

ETH Lectures 401-0663-00L Numerical Methods for Computer Science
401-2673-00L Numerical Methods for CSE

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2022, Version of December 2, 2022
(C) Seminar für Angewandte Mathematik, ETH Zürich

[Link](#) to the current version of this homework collection

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single “.pdf” file. Every week, we publish a list of problem numbers on the lecture Moodle page. These are the current assignments for that week. Details are given on the lecture Moodle page.

0.1.2 Importance of Homework

Homework assignments are **not** mandatory. However, it is **very important** that you constantly exercise with the material you learn. “Solving” the homework assignments one week before the main exam by looking at the solutions will likely result in failure.

We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck with a difficult problem and you tried to find a solution for a sufficient amount of time.

Most homework problems come with an estimate for the time required for its solution. These times are what we expect a bright and well-prepared student to require for that problem during an exam.

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by handing them to your tutor directly or uploading them via the Moodle upload interface, see the lecture Moodle page for instructions. The assistants will

Problem 11-2: Explicit Runge-Kutta methods

The most widely used class of numerical integrators for IVPs is that of *explicit* Runge-Kutta (RK) methods as defined in [Lecture → Def. 11.4.0.11]. They are usually described by giving their coefficients in the form of a Butcher scheme [Lecture → Eq. (11.4.0.13)].

Related to [Lecture → Section 11.4], [Lecture → Def. 11.4.0.11], and Butcher schemes as defined in [Lecture → Eq. (11.4.0.13)]

(11-2.a) 🕒 (30 min.) In the template file `rkintegrator.hpp` implement a header-only C++ class

```
template <class State>
class RKIntegrator {
public:
    // Constructor
    RKIntegrator(const MatrixXd & A, const VectorXd & b);
    // Explicit Runge-Kutta numerical integrator
    template <class Function>
    std::vector<State>
        solve(Function &f, double T,
              const State &y0, unsigned int N) const;
    // .... data (and private methods)
};
```

which provides a generic **explicit Runge-Kutta single-step method** (→ [Lecture → Def. 11.4.0.11]) given by a Butcher scheme [Lecture → Eq. (11.4.0.13)] to solve the autonomous initial-value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(t_0) = \mathbf{y}_0$. The matrix $\mathbf{A} \in \mathbb{R}^{s,s}$ and the vector $\mathbf{b} \in \mathbb{R}^s$ from the Butcher scheme are passed to the constructor, whereas the right-hand side vector field $\mathbf{f}$, the final time $T > 0$, the number N of *equidistant timesteps*, and the initial state $\mathbf{y}_0$ are arguments of the `solve()` method, which is supposed to return the sequence $(\mathbf{y}_k)_{k=0}^N$ of states generated by the explicit Runge-Kutta method.

SOLUTION of (11-2.a):

There are many ways to implement the formulas of [Lecture → Def. 11.4.0.11]. Thanks to the restriction to explicit RK-SSM all merely involve $\mathbf{f}$ -evaluations.

