Numerical methods in physics

Marc Wagner

Goethe-Universität Frankfurt am Main – summer semester 2018

Version: April 26, 2018

Contents

****** April 13, 2018 (1. lecture) ******

1 Introduction

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

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

2 Representation of numbers in computers, roundoff errors

2.1 Integers

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

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

- Typically $N = 32$ (sometimes also $N = 8, 16, 64, 128$)
  $\rightarrow 0 \leq z \leq 2^{32} - 1 = 4\,294\,967\,295$.
- Negative integers: very similar (homework: study Wiki,
  `https://en.wikipedia.org/wiki/Integer_(computer_science)`).
- 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 computes by floating point numbers,

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

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

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

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

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

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

– `double`:

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

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

2.3 Roundoff errors

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

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

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

2.3.1 Simple examples

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

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

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

****** April 20, 2018 (2. lecture) ******

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).
- Common approach: approximate $f'(x)$ numerically by finite difference, e.g.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  i.e. symmetric derivative superior to asymmetric derivative.

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

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

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

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

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

  // *****

  printf("h            rel_err_asym   rel_err_sym\n");

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

      double df_exact = cos(1.0);

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

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

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

  // *****

  return EXIT_SUCCESS;
}
```

---

[2] Throughout this lecture I use C.

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

---

3 Ordinary differential equations (ODEs), initial value problems

3.1 Physics motivation

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

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

initial conditions

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

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

3.2 Euler's method

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

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

  - Define

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

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

  - Then

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

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

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

XXXXX Figure 3.A XXXXX

- τ can be positive ($\rightarrow$ computation of future) or negative ($\rightarrow$ computation of past).

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

3.3 Runge-Kutta (RK) method

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

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

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

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

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

XXXXX Figure 3.B XXXXX

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

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

$$= \mathbf{f}\tau + \left(\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\frac{1}{2}\mathbf{f}\tau + \frac{\partial \mathbf{f}}{\partial t}\frac{1}{2}\tau\right)\tau + \mathcal{O}(\tau^3) = \mathbf{f}\tau + \frac{1}{2}\left(\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\dot{\mathbf{y}} + \frac{\partial \mathbf{f}}{\partial t}\right)\tau^2 + \mathcal{O}(\tau^3) =$$

$$= \mathbf{f}\tau + \frac{1}{2}\dot{\mathbf{f}}\tau^2 + \mathcal{O}(\tau^3) \tag{23}$$

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

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

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

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

– "3rd-order RK":

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

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

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

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

– "4th-order RK":

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

