## Contents

1. **Introduction** ................................. 5

2. **Representation of numbers in computers, roundoff errors** ......................... 7
   - 2.1 Integers .................................. 7
   - 2.2 Real numbers, floating point numbers ........................................... 7
   - 2.3 Roundoff errors ..................................... 8
     - 2.3.1 Simple examples ........................................ 8
     - 2.3.2 Another example: numerical derivative via finite difference ............. 9

3. **Ordinary differential equations (ODEs), initial value problems** ................. 12
   - 3.1 Physics motivation .................................................. 12
   - 3.2 Euler’s method ....................................................... 12
   - 3.3 Runge-Kutta (RK) method ......................................... 13
     - 3.3.1 Estimation of errors ........................................... 15
     - 3.3.2 Adaptive step size ............................................. 17

4. **Dimensionful quantities on a computer** ............................................. 22
   - 4.1 Method 1: define units for your computation ................................... 22
   - 4.2 Method 2: use exclusively dimensionless quantities .......................... 22

5. **Root finding, solving systems of non-linear equations** ......................... 24
   - 5.1 Physics motivation .................................................. 24
   - 5.2 Bisection (only for $N = 1$) ...................................... 24
   - 5.3 Secant method (only for $N = 1$) .................................. 25
   - 5.4 Newton-Raphson method (for $N = 1$) .................................. 26
   - 5.5 Newton-Raphson method (for $N > 1$) .................................. 27

6. **Ordinary differential equations, boundary value problems** ................... 29
   - 6.1 Physics motivation .................................................. 29
   - 6.2 Shooting method ...................................................... 29
     - 6.2.1 Example: QM, 1 dimension, infinite potential well ...................... 31
     - 6.2.2 Example: QM, 1 dimension, harmonic oscillator ......................... 33
     - 6.2.3 Example: QM, 3 dimensions, spherically symmetric potential .......... 37
   - 6.3 Relaxation methods .................................................. 38
7 Solving systems of linear equations

7.1 Problem definition, general remarks ............................................. 40
7.2 Gauss-Jordan elimination (a direct method) ................................. 40
  7.2.1 Pivoting .................................................................................. 42
7.3 Gauss elimination with backward substitution (a direct method) ....... 43
7.4 LU decomposition (a direct method) ............................................. 46
  7.4.1 Crout’s algorithm .................................................................. 46
  7.4.2 Computation of the solution of $Ax = b$ ............................... 47
  7.4.3 Computation of $\text{det}(A)$ ..................................................... 48
7.5 QR decomposition (a direct method) ............................................. 48
7.6 Iterative refinement of the solution of $Ax = b$ (for direct methods) ... 48
7.7 Conjugate gradient method (an iterative method) ......................... 49
  7.7.1 Symmetric positive definite $A$ ................................................ 49
  7.7.2 Example: static electric charge inside a grounded box in 2 dimensions 51
  7.7.3 Generalizations ..................................................................... 54
  7.7.4 Condition number, preconditioning ....................................... 54

8 Numerical integration

8.1 Numerical integration in 1 dimension ........................................... 56
  8.1.1 Newton-Cotes formulas ......................................................... 56
  8.1.2 Gaussian integration .............................................................. 59
8.2 Numerical integration in $D \geq 2$ dimensions ................................. 60
  8.2.1 Nested 1-dimensional integration ........................................... 60
  8.2.2 Monte Carlo integration ........................................................ 61
  8.2.3 When to use which method? ............................................... 62

9 Eigenvalues and eigenvectors

9.1 Problem definition, general remarks .......................................... 64
9.2 Basic principle of numerical methods for eigenvalue problems ......... 65
9.3 Jacobi method ........................................................................... 66
9.4 Example: molecule oscillations inside a crystal ......................... 68

10 Interpolation, extrapolation, approximation

10.1 Polynomial interpolation .......................................................... 73
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: $z = b_{N-1} \ldots b_2 b_1 b_0$ (stored in this way in computer memory, i.e. in the binary numeral system),
  \[ z = \sum_{j=0}^{N-1} b_j 2^j \]  
  (for positive integers). \hspace{1cm} (1)
- Typically $N = 32$ (sometimes also $N = 8, 16, 64, 128$)
  \[ 0 \leq z \leq 2^{32} - 1 = 4294967295. \]
- 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,
  \[ z = S \times M \times 2^{E-e}. \]  
  (2)
- Sign: $S = \pm 1$.
- Mantissa:
  \[ M = \sum_{j=0}^{N_M} m_j \left( \frac{1}{2} \right)^j, \]  
  \hspace{1cm} (3)
  \[ m_0 = 1 \text{ (phantom bit, i.e. } M = 1????, \text{ “normalized”), } m_j \in \{0, 1\} \text{ for } j \geq 1 \text{ (representation analogous to representation of integers).} \]
- 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 $\epsilon = (1/2)^{23} \approx 1.19 \times 10^{-7}$.

$\epsilon$ 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 floating point numbers.

- Equation (2):

  $z = S \times M \times 2^{E-e}$, \hspace{1cm} (4)

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

2.3.1 Simple examples

- $1 + \hat{\epsilon} = 1$, if $|\hat{\epsilon}| < \epsilon$.

- 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):

  $z_1 - z_2 = \underbrace{\alpha_1}_{\approx 1.???} 10^\beta - \underbrace{\alpha_2}_{1.???} 10^\beta = \left(\alpha_1 - \alpha_2\right) 10^\beta$. \hspace{1cm} (5)

  - When $\alpha_1 - \alpha_2$ is computed, the first $n$ digits cancel each other.
    → 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.

\[
\begin{align*}
    f(x + h) &= f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + O(h^3) \\
    \rightarrow f'(x) &= \frac{f(x + h) - f(x)}{h} + O(h) \quad \text{(asymmetric)} \quad \text{(7)} \\
    \rightarrow f'(x) &= \frac{f(x + h) - f(x - h)}{2h} + O(h^2) \quad \text{(symmetric)}. \quad \text{(8)}
\end{align*}
\]

---

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

- Problems:
  - If \( h \) is large
    \( \rightarrow O(h), O(h^2) \) large.
  - If \( h \) is small
    \( \rightarrow f(x + h) - f(x), f(x + h) - f(x - h) \) are differences of similar numbers (see section 2.3.1).

- Optimal choice \( h = h_{\text{opt}} \) for asymmetric finite difference [T]:

  - Relative error due to \( O(h) \):

    \[
    \begin{align*}
      \delta f'(x) &= f'(x) - f'_{\text{finite difference}}(x) = f'(x) - \frac{f(x + h) - f(x)}{h} = \\
      &= \frac{1}{2}f''(x)h + O(h^2) \\
      \rightarrow \delta f'(x) &= \frac{-f''(x)h/2}{f'(x)} \sim \frac{f''(x)h}{f'(x)} \\
      (\sim \text{ indicates “of the same order as”}).
    \end{align*}
    \]

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

    \[
    \begin{align*}
      f(x + h) - f(x) &\approx f'(x)h \\
      f(x + h) &\approx f(x) \\
      \rightarrow \text{relative loss of accuracy} &= \frac{f(x)}{f(x + h) - f(x)} \approx \frac{f(x)}{f'(x)h} \\
      (\text{e.g. } \approx 10^3 \text{ implies that 3 digits are lost}) \\
      \rightarrow \frac{\delta f'(x)}{f'(x)} &= \text{relative precision} \times \text{relative loss of accuracy} \sim \epsilon \frac{f(x)}{f'(x)h}. \quad \text{(10)}
    \end{align*}
    \]

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

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

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

\[ h_{\text{opt}} \sim \epsilon^{1/3}, \quad \frac{\delta f'(x)}{f'(x)}|_{\text{opt}} \sim \epsilon^{2/3}, \] (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

```c
// derivative of sin(x) at x = 1.0 via finite differences
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int j;
    // *****
    printf("h rel_err_asym rel_err_sym\n");
    for(j = 1; j <= 15; j++)
    {
        double h = pow(10.0, -(double)j);
        double df_exact = cos(1.0);
        double df_asym = (sin(1.0+h) - sin(1.0)) / h;
        double df_sym = (sin(1.0+h) - sin(1.0-h)) / (2.0 * h);
        double rel_err_asym = fabs((df_exact - df_asym) / df_exact);
        double rel_err_sym = fabs((df_exact - df_sym) / df_exact);
        printf("%.1e %.3e %.3e\n", h, rel_err_asym, rel_err_sym);
    }
    // *****
}
```

\(^{2}\) Throughout this lecture I use C.
```c
    return EXIT_SUCCESS;
}
```

<table>
<thead>
<tr>
<th>h</th>
<th>rel_err_asym</th>
<th>rel_err_sym</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0e-01</td>
<td>7.947e-02</td>
<td>1.666e-03</td>
</tr>
<tr>
<td>1.0e-02</td>
<td>7.804e-03</td>
<td>1.667e-05</td>
</tr>
<tr>
<td>1.0e-03</td>
<td>7.789e-04</td>
<td>1.667e-07</td>
</tr>
<tr>
<td>1.0e-04</td>
<td>7.787e-05</td>
<td>1.667e-09</td>
</tr>
<tr>
<td>1.0e-05</td>
<td>7.787e-06</td>
<td>2.062e-11</td>
</tr>
<tr>
<td>1.0e-06</td>
<td>7.787e-07</td>
<td>5.130e-11</td>
</tr>
<tr>
<td>1.0e-07</td>
<td>7.742e-08</td>
<td>3.597e-10</td>
</tr>
<tr>
<td>1.0e-08</td>
<td>5.497e-09</td>
<td>4.777e-09</td>
</tr>
<tr>
<td>1.0e-09</td>
<td>9.724e-08</td>
<td>5.497e-09</td>
</tr>
<tr>
<td>1.0e-10</td>
<td>1.082e-07</td>
<td>1.082e-07</td>
</tr>
<tr>
<td>1.0e-11</td>
<td>2.163e-06</td>
<td>2.163e-06</td>
</tr>
<tr>
<td>1.0e-12</td>
<td>8.003e-05</td>
<td>2.271e-05</td>
</tr>
<tr>
<td>1.0e-13</td>
<td>1.358e-03</td>
<td>3.309e-04</td>
</tr>
<tr>
<td>1.0e-14</td>
<td>6.861e-03</td>
<td>6.861e-03</td>
</tr>
<tr>
<td>1.0e-15</td>
<td>2.741e-02</td>
<td>2.741e-02</td>
</tr>
</tbody>
</table>
3 Ordinary differential equations (ODEs), initial value problems

3.1 Physics motivation

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

$$m_j \ddot{r}_j(t) = F_j(r_1(t), \ldots, r_N(t), \dot{r}_1(t), \ldots, \dot{r}_N(t), t), \quad j = 1, \ldots, N, \tag{13}$$

initial conditions

$$r_j(t = 0) = r_{j,0}, \quad \dot{r}_j(t = 0) = v_{j,0}. \tag{14}$$

- Calculate trajectories $r_j(t)$.
- 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

$$\dot{r}_j(t) = v_j(t), \quad \dot{v}_j(t) = \frac{F_j(r_1(t), \ldots, r_N(t), \dot{r}_1(t), \ldots, \dot{r}_N(t), t)}{m_j}. \tag{15}$$

  - Define

$$y(t) = (r_1(t), \ldots, r_N(t), v_1(t), \ldots, v_N(t)) \tag{16}$$

$$f(y(t), t) = \left(\frac{v_1(t), \ldots, v_N(t)}{m_1}, \ldots, \frac{F_N(y(t), t)}{m_N}\right). \tag{17}$$

  - Then

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

(left hand side (lhs) can be evaluated in a straightforward way for given $t$ and $y(t)$).

- 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 $\tau$:

$$y(t + \tau) = y(t) + \dot{y}(t)\tau + O(\tau^2) = y(t) + f(y(t), t)\tau + O(\tau^2). \tag{19}$$
• $\tau$ can be positive ($\rightarrow$ computation of future) or negative ($\rightarrow$ computation of past).

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

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

3.3 Runge-Kutta (RK) method

• Same idea as in section 3.2 but improved discretization (stronger suppression of errors with respect to $\tau$).

• “2nd-order RK”:

$$\begin{align*}
  k_1 &= f(y(t), t)\tau \quad \rightarrow \text{“full Euler step”} \\
  k_2 &= f\left(y(t) + \frac{1}{2}k_1, t + \frac{1}{2}\tau\right)\tau \\
  \rightarrow \text{“half Euler step”} \\
  y(t + \tau) &= y(t) + k_2 + O(\tau^3).
\end{align*}$$

- $f(y(t) + (1/2)k_1, t + (1/2)\tau)$ in (21): estimated derivative $\dot{y}(t + \tau/2)$, i.e. after half step.
- (22): 2nd order RK step (a full “Euler-like” step using the derivative after a half step).
**Proof of (22)**, i.e. that error per step is $O(\tau^3)$:

\[
\begin{align*}
k_2 &= f\left(y + (1/2)f\tau, t + (1/2)\tau\right)\tau = \\
&= f\tau + \left(\frac{\partial f}{\partial y} y + \frac{\partial f}{\partial t} t\right) + O(\tau^3) = f\tau + \frac{1}{2} \left(\frac{\partial f}{\partial y} y + \frac{\partial f}{\partial t} t\right)^2 + O(\tau^3) = \\
&= f\tau + \frac{1}{2} \dot{f}\tau^2 + O(\tau^3) \tag{23}
\end{align*}
\]

\[
\begin{align*}
y(t + \tau) &= y + \dot{y}\tau + \frac{1}{2} \ddot{y}\tau^2 + O(\tau^3) = y + f\tau + \frac{1}{2} \dot{f}\tau^2 + O(\tau^3) = \\
&= y + k_2 + O(\tau^3) \tag{24}
\end{align*}
\]

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

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

- Straightforward to derive discretizations with $O(\tau^4), O(\tau^5), \ldots$ error per step:
  - “3rd-order RK”:
    \[
    \begin{align*}
k_1 &= f(y(t), t)\tau \\
k_2 &= f\left(y(t) + k_1, t + \tau\right)\tau \\
k_3 &= f\left(y(t) + (1/4)(k_1 + k_2), t + (1/2)\tau\right)\tau \\
y(t + \tau) &= y(t) + \frac{1}{6}(k_1 + k_2 + 4k_3) + O(\tau^4). \tag{28}
    \end{align*}
    \]
  - “4th-order RK”:
    \[
    \begin{align*}
k_1 &= f(y(t), t)\tau \\
k_2 &= f\left(y(t) + (1/2)k_1, t + (1/2)\tau\right)\tau \\
k_3 &= f\left(y(t) + (1/2)k_2, t + (1/2)\tau\right)\tau \\
k_4 &= f\left(y(t) + k_3, t + \tau\right)\tau \\
y(t + \tau) &= y(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(\tau^5). \tag{33}
    \end{align*}
    \]
- ...