C++11-code 11.2.1: Solution of (11-2.a).

```
2  template <class State>
3  class RKIntegrator {
4  public:
5      /*!
6       *! \brief Constructor for the RK method.
7       *! Performs size checks and copies A and b into internal storage.
8       *! \param[in] A Matrix containing coefficients of the Butcher tableau,
9       *! must be (strictly) lower triangular (no check is done).
10      *! \param[in] b Vector containing coefficients of lower
11      *! part of Butcher tableau.
12      */
13      RKIntegrator(const MatrixXd & A, const VectorXd & b)
14          // TO DO: Use an initializer list to Initialize the private
15          // variables A, b, s.
16          // START
17          : A(A), b(b), s(b.size())
```

```

18 // END
19 {
20 // TO DO: Use assert() to make sure A is a square matrix of
21 // shape s*s with s=b.size().
22 // START
23 assert(A.cols() == A.rows() && "Matrix must be square.");
24 assert(A.cols() == b.size() && "Incompatible matrix/vector size.");
25 // END
26 }
27
28 /*!
29 *! \brief Perform the solution of the ODE.
30 *! Solve an autonomous ODE  $y' = f(y)$ ,  $y(0) = y_0$ , using a
31 *! RK scheme given in the Butcher tableau provided in the
32 *! constructor. Performs N equidistant steps up to time
33 *! T with initial data  $y_0$ .
34 *! \tparam Function type for function implementing the rhs function.
35 *! Must have State operator() (const State & x).
36 *! \param[in] f function handle for rhs in  $y' = f(y)$ , e.g. \
37 *! implemented using lambda function.
38 *! \param[in] T The final time T.
39 *! \param[in] y0 Initial data  $y(0) = y_0$  for  $y' = f(y)$ .
40 *! \param[in] N Number of steps to perform. Step size is  $h = T/N$ .
41 *! Steps are equidistant.
42 *! \return The vector containing all steps  $y^n$  (for each n)
43 *! including initial and final value.
44 */
45 template <class Function>
46 std::vector<State> solve(Function &&f, double T, const State &y0, unsigned
47 int N) const {
48 std::vector<State> res;
49 // Will contain all steps, reserve memory for efficiency
50 res.reserve(N+1);
51 // TO DO: Compute the solution of  $y' = f(y)$ ,  $y(0) = y_0$  up to time T using
52 // N Runge-Kutta steps, and store the solution at step n in res[n].
53 // Use the private method step() for the Runge-Kutta step.
54 // START
55 // Step size
56 double h = T / N;
57 // Will contain all steps, reserve memory for efficiency
58 res.reserve(N + 1);
59 // Store initial data
60 res.push_back(y0);
61
62 // Initialize some memory to store temporary values
63 State ytemp1 = y0;
64 State ytemp2 = y0;
65 // Pointers to swap previous value
66 State *yold = &ytemp1;
67 State *ynew = &ytemp2;
68
69 // Loop over all fixed steps
70 for (unsigned int k = 0; k < N; ++k) {
71 // Compute, save and swap next step
72 step(f, h, *yold, *ynew);
73 res.push_back(*ynew);
74 std::swap(yold, ynew);
75 }
76 // END
77 return res;

```

```

77     }
78
79 private:
80     /*!
81     *! \brief Perform a single step of the RK method.
82     *! Solve an autonomous ODE using an explicit Runge Kutta Method.
83     *! Compute a single explicit RK step  $y^{n+1} = y_n + \sum \dots$ 
84     *! starting from value  $y_0$  and storing next value in  $y_1$ .
85     *! \tparam Function type for function implementing the rhs.
86     *! Must have State operator() (State x)
87     *! \param[in] f function handle for rhs  $f$ , s.t.  $y' = f(y)$ 
88     *! \param[in] h step size
89     *! \param[in]  $y_0$  initial state
90     *! \param[out]  $y_1$  next step  $y^{n+1} = y^n + \dots$ 
91     */
92     template <class Function>
93     void step(const Function &f, double h, const State &y0, State &y1) const {
94         // TO DO: Compute a single step of the explicit Runge-Kutta method
95         // (defined by A, b, s) for  $y' = f(y)$  starting at State  $y_0$  and using
96         // step size  $h$ . Store the result in the State  $y_1$ .
97         // START
98         // create vector holding next value
99         y1 = y0;
100        // Reserve space for increments
101        std::vector<State> k;
102        k.reserve(s);
103
104        // Loop over the size of RK
105        for (unsigned int i = 0; i < s; ++i) {
106            // Compute increments and save them to k
107            State incr = y0;
108            for (unsigned int j = 0; j < i; ++j) {
109                incr += h * A(i, j) * k.at(j);
110            }
111            k.push_back(f(incr));
112            y1 += h * b(i) * k.back();
113        }
114        // END
115    }
116
117    //! Matrix A in Butcher scheme
118    const MatrixXd A;
119    //! Vector b in Butcher scheme
120    const VectorXd b;
121    //! Size of Butcher matrix and vector A and b
122    unsigned int s;
123 };

```

There is no real reason to introduce the method `step()`; everything can be implemented in `solve()`.



(11-2.b) 🕒 (30 min.) [depends on Sub-problem (11-2.a)]

In the template file `rk3prey.hpp` implement a C++ function

double RK3prey();

in order to test your implementation of the RK methods with the following test case:

As autonomous initial value problem, consider the predator/prey model (cf. [Lecture → Ex. 11.1.2.5]):

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t)) y_1(t), \quad (11.2.2)$$

$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2) y_2(t), \quad (11.2.3)$$

$$\mathbf{y}(0) = [100, 5]^\top, \quad (11.2.4)$$

with coefficients $\alpha_1 = 3, \alpha_2 = 2, \beta_1 = \beta_2 = 0.1$.