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

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

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

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

– ...

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

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

  - EOMs:

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

    i.e.

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

    with

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

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

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

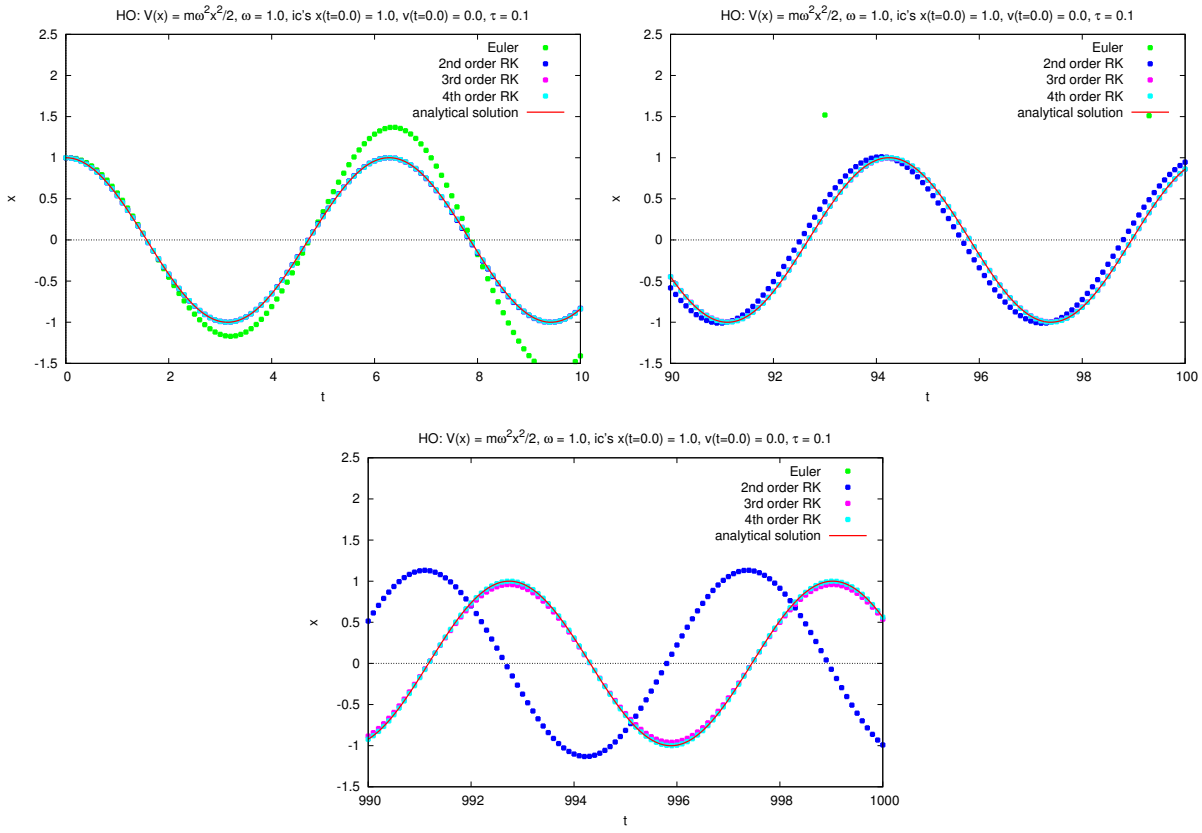

Figure 1: HO, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK.

– Corresponding C code: cf. appendix A.

****** April 27, 2018 (3. lecture) ******

3.3.1 Estimation of errors

- Error per step for n-th order RK can be estimated in the following way:
  - RK step with step size τ
    $\rightarrow \mathbf{y}_\tau(t+\tau)$
    $\rightarrow \vec{\delta}_\tau \approx \mathbf{c}\tau^{n+1}$.
    c depends on properties of the system
  - 2 RK steps with step size $\tau/2$
    $\rightarrow \mathbf{y}_{2\times\tau/2}(t+\tau)$
    $\rightarrow \vec{\delta}_{2\times\tau/2} \approx 2\mathbf{c}(\tau/2)^{n+1}$.
    XXXXX Figure 3.C XXXXX
  - Estimated absolute error for $\mathbf{y}_{2\times\tau/2}(t+\tau)$:

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

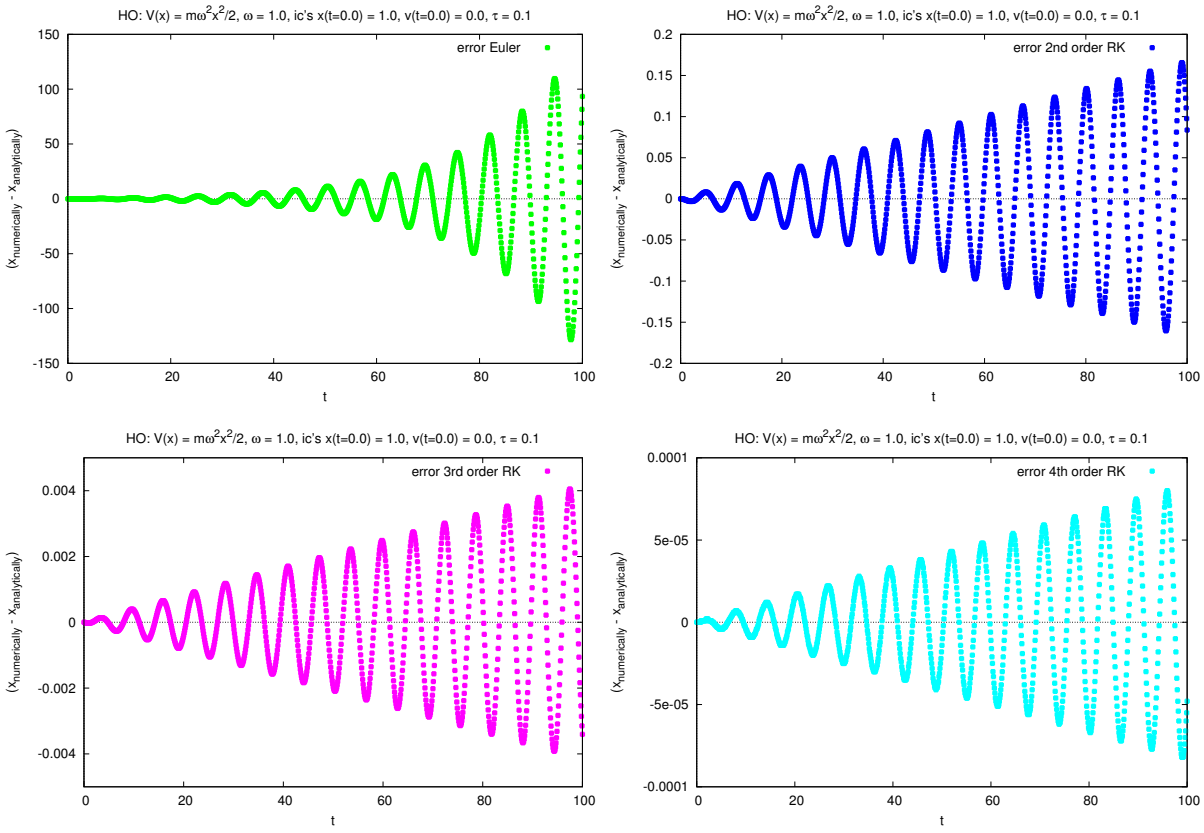

Figure 2: HO, errors of the trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK.

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

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

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

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

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

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

3.3.2 Adaptive step size

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

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

  XXXXX Figure 3.D XXXXX
- For given maximum tolerable error $\delta_{\text{abs,max}}$ or $\delta_{\text{rel,max}}$, estimated error allows to estimate corresponding step size τ_{max}:

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

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

  (1) RK steps:

$$\mathbf{y}(t) \rightarrow_{\tau} \mathbf{y}_{\tau}(t+\tau) \tag{43}$$
$$\mathbf{y}(t) \rightarrow_{\tau/2} \rightarrow_{\tau/2} \mathbf{y}_{2\times\tau/2}(t+\tau). \tag{44}$$