- Common choice is 4th-order RK.
- Even better: numerical tests with different order RKs (higher orders allow larger step sizes \( \tau \) [which is good], require larger numbers of arithmetic operations per step [which is bad]).
- Example: Compute the trajectory of the 1-dimensional harmonic oscillator (HO).
  - Lagrangian:
    \[
    L = \frac{m}{2} \dot{x}^2 - \frac{m \omega^2}{2} x^2.
    \] (34)
  - EOMs:
    \[
    m \ddot{x}(t) = -m \omega^2 x(t),
    \] (35)
    i.e.
    \[
    \dot{y}(t) = f(y(t), t)
    \] (36)
    with
    \[
    y(t) = (x(t), v(t)) \quad , \quad f(y(t), t) = (v(t), -\omega^2 x(t)).
    \] (37)
  - Initial conditions: \( x(t = 0) = x_0 \), \( \dot{x}(t = 0) = 0 \), i.e. \( y(t = 0) = (x_0, 0) \).
  - \( \omega = 1.0 \), \( x_0 = 1.0 \), step size \( \tau = 0.1 \) \(^4\)
  - 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 \( \tau \)
    \[
    \rightarrow y_{\tau}(t + \tau) \\
    \rightarrow \delta_{\tau} \approx c \tau^{n+1}.
    \]
  - 2 RK steps with step size \( \tau/2 \)
    \[
    \rightarrow y_{2\times\tau/2}(t + \tau) \\
    \rightarrow \delta_{2\times\tau/2} \approx 2c(\tau/2)^{n+1}.
    \]

\(^{4}\)Assigning dimensionless numbers to dimensionful quantities, e.g. \( \omega = 1.0 \) or \( x_0 = 1.0 \), is not always recommended. Usually it is advantageous to define and exclusively use equivalent dimensionless quantities (see section 4).
Figure 1: HO, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK.

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

- Estimated absolute error for $y_{2\times T/2}(t + \tau)$:

$$\delta_{abs} = \frac{|y_{2\times T/2}(t + \tau) - y_{T}(t + \tau)|}{2^n - 1},$$

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

$\text{Figure 3. C}$
Figure 2: HO, errors of the trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK.

- Estimated relative error for $y_{2\times\tau/2}(t+\tau)$ (might be more relevant than estimated absolute error):
  $$\delta_{rel} = \frac{\delta_{abs}}{|y(t)|}. \quad (39)$$
- Estimated error allows local extrapolation:
  - Correct by estimated error:
    $$y_{2\times\tau/2}(t+\tau) \rightarrow y_{2\times\tau/2}(t+\tau) + \frac{y_{2\times\tau/2}(t+\tau) - y_{\tau}(t+\tau)}{2^n - 1}. \quad (40)$$
  - However, no estimation of errors, when using (40).

3.3.2 Adaptive step size

- Small step size $\tau$
  $\rightarrow$ small errors, computation slow.

- Large step size $\tau$
  $\rightarrow$ large errors, computation fast.
Compromise needed: large $\tau$ in regions, where $y(t)$ is smooth, small $\tau$ otherwise.

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

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

Use e.g. the following algorithm to adapt $\tau$ in each RK step:

- **Input:**
  * Initial conditions $y(t = 0)$.
  * Maximum tolerable error $\delta_{\text{abs, max}}$.
  * Initial step size $\tau$ (can be coarse).
- $t = 0$.

1. **RK steps:**
   $$y(t) \rightarrow_{\tau} y_\tau(t + \tau)$$
   $$y(t) \rightarrow_{\tau/2} y_{\tau/2}(t + \tau).$$

2. **Estimated error:**
   $$\delta_{\text{abs}} = \frac{|y_{2\times\tau/2}(t + \tau) - y_{\tau}(t + \tau)|}{2^n - 1}.$$  
   (44)

3. **Change step size:**
   $$\tau_{\text{new}} = 0.9 \times \tau \left(\frac{\delta_{\text{abs, max}}}{\delta_{\text{abs}}}\right)^{1/(n+1)} \quad (45)$$

   ("0.9" reduces number of RK steps, which have to be repeated with smaller step size).

- Clamp $\tau_{\text{new}}$ to $[0.2 \times \tau, 5.0 \times \tau]$ (avoid tiny/huge step size, which might cause breakdown of algorithm).

- If $\delta_{\text{abs}} \leq \delta_{\text{abs, max}}$:
  - Accept $y_{2\times\tau/2}(t + \tau)$ (e.g. output to file).
    - $t = t + \tau$ (i.e. continue at time $t + \tau$).
    - $\tau = \tau_{\text{new}}$ (i.e. continue with estimated optimal step size).
  - Go to (1).
Else:
→ \( \tau = \tau_{\text{new}} \) (i.e. reduce step size).

Go to (1) (i.e. repeat RK steps with smaller step size).

• Modifications possible, e.g. estimate error and \( \tau_{\text{new}} \) by performing RK steps of \( n \)-th and \( n + 1 \)-th order instead of RK steps with step sizes \( \tau \) and \( \tau/2 \).

• Example: 1-dimensional anharmonic oscillator.

  – Lagrangian:
    \[
    L = \frac{m}{2} \dot{x}^2 - m\alpha x^n, \quad n \in \{2, 20\}. \tag{46}
    \]

  – EOMs:
    \[
    m\ddot{x}(t) = -m\alpha n(x(t))^{n-1}, \tag{47}
    \]
    i.e.
    \[
    \dot{y}(t) = f(y(t), t) \tag{48}
    \]
    with
    \[
    y(t) = (x(t), v(t)), \quad f(y(t), t) = (v(t), -\alpha n(x(t))^{n-1}). \tag{49}
    \]

  – Initial conditions: \( x(t = 0) = x_0, \dot{x}(t = 0) = 0 \), i.e. \( y(t = 0) = (x_0, 0) \).

  – \( \alpha = 0.5, 1.0 \) 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, $V(x) = mx^2$, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK using adaptive step size.
Figure 4: Anharmonic oscillator, $V(x) = m x^{20}$, 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 oscillator (same example as in section 3.3).

  - Lagrangian:
    \[ L = \frac{m}{2} \dot{x}^2 - \frac{m\omega^2}{2} x^2. \]  

  - EOMs:
    \[ m\ddot{x}(t) = -m\omega^2 x(t) \quad \rightarrow \quad \ddot{x}(t) = -\omega^2 x(t), \]  
    i.e. \( m \) irrelevant.

  - Measure time in units of \( 1/\omega \):
    \[ \hat{t} = \omega t \quad \rightarrow \quad \frac{d^2}{d\hat{t}^2} \hat{x}(\hat{t}) = -\hat{x}(\hat{t}). \]  

  - Moreover, initial conditions introduce length scale, e.g. \( x(t=0) = x_0, \dot{x}(t=0) = 0 \)  
    \rightarrow \ measure \( x \) in units of \( x_0 \):
    \[ \hat{x} = \frac{x}{x_0} \quad \rightarrow \quad \frac{d^2}{d\hat{t}^2} \hat{x}(\hat{t}) = -\hat{x}(\hat{t}). \]  

  - Now only dimensionless quantities in (53), i.e. straightforward to treat numerically.

  - Figure 5 showing trajectory \( \hat{x}(\hat{t}) \) is analog of Figure 1 (left top).

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

***** October 31, 2023 (5th lecture) *****
Figure 5: HO, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK (same data as in Figure 4 [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,
  \[
  f_j(x_1, \ldots, x_N) = 0 \quad j = 1, \ldots, N, 
  \]  
  or equivalently written in a more compact way
  \[
  f(x) = 0. 
  \]

- Find solutions \( x \) of (55), i.e. find roots of \( f(x) \).

- 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:
  \[
  (1) \quad \bar{x} = (x_1 + x_2)/2. 
  \]
  - If \( f(x_1)f(\bar{x}) < 0 \):
    \[
    \rightarrow x_2 = \bar{x}. 
    \]
    Else:
    \[
    \rightarrow x_1 = \bar{x}. 
    \]
  - If \( |x_1 - x_2| \) sufficiently small:
    \[
    \rightarrow x_1 \approx x_2 \text{ is approximate root.} 
    \]
    End of algorithm.
  Else:
  \[
  \rightarrow Go \ to \ (1). 
  \]

- Convergence:
  \[
  \text{Error of approximate root } \delta \text{ defined via } f(x_1 + \delta) = 0. 
  \]
  \[
  \text{After } n \text{ iterations} 
  \]
  \[
  \delta_n \leq \frac{|x_1 - x_2|}{2^n}, 
  \]
  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 (\( \delta_{n+1} \) linear in \( \delta_n \)).

- Advantages and disadvantages:

(+) Always finds a root.

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

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

- Starting point: \( x_1, x_2 \) fulfilling \( |f(x_2)| < |f(x_1)| \).

- Secant method might find a root of \( f(x) \), not necessarily between \( x_1 \) and \( x_2 \).

- Basic principle:
  - Iteration.
  - Each step as sketched below.

**Algorithm:**

- \( n = 2 \).

\[ (1) \]

\[ \Delta x = -f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, \quad x_{n+1} = x_n + \Delta x. \]  

- If \( |\Delta x| \) sufficiently small:
  → \( x_{n+1} \) is approximate root.
  
  \textit{End of algorithm.}

Else:

→ \( n = n + 1 \).

\textit{Go to (1).}
Convergence: $\delta_{n+1} \approx c(\delta_n)^{1.618...}$ (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
    → $f'$ has to be known analytically/must be cheap to evaluate numerically.
  
  – Each step as sketched below.

Algorithm:
\( n = 1. \) 

\[ (1) \]

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

\( (58) \)

- If \( |\Delta x| \) sufficiently small:
  \( \rightarrow x_{n+1} \) is approximate root.
  End of algorithm.

Else:
  \( \rightarrow n = n + 1. \)
  Go to (1).

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

- Advantages and disadvantages:
  (+) Converges faster than bisection and secant method.
  (−) Does not always find a root.
  (−) \( f' \) has to be known analytically/must be cheap to evaluate numerically.

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

- For \( N > 1 \) root finding is extremely difficult.
  - \( N = 2: \)
    \[ f_1(x_1, x_2) = 0, \quad f_2(x_1, x_2) = 0. \]
    One has to find intersections of isolines \( f_1(x_1, x_2) = 0 \) and \( f_2(x_1, x_2) = 0. \)

\[ \text{Figure 5. D} \]

- \( N > 2: \)
  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: \( x_1 \) (should be close to a root).

• Basic principle:

\[
0 = f_j(x_n + \delta) = f_j(x_n) + \left. \frac{\partial f_j(x)}{\partial x_k} \right|_{x=x_n} \delta_k + \mathcal{O}(\delta^2) , \quad j = 1, \ldots, N \tag{59}
\]

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

\[
0 = f(x_n) + J(x_n)\delta + \mathcal{O}(\delta^2). \tag{60}
\]

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

\[
0 = f(x_n) + J(x_n)\Delta x \tag{61}
\]

or equivalently

\[
\Delta x = -\left( J(x_n) \right)^{-1} f(x_n) \tag{62}
\]

\((\Delta x \approx \delta\), i.e. approximate difference between root and \(x_n\)).

– \((61)\) is system of linear equations (solve analytically for \(N = 2, 3\) or numerically as discussed in section 7).

– \(N = 1\): \(J(x_n) = f'(x_n)\) and \((62)\) becomes

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

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

• Algorithm:

– \(n = 1\).

\((1)\)

\[
\Delta x = -\left( J(x_n) \right)^{-1} f(x_n) \quad , \quad x_{n+1} = x_n + \Delta x. \tag{64}
\]

– If \(|\Delta x| \) sufficiently small:

\(\rightarrow x_{n+1} \) is approximate root.

\(\text{End of algorithm.}\)

Else:

\(\rightarrow n = n + 1.\)

\(\text{Go to (1).}\)
6 Ordinary differential equations, boundary value problems

6.1 Physics motivation

- Newton’s EOMs, $N$ point masses $m_j$,

$$m_j \ddot{r}_j(t) = F_j(r_1(t), \ldots, r_N(t), \dot{r}_1(t), \ldots, \dot{r}_N(t), t), \ j = 1, \ldots, N,$$

(boundary conditions)

$$r_j(t_1) = r_{j,1}, \quad r_j(t_2) = r_{j,2}$$

(“Compute trajectory of a particle, which is at $r_1$ at time $t_1$ and at $r_2$ at time $t_2$.”)

- QM, Schrödinger equation in 1 dimension,

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

(boundary conditions)

$$\psi(x_1) = \psi(x_2) = 0$$

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

- Example appropriate? $E$ is unknown, i.e. (67) and (68) is rather an eigenvalue problem, not an ordinary boundary value problem ...

- Yes, can be reformulated:

  * Consider $E$ as a function of $x$, i.e. $E = E(x)$.
  * Add another ODE: $E'(x) = 0$.
  → System of ODEs,

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

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,

$$y'(x) = f(y(x), x)$$

(both $y$ and $f$ have $N$ components) and boundary conditions

$$g_j(y(x_1)) = 0, \quad j = 1, \ldots, n < N$$

$$h_j(y(x_2)) = 0, \quad j = 1, \ldots, N - n.$$

- Basic principle:
Choose/guess initial conditions $y(x_1)$ such that
- boundary conditions $g_j(y(x_1)) = 0$, $j = 1, \ldots, n < N$ are fulfilled,
- boundary conditions $h_j(y(x_2)) = 0$, $j = 1, \ldots, N - n$ are approximately fulfilled
($y(x_2)$ 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 $y(x_1)$, i.e. such that $h_j(y(x_2)) = 0$.

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

Improve initial conditions, i.e. tune $\lambda$, using the Newton-Raphson method (see section 5.4):
- Interpret $h(y(t_2)) = x(t_2) - b$ as function of $\lambda$ ($x(t_2)$ depends on initial conditions $y(t_1)$, i.e. on $\lambda$).
- Compute derivative $dh(y(t_2))/d\lambda$ (needed by the Newton-Raphson method) numerically (see section 2.3.2).
- Newton-Raphson step to improve $\lambda$:
  $\lambda \rightarrow \lambda - \frac{h(y(t_2))}{dh(y(t_2))/d\lambda}$. \hspace{1cm} (73)
- Repeat RK computation and Newton-Raphson step, until $h(y(t_2)) = 0$ (numerically 0, e.g. up to 6 digits).

