree-2 polynomials of $N+1=10$ data points f_j.

- Replace (199) by

$$\chi^2 \quad = \quad \sum_{j=0}^{N} \left(\frac{g(x_j;\mathbf{a}) - f_j}{\sigma_j} \right)^2 \tag{205}$$

  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 (205) should be of order 1
    $\rightarrow$ reduced $\chi^2 = \chi^2/\mathrm{dof} = \chi^2/(N-M) \approx 1$.
  - $\chi^2/\mathrm{dof} \gg 1$
    $\rightarrow$ ansatz $g(x_j;\mathbf{a})$ not consistent with data points.
  - $\chi^2/\mathrm{dof} \ll 1$
    $\rightarrow$ 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

$$\chi^2 \quad = \quad \sum_{j=0}^{N} \left(\frac{a - f_j}{\sigma_j} \right)^2 \tag{206}$$

    is equivalent to solving

$$0 \quad = \quad \frac{d}{da}\chi^2 \quad = \quad 2\sum_{j=0}^{N} \frac{a - f_j}{(\sigma_j)^2}, \tag{207}$$

i.e.

$$a = \sum_{j=0}^{N} \underbrace{\frac{1/(\sigma_j)^2}{\sum_{k=0}^{N} 1/(\sigma_k)^2}}_{=w_j} f_j = \sum_{j=0}^{N} w_j f_j, \tag{208}$$

where w_j is the "weight of data point f_j" ($0 \leq w_j \leq 1$, $\sum_{j=0}^{N} w_j = 1$).

– An example is shown in Figure 14.

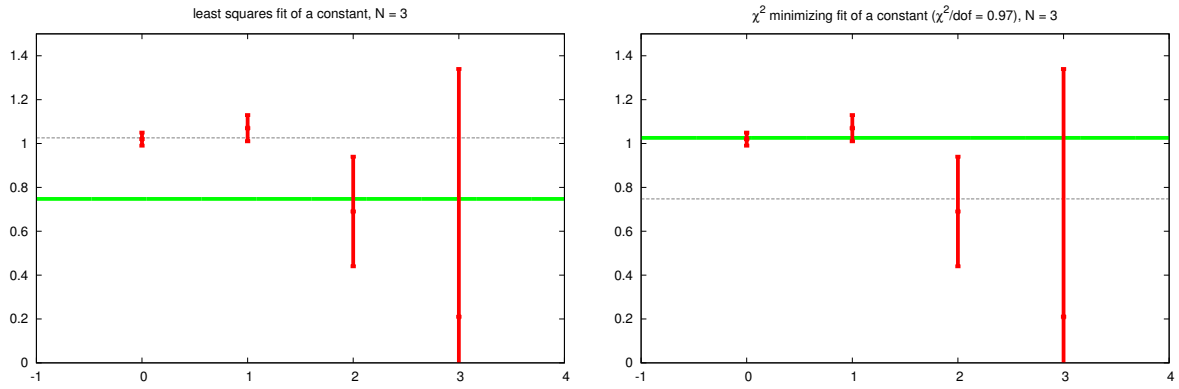


Figure 14: Comparison of a least squares fit (**left**) and a χ^2 minimizing fit (**right**) of a constant to $N + 1 = 4$ data points f_j.

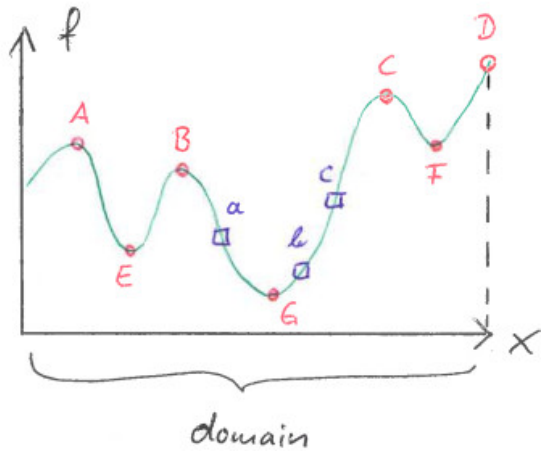
- Error estimates for fit parameters **a**:
  - Jackknife method.
  - Resampling.
  - Due to limited time not discussed.

