

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 2021, Version of October 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 12-2: Damped precession of a magnetic needle

This problem deals with a dynamical system from mechanics describing the movement of a rod-like magnet in a strong magnetic field. This can be modelled by an ODE with a particular invariant that can become stiff in the case of large friction.

Related to [Lecture → Chapter 12]

We consider the initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \mathbf{a} \times \mathbf{y} + c\mathbf{y} \times (\mathbf{a} \times \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0 = [1, 1, 1]^\top, \quad (12.2.1)$$

where $c > 0$ and $\mathbf{a} \in \mathbb{R}^3$, $\|\mathbf{a}\|_2 = 1$.

Note: $\mathbf{x} \times \mathbf{y}$ denotes the **cross product** of the two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$. It is defined by

$$\mathbf{x} \times \mathbf{y} = \begin{bmatrix} x_2 y_3 - x_3 y_2 \\ x_3 y_1 - x_1 y_3 \\ x_1 y_2 - x_2 y_1 \end{bmatrix}.$$

It satisfies $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$. In Eigen, it is available as `x.cross(y)` 🐘 [EIGEN documentation](#), `#include <Eigen/Geometry>` required.

(12-2.a) 🕒 (10 min.) Show that $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}_0\|_2$ for every solution $\mathbf{y}$ of the ODE.

HINT 1 for (12-2.a): Examine the time derivative $\frac{d}{dt} \|\mathbf{y}(t)\|_2^2$ and use the product rule. ┘

SOLUTION of (12-2.a):

Since $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$ for all $\mathbf{x}, \mathbf{y}$, we have

$$(\mathbf{a} \times \mathbf{y} + c(\mathbf{y} \times (\mathbf{a} \times \mathbf{y}))) \cdot \mathbf{y} = 0 \Rightarrow D_t \|\mathbf{y}(t)\|_2^2 = 2\dot{\mathbf{y}}(t) \cdot \mathbf{y}(t) = 0 \Rightarrow \|\mathbf{y}(t)\|_2^2 = \text{const. } \forall t.$$

Thus $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}(0)\|_2 = \|\mathbf{y}_0\|_2$.



(12-2.b) 🕒 Compute the Jacobian $D\mathbf{f}(\mathbf{y}) \in \mathbb{R}^{3,3}$ and its **spectrum** (= set of eigenvalues) $\sigma(D\mathbf{f}(\mathbf{y}))$ in the stationary state $\mathbf{y} = \mathbf{a}$, for which $\mathbf{f}(\mathbf{y}) = 0$. For simplicity, you may consider only the case $\mathbf{a} = [1, 0, 0]^\top$.

SOLUTION of (12-2.b):

The right-hand side function $\mathbf{f}$ in (12.2.1) is a quadratic polynomial in the components of $\mathbf{y}$, hence smooth and defined on all of $\mathbb{R}^3$. Based on the **product rule** [Lecture → Eq. (8.5.1.17)] and appealing to the bilinearity of the cross product, we find

$$D\mathbf{f}(\mathbf{y})\mathbf{h} = \mathbf{a} \times \mathbf{h} + c\mathbf{h} \times (\mathbf{a} \times \mathbf{y}) + c\mathbf{y} \times (\mathbf{a} \times \mathbf{h}).$$

The Jacobian $D\mathbf{f}(\mathbf{y})$ is the matrix representation of this mapping $\mathbf{h} \mapsto D\mathbf{f}(\mathbf{y})$. Using the definition of cross product, a simple but tedious calculation shows that

$$D\mathbf{f}(\mathbf{y}) = \begin{bmatrix} -ca_2y_2 - ca_3y_3 & -a_3 + 2ca_1y_2 - ca_2y_1 & a_2 - ca_3y_1 + 2ca_1y_3 \\ a_3 - ca_1y_2 + 2ca_2y_1 & -ca_3y_3 - ca_1y_1 & -a_1 + 2ca_2y_3 - ca_3y_2 \\ -a_2 + 2ca_3y_1 - ca_1y_3 & a_1 - ca_2y_3 + 2ca_3y_2 & -ca_1y_1 - ca_2y_2 \end{bmatrix}.$$

Thus for $\mathbf{y} = \mathbf{a}$ we obtain

$$\begin{aligned} D\mathbf{f}(\mathbf{a}) &= \begin{bmatrix} -c(a_2^2 + a_3^2) & -a_3 + ca_1a_2 & a_2 + ca_3a_1 \\ a_3 + ca_1a_2 & -c(a_1^2 + a_3^2) & -a_1 + ca_2a_3 \\ -a_2 + ca_1a_3 & a_1 + ca_2a_3 & -c(a_1^2 + a_2^2) \end{bmatrix} \\ &= \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} + c\mathbf{a}\mathbf{a}^\top - c\|\mathbf{a}\|^2\mathbf{I}. \end{aligned}$$