**** November 07, 2023 (7th lecture) ****
6.2.1 Example: QM, 1 dimension, infinite potential well

- Infinite potential well:
  \[ V(x) = \begin{cases} 
  0 & \text{if } 0 \leq x \leq L \\
  \infty & \text{otherwise} 
  \end{cases} \quad (74) \]

- Schrödinger equation and boundary conditions:
  \[-\frac{\hbar^2}{2m}\psi''(x) = E\psi(x), \quad \psi(x = 0) = \psi(x = L) = 0. \quad (75)\]

- Reformulate equations using exclusively dimensionless quantities:
  \[
  \hat{x} = \frac{x}{L} \\
  \rightarrow \frac{d}{d\hat{x}} = L \frac{d}{dx} \quad (76)\
  \rightarrow -\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) = \frac{2mEL^2}{\hbar^2} \psi(\hat{x}) \quad (77)\
  \]
  \((\hat{E} \text{ is "dimensionless energy"}), \text{i.e.}\)
  \[-\frac{d^2}{d\hat{x}^2}\psi(\hat{x}) = \hat{E}\psi(\hat{x}), \quad \psi(\hat{x} = 0) = \psi(\hat{x} = 1) = 0. \quad (79)\]

- Analytical solution (to check numerical results):
  \[
  \psi(\hat{x}) = \sqrt{2} \sin(n\pi \hat{x}), \quad \hat{E} = \pi^2 n^2, \quad n = 1, 2, \ldots \quad (80)\]

- Numerical solution:
  - Rewrite Schrödinger equation to system of first order ODEs:
    \[
    \psi'(\hat{x}) = \phi(\hat{x}), \quad \phi'(\hat{x}) = -\hat{E}(\hat{x})\psi(\hat{x}), \quad \hat{E}'(\hat{x}) = 0, \quad (81)\]
    \((\prime \text{ denotes } d/d\hat{x}) \text{i.e.}\)
    \[
    \mathbf{y}(x) = \left(\psi(\hat{x}), \phi(\hat{x}), \hat{E}(\hat{x})\right), \quad \mathbf{f}(\mathbf{y}(x), x) = \left(\phi(\hat{x}), -\hat{E}(\hat{x})\psi(\hat{x}), 0\right). \quad (82)\]
  - Initial conditions for RK/shooting method:
    \[
    * \quad \psi(\hat{x} = 0.0) = 0.0 \quad \text{(boundary condition at } \hat{x} = 0), \quad (83)\]
    \[
    * \quad \phi(\hat{x} = 0.0) = 1.0 \quad \text{(must be } \neq 0, \text{ apart from that arbitrary; different choices result in differently normalized wavefunctions).} \]
\[ \hat{E}(\hat{x} = 0.0) = \mathcal{E} \]

(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.

![Potential well: graphical determination of energy eigenvalues](image)

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

- There are 3 eigenvalues in the range \( 0.0 < \hat{E} < 100.0 \):
  \( \hat{E}_0 \approx 10.0, \hat{E}_1 \approx 40.0, \hat{E}_2 \approx 90.0 \).
- **Shooting method with** \( \mathcal{E} \in \{10.0, 40.0, 90.0\} \).
  
  * Figure 7 (top) illustrates the first Newton-Raphson step for the second excitation (4th order RK).
  
  * Figure 7 (bottom) shows the resulting non-normalized wave functions of the three lowest states (4th order RK).
  
  * Convergence after three Newton-Raphson steps (4th order RK, 7 digits of accuracy); see program output below.

---

**ground state:**

\[ \begin{align*}
\text{E} & \text{_num} = +10.000000 , \\
\text{E} & \text{_num} = +9.868296 , \ E_\text{ana} = +9.869604 , \ \psi(\text{x=1}) = -0.006541 . \\
\text{E} & \text{_num} = +9.869604 , \ E_\text{ana} = +9.869604 , \ \psi(\text{x=1}) = +0.000066 . \\
\text{E} & \text{_num} = +9.869604 , \ E_\text{ana} = +9.869604 , \ \psi(\text{x=1}) = +0.000000 . \\
\end{align*} \]

**1st excitation:**

\[ \begin{align*}
\text{E} & \text{_num} = +40.000000 , \\
\text{E} & \text{_num} = +39.472958 , \ E_\text{ana} = +39.478418 , \ \psi(\text{x=1}) = +0.006539 . \\
\text{E} & \text{_num} = +39.478417 , \ E_\text{ana} = +39.478418 , \ \psi(\text{x=1}) = -0.000069 . \\
\text{E} & \text{_num} = +39.478418 , \ E_\text{ana} = +39.478418 , \ \psi(\text{x=1}) = -0.000000 . \\
\end{align*} \]
2nd excitation:
E_{num} = +90.000000
E_{num} = +88.813303 , E_{ana} = +88.826440 , \psi(x=1) = -0.006537
E_{num} = +88.826438 , E_{ana} = +88.826440 , \psi(x=1) = +0.000074
E_{num} = +88.826440 , E_{ana} = +88.826440 , \psi(x=1) = +0.000000

6.2.2 Example: QM, 1 dimension, harmonic oscillator

- Schrödinger equation and boundary conditions:
  \[-\frac{\hbar^2}{2m} \psi''(x) + \frac{m\omega^2}{2} x^2 \psi(x) = E \psi(x) , \quad \psi(x = -\infty) = \psi(x = +\infty) = 0 \]  \hspace{1cm} (83)
  (numerical challenge are boundary conditions at \( x = \pm \infty \)).
- Reformulate equations using exclusively dimensionless quantities:
  - Length scale from \( \hbar, m, \omega \):
    \([\hbar] = \text{kg m}^2/\text{s}\)
$[m] = \text{kg}$
$[\omega] = 1/\text{s}$
→ length scale $a = (\hbar/m\omega)^{1/2}$.

\[\hat{x} = \frac{x}{a}\quad \text{(84)}\]
\[\frac{d}{d\hat{x}} = a \frac{d}{dx}\quad \text{(85)}\]
\[\frac{d^2}{d\hat{x}^2} \psi(\hat{x}) + \hat{x}^2 \psi(\hat{x}) = \frac{2E}{\hbar \omega} \psi(\hat{x})\quad \text{(86)}\]

($\hat{E}$ is “dimensionless energy”), i.e.
\[\frac{d^2}{d\hat{x}^2} \psi(\hat{x}) + \hat{x}^2 \psi(\hat{x}) = \hat{E}\psi(\hat{x})\quad ,\quad \psi(\hat{x} = -\infty) = \psi(\hat{x} = +\infty) = 0.\quad \text{(87)}\]

- **Parity:**
  - Parity $P$: spatial reflection, i.e. $PxP = -x$, $P\psi(+x) = \psi(-x)$.
  - Eigenvalues and eigenfunctions of $P$:
    \[P\psi(x) = \lambda \psi(x) \rightarrow \left\{ \begin{array}{l} P\psi(x) = \lambda^2 \psi(x) \\ \lambda = \pm 1. \end{array} \right.\quad \text{(88)}\]
  * Common notation: $P = \pm$ (instead of $\lambda = \pm$).
  * $P = +$: $P\psi(x) = \psi(-x)$ and $P\psi(x) = +\psi(x) \rightarrow \psi(x) = +\psi(-x)$, i.e. even eigenfunction.
  * $P = -$: $P\psi(x) = \psi(-x)$ and $P\psi(x) = -\psi(x) \rightarrow \psi(x) = -\psi(-x)$, i.e. odd eigenfunction.
  - $[H, P] = 0$, if $V(+x) = V(-x)$, i.e. for symmetric potentials.
    → Eigenfunctions $\psi(x)$ of $H$ can be chosen such that they are also eigenfunctions of $P$.
    → $P = +$
    \[\psi(x) = +\psi(-x) \rightarrow \psi'(x = 0) = 0.\quad \text{(89)}\]
    → $P = -$
    \[\psi(x) = -\psi(-x) \rightarrow \psi(x = 0) = 0.\quad \text{(90)}\]

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

- Numerical problems with boundary conditions $\psi(\hat{x} = -\infty) = \psi(\hat{x} = +\infty) = 0$ (eq. (87)).
- Numerical solution, first attempt:
  - Use either $\psi'(\hat{x} = 0) = 0$ or $\psi(\hat{x} = 0) = 0$ (89) or (90) instead of $\psi(\hat{x} = -\infty) = 0$.
  - Use $\psi(\hat{x} = L/a) = 0$, where $x = L$ is far in the classically forbidden region ($E \ll V(L)$), i.e. where $\psi$ is exponentially suppressed.
– Rewrite Schrödinger equation to system of first order ODEs:
\[ \psi'(\hat{x}) = \phi(\hat{x}), \quad \phi'(\hat{x}) = (\hat{x}^2 - \hat{E}(\hat{x}))\psi(\hat{x}), \quad \hat{E}'(\hat{x}) = 0, \]
(91)
i.e.
\[ \mathbf{y}(x) = \left(\psi(\hat{x}), \phi(\hat{x}), \hat{E}(\hat{x})\right), \quad \mathbf{f}(\mathbf{y}(x), x) = \left(\phi(\hat{x}), (\hat{x}^2 - \hat{E}(\hat{x}))\psi(\hat{x}), 0\right). \]
(92)
– Initial conditions for \( P = + \) for RK/shooting method:
  * \( \psi(\hat{x} = 0.0) = 1.0 \)
    (must be \( \neq 0 \), apart from that arbitrary; different choices result in differently normalized wavefunctions),
  * \( \phi(\hat{x} = 0.0) = 0.0 \)
    (boundary condition at \( \hat{x} = 0 \)),
  * \( \hat{E}(\hat{x} = 0.0) = \mathcal{E} \)
    (will be tuned by Newton-Raphson method such that boundary condition \( \psi(\hat{x} = L/a) = 0 \) is fulfilled; has to be close to the energy eigenvalue one is interested in [e.g. ground state: \( E = \hbar \omega / 2 \), i.e. \( \hat{E} = 1 \), i.e. choose \( \mathcal{E} \approx 1 \)]; typically \( \mathcal{E} \) is the result of a crude graphical determination of energy eigenvalues [see section 6.2.1]).

(For \( P = - \) use \( \psi(\hat{x} = 0.0) = 0.0, \phi(\hat{x} = 0.0) = 1.0. \))
– Boundary condition \( \psi(\hat{x} = L/a) = 0 \) numerically hard to implement; a tiny admixture of the exponentially increasing solution will dominate for large \( \hat{x} \), as shown in Figure 8 (4th order RK).

• Numerical solution, more practical approach:
  – Use “... a tiny admixture of the exponentially increasing solution will dominate for large \( \hat{x} \) ...” to your advantage:
Figure 8: HO, numerical problems with boundary condition $\psi(\hat{x} = L/a) = 0$.

* Start far in the classically forbidden region using arbitrary initial conditions, e.g. $\psi(\hat{x} = L/a) = 1.0$, $\phi(\hat{x} = L/a) = 0.0$, $\hat{E}(\hat{x} = L/a) = \mathcal{E}$
  or $\psi(\hat{x} = L/a) = 0.0$, $\phi(\hat{x} = L/a) = 1.0$, $\hat{E}(\hat{x} = L/a) = \mathcal{E}$
  or ...
* Tune $\mathcal{E}$ via the RK/shooting method such that $\psi'(\hat{x} = 0) = 0$ for $P = +$ (or $\psi(\hat{x} = 0) = 0$ for $P = -$).
  
  From Figure 9 (top) one can read off rough estimates for the energy eigenvalues, which can be used to intialize $\mathcal{E}$ (4th order RK).
  
  Figure 9 (bottom) shows the resulting wave functions of the four lowest states (4th order RK).
  
  For initial values $\mathcal{E} \in \{0.9, 2.9, 4.9, 6.9\}$ convergence after three Newton-Raphson steps (7 digits of accuracy); see program output below.

---

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

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

2nd excitation:
E_num = +4.900000 .
E_num = +4.990699.
E_num = +4.999911.
Figure 9: HO. **(top)** Crude graphical determination of energy eigenvalues. **(bottom)** Wave functions of the four lowest states.

\[ E_{\text{num}} = +5.00000. \]

3rd excitation:
\[ E_{\text{num}} = +6.90000. \]
\[ E_{\text{num}} = +6.990720. \]
\[ E_{\text{num}} = +6.999911. \]
\[ E_{\text{num}} = +7.000000. \]

### 6.2.3 Example: QM, 3 dimensions, spherically symmetric potential

- Spherically symmetric potential: \( V(r) = V(r) \), where \( r = |r| \).
- Rewrite Schrödinger equation in spherical coordinates ...
- ... angular dependence of wavefunctions proportional to spherical harmonics, i.e. \( \psi(r, \theta, \varphi) \propto Y_{lm}(\theta, \varphi) \) ...
- ... remaining radial equation is second order ODE in \( r \), which can be solved using RK/shooting.
- For details see e.g. Ref. [2].

- Solving such radial equations numerically is common in up-to-date research.
  - For example [3]: The potential of two heavy $\bar{b}$ quarks in the presence of two light $u$ and/or $d$ quarks can be computed with lattice QCD (a numerical method to study QCD). This potential can be used in a standard non-relativistic Schrödinger equation to check, whether the quarks may form a stable $\bar{b}b$ $u/d$ tetraquark.

  "C. Numerical solution of Schrödinger’s equation

  To investigate the existence of a bound state rigorously, we numerically solve the Schrödinger equation with the Hamiltonian (6). The strongest binding is expected in an $s$-wave, for which the radial equation is

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

  (93)

  with the wave function $\psi(r) = R(r)/r$. We impose Dirichlet boundary conditions $R(r_{\text{max}}) = 0$ at sufficiently large $r_{\text{max}}$ (we checked that results are stable for $r_{\text{max}} \gtrsim 10\, \text{fm}$). The radial equation (9) can be solved by standard methods (e.g. 4th order Runge-Kutta shooting) up to arbitrary numerical precision."

  - For example [4]: The potential of a $\bar{b}b$ (quark-antiquark) pair and of a $\bar{B}B$ (meson-antimeson) pair (a $B$ meson is composed of a $\bar{b}$ quark and a light $u$ or $d$ quark) can be computed with lattice QCD. These potentials can be used in a coupled-channel non-relativistic Schrödinger equation to predict and explore the spectrum and properties of bottomonium (both stable states and resonances; bottomonium = $\bar{b}b$).

  "A. Numerical methods to solve the coupled channel Schrödinger equation and to determine the poles of the $T$ matrix

  In Section II C we defined the entries of the $T$ matrix (19) as the a priori unknown coefficients $t_\ldots$ appearing in the $r \to \infty$ boundary conditions (17) and (18). To determine these coefficients, one has to solve the coupled channel Schrödinger equation (11). To cross check our results, we used two rather different numerical methods. The first method corresponds to discretizing the radial coordinate by a uniform grid and solving the resulting system of linear equations using methods from standard textbooks (for details see Ref. [28]). The second method corresponds to using an ordinary 4th order Runge-Kutta algorithm.