11 Function minimization, optimization

11.1 Problem definition, general remarks

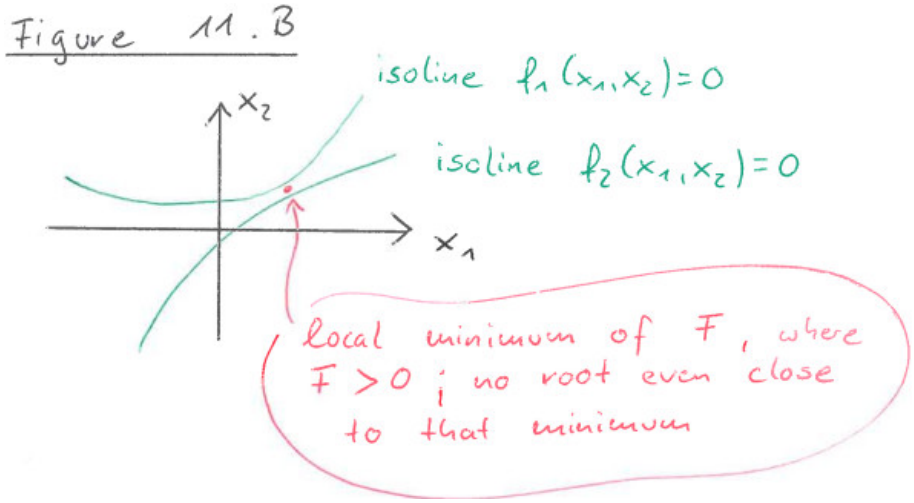
- $f(\mathbf{x})$: real valued function ($\mathbf{x} = (x_1, \ldots, x_D)$).
- Problem: find a local or the global minimum of $f(\mathbf{x})$, i.e. a value $\mathbf{x}$ minimizing $f(\mathbf{x})$ locally or globally.
- Function maximization is equivalent to function minimization, because maximum of $f(\mathbf{x})$ 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 $f(x)$ (i.e. 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).

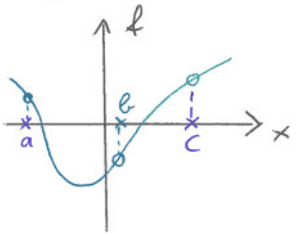
- Root finding versus function minimization:
  - At first glance problems seem to be very similar: $\mathbf{f}(\mathbf{x}) = 0$ versus $\nabla f(\mathbf{x}) = 0$.
  - There are, however, significant differences:
    * Root finding:
      $f_1(\mathbf{x})$, $f_2(\mathbf{x})$, ... are independent functions.
      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 $\mathbf{f}(\mathbf{x})$.



11.2 Golden section search in $D = 1$ dimension

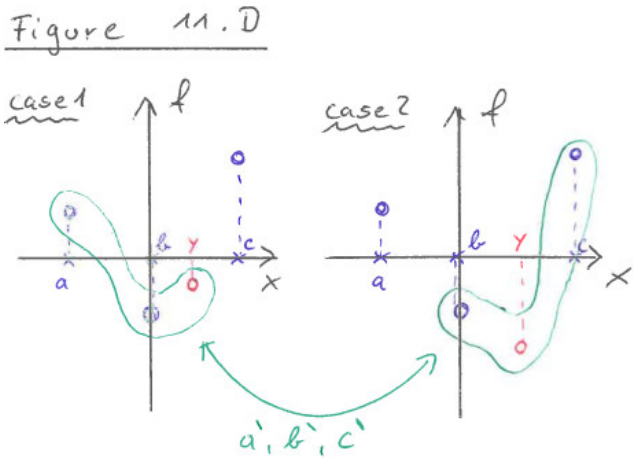
- Similar to bisection for root finding.
- Starting point: minimum localized inside interval $[a, c]$ via $f(a) > f(b) < f(c)$, where $a < b < c$ (minimum can be left or right of b).

Figure 11.C



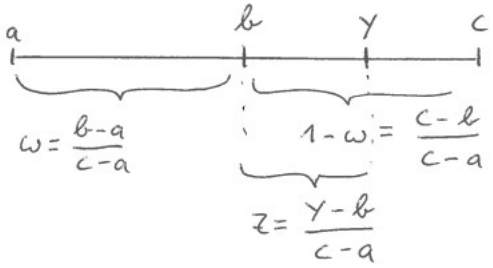
- Evaluate $f(y)$:
  - $b < y < c$:
    - If $f(y) > f(b)$ (case 1):
      → Replace (a, b, c) by $(a', b', c') = (a, b, y)$, i.e. minimum now localized inside interval $[a, y]$.
    - If $f(y) < f(b)$ (case 2):
      → Replace (a, b, c) by $(a', b', c') = (b, y, c)$, i.e. minimum now localized inside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