  - *Estimated error:*

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

  - *Change step size:*

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

    conservative solution (0.9)

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

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

    Else:

    $\rightarrow$ $\tau = \tau_{new}$ *(i.e. reduce step size).*
    Go to (1) (i.e. repeat RK steps with smaller step size).
- Modifications possible, e.g. estimate error and τ_{new} by performing RK steps of n-th and $n+1$-th order instead of RK steps with step sizes τ and $\tau/2$.

- Example: 1D anharmonic oszillator.
  - Lagrangian:

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

  - EOMs:

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

    i.e.

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

    with

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

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

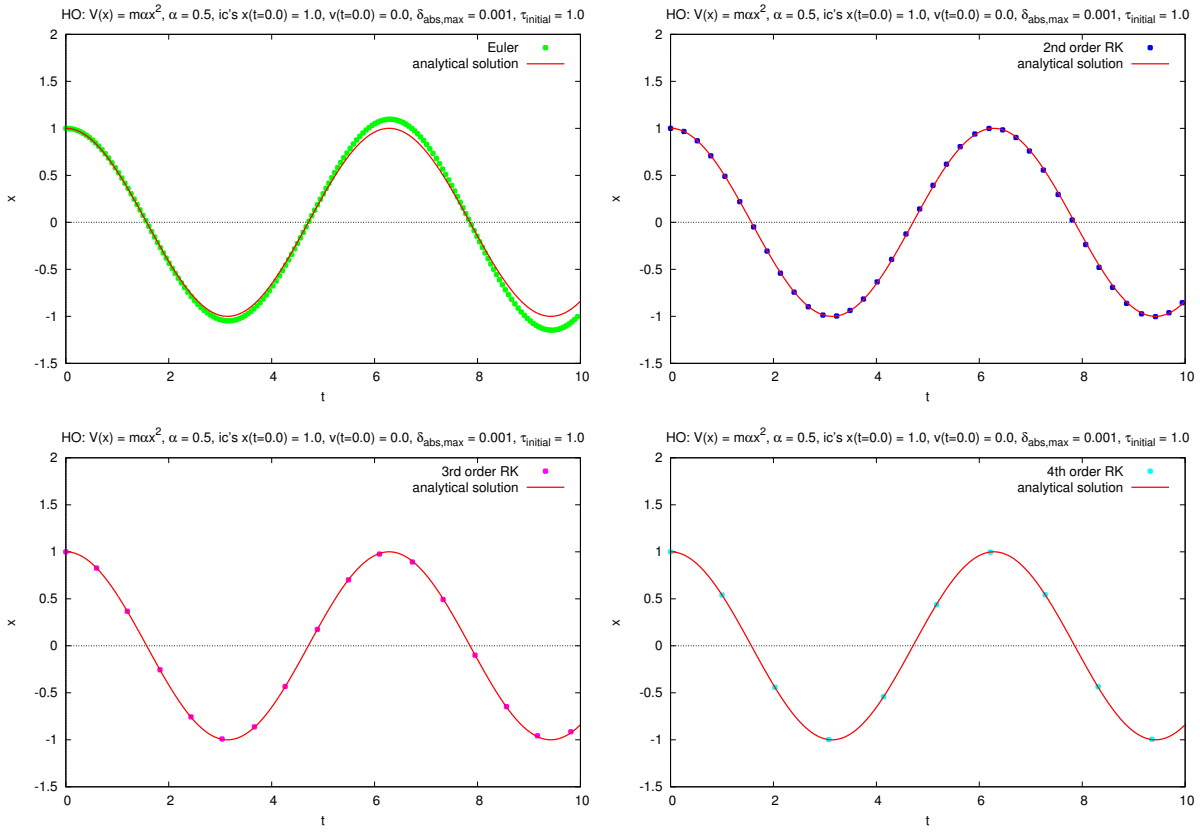

Figure 3: Harmonic oscillator, $V(x) = m\alpha x^2$, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK using adaptive step size.

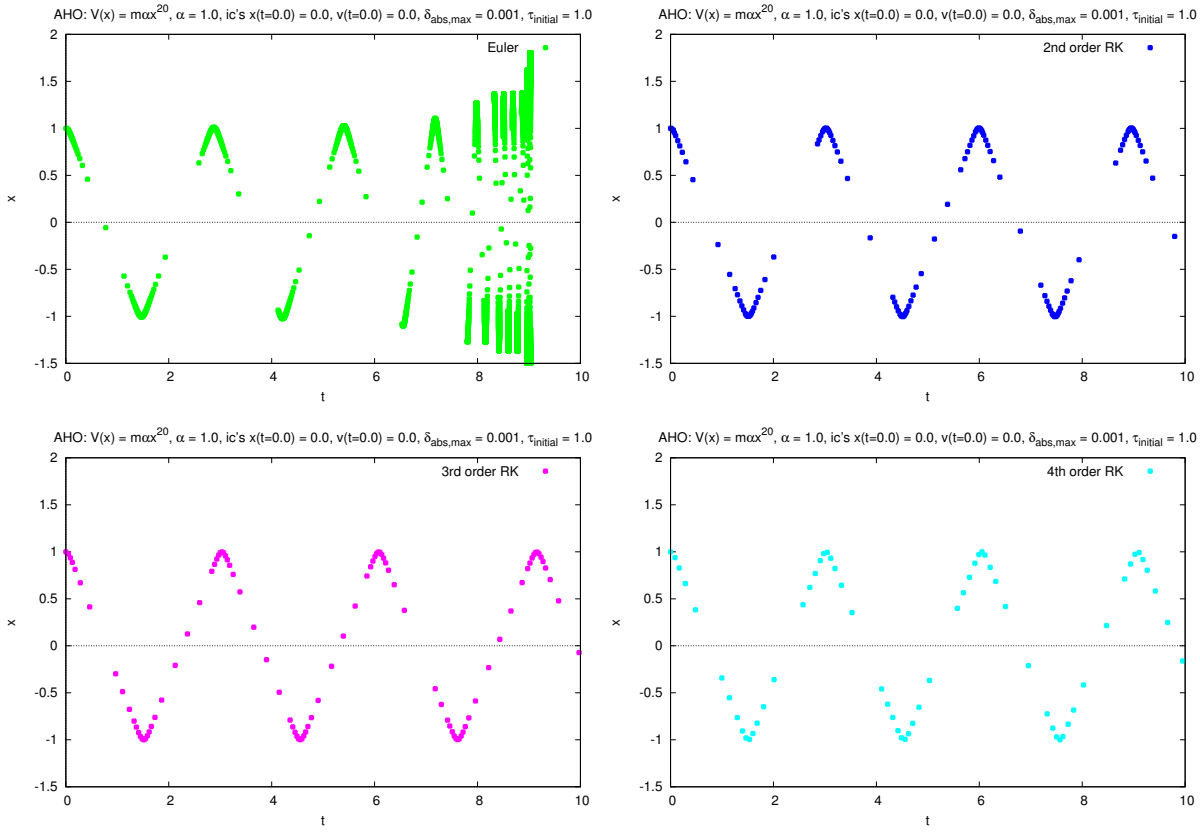

Figure 4: Anharmonic oscillator, $V(x) = m\alpha x^{20}$, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK using adaptive step size.

4 Dimensionful quantities on a computer

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

4.1 Method 1: define units for your computation

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

4.2 Method 2: use exclusively dimensionless quantities

- Reformulate the problem using exclusively dimensionless quantities.
- Example: compute the trajectory of the 1D harmonic oszillator (same example as in section 3.3).
  - Lagrangian:
    $$L = \frac{m}{2}\dot{x}^2 - \frac{m\omega^2}{2}x^2. \tag{51}$$
  - EOMs:
    $$m\ddot{x}(t) = -m\omega^2 x(t) \quad \rightarrow \quad \ddot{x}(t) = -\omega^2 x(t), \tag{52}$$
    i.e. m irrelevant.
  - Measure time in units of ω:
    $$\hat{t} = \omega t \quad \rightarrow \quad \frac{d^2}{d\hat{t}^2}x(\hat{t}) = -x(\hat{t}). \tag{53}$$
  - Moreover, initial conditions introduce length scale, e.g. $x(t=0) = x_0$, $\dot{x}(t=0) = 0$ $\rightarrow$ measure x in units of x_0:
    $$\hat{x} = \frac{x}{\hat{x}} \quad \rightarrow \quad \frac{d^2}{d\hat{t}^2}\hat{x}(\hat{t}) = -\hat{x}(\hat{t}). \tag{54}$$
  - Now only dimensionless quantities in (54), i.e. straightforward to treat numerically.
  - Figure 5 showing trajectory $\hat{x}(\hat{t})$ is analog of Figure 1 (left top).