  To find the poles of $T_J$ in the complex energy plane, characterized by at least one of its eigenvalues approaching infinity, we applied the Newton-Raphson method to find the roots of $1/\det(T_J)$.”

6.3 Relaxation methods

- See e.g. Ref. [1], section 18.0.

- Discretize time, guess solution ...

- ... then iteratively improve the solution, until the ODE is fulfilled.
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).
- $b_j, j = 0, \ldots, M - 1$: vectors with $N$ components.

Typical problems:
- Solve $Ax_j = b_j$ (possibly for several vectors $b_j$).
- Compute $A^{-1}$.
  Do not compute $A^{-1}$ and solve $Ax_j = b_j$ via $x_j = A^{-1}b_j$ ... 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"): $N \gtrsim O(1000)$.
- Sparse matrices: $N \gtrsim O(10^6)$.

It might be a good idea to check your result, e.g. by computing $Ax_j$ and comparing to $b_j$, e.g. to exclude large roundoff errors.

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

7.2 Gauss-Jordan elimination (a direct method)

- Goal: solve $Ax_j = b_j$.
- Problem "compute $A^{-1}$" included: choose $b_j = e_j$, then $A^{-1} = (x_1 \ x_2 \ \ldots \ x_N)$.
- Basic idea: add/subtract multiples of the linear equations, until solution $x_j$ is obvious.
Notation

\[ Ax_j = b_j \rightarrow \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & b_{0,0} & \cdots & b_{0,M-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & b_{1,0} & \cdots & b_{1,M-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & b_{2,0} & \cdots & b_{2,M-1} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \end{bmatrix}. \] (94)

Step 1: elimination of column 0,

\[ a_{0,k}^{(1)} = \frac{a_{0,k}}{a_{0,0}}, \quad k = 0, \ldots, N - 1 \] (95)
\[ b_{0,k}^{(1)} = \frac{b_{0,k}}{a_{0,0}}, \quad k = 0, \ldots, M - 1 \] (96)
\[ a_{j,k}^{(1)} = a_{j,k} - a_{j,0}a_{0,k}^{(1)}, \quad j = 1, \ldots, N - 1, \quad k = 0, \ldots, N - 1 \] (97)
\[ b_{j,k}^{(1)} = b_{j,k} - a_{j,0}b_{0,k}^{(1)}, \quad j = 1, \ldots, N - 1, \quad k = 0, \ldots, M - 1 \] (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.

\[
\begin{bmatrix}
1 & a_{0,1}^{(1)} & a_{0,2}^{(1)} & \cdots & b_{0,1}^{(1)} & \cdots & b_{0,M-1}^{(1)} \\
0 & a_{1,1}^{(1)} & a_{1,2}^{(1)} & \cdots & b_{1,1}^{(1)} & \cdots & b_{1,M-1}^{(1)} \\
0 & a_{2,1}^{(1)} & a_{2,2}^{(1)} & \cdots & b_{2,1}^{(1)} & \cdots & b_{2,M-1}^{(1)} \\
\vdots & \vdots & \vdots & & \vdots & & \vdots
\end{bmatrix}.
\] (99)

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

\[ a_{n-1,k}^{(n)} = \frac{a_{n-1,k}^{(n-1)}}{a_{n-1,n-1}^{(n-1)}}, \quad k = n - 1, \ldots, N - 1 \] (100)
\[ b_{n-1,k}^{(n)} = \frac{b_{n-1,k}^{(n-1)}}{a_{n-1,n-1}^{(n-1)}}, \quad k = 0, \ldots, M - 1 \] (101)
\[ a_{j,k}^{(n)} = a_{j,k}^{(n-1)} - a_{j,n-1}^{(n-1)}a_{n-1,k}^{(n)}, \quad j \neq n - 1, \quad k = n - 1, \ldots, N - 1 \] (102)
\[ b_{j,k}^{(n)} = b_{j,k}^{(n-1)} - a_{j,n-1}^{(n-1)}b_{n-1,k}^{(n)}, \quad j \neq n - 1, \quad k = 0, \ldots, M - 1 \] (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 \( 0 \ldots 0 \ldots 0 \).

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 \( N \):

\[ \begin{bmatrix}
1 & 0 & 0 & \cdots & b_{0,0}^{(N)} & \cdots & b_{0,M-1}^{(N)} \\
0 & 1 & 0 & \cdots & b_{1,0}^{(N)} & \cdots & b_{1,M-1}^{(N)} \\
0 & 0 & 1 & \cdots & b_{2,0}^{(N)} & \cdots & b_{2,M-1}^{(N)} \\
\vdots & \vdots & \vdots & & \vdots & & \vdots
\end{bmatrix} \rightarrow \mathbb{I}x_j = b_j^{(N)} \] (104)
i.e. “b columns” are solutions $x_j$.

- Advantages and disadvantages:
  
  
  - Rather slow to solve $Ax_j = b_j$.
  - All vectors $b_j$ 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 $b_j$, in a numerically advantageous way.

- **Partial pivoting:**
  
  - Before step $n$ swap row $n - 1$ and row $j$, where $n - 1 \leq j \leq N - 1$ and
  
  $|a^{(n-1)}_{j,n-1}| = \max_k |a^{(n-1)}_{k,n-1}|$. 
  
  \[ (105) \]

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

  - Significantly smaller roundoff errors.

- **Scaled partial pivoting:**
  
  - Problem with partial pivoting:
    * E.g., if $a_{0,0} = |a_{0,0}^{(0)}|$ is small, then partial pivoting will swap line 0 with another line before step 1.
    * However, if you multiply line 0 with a huge number, before using the Gauss-Jordan elimination method, you solve an equivalent system of linear equations, which has the same solution; $a_{0,0} = |a_{0,0}^{(0)}|$ is now large and pivoting will not swap line 0 with another line before step 1; roundoff errors might then be rather large.

  - Before step $n$ swap row $n - 1$ and row $j$, where $n - 1 \leq j \leq N - 1$ and

  $\frac{|a^{(n-1)}_{j,n-1}|}{\max_l |a^{(0)}_{j,l}|} = \max_k \frac{|a^{(n-1)}_{k,n-1}|}{\max_l |a^{(0)}_{k,l}|}$

  \[ (106) \]

  (“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 \( a_{n-1,n-1} \), but not above.

\[
\begin{pmatrix}
  a_{0,0}^{(0)} & a_{0,1}^{(0)} & a_{0,2}^{(0)} & \cdots & b_{0,0}^{(0)} & \cdots & b_{0,M-1}^{(0)} \\
  0 & a_{1,1}^{(1)} & a_{1,2}^{(1)} & \cdots & b_{1,0}^{(1)} & \cdots & b_{1,M-1}^{(1)} \\
  0 & 0 & a_{2,2}^{(2)} & \cdots & b_{2,0}^{(2)} & \cdots & b_{2,M-1}^{(2)} \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\
\end{pmatrix}
\]  

(107)

- Then backward substitution, i.e. computation of \( \mathbf{x}_n \):

  * Start with

  \[
  x_{N-1,n} = \frac{b_{N-1,n}^{(N-1)}}{a_{N-1,N-1}^{(N-1)}}. 
  \]  

(108)

  * Then

  \[
  x_{N-2,n} = \frac{b_{N-2,n}^{(N-2)} - a_{N-2,N-1}^{(N-2)}x_{N-1,n}}{a_{N-2,N-2}^{(N-2)}}. 
  \]  

(109)

  * I.e. perform \( N \) steps \( j = N - 1, N - 2, \ldots, 0 \),

  \[
  x_{j,n} = \frac{b_{j,n}^{(j)} - \sum_{k=j+1}^{N-1} a_{j,k}^{(j)}x_{k,n}}{a_{j,j}^{(j)}}. 
  \]  

(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.

\[
\begin{array}{cccc|c}
N = 4 \\
\hline
\hline
\text{no pivoting:} & +0.68 & -0.21 & +0.57 & +0.60 \mid +0.82 \\
\hline
\end{array}
\]
\[-0.60 \quad -0.33 \quad +0.54 \quad -0.44 \quad | \quad +0.11\]
\[-0.05 \quad +0.26 \quad -0.27 \quad +0.03 \quad | \quad +0.90\]
\[+0.83 \quad +0.27 \quad +0.43 \quad -0.72 \quad | \quad +0.21\]
\[+0.68 \quad -0.21 \quad +0.57 \quad +0.60 \quad | \quad +0.82\]
\[+0.00 \quad -0.52 \quad +1.04 \quad +0.09 \quad | \quad +0.84\]
\[+0.00 \quad +0.24 \quad -0.23 \quad +0.07 \quad | \quad +0.96\]
\[+0.00 \quad +0.53 \quad -0.26 \quad -1.45 \quad | \quad -0.79\]
\[+0.68 \quad -0.21 \quad +0.57 \quad +0.60 \quad | \quad +0.82\]
\[+0.00 \quad -0.52 \quad +1.04 \quad +0.09 \quad | \quad +0.84\]
\[+0.00 \quad +0.00 \quad +0.26 \quad +0.11 \quad | \quad +1.35\]
\[+0.00 \quad +0.00 \quad +0.81 \quad -1.36 \quad | \quad +0.07\]
\[+0.68 \quad -0.21 \quad +0.57 \quad +0.60 \quad | \quad +0.82\]
\[+0.00 \quad -0.52 \quad +1.04 \quad +0.09 \quad | \quad +0.84\]
\[+0.00 \quad +0.00 \quad +0.26 \quad +0.11 \quad | \quad +1.35\]
\[+0.00 \quad +0.00 \quad +0.81 \quad -1.36 \quad | \quad +0.07\]
\[x = ( \begin{array}{c}
-2.22 \\
+7.31 \\
+4.24 \\
+2.47 
\end{array} ).
\]
\[b_{\text{check}} = ( \begin{array}{c}
+0.82 \\
+0.11 \\
+0.90 \\
+0.21 
\end{array} ).
\]
\[b_{\text{check}} - b = ( \begin{array}{c}
-2.2e-16 \\
+1.5e-16 \\
-1.1e-16 \\
-1.7e-15 
\end{array} ).
\]
\[|b_{\text{check}} - b| = +1.74535e-15.
\]
\[------------------------
\]
partial pivoting:
\[+0.68 \quad -0.21 \quad +0.57 \quad +0.60 \quad | \quad +0.82\]
\[-0.60 \quad -0.33 \quad +0.54 \quad -0.44 \quad | \quad +0.11\]
\[-0.05 \quad +0.26 \quad -0.27 \quad +0.03 \quad | \quad +0.90\]
\[+0.83 \quad +0.27 \quad +0.43 \quad -0.72 \quad | \quad +0.21\]
\[+0.83 \quad +0.27 \quad +0.43 \quad -0.72 \quad | \quad +0.21\]
\[+0.00 \quad -0.13 \quad +0.85 \quad -0.97 \quad | \quad +0.26\]
\[+0.00 \quad +0.27 \quad -0.25 \quad -0.01 \quad | \quad +0.92\]
\[+0.00 \quad -0.43 \quad +0.21 \quad +1.18 \quad | \quad +0.65\]
\[+0.83 \quad +0.27 \quad +0.43 \quad -0.72 \quad | \quad +0.21\]
\[+0.00 \quad -0.43 \quad +0.21 \quad +1.18 \quad | \quad +0.65\]
\[+0.00 \quad +0.00 \quad -0.11 \quad +0.73 \quad | \quad +1.32\]
\[+0.00 \quad +0.00 \quad +0.79 \quad -1.33 \quad | \quad +0.67\]
\[+0.83 \quad +0.27 \quad +0.43 \quad -0.72 \quad | \quad +0.21\]
\[+0.00 \quad -0.43 \quad +0.21 \quad +1.18 \quad | \quad +0.65\]
\[+0.00 \quad +0.00 \quad +0.79 \quad -1.33 \quad | \quad +0.67\]
\[+0.00 \quad +0.00 \quad +0.00 \quad +0.54 \quad | \quad +1.33\]
\[x = ( \begin{array}{c}
-2.22 \\
+7.31 \\
+4.24 \\
+2.47 
\end{array} ).
\]
\[b_{\text{check}} = ( \begin{array}{c}
+0.82 \\
+0.11 \\
+0.90 \\
+0.21 
\end{array} ).
\]
\[b_{\text{check}} - b = ( \begin{array}{c}
-1.1e-16 \\
-3.6e-16 \\
-3.3e-16 \\
+1.1e-16 
\end{array} ).
\]
\[ |b_{\text{check}} - b| = +5.15537e-16. \]

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

scaled partial pivoting:

\[
\begin{array}{cccc|c}
+0.68 & -0.21 & +0.57 & +0.60 & +0.82 \\
-0.60 & -0.33 & +0.54 & -0.44 & +0.11 \\
-0.05 & +0.26 & -0.27 & +0.03 & +0.90 \\
+0.83 & +0.27 & +0.43 & -0.72 & +0.21 \\
\end{array}
\]

\[
\begin{array}{cccc|c}
+0.68 & -0.21 & +0.57 & +0.60 & +0.82 \\
+0.00 & -0.52 & +1.04 & +0.09 & +0.84 \\
+0.00 & +0.24 & -0.23 & +0.07 & +0.96 \\
+0.00 & +0.53 & -0.26 & -1.45 & -0.79 \\
\end{array}
\]

\[
\begin{array}{cccc|c}
+0.68 & -0.21 & +0.57 & +0.60 & +0.82 \\
+0.00 & +0.24 & -0.23 & +0.07 & +0.96 \\
+0.00 & +0.00 & +0.55 & +0.23 & +2.88 \\
+0.00 & +0.00 & +0.25 & -1.59 & -2.88 \\
\end{array}
\]

\[
\begin{array}{cccc|c}
+0.68 & -0.21 & +0.57 & +0.60 & +0.82 \\
+0.00 & +0.24 & -0.23 & +0.07 & +0.96 \\
+0.00 & +0.00 & +0.55 & +0.23 & +2.88 \\
+0.00 & +0.00 & +0.00 & -1.69 & -4.19 \\
\end{array}
\]

\[ x = ( -2.22 +7.31 +4.24 +2.47 ). \]

\[ b_{\text{check}} = ( +0.82 +0.11 +0.90 +0.21 ). \]

\[ b_{\text{check}} - b = ( -2.2e-16 -6.9e-17 +1.1e-16 -1.5e-15 ). \]

\[ |b_{\text{check}} - b| = +1.52081e-15. \]

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

\[ N = 100 \]

--------

no pivoting:

\[ |b_{\text{check}} - b| = +2.25693e-11. \]

partial pivoting:

\[ |b_{\text{check}} - b| = +1.46047e-12. \]

scaled partial pivoting:

\[ |b_{\text{check}} - b| = +3.28886e-13. \]

