

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 11-9: Drawing Isolines of a Function

Isoline/contour plots of functions are a popular way to visualize real-valued functions or data defined on a subset of the plane. In this exercise we will use numerical integration for finding isolines.

This problem assumes familiarity with explicit adaptive Runge-Kutta single-step methods as introduced in [Lecture → Section 11.4] and [Lecture → Section 11.5]. It relies on the **ode45** integrator class from [Lecture → ??].

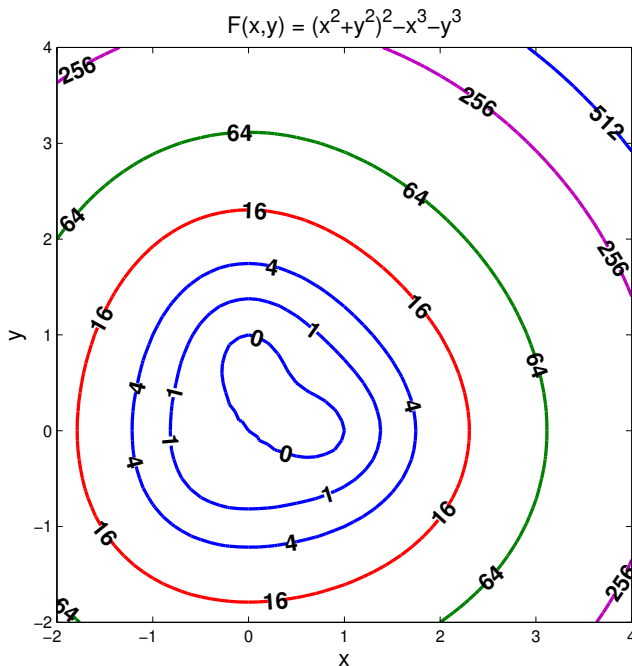


Fig. 101

An isoline is a curve in the plane $\mathbb{R}^2$ that can be defined as the set of zeros of a continuous function $F : \mathbb{R}^2 \mapsto \mathbb{R}$:

$$\mathcal{C} := \{x \in \mathbb{R}^2 : F(x) = 0\}.$$

In PYTHON (matplotlib) or MATLAB they can be plotted with the function `contour`

◁ Contour plot of the function $F(x,y) = (x^2 + y^2)^2 - x^3 - y^3$

If F is continuously differentiable, we can obtain connected components of $\mathcal{C}$ as solution curves of the initial value problems

$$\dot{\mathbf{y}}(t) = \frac{\text{grad } F(\mathbf{y}(t))^\perp}{\|\text{grad } F(\mathbf{y}(t))\|}, \quad F(\mathbf{y}(0)) = 0. \quad (11.9.1)$$

Here $^\perp$ stands for the counterclockwise rotation of a vector $\in \mathbb{R}^2$ by 90° , that is,

$$\begin{bmatrix} x \\ y \end{bmatrix}^\perp = \begin{bmatrix} -y \\ x \end{bmatrix}, \quad x, y \in \mathbb{R}. \quad (11.9.2)$$

Then (11.9.1) is a consequence of the fact that an isoline always runs perpendicular to the direction of steepest ascent of F , which is provided by $\text{grad } F$.

(11-9.a) 🕒 (5 min.)

State the ordinary differential equation (11.9.1) for the concrete case of the function

$$F : \mathbb{R}^2 \rightarrow \mathbb{R}, \quad F(x,y) := (x^2 + y^2)^2 - x^3 - y^3. \quad (11.9.3)$$

SOLUTION of (11-9.a):

By straightforward computation we find

$$\text{grad } F(x,y) = \begin{bmatrix} 4(x^2 + y^2)x - 3x^2 \\ 4(x^2 + y^2)y - 3y^2 \end{bmatrix}, \quad x, y \in \mathbb{R}. \quad (11.9.4)$$

Then we rotate the gradient counterclockwise using (11.9.2). This formula is easily explained: The linear mapping which performs this rotation is

$$x \mapsto \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} x, \quad x \in \mathbb{R}^2$$

This is easily verified by looking at the unit vectors. The first unit vector pointing in x -direction $[1, 0]^T$ is rotated, such that it points in y -direction, thus, it is mapped onto the second unit vector $[0, 1]^T$. The second unit vector $[0, 1]^T$ is further rotated in the $-x$ -direction, thus, it is mapped on the opposite of the first unit vector $[-1, 0]^T$.

Using (11.9.4), the differential equation (11.9.1) for F from (11.9.3) reads, with $\mathbf{y}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \frac{1}{\sqrt{(4(x^2 + y^2)x - 3x^2)^2 + (4(x^2 + y^2)y - 3y^2)^2}} \begin{bmatrix} -4(x^2 + y^2)y + 3y^2 \\ 4(x^2 + y^2)x - 3x^2 \end{bmatrix}. \quad (11.9.5)$$

The time argument of x and y is suppressed for the sake brevity.



(11-9.b) 🕒 (30 min.)