- Advantage: a single computation for different parameter sets (above example: trajectory $\hat{x}(\hat{t})$ shown in Figure 5 valid for arbitrary m, ω and x_0).

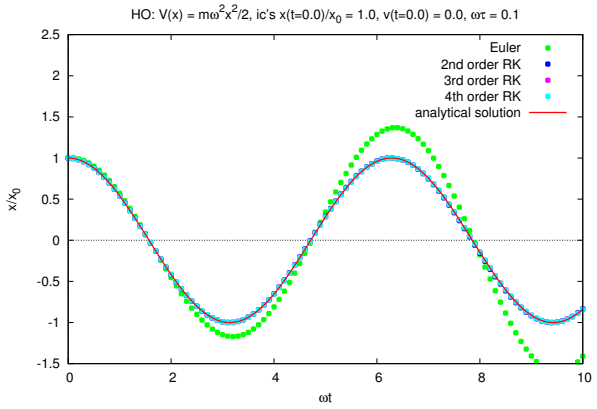

Figure 5: HO, resulting trajectories for Euler, 2nd-order RK, 3rd-order RK and 4th-order RK (same data as in Figure 1 [left top], but coordinate axes correspond to dimensionless quantities $\hat{t} = \omega t$ and $\hat{x} = x/x_0$).

5 Root finding, solving systems of non-linear equations

5.1 Physics motivation

- N non-linear equations with N unknowns,

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

or equivalently written in a more compact way

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

- Find solutions $\mathbf{x}$ of (56), i.e. find zeros of $\mathbf{f}(\mathbf{x})$.
- Standard problem in physics, e.g. needed to solve the Schrödinger equation (cf. section 6).
- For systems of linear equations cf. 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 zero of $f(x)$, somewhere between x_1 and x_2.
- Algorithm:

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

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

    Else:

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

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

    Else:

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

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

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

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

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

- Advantages and disadvantages:

  (+) Always finds a zero.

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

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

- Starting point: x_1, x_2 fulfilling $|f(x_2)| < |f(x_1)|$.
- Secant method might find a zero of $f(x)$, not necessarily between x_1 and x_2.
- Basic principle:
  - Iteration.
  - Each step as sketched below.
    XXXXX Figure 5.A XXXXX
- Algorithm:
  - $n = 2$.

  (1)

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

  - *If Δx sufficiently small:*

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

    Else:

    → $n = n + 1$.
    Go to (1).

  XXXXX Figure 5.B XXXXX

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

  (+) Converges faster than bisection.

  (−) Does not always find a zero.

6 Ordinary differential equations, boundary value problems

7 Solving systems of linear equations

A C Code: trajectories for the HO with RK