***** November 16, 2023 (10th lecture) *****
7.4  *LU* decomposition (a direct method)

- *LU* decomposition of *A*:

\[
A = LU
\]

\[
L = \begin{pmatrix}
1 & 0 & 0 & 0 & \ldots \\
\alpha_{1,0} & 1 & 0 & 0 & \ldots \\
\alpha_{2,0} & \alpha_{2,1} & 1 & 0 & \ldots \\
\alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & 1 & \ldots \\
\vdots & \vdots & \vdots & \vdots & \ddots \\
\end{pmatrix}
\]  \hspace{1cm} (111)

\[
U = \begin{pmatrix}
\beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} & \ldots \\
0 & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \ldots \\
0 & 0 & \beta_{2,2} & \beta_{2,3} & \ldots \\
0 & 0 & 0 & \beta_{3,3} & \ldots \\
\vdots & \vdots & \vdots & \vdots & \ddots \\
\end{pmatrix}
\]  \hspace{1cm} (112)

- *L*: lower triangular matrix.
- *U*: upper triangular matrix.
- Allows efficient computation of the solution of *Ax = 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,

\[
a_{j,k} = \sum_{l=0}^{N-1} \alpha_{j,l} \beta_{l,k}, \hspace{1cm} (114)
\]

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

- Solving these equations is trivial, when considering them in a particular order:

  - For \( k = 0, 1, \ldots, N - 1 \), i.e. for all columns:

    * Step 1:

    \[
    \beta_{j,k} = a_{j,k} - \sum_{l=0}^{j-1} \alpha_{j,l} \beta_{l,k}, \hspace{1cm} j = 0, 1, \ldots, k. \hspace{1cm} (115)
    \]

    * Step 2:

    \[
    \alpha_{j,k} = \frac{1}{\beta_{k,k}} \left( a_{j,k} - \sum_{l=0}^{k-1} \alpha_{j,l} \beta_{l,k} \right), \hspace{1cm} j = k + 1, k + 2, \ldots, N - 1. \hspace{1cm} (116)
    \]
Pivoting as important as for Gauss-Jordan elimination and for Gauss elimination with backward substitution.

- Proceed as discussed 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 \( Ax = b \)

- Proceed in two steps:

  (1) Compute \( y \), defined by
  \[
  Ax = L \underbrace{Ux}_{y} = b, \tag{117}
  \]
  via forward substitution,
  \[
  y_j = b_j - \sum_{k=0}^{j-1} \alpha_{j,k} y_k , \quad j = 0, 1, \ldots, N - 1, \tag{118}
  \]
i.e. solve $Ly = b$ (note that, when pivoting has been used in the computation of the $LU$ decomposition, the components of $b$ 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 $x$ via backward substitution (as in section 7.3),

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

i.e. solve $Ux = y$.

- Advantages and disadvantages:
  
  (+) $LU$ decomposition independent of vectors $b_j$, i.e. corresponding solutions $x_j$ do not have to be computed at the same time.
  
  (+) Not slower than Gauss-Jordan elimination and Gauss elimination with backward substitution for $Ax_j = b_j$ and for $A^{-1}$.
  
  (+) Allows computation of $\det(A)$ (see section 7.4.3)

### 7.4.3 Computation of $\det(A)$

- 

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

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

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

### 7.5 $QR$ decomposition (a direct method)

- Due to limited time not discussed.

### 7.6 Iterative refinement of the solution of $Ax = b$ (for direct methods)

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

- Refine $x$ as follows:

$$A(x + \delta x) = b \rightarrow A\delta x = b - \underbrace{Ax}_{=b} = \delta b, \quad (122)$$
i.e. solve
\[ A\delta x = \delta b; \]  \hspace{1cm} (123)
refined solution is \( x + \delta x \).

- 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 \( 10000 \times 10000 \) 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 \( 10000 \times 10000 \) 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 \( Ax = b \), when \( A \) is a large sparse matrix.

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

#### 7.7.1 Symmetric positive definite \( A \)

- Goal: solve \( Ax = b \) (a single vector \( b \), no computation of \( A^{-1} \) or \( \det(A) \)).
- Basic idea:
  - Minimize
    \[ f(x) = \frac{1}{2}xAx - bx, \]  \hspace{1cm} (124)
    which describes an \( N \)-dimensional paraboloid, with respect to \( x \).
  - The minimum is characterized by
    \[ \nabla f(x) = Ax - b = 0, \]  \hspace{1cm} (125)
    i.e. it is the solution of \( Ax = b \).
- Algorithm:
  - Guess solution \( x_0 \), e.g. \( x_0 = 0 \) (can be far away from the correct solution).
- $n = 0$.

(1) Select direction $p_n$ (details below).
- Minimize $f(x_n + \alpha_n p_n)$ with respect to $\alpha_n$.
- $x_{n+1} = x_n + \alpha_n p_n$.
- If $|b - Ax_{n+1}|$ sufficiently small:
  - $x_{n+1}$ is approximate solution.
  End of algorithm.

Else:
  - $n = n + 1$.
  Go to (1).

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

- Detailed equations:
  - $r_0 = b - Ax_0$ ("residual", gradient of the paraboloid at $x_0$), $p_0 = r_0$.
  - During each iteration:
    \begin{align*}
    \alpha_n &= \frac{r_n r_n}{p_n A p_n} \quad (126) \\
    r_{n+1} &= r_n - \alpha_n A p_n \quad (127) \\
    \beta_n &= \frac{r_{n+1} r_{n+1}}{r_n r_n} \quad (128) \\
    p_{n+1} &= r_{n+1} + \beta_n p_n \quad (129)
    \end{align*}
  (see Ref. [1], section 2.7.6).

- One can show: after $n$ steps $x_n$ is not just minimum with respect to direction $p_{n-1}$, but also minimum with respect to all previous directions $p_0, p_1, \ldots, p_{n-2}$.
  $\rightarrow$ Solution of $Ax = b$ after $N$ steps or less.
Typically solution of $A\mathbf{x} = \mathbf{b}$ obtained after significantly less than $N$ steps.

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

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

\[ \Delta \phi(x, y) = q \delta(x, y) \]  \hspace{1cm} (130)

with boundary conditions

\[ \phi(x, y) = 0 \quad \text{if} \quad |x| \geq R \quad \text{or} \quad |y| \geq R. \]  \hspace{1cm} (131)

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

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

\[ \phi(x, y) \rightarrow \phi_{j,k} \equiv \phi(x_j, y_k) \]  \hspace{1cm} (133)

\[ \Delta \phi(x, y) \rightarrow \frac{\phi_{j+1,k} + \phi_{j-1,k} + \phi_{j,k+1} + \phi_{j,k-1} - 4\phi_{j,k}}{a^2} \equiv \Delta \phi(x, y) \bigg|_{(x,y)=(x_j,y_k)} \]  \hspace{1cm} (134)

\[ \delta(x, y) \rightarrow \frac{1}{a^2} \delta_{j,0} \delta_{k,0} \equiv \delta(x_j, y_k) \]  \hspace{1cm} (135)

\footnote{There might be better ways to treat this partial differential equation numerically (see section ??). The main intention of this example is to demonstrate the conjugate gradient method in the context of a simple physics example.}
\( \delta_{j,k} \) denotes the Kronecker delta.

- **Dimensionless discretized Poisson equation:**

\[
\Delta \phi(x, y) = q \delta(x, y) \quad (136)
\]

\[
\Rightarrow \Delta \phi(x, y) = \phi_{j+1,k} + \phi_{j-1,k} + \phi_{j,k+1} + \phi_{j,k-1} - 4\phi_{j,k} = \delta_{j,0} \delta_{k,0} \quad (138)
\]

with \( \hat{\phi} = \phi/q \).

- **Discretized boundary conditions:**

\[
\hat{\phi}_{-n,k} = \hat{\phi}_{n,k} = \hat{\phi}_{j,-n} = \hat{\phi}_{j,n} = 0, \quad (139)
\]

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

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

- These linear equations can be written in the standard form

\[
A_{(j,k),(j',k')} x_{(j',k')} = b_{(j,k)}, \quad j, k, j', k' = -(n-1), -(n-2), \ldots, (n-1) \quad (140)
\]

by defining

\[
A_{(j,k),(j',k')} = \delta_{j+1,j'} \delta_{k,k'} + \delta_{j-1,j'} \delta_{k,k'} + \delta_{j,j'} \delta_{k+1,k'} + \delta_{j,j'} \delta_{k-1,k'} - 4 \delta_{j,j'} \delta_{k,k'} \quad (141)
\]

\[
x_{(j,k)} = \hat{\phi}_{j,k} \quad (142)
\]

\[
b_{(j,k)} = \delta_{j,0} \delta_{k,0}. \quad (143)
\]

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

\[
r = (2n-1) \left( k + (n-1) \right) + \left( j + (n-1) \right), \quad r = 0, 1, \ldots, N-1, \quad (144)
\]

which is equivalent to

\[
j = (r \mod (2n-1)) - (n-1), \quad k = \left\lfloor r/(2n-1) \right\rfloor - (n-1) \quad (145)
\]

(see corresponding C code in appendix F).

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

***** November 23, 2023 (12th lecture) *****
In the program code the matrix $A$ should not be stored as an array (this would unnecessarily occupy a huge amount of memory and prevent computations with a large number of lattice sites). $A$ can conveniently be implemented as a function (see corresponding C code in appendix E).

Comparison of several lattice sizes:
- $n = 100 \rightarrow 420$ iterations, 0.7 sec,
- $n = 400 \rightarrow 1598$ iterations, 40.5 sec,
- $n = 1000 \rightarrow 3958$ iterations, 10 min 14 sec

(CPU time on a standard laptop, stopping criterion $|Ax - b| < 10^{-10}$).

Note that $n = 1000$ corresponds to a matrix $A$ with $N \approx 4 \times 10^6$ rows and columns. Using a direct method would require around $(4 \times 10^6)^2 \times 8$ byte $= 128$ 000 GB.

Figure 10
- Shows $\phi/q$ for $n = 100$ plotted along one of the axes (red dots) and along one of the diagonals (green dots).

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

\[ \Delta \phi_{\text{no box}}(x, y) = q \delta(x, y) \]

\[ \int d^2r \Delta \phi_{\text{no box}}(x, y) = \oint d\mathbf{n} \nabla \phi_{\text{no box}}(x, y) = 2\pi r E_r(r) = q \]

\[ E_r(r) = \frac{q}{2\pi r} \]

\[ \phi_{\text{no box}}(r) = \int dr E_r(r) = \frac{q}{2\pi} \ln(r/R). \tag{146} \]

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

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

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

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

### 7.7.3 Generalizations

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

### 7.7.4 Condition number, preconditioning

- Any \( N \times N \) matrix can be decomposed according to

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

where \( U \) and \( V \) are orthogonal matrices and \( \omega_j \geq 0 \) are the singular values of \( A \)

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

\[
\text{cond}(A) = \frac{\max_j(\omega_j)}{\min_j(\omega_j)} \tag{148}
\]

- If \(\text{cond}(A)\) is large, the conjugate gradient method is inefficient:
  - Many iterations necessary.
  - Numerical accuracy limited.

- Illustration for symmetric positive definite \(A\):
  - \(\omega_j\) are eigenvalues of \(A\).
  - Semi-axes of the ellipsoids \(f(x) = (1/2)xx - bx = \text{const}\) are proportional to \((\omega_j)^{-1/2}\).
  - Numerically problematic, if ellipsoids have significantly different semi-axes.
  - Numerically ideal, if ellipsoids are spheres (solution obtained after one step; see “Figure 7.B”).

- Caution:
  - If \(A\) is not symmetric and positive definite, one could solve \(A^TAx = A^Tb\) instead of \(Ax = b\); then one could use the conjugate gradient method, since \(A^TA\) is symmetric and positive definite.
  - Do not do that, because \(\text{cond}(A^TA) = (\text{cond}(A))^2\), i.e. the conjugate gradient method applied to \(A^TA\) is significantly more inefficient than other generalized methods (see section 7.7.3) applied to \(A\).

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

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

\[ I = \int_{a}^{b} dx \, f(x). \quad (149) \]

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

8.1.1 Newton-Cotes formulas

- Basic principle: Approximate \( f(x) \) using a polynomial, integrate the polynomial analytically.
- In detail:
  - Approximate \( f(x) \) using Lagrange polynomials:
    * Select \( n + 1 \) points \( x_j, j = 0, 1, \ldots, n \).
    * Compute/evaluate samples \( f_j = f(x_j) \).
    * Langrange polynomials:
      \[
      l_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k},
      \quad (150)
      \]
      i.e. \( l_j(x_k) = \delta_{jk} \).
    * Approximation of \( f(x) \):
      \[
      f(x) \approx g(x) = \sum_{j=0}^{n} f_j l_j(x).
      \quad (151)
      \]
– Integrate \( g(x) \) instead of \( f(x) \) to obtain an approximation of the integral,

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

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

* Even \( n \):

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

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

* Odd \( n \):

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

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

* Error estimates \( \Delta I \) are only useful, if derivatives \( f^{(n+2)} \) and \( f^{(n+1)} \) are bounded (not the case, if \( f \) has singularities).

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

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

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

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

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

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

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

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

(coefficients 1/3, 4/3 and 1/3 can be obtained in a straightforward way from [152]).
There are further common “integration rules”, e.g. Simpson’s 3/8 rule or Boole’s rule.

Examples (see Ref. [5]):

\[
I_1 = \int_0^1 \frac{1}{1+x^2} \, dx = \frac{\pi}{4}.
\]

* \( I_1 = +0.7853 \ldots \) (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).

\[
I_2 = \int_0^1 e^x \, dx = e - 1.
\]

* \( I_2 = +1.7182 \ldots \) (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:

\[
I = \int_a^b f(x) \, dx \approx \left( \frac{1}{2} f_0 + f_1 + f_2 + \ldots + f_{N-1} + \frac{1}{2} f_N \right) h = T_N
\]

\[
\Delta I = N \times O(h^3) = O(1/N^2).
\]

(\(h = (b-a)/N\)).