Figure 11.D



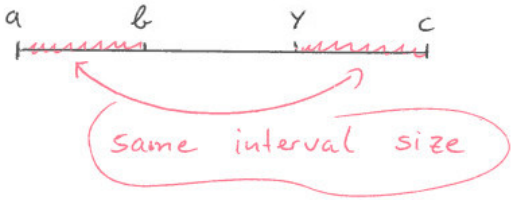
  - $a < y < b$:
    - Analogous to $b < y < c$.
  - Iterate this step, until $c - a$ is sufficiently small.
- How to choose y, to reduce the size of the interval $[a, c]$ as quickly as possible?
  - Define relative sizes of subintervals:
    - $w = (b - a)/(c - a)$.
    - $1 - w = (c - b)/(c - a)$.
 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

Figure 11.E



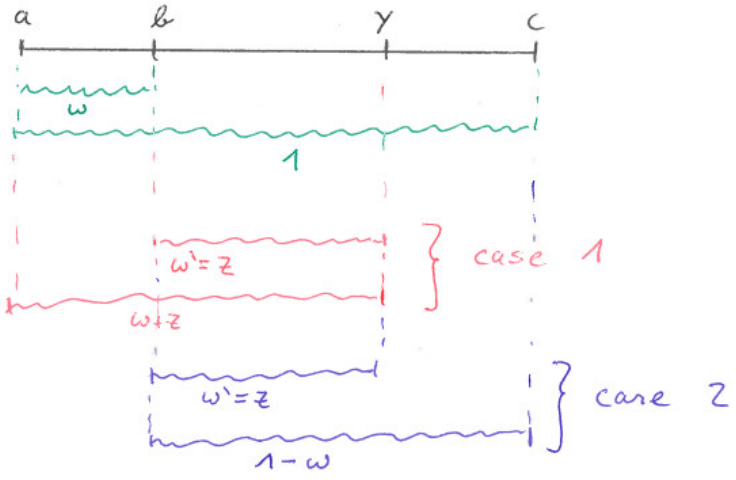
- Relative size of "new interval" $[a', c']$:
  - Case 1: $w + z$.
  - Case 2: $1 - w$.
- Determine z (and thereby y) via $w + z = 1 - w$
  $\rightarrow z = 1 - 2w = (1 - w) - w$, i.e. z is chosen such that the reduction of the size of $[a, b]$ is in both cases the same ("no bad case").

Figure 11.F



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

Figure 11.G



* 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": $(1 + \sqrt{5})/2$.

– $w = (3 - \sqrt{5})/2$ is a stable fixed point (can be shown), i.e. even when starting with $w \neq (3 - \sqrt{5})/2$, w will be closer to $(3 - \sqrt{5})/2$ after every step.

– Consequently,

$$c' - a' = \underbrace{\left(1 - \frac{3 - \sqrt{5}}{2}\right)}_{0.618\ldots}(c - a), \tag{209}$$

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

- Accuracy:

  – For $x \approx x_{\min}$

$$f(x) \approx f_{\min} + \frac{f''(x_{\min})}{2}(x - x_{\min})^2. \tag{210}$$

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

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

$$\frac{(f''(x_{\min})/2)(x - x_{\min})^2}{f_{\min}} > \epsilon, \tag{211}$$

  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 (211), is limited,

$$x_{\min,\text{numerically}} - x_{\min} \gtrsim \left(\frac{2f_{\min}}{f''(x_{\min})}\right)^{1/2}\sqrt{\epsilon} \sim \sqrt{\epsilon}. \tag{212}$$

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.