```
// solve system of ODEs
//
//   \vec{\dot{y}}(t) = \vec{f}(\vec{y}(t),t) ,
//
// initial conditions
//
//   \vec{y}(t=0) = \vec{y}_0 ,
//
// HO, potential
//
//   V(x) = m \omega^2 x^2 / 2

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

#define __EULER__
// #define __RK_2ND__
// #define __RK_3RD__
// #define __RK_4TH__

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

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

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

const int N = 2;  // number of components of \vec{y} and \vec{f}

const double omega = 1.0;  // frequency

const int num_steps = 10000;  // number of steps
const double tau = 0.1;  // step size

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

double y[N][num_steps+1];  // discretized trajectories

double y_0[N] = { 1.0 , 0.0 };  // initial conditions

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

int main(int argc, char **argv)
{
  int i1, i2;

  // *****

  // initialize trajectories with initial conditions

  for(i1 = 0; i1 < N; i1++)
    y[i1][0] = y_0[i1];
```

```
  // *****

  // Euler/RK steps

  for(i1 = 1; i1 <= num_steps; i1++)
    {
      // 1D HO:
      // y(t) = (x(t) , \dot{x}(t)) ,
      // \dot{y}(t) = f(y(t),t) = (\dot{x}(t) , F/m) ,
      //   where force F = -m \omega^2 x(t)

#ifdef __EULER__

      // k1 = f(y(t),t) * tau

      double k1[N];

      k1[0] = y[1][i1-1]   * tau;
      k1[1] = -pow(omega, 2.0) * y[0][i1-1]   * tau;

      // *****

      for(i2 = 0; i2 < N; i2++)
y[i2][i1] = y[i2][i1-1] + k1[i2];

#endif

#ifdef __RK_2ND__

      // k1 = f(y(t),t) * tau

      double k1[N];

      k1[0] = y[1][i1-1]   * tau;
      k1[1] = -pow(omega, 2.0) * y[0][i1-1]   * tau;

      // *****

      // k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau

      double k2[N];

      k2[0] = (y[1][i1-1] + 0.5*k1[1])   * tau;
      k2[1] = -pow(omega, 2.0) * (y[0][i1-1] + 0.5*k1[0])   * tau;

      // *****

      for(i2 = 0; i2 < N; i2++)
y[i2][i1] = y[i2][i1-1] + k2[i2];

#endif

#ifdef __RK_3RD__

...

#endif
```

```
#ifdef __RK_4TH__

...

#endif
    }

  // *****

  // output

  for(i1 = 0; i1 <= num_steps; i1++)
    {
      double t = i1 * tau;
      printf("%9.6lf %9.6lf %9.6lf\n", t, y[0][i1], y[0][i1]-cos(t));
    }

  // *****

  return EXIT_SUCCESS;
}
```

B C Code: trajectories for the anharmonic oscillator with RK with adaptive step size

---

```
// solve system of ODEs
//
// \vec{\dot{y}}(t) = \vec{f}(\vec{y}(t),t) ,
//
// initial conditions
//
// \vec{y}(t=0) = \vec{y}_0 ,
//
// anharmonic oscillator, potential
//
// V(x) = m \alpha x^n ,
//
// adaptive stepsize

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

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

// **********
// physics parameters and functions
// **********

// anharmonic oscillator, V(x) = m \alpha x^n,
//   y = (x , v)
//   f = (v , -\alpha n x^n-1)

const int N = 2;  // number of components of \vec{y} and \vec{f}

// const int n = 2;
// const double alpha = 0.5;
const int n = 20;
const double alpha = 1.0;

double y_0[N] = { 1.0 , 0.0 };  // initial conditions

// function computing f(y(t),t) * tau

void f_times_tau(double *y_t, double t, double *f_times_tau_, double tau)
{
  if(N != 2)
    {
      fprintf(stderr, "Error: N != 2!\n");
      exit(EXIT_FAILURE);
    }

  f_times_tau_[0] = y_t[1] * tau;
  f_times_tau_[1] = -alpha * ((double)n) * pow(y_t[0], ((double)(n-1))) * tau;
}
```