Iteratively increase the number of sub-intervals by a factor of 2 in each step, i.e. \(N \to 2N\), until the approximation of \(I\) is sufficiently accurate (error is reduced by a factor of around 4 in each step).
• Iterated Simpson’s rule:
  – Approximate \( I \) according to
    \[
    I = \int_a^b dx f(x) \approx \frac{4}{3} T_{2N} - \frac{1}{3} T_N. \tag{163}
    \]
  – Naive expectation: \( \Delta I = \mathcal{O}(1/N^2) \).
  – Closer inspection shows that \( \Delta \) is much smaller:
    \[
    \frac{4}{3} T_{2N} - \frac{1}{3} T_N = \left(\frac{1}{3} f_0 + \frac{4}{3} f_1 + \frac{2}{3} f_2 + \frac{4}{3} f_3 + \ldots + \frac{2}{3} f_{2N-2} + \frac{4}{3} f_{2N-1} + \frac{1}{3} f_{2N}\right) h \tag{164}
    \]
    \((h = (b - a)/2N)\), which is the iterated Simpson’s rule, i.e. \( \Delta I = \mathcal{O}(1/N^4) \).

\[\text{Figure 8.1} \]

```
\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure8.1}
\caption{Diagram of Iterated Simpson’s rule}
\end{figure}
```

• Algorithm:
  1. Compute \( T_N \) (eq. 161).
  2. Compute \( T_{2N} \) (eq. 161), reuse even samples \( f_0, f_1, f_2, \ldots, f_N \) from step (1) as “even samples” \( f_0, f_2, f_4, \ldots, f_{2N} \) (evaluating \( f(x) \) might be expensive).
    – Approximate \( I \) according to \((4/3)T_{2N} - (1/3)T_N\), to reduce the error from \( \mathcal{O}(1/N^2) \) to \( \mathcal{O}(1/N^4) \).
    – If the approximation of \( I \) is sufficiently accurate:
      → End of algorithm.
    Else:
      \( N \to 2N. \)
      Go to (1).

8.1.2 Gaussian integration

• Due to limited time not discussed.
8.2 Numerical integration in $D \geq 2$ dimensions

- Goal: Compute the definite integral

$$I = \int_R d^D x \, f(x),$$

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

- More difficult than in 1 dimension, because:
  - Number of samples $f_j = f(x_j)$ can be very large ($N$ samples in 1 dimension $\rightarrow N^D$ samples in $D$ dimensions).
  - $R$ might be “complicated”.

\[ \text{Figure 8.6} \]

\[ \text{D = 1} \]

\[ \text{D = 2} \]

```
"simple"
```

```
"complicated"
```

8.2.1 Nested 1-dimensional integration

- $D = 3$ in the following (generalization to arbitrary $D$ obvious).
- Notation: $x = (x, y, z)$.
- 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,

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

\[ = I(x, y) \]

\[ = I(x) \]
Nested integrals might be more complicated, if $R$ is not convex. e.g.

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

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

Step 2:
Compute samples of $I(x)$ (typically $N$ samples) using e.g. techniques from section 8.1.

Step 3:
Compute $I$ using e.g. techniques from section 8.1.

8.2.2 Monte Carlo integration

- Statistical approximation of $I$ using random numbers; similar to an experimental measurement the result has an error bar.

- Select $N$ points $x_1, x_2, \ldots, x_N \in R$ randomly and uniformly.

$$I = \int_{R} d^D x f(x) = \underbrace{\frac{V(R)\langle f \rangle}{N-1}}_{=\tilde{T}} \pm \underbrace{\frac{V(R)\left(\langle (f - \langle f \rangle)^2 \right)}{N-1}}_{=\Delta I}^{1/2},$$  \hfill (168)

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

$$\langle f \rangle = \frac{1}{N} \sum_{j=1}^{N} f(x_j)$$  \hfill (169)

$$\langle f^2 \rangle = \frac{1}{N} \sum_{j=1}^{N} \left( f(x_j) \right)^2.$$

\hfill (170)
• Error $\Delta I$: probability for $I \in [I - \Delta I, I + \Delta I]$ 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 $f(x)$ is “smooth”; very inefficient, if $f(x)$ has strongly localized peaks).

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

– “Complicated domains” $R$ possible, if $R$ can be defined by a function

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

* Define “simple domain” $\hat{R} \supseteq R$, e.g. a $D$-dimensional box.

* Figure 8.6

\[\hat{R} \text{ (a simple box)}\]

*\[I = \int_{\hat{R}} d^D x f(x)g(x). \hspace{1cm} (172)\]

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

8.2.3 When to use which method?

• For very precise computations (many digits of accuracy)
  → nested 1-dimensional integration
  (convergence of Monte Carlo integration too slow, $\Delta I \propto 1/\sqrt{N}$).

• Complicated domain easy for Monte Carlo integration, more difficult for nested 1-dimensional integration.

• Nested 1-dimensional integration requires smooth integrands, otherwise error estimates useless.

• Monte Carlo integration requires integrands, which are not strongly peaked, otherwise huge statistical errors.
• Complicated domain, integrand, which is not strongly peaked, limited accuracy o.k.  
  → Monte Carlo integration.

• Simple domain, integrand smooth  
  → nested 1-dimensional integration.

• Strongly oscillating or discontinuous integrand  
  → Monte Carlo integration.
9 Eigenvalues and eigenvectors

9.1 Problem definition, general remarks

- **Eigenvalue problem**: Find eigenvalues \( \lambda_j \) and eigenvectors \( v_j \neq 0 \), \( j = 1, \ldots, N \) fulfilling
  \[
  Av_j = \lambda_j v_j \tag{173}
  \]
  \((A: N \times N \text{ matrix})\).

- Eigenvalues are roots of the characteristic polynomial \( \det(A - \lambda_j) \), i.e. solutions of
  \[
  \det(A - \lambda_j) = 0. \tag{174}
  \]

- The characteristic polynomial is a degree-\( N \) polynomial, i.e.
  \begin{itemize}
  \item \( N \) roots (= eigenvalues) \( \lambda_j \) (might be complex, not necessarily different),
  \item \( N \) eigenvectors \( v_j \) (might be complex, not necessarily linearly independent).
  \end{itemize}

- Properties of eigenvalues and eigenvectors for specific classes of matrices:
  \begin{itemize}
  \item \( A \) real, symmetric \( (A^T = A) \):
    \begin{itemize}
    \item \( \lambda_j \) real, \( v_j \) can be chosen real.
    \end{itemize}
  \item \( A \) complex, hermitian \( (A^\dagger = A) \):
    \begin{itemize}
    \item \( \lambda_j \) real.
    \end{itemize}
  \item \( A \) not symmetric/not hermitian:
    \begin{itemize}
    \item \( \lambda_j \) and \( v_j \) typically complex.
    \end{itemize}
  \item \( A \) normal \( (AA^\dagger = A^\dagger A) \):
    \begin{itemize}
    \item \( \lambda_j \) pairwise distinct:
      \begin{itemize}
      \item \( v_j \) orthogonal.
      \end{itemize}
    \item \( \lambda_j \) degenerate:
      \begin{itemize}
      \item can be chosen orthogonal (e.g. using Gram-Schmidt orthogonalization).
      \end{itemize}
    \end{itemize}
  \end{itemize}

- **Generalized eigenvalue problem**: find eigenvalues \( \lambda_j \) and eigenvectors \( v_j \neq 0 \), \( j = 1, \ldots, N \) fulfilling
  \[
  Av_j = \lambda_j B v_j \tag{175}
  \]
  \((A, B: N \times N \text{ matrices})\).

  - Can be rewritten as a standard eigenvalue problem:
    \[
    B^{-1} Av_j = \lambda_j v_j. \tag{176}
    \]

  - If \( A \) symmetric and \( B \) symmetric and positive definite, use the following method:
    \begin{itemize}
    \item Cholesky decomposition (“square root of a matrix”; see e.g. Ref. \[II\]): \( B = LL^T \),
      where \( L \) is a lower triangular matrix.
    \end{itemize}
Then
\[ Av_j = \lambda_j LL^T v_j \] (177)
\[ L^{-1}A(L^T)^{-1}L^Tv_j = \lambda_j L^Tv_j, \] (178)
which is a standard eigenvalue problem with a symmetric matrix \( A' \)
\((L^{-1})^T = (L^T)^{-1} \) can be shown in a straightforward way). Note that \( B^{-1}A \) in
\[ 176 \] is typically not symmetric.

* Computing \( L^{-1} \) and solving \( L^Tv_j = v'_j \), simple, because \( L \) is a lower triangular

\[ \text{9.2 Basic principle of numerical methods for eigenvalue problems} \]

- Iterative procedure: apply similarity transformations
  \[ A \rightarrow \( (P_1)^{-1}AP_1 \rightarrow \) \]
  \[ (P_2)^{-1}(P_1)^{-1}AP_1P_2 \rightarrow \]
  \[ \ldots \]
  \[ (P_n)^{-1} \ldots (P_1)^{-1}A P_1P_2 \ldots P_n = Q^{-1}AQ \] (179)
  such that \( Q^{-1}AQ \) is diagonal.

- In practice: stop iteration, as soon as \( Q^{-1}AQ \) is “almost a diagonal matrix” (e.g. absolute
  values of off-diagonal elements \(< \epsilon = 10^{-6}\)).

- Matrix \( Q^{-1}AQ = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_N) \):
  - Eigenvalues \( \lambda_j \).
  - Eigenvectors \( e_j \).

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

- Matrix \( A \):
  - Eigenvalues \( \lambda_j \), because
  \[ \det(A - \lambda_j) = \det(Q^{-1}) \det \left( A - \lambda_j \right) \det(Q) = \det \left( Q^{-1} \left( A - \lambda_j \right) Q \right) = \]
  \[ = \det \left( Q^{-1}AQ - \lambda_j \right), \] (180)
  i.e. \( A \) and \( Q^{-1}AQ \) have the same characteristic polynomial and, consequently, the
  same eigenvalues \( \lambda_j \).
  - Eigenvectors \( Qe_j \), i.e. the columns of \( Q \) are the eigenvectors, because
  \[ Q^{-1}AQe_j = \lambda_j e_j \] (181)
  \[ \rightarrow A(Qe_j) = \lambda_j(Qe_j). \] (182)
• Summary:

(1) Iteratively apply similarity transformations, until $Q^{-1}AQ$ is diagonal.
(2) Eigenvalues are the diagonal elements of $Q^{-1}AQ$.
(3) If eigenvectors are needed, $Q = P_1P_2 \ldots P_n$ has to be computed; eigenvectors are columns of $Q$.

9.3 Jacobi method

• $A$ must be real and symmetric:
  → $A$ is normal.
  → $\lambda_j$ real, $v_j$ can be chosen real and orthogonal.
  → Eigenvectors form an orthogonal matrix,

$$Q = \begin{pmatrix} v_1 & v_2 & \ldots & v_n \end{pmatrix} \rightarrow Q^{-1} = Q^T = \begin{pmatrix} (v_1)^T \\ (v_2)^T \\ \ldots \\ (v_n)^T \end{pmatrix},$$

which diagonalizes $A$, i.e.

$$Q^T AQ = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_N).$$

• 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,

$$A \rightarrow A' = (P_j)^T AP_j,$$

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

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

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

$$A'_{kl} = \frac{((P_j)^T)_{k,m} A_{m,n}(P_j)_{n,l}}{= (P_j)_{m,k}}.$$
\[ A'_{p,p} = (P_j)_{m,p}A_{m,n}(P_j)_{n,p} = c^2A_{p,p} + s^2A_{q,q} - 2csA_{p,q}, \]
\[ A'_{q,q} = c^2A_{q,q} + s^2A_{p,p} + 2csA_{p,q}, \]
\[ A'_{p,q} = A'_{q,p} = (c^2 - s^2)A_{p,q} + sc(A_{p,p} - A_{q,q}), \]
\[ k \neq p, q: A'_{k,p} = A'_{p,k} = cA_{k,p} - sA_{k,q}, \]
\[ k \neq p, q: A'_{k,q} = A'_{q,k} = cA_{k,q} + sA_{k,p}, \]
\[ k, l \neq p, q: A'_{k,l} = A_{k,l}. \]

Choose \( \varphi \) such that \( A'_{p,q} = A'_{q,p} = 0 \), i.e.
\[
\frac{c^2 - s^2}{2sc} = \frac{A_{q,q} - A_{p,p}}{2A_{p,q}} \tag{188}
\]
and after defining \( \theta = (A_{q,q} - A_{p,p})/2A_{p,q} \) and using \( (c^2 - s^2)/2sc = (1/t - t)/2 \)
\[ t^2 + 2\theta t - 1 = 0 \]
\[ \to t = -\theta \pm \left(\theta^2 + 1\right)^{1/2} \tag{189}, \]
where \( t = \tan(\varphi) \); numerical tests have shown that it is advantageous to choose the smaller \(|t|\), i.e.
\[
t = \frac{\text{sign}(\theta)}{\mid\theta\mid + (\theta^2 + 1)^{1/2}}, \tag{190}
\]
implying \(|\varphi| \leq \pi/4.

- Using (188) the above equations can be implemented in the following equivalent, more convenient form:
  - \( A'_{p,p} = A_{p,p} - tA_{p,q} \).
  - \( A'_{q,q} = A_{q,q} + tA_{p,q} \).
  - \( A'_{p,q} = A'_{q,p} = 0. \)
  - \( k \neq p, q: A'_{k,p} = A'_{p,k} = cA_{k,p} - s(1 + c)A_{k,q} \).
  - \( k \neq p, q: A'_{k,q} = A'_{q,k} = cA_{k,q} + s(1 + c)A_{k,p} \).
  - \( k, l \neq p, q: A'_{k,l} = A_{k,l}. \)

- Convergence of the Jacobi method:
  - Applying the Jacobi rotation (185) results in \( A'_{p,q} = A'_{q,p} = 0 \), but other off-diagonal elements might become larger.
  - Is convergence guaranteed?
  - Define “deviation from diagonal matrix”: \( S = \sum_{k \neq l} (A_{k,l})^2. \)
  - One can show: \( S' = S - 2(A_{p,q})^2 \), i.e. \( S \) will approach 0, if “large off-diagonal elements \( A_{p,q} \) are rotated to 0”.
- How to choose \( p \) and \( q \)?
  - Jacobi 1846: “rotate the largest off-diagonal elements \( |A_{p,q}| = |A_{q,p}| \) to 0”.
  - Jacobi’s strategy is numerically too expensive (finding the largest off-diagonal elements is \( \mathcal{O}(N^2) \), while a Jacobi rotation is only \( \mathcal{O}(N) \)).
Nowadays: cyclic Jacobi method, pick off-diagonal elements in a fixed order, e.g. $A_{0,1}$, $A_{0,2}$, ..., $A_{0,N-1}$, $A_{1,2}$, $A_{1,3}$, ..., $A_{1,N-1}$, $A_{2,3}$, ...