The next step is an exercise in advanced linear algebra: We see that $D\mathbf{f}(\mathbf{a})\mathbf{a} = \mathbf{0}$ and on the orthogonal complement $\mathbf{a}^\perp$ the matrix acts like a rotation (scaled with $\|\mathbf{a}\|$) minus $c\|\mathbf{a}\|\mathbf{I}$. This reveals that the spectrum is given by

$$\sigma(D\mathbf{f}(\mathbf{a})) = \{0, -c\|\mathbf{a}\|_2^2 \pm i\|\mathbf{a}\|_2\} = \{0, -c \pm i\}.$$

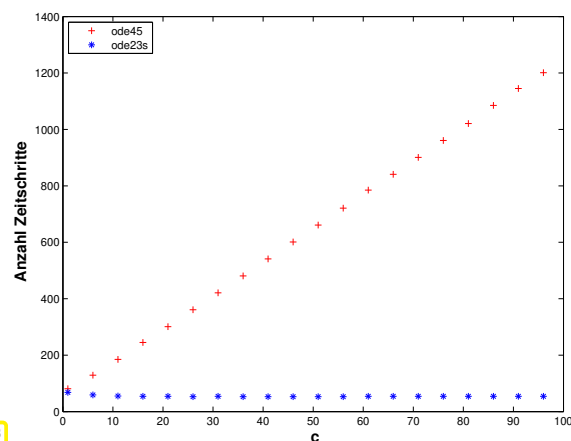
Applying the criteria [Lecture → Eq. (12.2.0.13)] and [Lecture → Eq. (12.2.0.14)] we find that at the state $\mathbf{y} = \mathbf{a}$ the ODE (12.2.1) is stiff for large $c > 0$ (see [Lecture → § 12.2.0.12]).



(12-2.c) [depends on Sub-problem (12-2.b)]

For $\mathbf{a} = [1, 0, 0]^\top$, (12.2.1) was solved with the standard explicit adaptive Runge-Kutta numerical integrator **ode45** and another A-stable semi-implicit adaptive numerical integrator up to the point $T = 10$ (same tolerances for the stepsize control). Explain the different dependence of the total number of steps from the parameter c observed in the figure.

Fig. 103



SOLUTION of (12-2.c):

In the plot, we see that, for the integrator **ode45**, the number of steps rises with c . On the other hand, the A-stable method uses roughly the same amount of steps, regardless of the value of c chosen.

As **ode45** is an explicit solver, it suffers from stability-induced step-size constraints for large $c > 0$. The A-stable integrator need not use smaller timesteps to satisfy the tolerance for big c .

This demonstrates [Lecture → Notion 12.2.0.7]: We see that the problem becomes the stiffer the larger the parameter c . This is expected from the analysis of Sub-problem (12-2.b).



(12-2.d) Formulate the non-linear equation given by the implicit mid-point method from [Lecture → Section 11.2.3] for the initial value problem (12.2.1).

SOLUTION of (12-2.d):

Recall the recursion formula for the implicit mid-point method applied to an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$,

$$\mathbf{y}_{k+1} = \mathbf{y}_k + hf\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right).$$

Its specialization for (12.2.1) reads

$$\frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h} = \mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) + c \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) \times \left(\mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right)\right).$$



(12-2.e)  By means of a C++ function

```
void tab_crossprod(void);
```

solve (12.2.1) with $\mathbf{a} = [1, 0, 0]^\top$, $c = 1$ up to $T = 10$. Use $N = 128$ uniform time steps of the implicit mid-point method. Tabulate $\|\mathbf{y}_k\|_2$ for the sequence of approximate states generated by the implicit midpoint method. What do you observe?

For this task rely on the helper class **implicit_RKIntegrator**:

```
class implicit_RKIntegrator {
public:
    implicit_RKIntegrator(
        const Eigen::MatrixXd &A,
        const Eigen::VectorXd &b);
    template <class Function, class Jacobian>
    std::vector<Eigen::VectorXd> solve(
        Function &&f, Jacobian &&Jf, double T,
        const Eigen::VectorXd &y0, unsigned int N) const;
private:
    .....
}
```

It implements a generic implicit Runge-Kutta single-step method for an autonomous ODE according to [Lecture → Def. 12.3.3.1]. The method is specified through its Butcher matrix $\mathfrak{A} \in \mathbb{R}^{s,s}$ and the weight vector $\mathbf{b} \in \mathbb{R}^s$ that are passed to the constructor as arguments $\mathbf{A}$ and $\mathbf{b}$.