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.* $|\mathbf{x}_{n+1} - \mathbf{x}_n| < \epsilon$*):*

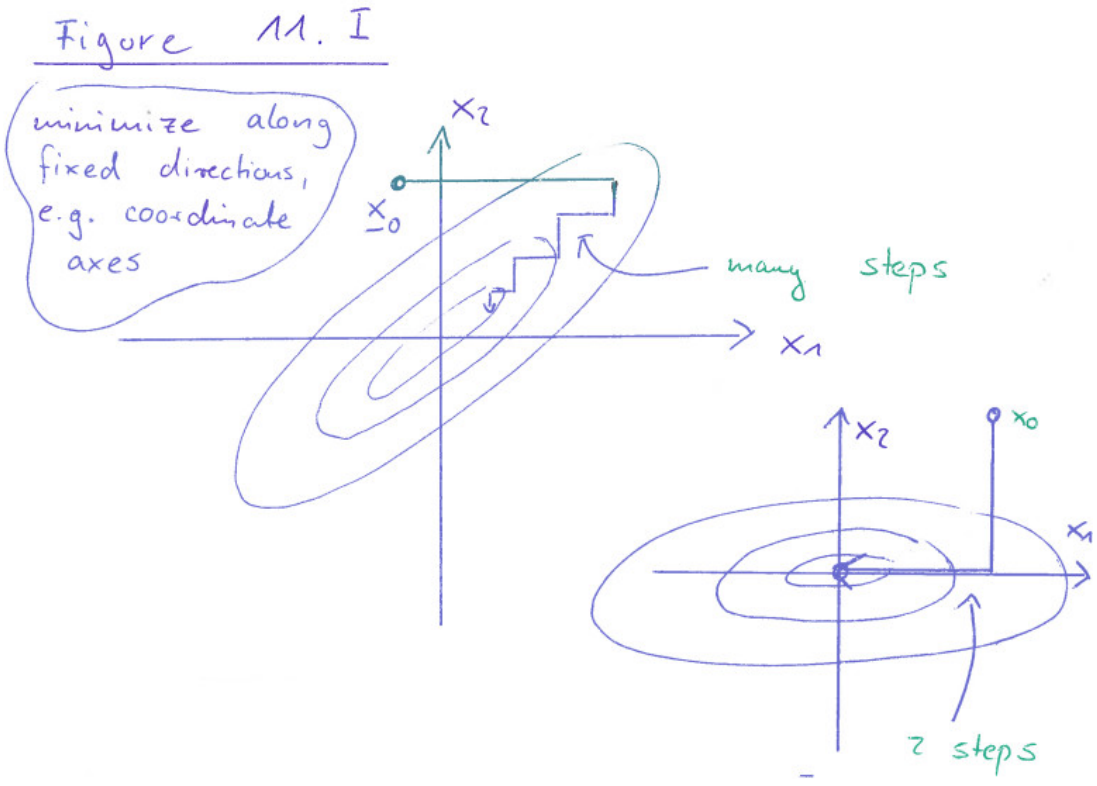
    $\rightarrow$ $\mathbf{x}_{n+1}$ *is approximate minimum.*
    End of algorithm.

    Else:

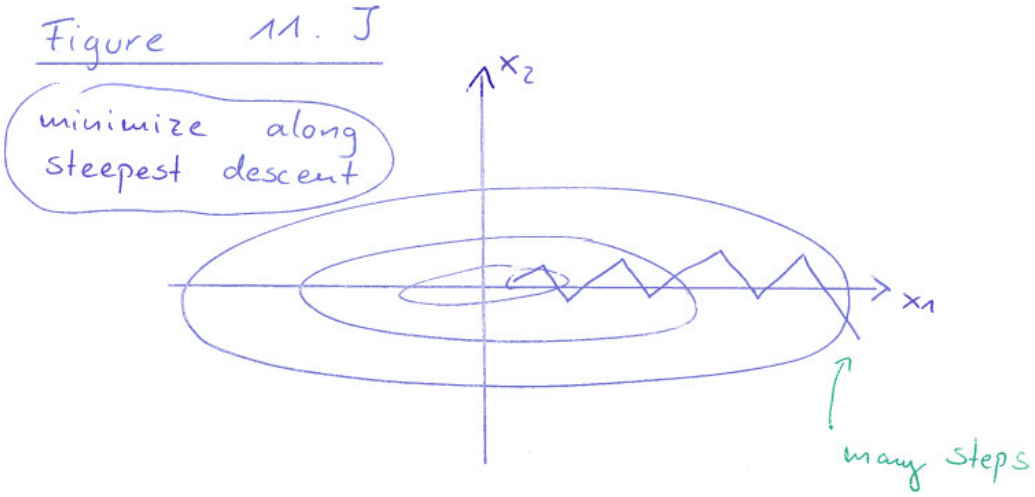
    $\rightarrow$ $n = n + 1$.
    Go to (1).
- Efficiency of the algorithm strongly depends, on how the directions $\mathbf{p}_n$ are chosen.
  - Simple example:
    - $D = 2$ dimensions.
    - $f(x_1, x_2)$: a paraboloid, which is "wide in $(+1, +1)/\sqrt{2}$ direction and narrow in $(-1, +1)/\sqrt{2}$ direction ",