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

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):

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

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

  \[
  m \ddot{x}_1 = +k(x_2 - x_1) \tag{192}
  \]

  \[
  m \ddot{x}_2 = +k(x_3 - x_2) + k(x_1 - x_2) \tag{193}
  \]

  \[
  \ldots \tag{194}
  \]

  i.e. can be written in matrix form,

  \[
  M\ddot{x} = -Kx \tag{195}
  \]

  \[
  M = \begin{pmatrix} m & & & \\ & m & & \\ & & \ddots & \\ & & & m \end{pmatrix} \text{ (mass matrix)} \tag{196}
  \]

  \[
  K = \begin{pmatrix} +k & -k & & & \\ -k & +2k & -k & & \\ & -k & +2k & \ddots & \\ & & \ddots & \ddots & \ddots \end{pmatrix} \text{ (stiffness matrix)} \tag{197}
  \]

  where $x = (x_1, x_2, \ldots, x_N)$. 
EOMs reformulated using dimensionless quantities:

\[
\frac{d^2}{dt^2} \ddot{x} = -\hat{K} \dot{x}
\]

(198)

\[
\hat{K} = \begin{pmatrix}
+1 & -1 & & \\
-1 & +2 & -1 & \\
& -1 & +2 & \\
& & & \ddots
\end{pmatrix},
\]

(199)

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

The ansatz

\[
\dot{x} = v_j e^{i\hat{\omega}_j \hat{t}}
\]

(200)

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

\[
-\hat{\omega}_j^2 v_j = -\hat{K} v_j.
\]

(201)

Since \( \hat{K} \) is real and symmetric, the Jacobi method can be used to solve the eigenvalue problem.

Computation for \( N = 10 \):

– C code to compute eigenvalues and eigenvectors of \( \hat{K} \): see appendix F.

\[
N = 10
\]

initial matrix

\[
\begin{array}{cccccccccc}
+1.00 & -1.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 \\
-1.00 & +2.00 & -1.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 \\
+0.00 & -1.00 & +2.00 & -1.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 \\
+0.00 & +0.00 & -1.00 & +2.00 & -1.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 \\
+0.00 & +0.00 & +0.00 & -1.00 & +2.00 & -1.00 & +0.00 & +0.00 & +0.00 & +0.00 \\
+0.00 & +0.00 & +0.00 & +0.00 & -1.00 & +2.00 & -1.00 & +0.00 & +0.00 & +0.00 \\
+0.00 & +0.00 & +0.00 & +0.00 & +0.00 & -1.00 & +2.00 & -1.00 & +0.00 & +0.00 \\
+0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & -1.00 & +2.00 & -1.00 & +0.00 \\
+0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & -1.00 & +2.00 & -1.00 \\
+0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & +0.00 & -1.00 & +2.00 \\
\end{array}
\]

\( S = 1.80000e+01 \).

**********

Sweep 1 ...
\[
\begin{array}{cccccccccccc}
+0.16 & -0.11 & -0.22 & +0.09 & +0.11 & -0.05 & -0.04 & +0.03 & +0.01 & +0.02 \\
-0.11 & +3.63 & -0.11 & +0.33 & +0.09 & -0.13 & -0.06 & +0.04 & +0.04 & +0.02 \\
-0.22 & -0.11 & +0.48 & -0.08 & -0.36 & +0.14 & +0.15 & +0.02 & -0.12 & +0.01 \\
+0.09 & +0.33 & -0.08 & +3.40 & -0.08 & +0.40 & +0.07 & -0.14 & -0.03 & -0.07 \\
+0.11 & +0.09 & -0.36 & -0.08 & +0.63 & +0.01 & -0.38 & +0.31 & -0.14 & +0.23 \\
-0.05 & -0.13 & +0.14 & +0.40 & +0.01 & +3.23 & -0.05 & +0.34 & +0.20 & +0.03 \\
-0.04 & -0.06 & +0.15 & +0.07 & -0.38 & -0.05 & +0.72 & +0.17 & -0.50 & -0.27 \\
+0.03 & +0.04 & -0.14 & -0.14 & +0.31 & +0.34 & +0.17 & +2.86 & -0.09 & -0.04 \\
+0.01 & +0.04 & -0.12 & -0.03 & -0.14 & +0.20 & -0.50 & -0.09 & +1.89 & +0.00 \\
+0.02 & +0.02 & +0.01 & -0.07 & +0.23 & +0.03 & -0.27 & -0.04 & +0.00 & +0.99 \\
\end{array}
\]

\[ S = 2.91374e+00. \]

**********

sweep 2 ...

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

\[ S = 2.53839e+01. \]

**********

sweep 3 ...

...

\[ S = 2.12206e-02. \]

**********

sweep 4 ...

...

\[ S = 7.26279e-06. \]

**********

sweep 5 ...

...

\[ S = 2.26242e-10. \]

**********
\[ S = 1.12777e-32. \]

**********

\[
\lambda_{00} = +0.000000.
\]
\[
v_{00} = (+0.32, +0.32, +0.32, +0.32, +0.32, +0.32, +0.32, +0.32, +0.32, +0.32). 
\]

\[
\lambda_{01} = +3.902113.
\]
\[
v_{01} = (-0.07, +0.20, -0.32, +0.40, -0.44, +0.44, -0.40, +0.32, -0.20, +0.07). 
\]

\[
\lambda_{02} = +0.097887.
\]
\[
v_{02} = (-0.44, -0.40, -0.32, -0.20, -0.07, +0.07, +0.20, +0.32, +0.40, +0.44). 
\]

\[
\lambda_{03} = +2.618034.
\]
\[
v_{03} = (+0.26, -0.43, +0.43, -0.26, +0.26, +0.43, +0.00, +0.43, -0.43, +0.26). 
\]

\[
\lambda_{04} = +1.381966.
\]
\[
v_{04} = (+0.36, -0.14, -0.45, +0.14, +0.36, -0.14, +0.45, -0.14, +0.36). 
\]

\[
\lambda_{05} = +3.618034.
\]
\[
v_{05} = (+0.14, -0.36, +0.45, -0.36, +0.14, +0.14, -0.36, +0.45, -0.36, +0.14). 
\]

\[
\lambda_{06} = +0.381966.
\]
\[
v_{06} = (+0.43, +0.26, +0.00, -0.26, -0.43, -0.00, -0.26, +0.43, -0.43, +0.26). 
\]

\[
\lambda_{07} = +3.175571.
\]
\[
v_{07} = (-0.20, +0.44, -0.32, -0.07, +0.40, -0.40, +0.07, +0.32, -0.44, +0.20). 
\]

\[
\lambda_{08} = +2.000000.
\]
\[
v_{08} = (+0.32, -0.32, -0.32, +0.32, +0.32, -0.32, -0.32, +0.32, +0.32, -0.32). 
\]

\[
\lambda_{09} = +0.824429.
\]
\[
v_{09} = (-0.40, -0.07, +0.32, +0.44, +0.20, -0.20, -0.44, -0.32, +0.07, +0.40). 
\]

---

- General solution:

\[
\hat{x} = \sum_{j=1}^{N} v_{j-1} \left( A_j \cos(\hat{\omega}_j \hat{t}) + B_j \sin(\hat{\omega}_j \hat{t}) \right), \tag{202}
\]

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

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

* \[
\hat{x}(\hat{t} = 0) = \sum_{j=1}^{N} v_{j-1} B_j \hat{\omega}_j = 0 \rightarrow B_j = 0, \tag{203}
\]

because eigenvectors \( \mathbf{v}_j \) are orthogonal and, thus, linearly independent.
\[
\hat{x}(\hat{t} = 0) = \sum_{j=1}^{N} v_{j-1} A_j = (1, 0, \ldots, 0) \rightarrow A_j = v_{j-1,1}
\]  
(204)
(first index of \(v_{j-1,1}\) is eigenvector index, second index is component index), where \(v_j v_k = \delta_{j,k}\) has been used.

* Solution:
\[
\hat{x} = \sum_{j=1}^{N} v_{j-1} v_{j-1,1} \cos(\hat{\omega}_j \hat{t})
\]
(205)
(see Figure 11 from which e.g. the speed of sound can be read off).

Figure 11: “Molecule oscillations in a 1-dimensional crystal” \((N = 10\) molecules).

- Can be generalized in a straightforward way to study small oscillations of any system of \(N\) point masses (after first order Taylor expansion, EOMs are of the form \(M\ddot{x} = -Kx\)).
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 $x_0 < y < x_N$ → interpolation, otherwise → extrapolation.
  - $g(x_j) = f_j = f(x_j)$, $j = 0, \ldots, N$ → interpolation
    - otherwise (i.e. $g(x_j) \approx f_j = f(x_j)$) → 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.
  - Approximation $g(x) \approx f(x)$ often needed, e.g. for a subsequent analytical calculation.
  - For example [3]:
    The potential of two heavy $\bar{b}$ quarks in the presence of two light $u$ and/or $d$ quarks can be computed with lattice QCD (a numerical method to solve QCD) for discrete $\bar{b}b$ separations $r = na$ ($n = 1, 2, \ldots$; $a$: lattice spacing). To use this potential in a standard non-relativistic Schrödinger equation (see also section 6.2.3), the lattice data points need to be parameterized by a continuous function. A physically motivated ansatz is
    \[
    V(r) = -\frac{\alpha}{r} \exp \left( -\left( \frac{r}{d} \right)^2 \right) \tag{206}
    \]
    (1/r, because of 1-gluon-exchange at small $r$ [leading order perturbation theory; see lectures on quantum field theory]; $\exp(-r^2/d^2)$, because of color screening at large $r$). One has to fit this ansatz to the lattice data points, i.e. one has to determine the optimal values for $\alpha$ and $d$.

10.1 Polynomial interpolation

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

73
• Unique solution (easy to show).

• \( g(x) \) can be obtained e.g. using Lagrange polynomials (see section 8.1.1):

\[
l_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k}
\]

\[
g(x) = \sum_{j=0}^{n} f_j l_j(x).
\]

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

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

10.2 Cubic spline interpolation

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

\[
g(x) = y_j(x) \quad \text{for} \quad x_j \leq x \leq x_{j+1},
\]

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 \( (x_{j+1}, f_{j+1}) \)).

**** December 14, 2023 (18th lecture) ****
- \( y'_j(x_{j+1}) = y'_{j+1}(x_{j+1}) \) and \( y''_j(x_{j+1}) = y''_{j+1}(x_{j+1}) \) (the piecewise defined function \( g(x) \) is \( C^2 \) continuous).

Such a piecewise defined polynomial is called “cubic spline”.

- Advantage (compared to polynomial interpolation discussed in section 10.1): only degree-3 polynomials, i.e. polynomial degree is small, even though the number of data points \( (N+1) \) might be large; thus, no unnecessary oscillations.

- Construction of a cubic spline:
  - To interpolate data points \( f_j, j = 0, \ldots, N \), degree-1 polynomials are sufficient:
    \[
    y_j(x) = f_j A(x) + f_{j+1} B(x),
    \]
    where
    \[
    A(x) = \frac{x_{j+1} - x}{x_{j+1} - x_j}, \quad B(x) = \frac{x - x_j}{x_{j+1} - x_j}
    \]
    are the Lagrange polynomials (207) for \( N = 1 \).
  - If the second derivatives \( f''_j = f''(x_j), j = 0, \ldots, N \) are given (in addition to the data points \( f_j \)),
    \[
    y_j(x) = f_j A(x) + f_{j+1} B(x) + f''_j C(x) + f''_{j+1} D(x),
    \]
    where
    \[
    C(x) = \frac{1}{6} \left( A(x)^3 - A(x) \right) (x_{j+1} - x_j)^2 \quad \text{(214)}
    \]
    \[
    D(x) = \frac{1}{6} \left( B(x)^3 - B(x) \right) (x_{j+1} - x_j)^2. \quad \text{(215)}
    \]
  - Determine \( f''_j, j = 0, \ldots, N \) such that the resulting spline \( g(x) \) is \( C^2 \) continuous:
    * Impose \( y'_{j-1}(x_j) = y'_j(x_j), j = 1, \ldots, N - 1. \)
* Insert (213):
\[
\frac{x_j - x_{j-1}}{6} f''_{j-1} + \frac{x_{j+1} - x_j}{3} f''_j + \frac{x_{j+1} - x_j}{6} f''_{j+1} = \frac{f_{j+1} - f_j}{x_{j+1} - x_j} - \frac{f_j - f_{j-1}}{x_j - x_{j-1}}.
\]
(216)

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

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

![Figure 13: Cubic spline interpolation of \(N + 1 = 10\) data points \(f_j\).](image)

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

10.3 Method of least squares

- Data points \(f_j\) often exhibit statistical fluctuations (e.g. \(f_j\) can be experimental measurements, results of Monte Carlo integrations or simulations, ...).
- \(g(x)\) should not reflect these statistical fluctuations, i.e. in such cases approximation more suited than interpolation.
- Select an ansatz \(g(x; \mathbf{a})\) (\(\mathbf{a} = (a_0, \ldots, a_M)\) are parameters, which will be determined such that \(g(x; \mathbf{a})\) approximates the data points in an optimal way).
– E.g. a low degree polynomial, \( g(x; a) = a_0 + a_1 x + a_2 x^2 \) ...
– ... or \( g(x; a) = a_0 / x \) (if \( f_j \) describe a Coulomb-like potential) ...
– ... or \( g(x; a) = (a_0 / x) \exp(-a_1 x) \) (if \( f_j \) describe a potential with a limited range) ...
– ...

• Determine \( a \) by minimizing

\[
G(a) = \sum_{j=0}^{N} \left(g(x_j; a) - f_j\right)^2
\]  

(217)

with respect to \( a \).

– \( (g(x_j; a) - f_j)^2 \): squared difference of approximating function and data points
  (→ “method of least squares”).
– Minimization equivalent to solving

\[
\nabla^{(a)} G(a) = 0.
\]  

(218)

• \( g(x; a) \) linear in \( a_j \),

\[
g(x; a) = \sum_{j=0}^{M} a_j g_j(x)
\]  

(219)

(e.g. \( g_j(x) = x^j \), if \( g(x; a) \) is a degree-\( M \) polynomial):
– Insert (219) in (217):

\[
G(a) = \sum_{j=0}^{N} \left( \sum_{k=0}^{M} a_k g_k(x_j) - f_j \right) \left( \sum_{l=0}^{M} a_l g_l(x_j) - f_j \right) = \sum_{j=0}^{N} \left( \sum_{k=0}^{M} A_{j,k} a_k - f_j \right) \left( \sum_{l=0}^{M} A_{j,l} a_l - f_j \right),
\]  

(220)

where

\[
A = \begin{pmatrix}
g_0(x_0) & g_1(x_0) & \ldots & g_M(x_0) 
g_0(x_1) & g_1(x_1) & \ldots & g_M(x_1) 
\vdots & \vdots & \ddots & \vdots 
g_0(x_N) & g_1(x_N) & \ldots & g_M(x_N)
\end{pmatrix}.
\]  

(221)

– (218):

\[
\frac{\partial}{\partial a_m} G(a) = 2 \sum_{j=0}^{N} \left( \sum_{k=0}^{M} A_{j,k} a_k - f_j \right) A_{j,m} = 0
\]

\[
\rightarrow A^T A a = A^T f
\]  

(222)

i.e. one has to solve a system of linear equations to determine the parameters \( a \) (e.g. by using methods from section 7).
• $g(x; a)$ not linear in $a_j$:
  - (218) is system of non-linear equations.
  - Solving such systems is difficult (see section 5.5).
  - Typically a good estimate of the parameters $a$ is needed to solve such systems of non-linear equations, e.g. by using the Newton-Raphson method.