The `solve()` method carries out N equidistant timesteps of the method for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ on the time interval $[0, T]$, $T > 0$, and with initial value $\mathbf{y}_0$. The right-hand side function $\mathbf{f}$ and its Jacobian $D\mathbf{f}$ are passed through the functor objects $\mathfrak{f}$ and $\mathfrak{J}$ equipped with suitable evaluation operators. The `solve()` method relies on the damped Newton iteration to solve the stage equations, see [Lecture → Rem. 12.3.3.4] and [Lecture → Rem. 12.3.3.7].

SOLUTION of (12-2.e):

We observe that norm of the approximate states generated by the single step method *is constant*, $\|\mathbf{y}_k\|_2 = \|\mathbf{y}_0\|$ for all k , and equal to $1.73205 = \|[1, 1, 1]^\top\|_2$.

Our implementation relies on an auxiliary function that calls **implicit_RKIntegrator** with parameters characteristic of the implicit midpoint method. The Butcher scheme for the implicit midpoint method can be found in [Lecture → Ex. 12.3.4.2].

C++11-code 12.2.2: Example code.

```

2  template <class Function, class Jacobian>
3  std::vector<VectorXd> solve_imp_mid(Function &f, Jacobian &&Jf, double T,
4                                     const VectorXd &y0, unsigned int N) {
5      std::vector<VectorXd> res(N + 1);
6      // TO DO (13-1.e): Construct the implicit mid-point method with the class
7      // implicit_RKIntegrator and execute the .solve() method.
8      // Return the vector containing all steps including initial and final value.
9      // START
10     // Initialize implicit RK with Butcher scheme
11     unsigned int s = 1;
12     // Initialize coefficients for the implicit midpoint method
13     MatrixXd A(s, s);
14     VectorXd b(s);
15     A << 1. / 2.;
16     b << 1.;
17     implicit_RKIntegrator RK(A, b);
18     res = RK.solve(f, Jf, T, y0, N);
19     // END
20     return res;
21 }

```

The right-hand side function for (12.2.1) and its Jacobian as computed in Sub-problem (12-2.b) are realized as lambda functions.

C++11-code 12.2.3: Example code.

```

2  void tab_crossprod(void) {
3      // TO DO (13-1.e): solve the cross-product ODE with the implicit RK method
4      // defined in solve_imp_mid. Tabulate the norms of the results at all steps.
5      // START
6      double T = 10.;
7      int N = 128;
8      // set data
9      double c = 1.;
10     Vector3d y0;
11     y0 << 1., 1., 1.;
12     Vector3d a;
13     a << 1., 0., 0.;
14     // define rhs
15     auto f = [&a, &c](const Vector3d &y) -> Vector3d {
16         return a.cross(y) + c * y.cross(a.cross(y));
17     };
18     // define Jacobian of rhs
19     auto Jf = [&a, &c](const Vector3d &y) {
20         Matrix3d temp;
21         temp << -c * (a(1) * y(1) + a(2) * y(2)),
22                 c * (2 * a(0) * y(1) - a(1) * y(0)) - a(2),
23                 a(1) + c * (2 * a(0) * y(2) - a(2) * y(0)),
24                 a(2) - c * (a(0) * y(1) - 2 * a(1) * y(0)),
25                 -c * (a(0) * y(0) + a(2) * y(2)),
26                 c * (2 * a(1) * y(2) - a(2) * y(1)) - a(0),
27                 -a(1) - c * (a(0) * y(2) - 2 * a(2) * y(0)),
28                 a(0) - c * (a(1) * y(2) - 2 * a(2) * y(1)),
29                 -c * (a(0) * y(0) + a(1) * y(1));
30         return temp;
31     };
32 }

```

```

33 std::vector<VectorXd> res_imp = solve_imp_mid(f, Jf, T, y0, N);
34
35 std::cout << "1. Implicit midpoint method" << std::endl;
36 std::cout << std::setw(10) << "t" << std::setw(15) << "norm(y(t))"
37         << std::endl;
38
39 for (int i = 0; i < N + 1; ++i) {
40     std::cout << std::setw(10) << T * i / N << std::setw(15)
41             << res_imp[i].norm() << std::endl;
42 }
43 // END

```



(12-2.f)  The so-called **linear-implicit mid-point method** can be obtained by a simple *linearization* of the (single) increment equation of the implicit mid-point method around the current state.