Use the **explicit 3-stage Runge-Kutta single step method** described by the following **Butcher scheme** (cf. [Lecture → Def. 11.4.0.11]):

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{1}{3} & \frac{1}{3} & 0 & \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array} \quad (11.2.5)$$

Compute an approximated solution up to time $T = 10$ for the number of equidistant time steps $N = 2^j, j = 7, \dots, 14$.

As reference solution, use $\mathbf{y}(10) = [0.319465882659820, 9.730809352326228]^\top$.

Tabulate the error at final time and estimate the empiric rate of algebraic convergence of the method by means of linear regression using the supplied function `polyfit` (file `polyfit.hpp`). Your function should return the estimated convergence rate.

SOLUTION of (11-2.b):

The main task is to prepare the inputs for the application of **RKIntegrator**. The right-hand-side function **f** can be passed as lambda function, see [Lecture → Section 0.3.3].

C++11-code 11.2.6: Solution of (11-2.b)

```
2  double RK3prey() {
3      double conv_rate = 0;
4      // Dimension of state space
5      unsigned int d = 2;
6      // Exact value y(10) at final time T = 10 (approximated)
7      Eigen::VectorXd yex(d);
8      yex << 0.319465882659820, 9.730809352326228;
9
10     // TO DO: Use an RKIntegrator to solve the predator/prey IVP up to T=10,
11     // as described in (12-2.b). Tabulate the error as the number of steps
12     // increases N = 2^7, 2^8, ..., 2^14, and estimate the convergence rate.
13     // HINT: You may use polyfit() to calculate the convergence rate.
14     // START
15
16     // Implementation of butcher scheme
17     unsigned int s = 3;
18     Eigen::MatrixXd A(s,s);
19     Eigen::VectorXd b(s);
20     A << 0,      0,      0,
21          1./3.,  0,      0,
22          0,      2./3.,  0;
23     b << 1./4.,  0,      3./4.;
24 }
```

```

25 // Initialize RK with Butcher scheme
26 RKIntegrator<Eigen::VectorXd> RK(A,b);
27
28 // Coefficients and handle for prey/predator model
29 double alpha1 = 3.;
30 double alpha2 = 2.;
31 double beta1 = 0.1;
32 double beta2 = 0.1;
33 auto f = [&alpha1, &alpha2, &beta1, &beta2] (const Eigen::VectorXd & y) {
34     auto temp = y;
35     temp(0) *= alpha1 - beta1*y(1);
36     temp(1) *= -alpha2 + beta2*y(0);
37     return temp;
38 };
39
40 // Initial value for model
41 Eigen::VectorXd y0(d);
42 y0 << 100, 5;
43
44 // Final time for model
45 double T = 10.;
46
47 // Array of number of steps (for convergence study)
48 ArrayXd N(8);
49 N << 128, 256, 512, 1024, 2048, 4096, 8192, 16384;
50 ArrayXd Error(N.size());
51
52 // Start convergence study
53 std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::endl;
54 for(unsigned int i = 0; i < N.size(); ++i) {
55     auto res = RK.solve(f, T, y0, N[i]);
56     double err = (res.back() - yex).norm();
57     std::cout << std::setw(15) << N[i] << std::setw(15) << err << std::endl;
58     Error[i] = err;
59 }
60
61 VectorXd coeffs = polyfit(N.log(), Error.log(), 1);
62 conv_rate = -coeffs(0);
63 // END
64 return conv_rate;
65 }

```

N	error norm	rate
128	8.9601	
256	0.60584	3.88651
512	0.0688263	3.13791
1024	0.00838567	3.03696
2048	0.00103847	3.01347
4096	0.000129287	3.00581
8192	1.61305e-05	3.00271
16384	2.01448e-06	3.00131

In the table beside the column “rate” gives a “local” estimate of the rate α obtained from the quotient of two successive error norms:

$$\text{err}(\tfrac{1}{2}h) : \text{err}(h) \approx 2^{-\alpha} \Leftrightarrow \alpha \approx \frac{\log \text{err}(\tfrac{1}{2}h) - \log \text{err}(h)}{\log 2}.$$

Obviously, the 3-stage RK-SSM given in (11.2.5) is of **order 3**.



End Problem 11-2 , 60 min.