• Figure 14 shows the least squares approximation of data points already used in Figure 12 right ($N = 9$ example) and Figure 13 using degree-0, degree-1 and degree-2 polynomials; in contrast to Figure 12 and Figure 13 there are no oscillations.

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

10.4 $\chi^2$ minimizing fits

• Quite often, data points have errors, which have been ignored so far.

• Notation: $\sigma_j$ is the error of data point $f_j$ (i.e. “value $f_j \pm \sigma_j$ at $x_j$”).

• When approximating data points using an ansatz $g(x; a)$ (as e.g. in section 10.3), data points with small errors should have a stronger influence on the parameters $a$ than data points with large errors.

• Replace (217) by

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

(223)

to fit $g(x_j; a)$ to the data points $f_j$ with errors $\sigma_j$ in an optimal way (“if $\sigma_j$ is small, $g(x_j; a)$ must be close to $f_j$ ... otherwise $\chi^2$ would be large”).

• Resulting, i.e. minimal $\chi^2$ indicates the quality of the fit:
“Good fit”

→ each term in (223) should be of order 1

→ “reduced \( \chi^2 \) = \( \chi^2 / \text{dof} \) = \( \chi^2 / (N - M) \) \( \approx 1 \).

- \( \chi^2 / \text{dof} \gg 1 \)
  → ansatz \( g(x_j; a) \) not consistent with data points.

- \( \chi^2 / \text{dof} \ll 1 \)
  → errors are either overestimated or data points are correlated.

* For details see textbooks on data analysis.

* Simple and common example: \( \chi^2 \) minimizing fit of a constant \( a \).

  - Ansatz: \( g(x; a) = a \).

  - Minimizing

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

    is equivalent to solving

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

    i.e.

    \[
    a = \sum_{j=0}^{N} \frac{1}{(\sigma_j)^2} f_j = \sum_{j=0}^{N} w_j f_j,
    \] (226)

    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 15.

---

Figure 15: Comparison of a least squares fit (left) and a \( \chi^2 \) 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.
A C Code: trajectories for the HO with the RK method

```c
// 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
#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;
}
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_2ND__

84
// 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
#endif __RK_3RD__
...
#endif
#endif __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 \times \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 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__

#if defined __EULER__
    const int order = 1;
#endif

#if defined __RK_2ND__
    const int order = 2;
#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

/**
 * @param y_t current state vector
 * @param t current time
 * @param y_t_plus_tau vector of states plus tau
 * @param tau step size
 */
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
#endif __RK_3RD__
...
#endif
#endif __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\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 initial condition (energy)
        y[0][N-1] = E;

        // RK computation of the trajectory (= wave function)
        double psi_1 = RK(false);
        printf("%.5e %.5e
", E, psi_1);
    }

    // shooting method


```c
// initial condition (energy)
double E = 10.0;
double E = 40.0;
double E = 90.0;

fprintf(stderr, "E_num = %+10.6lf .\n", E);

while(1)
{
    // change initial condition (energy)
    y[0][N-1] = E;

    // RK computation of the trajectory (= wave function)
    double psi_1_E = RK(false);

    // numerical derivative (d/dh) psi(x=1)
    y[0][N-1] = E-h;
    double psi_1_E_mi_h = RK(false);
    y[0][N-1] = E+h;
    double psi_1_E_pl_h = RK(false);
    double dpsi_1_E = (psi_1_E_pl_h - psi_1_E_mi_h) / (2.0 * h);

    // Newton-Raphson step
    double dE = psi_1_E / dpsi_1_E;
    if(fabs(dE) < dE_min)
        break;
    E = E - dE;

    fprintf(stderr, "E_num = %+10.6lf , E_ana = %+10.6lf , \psi(x=1) = %+.6lf .\n", E, M_PI*M_PI, psi_1_E);
    fprintf(stderr, "E_num = %+10.6lf , E_ana = %+10.6lf , \psi(x=1) = %+.6lf .\n", E, 4.0*M_PI*M_PI, psi_1_E);
    fprintf(stderr, "E_num = %+10.6lf , E_ana = %+10.6lf , \psi(x=1) = %+.6lf .\n", E, 9.0*M_PI*M_PI, psi_1_E);
}

// output
RK(true);
// */
```

// **********

    return EXIT_SUCCESS;
}
D  C Code: Gauss elimination with backward substitution, different pivoting strategies

// solve
// A x = b
// using Gauss elimination with backward substitution and different pivoting strategies

// **********
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// **********
#define __PARTIAL_PIVOTING__
#define __SCALED_PARTIAL_PIVOTING__

// **********
// size of A, b and x
// const int N = 4;
const int N = 100;

// matrix A (elements will be modified during computation)
double A[N][N];

// vector (elements will be modified during computation)
double b[N];

// solution
double x[N];

// permutation of rows due to pivoting
int p[N];

// **********
// generates a uniformly distributed random number in [min,max]
double DRand(double min, double max)
{
    return min + (max-min) * ( (rand()) + 0.5) / (RAND_MAX + 1.0) ;
}

// **********
// print A | b

void Print()
{
    int i1, i2;
for(i1 = 0; i1 < N; i1++)
{
    for(i2 = 0; i2 < N; i2++)
        fprintf(stdout, "%+5.2lf ", A[p[i1]][i2]);
    fprintf(stdout, "| %+5.2lf\n", b[p[i1]]);
}
fprintf(stdout, "\n");

// **********

int main(int argc, char **argv)
{
    double d1, d2, d3;
    int i1, i2, i3;
    srand(0);
    srand((unsigned int)time(NULL));
    // generate random matrix A and vector b, elements in [-1.0,+1.0]
    for(i1 = 0; i1 < N; i1++)
    {
        for(i2 = 0; i2 < N; i2++)
            A[i1][i2] = DRand(-1.0, +1.0);
        b[i1] = DRand(-1.0, +1.0);
    }
    // initialize permutation of rows
    for(i1 = 0; i1 < N; i1++)
        p[i1] = i1;
    Print();
    // *****
    // copy matrix A und vektor b (needed at the end to investigate roundoff errors)
    double A_org[N][N];
    double b_org[N];
    for(i1 = 0; i1 < N; i1++)
    {
        for(i2 = 0; i2 < N; i2++)
            A_org[i1][i2] = A[i1][i2];
        b_org[i1] = b[i1];
    }
    // *****
// elimination

#ifdef __SCALED_PARTIAL_PIVOTING__

// store maximum of each row of A, before elements are modified

double A_ij_max[N];

for(i1 = 0; i1 < N; i1++)
{
    A_ij_max[i1] = fabs(A[i1][0]);
    for(i2 = 1; i2 < N; i2++)
    {
        if(fabs(A[i1][i2]) > A_ij_max[i1])
            A_ij_max[i1] = fabs(A[i1][i2]);
    }
}
#endif

for(i1 = 0; i1 < N-1; i1++)
    // N-1 elimination steps
    {
        // determine "optimal row" according to pivoting strategy
        int index = i1;

#ifdef __PARTIAL_PIVOTING__
        for(i2 = i1+1; i2 < N; i2++)
        {
            if(fabs(A[p[i2]][i1]) > fabs(A[p[i1]][i1]))
                index = i2;
        }
#endif

#ifdef __SCALED_PARTIAL_PIVOTING__
        d1 = fabs(A[p[i1]][i1]) / A_ij_max[p[i1]];
        for(i2 = i1+1; i2 < N; i2++)
        {
            d2 = fabs(A[p[i2]][i1]) / A_ij_max[p[i2]];
            if(d2 > d1)
                index = i2;
        }
#endif

        i2 = p[i1];
        p[i1] = p[index];
        p[index] = i2;

        // ***
for(i2 = i1+1; i2 < N; i2++)
// for all remaining rows ...
{
    d1 = A[p[i2]][i1] / A[p[i1]][i1];
    A[p[i2]][i1] = 0.0;
    for(i3 = i1+1; i3 < N; i3++)
        A[p[i2]][i3] -= d1 * A[p[i1]][i3];
    b[p[i2]] -= d1 * b[p[i1]];
}
Print();
}

// *****
// backward substitution
for(i1 = N-1; i1 >= 0; i1--)
// Für alle Komponenten von x ...
{
    x[i1] = b[p[i1]];
    for(i2 = i1+1; i2 < N; i2++)
        x[i1] -= A[p[i1]][i2] * x[i2];
    x[i1] /= A[p[i1]][i1];
}
fprintf(stdout, "x = ( ");
for(i1 = 0; i1 < N-1; i1++)
{
    fprintf(stdout, "%+5.2lf ", x[i1]);
}
fprintf(stdout, "%+5.2lf ).\n\n", x[N-1]);

// *****
// check solution, investigate roundoff errors
double b_check[N];
for(i1 = 0; i1 < N; i1++)
{
    b_check[i1] = 0.0;
    for(i2 = 0; i2 < N; i2++)
        b_check[i1] += A_org[i1][i2] * x[i2];
}
fprintf(stdout, "b_check = ( ");
for(i1 = 0; i1 < N-1; i1++)
{
fprintf(stdout, "%+5.2lf ", b_check[i1]);

fprintf(stdout, "%+5.2lf 

", 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 ).

", 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.

", norm);

// *****

return EXIT_SUCCESS;
}
// solve the discretized Poisson equation with the conjugate gradient method

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

// number of lattice site in each direction is 2*N+1 (boundary included)
const int n = 100;

// dimension of the matrix A and the vectors x and b
const int N = (2*n - 1) * (2*n - 1);

// stop conjugate gradient method as soon as residual is smaller than eps
double eps_eps = 0.0000000001 * 0.0000000001;

// allocate vector

double *Alloc_Vector()
{
    double *v = (double *)malloc(N * sizeof(double));

    if(v == NULL)
    {
        fprintf(stderr, "Error: double *Alloc_Vector(...\n");
        exit(EXIT_FAILURE);
    }

    return v;
}

// superindex

int Index(int ix, int iy)
{
    if(ix <= -n || ix >= +n || iy <= -n || iy >= +n)
        return -1;

    return (2*n-1)*(iy+(n-1)) + (ix+(n-1));
}

// compute y = A x

void A(double *y, const double *x)
```c
{ int ix, iy;

for(ix = -(n-1); ix <= +(n-1); ix++)
{
  for(iy = -(n-1); iy <= +(n-1); iy++)
  {
    y[Index(ix, iy)] = -4.0 * x[Index(ix, iy)];

    if(Index(ix-1, iy) != -1)
      y[Index(ix, iy)] += x[Index(ix-1, iy)];

    if(Index(ix+1, iy) != -1)
      y[Index(ix, iy)] += x[Index(ix+1, iy)];

    if(Index(ix, iy-1) != -1)
      y[Index(ix, iy)] += x[Index(ix, iy-1)];

    if(Index(ix, iy+1) != -1)
      y[Index(ix, iy)] += x[Index(ix, iy+1)];
  }
}

// **********

int main(int argc, char **argv)
{
  int i1;
  double *v1 = Alloc_Vector();

  double *x = Alloc_Vector();
  double *b = Alloc_Vector();

  double *r = Alloc_Vector();
  double *p = Alloc_Vector();

  // *****

  // intitialize right hand side b
  for(i1 = 0; i1 < N; i1++)
    b[i1] = 0.0;

  b[Index(0, 0)] = 1.0;

  // *****

  // initialize solution x with 0.0
  for(i1 = 0; i1 < N; i1++)
    x[i1] = 0.0;

  // *****

  // r = b - A x
```
A(v1, x);

for(i1 = 0; i1 < N; i1++)
    r[i1] = b[i1] - v1[i1];

// p = r
for(i1 = 0; i1 < N; i1++)
    p[i1] = r[i1];

// *****
// conjugate gradient iteration

int ctr = 0;

double r_r = 0.0;

for(i1 = 0; i1 < N; i1++)
    r_r += r[i1] * r[i1];

while(1)
{
    ctr++;
    fprintf(stderr, "ctr = %4d.\n", ctr);

    if(ctr == 1000000)
        break;

    // ***
    // alpha = r^2 / (p A p)
    A(v1, p);

    double p_A_p = 0.0;

    for(i1 = 0; i1 < N; i1++)
        p_A_p += p[i1] * v1[i1];

    double alpha = r_r / p_A_p;

    // x = x + alpha p
    for(i1 = 0; i1 < N; i1++)
        x[i1] += alpha * p[i1];

    // r = r - alpha A p
    for(i1 = 0; i1 < N; i1++)
        r[i1] -= alpha * v1[i1];

    // beta = (r_new)^2 / (r_old)^2
    double r_r_old = r_r;
    r_r = 0.0;
}
for(i1 = 0; i1 < N; i1++)
    r_r += r[i1] * r[i1];

fprintf(stderr, "  r_r = %.5e (%.5e).
", r_r, r_r_old);

if(r_r < eps_eps)
    // Hinreichend genaues x erreicht.
    break;

double beta = r_r / r_r_old;

    // p = r + beta p

for(i1 = 0; i1 < N; i1++)
    p[i1] = r[i1] + beta * p[i1];
}

// *****
// check result by comparing A x to b

double *b_check = Alloc_Vector();
A(b_check, x);
double norm = 0.0;
for(i1 = 0; i1 < N; i1++)
    norm += pow(b_check[i1] - b[i1], 2.0);

norm = sqrt(norm);
fprintf(stderr, "\n|b_check - b| = %.5e.\n", norm);

// *****
// output of electrostatic potential \phi = x

for(i1 = -(n-1); i1 <= +(n-1); i1++)
    fprintf(stdout, "%.5e %.5e\n", ((double)i1) / ((double)n), x[Index(i1, 0)]); // on axis
    // fprintf(stdout, "%.5e %.5e\n", ((double)i1) / ((double)n) * sqrt(2.0), x[Index(i1, i1)]); // diagonal

// *****
free(v1);
free(b);
free(x);
free(r);
free(p);
free(b_check);

    // *****

return EXIT_SUCCESS;
}
F    C Code: eigenvalues and eigenvectors of a $10 \times 10$ 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 < epsilon
// **********
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 division 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[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