Equivalently, we can obtain it from the standard implicit midpoint method by carrying out a single step of the Newton iteration for the increment equations with initial guess **0**.

Give the defining equation of the linear-implicit mid-point method for the general autonomous differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$$

with smooth right-hand-side function $\mathbf{f} : \mathbb{R}^d \rightarrow \mathbb{R}^d$.

SOLUTION of (12-2.f):

The linearization of increment equations is also discussed in [Lecture → Ex. 12.4.0.1] and [Lecture → Eq. (12.4.0.2)].

The linear implicit mid-point method is obtained by expanding the right-hand side of the increment equation the implicit mid point method in its Taylor series in h around $h = 0$:

$$\mathbf{k}_1 = \mathbf{f}\left(\mathbf{y}_k + \frac{h}{2}\mathbf{k}_1\right) = \mathbf{f}(\mathbf{y}_k) + \frac{h}{2}D\mathbf{f}(\mathbf{y}_k)\mathbf{k}_1 + O(h^2)$$

and only taking the linear terms. Since the non-linear method is given by $\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1$, the linearization reads

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1^{\text{lin.}} \quad , \quad \mathbf{k}_1^{\text{lin.}} := \left(I - \frac{h}{2}D\mathbf{f}(\mathbf{y}_k)\right)^{-1} \mathbf{f}(\mathbf{y}_k) \quad .$$

A step of the method just involves the solution of a single $d \times d$ linear system of equations.



(12-2.g)  Implement the linear-implicit midpoint method in the C++ function

```

std::vector<VectorXd> solve_lin_mid(
    const Function &f,
    const Jacobian &Jf,

```

```
double T,
const Eigen::VectorXd & y0,
unsigned int N);
```

The signature of this function is the same as that of the `solve()` method of **implicit_RKIntegrator**.

Extend your implementation of `tab_crossprod()` so that it uses `solve_lin_mid()` to solve (12.2.1) with $\mathbf{a} = [1, 0, 0]^\top$, $c = 1$ up to $T = 10$ and $N = 128$. Tabulate $\|\mathbf{y}_k\|_2$ for the sequence of approximate states generated by the linear implicit midpoint method. What do you observe?

SOLUTION of (12-2.g):

See file `cross.hpp`. The sequence of the Euclidean norms of the states is not exactly constant: it starts at 1.73205 and converges to 1.72853. This is due to the approximation introduced with the linearization.

C++11-code 12.2.4:

```
2  template <class Function, class Jacobian>
3  std::vector<VectorXd> solve_lin_mid(Function &&f, Jacobian &&Jf, double T,
4                                     const VectorXd &y0, unsigned int N) {
5      std::vector<VectorXd> res(N + 1);
6      // TO DO (13-1.g): Implement the linear implicit mid-point method for
7      // an autonomous ODE  $y' = f(y)$ ,  $y(0) = y_0$ . Return the vector containing
8      // all steps including initial and final value.
9      // START
10     // Initial step size
11     double h = T / N;
12     int d = y0.size();
13     // Store initial data
14     res.push_back(y0);
15
16     // Initialize some memory to store temporary values
17     VectorXd ytemp1 = y0;
18     VectorXd ytemp2 = y0;
19     // Pointers to swap previous value
20     VectorXd *yold = &ytemp1;
21     VectorXd *ynew = &ytemp2;
22     MatrixXd eye = MatrixXd::Identity(3, 3);
23
24     // Loop over all fixed steps
25     for (unsigned int k = 0; k < N; ++k) {
26         // Compute, save and swap next step
27         *ynew = *yold + h * (eye - h / 2. * Jf(*yold)).lu().solve(f(*yold));
28         res.push_back(*ynew);
29         std::swap(yold, ynew);
30     }
31     // END
32     return res;
33 }
```

C++11-code 12.2.5:

```
2  // TO DO (13-1.g): solve the cross-product ODE with the implicit RK method
3  // defined in solve_lin_mid. Tabulate the norms of the results at all steps.
4  // START
5  std::vector<VectorXd> res_lin = solve_lin_mid(f, Jf, T, y0, N);
```

```
6
7  std::cout << "\n2. Linear implicit midpoint method" << std::endl;
8  std::cout << std::setw(10) << "t" << std::setw(15) << "norm(y(t))"
9    << std::endl;
10 for (int i = 0; i < N + 1; ++i) {
11     std::cout << std::setw(10) << T * i / N << std::setw(15)
12       << res_lin[i].norm() << std::endl;
13 }
14 // END
```



End Problem 12-2 , 10 min.