      $$f(x_1, x_2) = \frac{x^2 + y^2 + 2xy}{2a^2} + \frac{x^2 + y^2 - 2xy}{2b^2} \tag{213}$$

      with $a \gg b$.
    - Directions for minimization $\mathbf{p}_n$: $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_1$, $\mathbf{e}_2$, ...
      $\rightarrow$ Many minimizations in 1 dimension required, algorithm quite inefficient.
    - Directions for minimization $\mathbf{p}_n$: $(+1, +1)/\sqrt{2}$, $(-1, +1)/\sqrt{2}$.
      $\rightarrow$ Only two minimizations in 1 dimension required, algorithm very efficient.
    - See also Ref. [1], Figure 10.7.1.



* 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 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 conjugate direction $\mathbf{p}_1$ such that 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,
  $$f(\mathbf{x}) \quad = \quad c - \mathbf{b}\mathbf{x} + \frac{1}{2}\mathbf{x}A\mathbf{x}. \tag{214}$$
  · Then
  $$\nabla f(\mathbf{x}) \quad = \quad -\mathbf{b} + A\mathbf{x}. \tag{215}$$
  · $\mathbf{p}_0 \nabla f(\mathbf{x}_1) = 0$ becomes
  $$\mathbf{p}_0 \Big(-\mathbf{b} + A\mathbf{x}_1 \Big) \quad = \quad 0. \tag{216}$$
  · The condition for conjugate directions $\mathbf{p}_0 \nabla f(\mathbf{x}_1 + \lambda \mathbf{p}_1) = 0$ becomes
  $$\mathbf{p}_0 \Big(-\mathbf{b} + A(\mathbf{x}_1 + \lambda \mathbf{p}_1) \Big) \quad = \quad \mathbf{p}_0 A \lambda \mathbf{p}_1 \quad = \quad 0, \tag{217}$$
  i.e. $\mathbf{p}_0$ and $\mathbf{p}_1$ are conjugate, if
  $$\mathbf{p}_0 A \mathbf{p}_1 \quad = \quad 0. \tag{218}$$
  · If $f(\mathbf{x})$ is not quadratic, select "approximate conjugate direction" via Taylor expansion of $f(\mathbf{x})$,
  $$\begin{aligned} f(\mathbf{x}) \quad = \quad & \underbrace{f(\mathbf{x}_1)}_{=c} + \sum_{j=1}^{D} \underbrace{\partial_j f(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_1}}_{=b_j} (x_j - x_{1,j}) \\ & + \frac{1}{2} \underbrace{\sum_{j,k=1}^{D} \partial_j \partial_k f(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_1}}_{=A_{j,k}} (x_j - x_{1,j})(x_k - x_{1,k}) + \ldots, \end{aligned} \tag{219}$$
  i.e. use A from (219) in (218).
  · 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 see Ref. [1], section 10.7 and section 10.8.