```
// **********
// RK parameters
// **********

// #define __EULER__
#define __RK_2ND__
// #define __RK_3RD__
// #define __RK_4TH__

#ifdef __EULER__
const int order = 1;
#endif

#ifdef __RK_2ND__
const int order = 2;
#endif

#ifdef __RK_3RD__
const int order = 3;
#endif

#ifdef __RK_4TH__
const int order = 4;
#endif

// maximum number of steps
const int num_steps_max = 10000;

// compute trajectory for 0 <= t <= t_max
const double t_max = 10.0;

// maximum tolerable error
const double delta_abs_max = 0.001;

double tau = 1.0;  // initial step size

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

double t[num_steps_max+1];  // discretized time
double y[num_steps_max+1][N];  // discretized trajectories

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

#ifdef __EULER__

...

#endif

#ifdef __RK_2ND__

// RK step (2nd order), step size tau

void RK_step(double *y_t, double t, double *y_t_plus_tau, double tau)
{
  int i1;
```

```
  // *****

  // k1 = f(y(t),t) * tau

  double k1[N];
  f_times_tau(y_t, t, k1, tau);

  // *****

  // k2 = f(y(t)+(1/2)*k1 , t+(1/2)*tau) * tau

  double y_[N];

  for(i1 = 0; i1 < N; i1++)
    y_[i1] = y_t[i1] + 0.5*k1[i1];

  double k2[N];
  f_times_tau(y_, t + 0.5*tau, k2, tau);

  // *****

  for(i1 = 0; i1 < N; i1++)
    y_t_plus_tau[i1] = y_t[i1] + k2[i1];
}

#endif

#ifdef __RK_3RD__

...

#endif

#ifdef __RK_4TH__

...

#endif

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

int main(int argc, char **argv)
{
  double d1;
  int i1, i2;

  // *****

  // initialize trajectories with initial conditions

  t[0] = 0.0;

  for(i1 = 0; i1 < N; i1++)
    y[0][i1] = y_0[i1];

  // *****
```

```
// RK steps

for(i1 = 0; i1 < num_steps_max; i1++)
  {
    if(t[i1] >= t_max)
     break;

    // *****

    double y_tau[N], y_tmp[N], y_2_x_tau_over_2[N];

    // y(t)   --> \tau   y_{\tau}(t+\tau)
    RK_step(y[i1], t[i1], y_tau, tau);

    // y(t)   --> \tau/2   --> \tau_2   y_{2 * \tau / 2}(t+\tau)
    RK_step(y[i1], t[i1], y_tmp, 0.5*tau);
    RK_step(y_tmp, t[i1]+0.5*tau, y_2_x_tau_over_2, 0.5*tau);

    // *****

    // estimate error

    double delta_abs = fabs(y_2_x_tau_over_2[0] - y_tau[0]);

    for(i2 = 1; i2 < N; i2++)
    {
      d1 = fabs(y_2_x_tau_over_2[i2] - y_tau[i2]);

      if(d1 > delta_abs)
        delta_abs = d1;
    }

    delta_abs /= pow(2.0, (double)order) - 1.0;

    // *****

    // adjust step size (do not change by more than factor 5.0).

    d1 = 0.9 * pow(delta_abs_max / delta_abs, 1.0 / (((double)order)+1.0));

    if(d1 < 0.2)
    d1 = 0.2;

    if(d1 > 5.0)
    d1 = 5.0;

    double tau_new = d1 * tau;

    // *****

    if(delta_abs <= delta_abs_max)
    {
      // accept RK step

       for(i2 = 0; i2 < N; i2++)
        y[i1+1][i2] = y_2_x_tau_over_2[i2];
```

```
        t[i1+1] = t[i1] + tau;

       tau = tau_new;
       }
     else
     {
       // repeat RK step with smaller step size
      tau = tau_new;

        i1--;
     }
   }

  int num_steps = i1;

  // *****

  // output

  for(i1 = 0; i1 <= num_steps; i1++)
    {
      printf("%9.6lf %9.6lf\n", t[i1], y[i1][0]);
    }

  // *****

  return EXIT_SUCCESS;
}
```

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical recipes 3rd edition: the art of scientific computing," Cambridge University Press (2007).