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

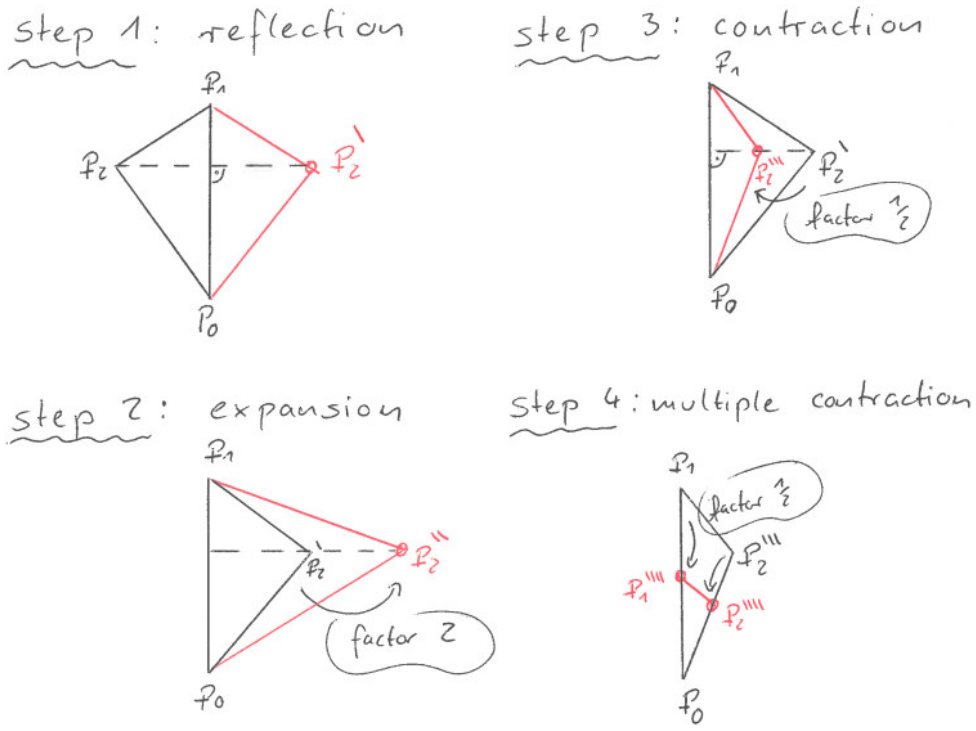
- Downhill simplex method: simple algorithm for function minimization in $D \geq 2$ dimensions, however, not very efficient.
  → Suited for rather simple optimization problems.

- 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 $f(\mathbf{p}_0)\le f(\mathbf{p}_1)\le\ldots\le f(\mathbf{p}_D)$.

- 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.

Figure 11.4



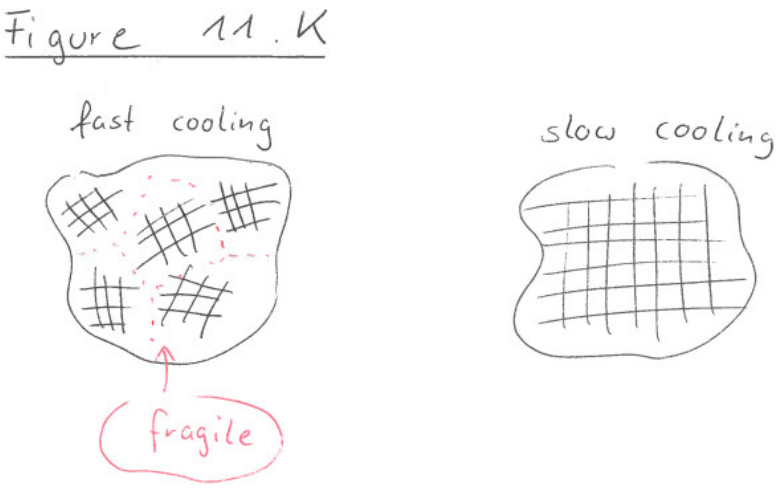
- Stopping criteria:
  - Less obvious than in $D = 1$ dimension.
  - E.g. if $|\mathbf{p}_D - \mathbf{p}_0| < \epsilon$ (where $f(\mathbf{p}_D) > f(\mathbf{p}_j)$, $j = 0, \ldots, D-1$ and $f(\mathbf{p}_0) < f(\mathbf{p}_j)$, $j = 1, \ldots, D$) ...
  - ... or if deformation of simplex is almost negligible.
  - 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 $\mathbf{x}$ 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.



- 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 and 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* $f(s^{(n+1)}) \leq f(s^{(n)})$*:*

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

    Else:

    $\rightarrow$ *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:

$$f(s_j) \quad = \quad \sum_{k=1}^{N} \left(\left(x_{p_j(k)} - x_{p_j(k-1)} \right)^2 + \left(y_{p_j(k)} - y_{p_j(k-1)} \right)^2 \right)^{1/2} \tag{220}$$

    $((x_0, y_0) \equiv (x_N, y_N)$; 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:

$$f(s_j) \quad \rightarrow \quad f(s_j) + \lambda \sum_{k=1}^{N} \Big(\mu_{p(k)} - \mu_{p(k-1)} \Big)^2, \tag{221}$$

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 (he might be a smuggler, etc.).
* 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 $\mathbf{x}^{(n)} \rightarrow \mathbf{x}^{(n+1)}$.
  – See Ref. [1], section 10.12.2.

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--)
  // Fr 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: 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;
}
```

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] H. Grabmüller, "Numerik II (für Ingenieure)," lecture notes, Friedrich-Alexander-Universität Erlangen-Nürnberg (2001).