Use (with `rtol = 0.00001`, `atol = 1e-9`) the provided EIGEN-based integrator class **ode45** that employs an embedded explicit Runge-Kutta pair of methods to implement (in the file `contour.hpp`) a C++ function

```
template<typename GradientFunctor>
Eigen::Matrix<double, 2, Eigen::Dynamic>
computeIsolinePoints(GradientFunctor &&gradF,
                    Eigen::Vector2d y0, double T);
```

that solves an initial value problem for (11.9.1) with initial state $\mathbf{y}_0 \in \mathbb{R}^2$ passed in `y0` from initial time 0 up to final time T (given in `T`). The argument `gradF` supplies a functor object equipped with an evaluation operator

```
Eigen::Vector2d operator (Eigen::Vector2d x) const;
```

that returns $\text{grad } F(\mathbf{x})$. The function `computeIsolinePoints()` should return the sequence of computed states $\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_N$, $N \in \mathbb{N}$, in the columns of an $2 \times (N+1)$ -matrix, where N is the number of steps taken by the adaptive integrator of **ode45**.

Instructions on the use of **ode45, see [Lecture → ??].**

The class **ode45** is header-only, meaning you just include the file and use it right away (no linking required). The file `ode45.hpp` defines the class **ode45** implementing an embedded Runge-Kutta-Fehlberg method of order 4(5), see [Lecture → Section 11.5.3] and [Lecture → Ex. 11.5.3.2], with an adaptive stepsize control as presented in [Lecture → § 11.5.2.11].

1. Construct an object of **ode45** type: create an instance of the class, passing the right-hand-side function $\mathbf{f}$ as a functor object to the constructor

```
template <class StateType,
class Rhstype = std::function<StateType(const StateType &)>>
class ode45 {
public:
    ode45(const Rhstype &rhs);
    // .....
}
```

Template parameters are

- **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `*=`, `+=` and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.
- **Rhstype**: type of rhs function (automatically deduced).

The argument `rhs` must be of a functor type that provides an evaluation operator

```
StateType operator() (const StateType & vec);
```

It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **ode45.options** to configure the solver:

```
O.options.<option_you_want_to_set> = <value>;
```

Examples:

- `rtol`: relative tolerance for error control (default is $10e-6$)
- `atol`: absolute tolerance for error control (default is $10e-8$)

e.g.:

```
O.options.rtol = 10e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ($y_0 = y_0$)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t), t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

SOLUTION of (11-9.b):

The right-hand side function to be passed to the constructor of `ode45` can conveniently be realized as a lambda function.

A conversion of the return value of `ode45::solve()` into the correct return type is needed.

C++ code 11.9.6: Sub-problem (11-9.b): The function `computeIsolinePoints()`

```

2  template <typename GradientFunc>
3  Eigen::Matrix<double, 2, Eigen::Dynamic>
4  computeIsolinePoints(GradientFunc &&gradF, Eigen::Vector2d y0, double T) {
5      Matrix<double, 2, Dynamic> States;
6      // To do: (0-2.b)
7      // START
8      // Right-hand-side vector field of isoline ODE
9      auto rhs = [gradF](const Vector2d &x) {
10         Vector2d gradFx = gradF(x);
11         Vector2d out(-gradFx(1), gradFx(0)); // rotate
12         out /= gradFx.norm(); // scale
13         return out;
14     };
15     // Adaptive explicit Runge-Kutta method
16     ode45<Vector2d> integrator(rhs);
17     // Set tolerances for timestep control
18     integrator.options.rtol = 0.00001;
19     integrator.options.atol = 1e-9;
20     // Perform explicit timestepping
21     std::vector<std::pair<Vector2d, double>> sol = integrator.solve(y0, T);
22     // Convert output into requested format: points on isoline arranged into the
23     // columns of a matrix.
24     int N = sol.size() - 1;
25     States = MatrixXd::Zero(2, N + 1);
26     for (int i = 0; i <= N; i++) {
27         States.col(i) = sol[i].first;
28     }
29     // END
30     return States;
31 }

```



(11-9.c) 🕒 (10 min.) [depends on Sub-problem (11-9.a) and Sub-problem (11-9.b)]

The 0-isoline of F from (11.9.3)

$$\mathcal{C}_{\text{egg}} := \{x = (x, y) \in \mathbb{R}^2 \setminus \{0\} : F(x) := (x^2 + y^2)^2 - x^3 - y^3 = 0\}.$$

is known as “crooked egg curve”. Based on `computeIsolinePoints()` and with $y_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \in \mathcal{C}_{\text{egg}}$ and $T = 4$ implement a C++ function (in the file `contour.hpp`)

```
Eigen::Matrix<double, 2, Eigen::Dynamic> crookedEgg();
```

that returns coordinates of densely spaced points $\in \mathcal{C}_{\text{egg}}$ in the columns of the produced matrix.

HINT 1 for (11-9.c): It is recommended to call `computeIsolinePoints()` from Sub-problem (11-9.b).



SOLUTION of (11-9.c):

The normalized rotated gradient of F from (11.9.3) can be encapsulated in a lambda function.

C++ code 11.9.7: Sub-problem (11-9.c): The function `crookedEgg()`

```

2  Eigen::Matrix<double, 2, Eigen::Dynamic> crookedEgg() {
3  // crookedEggCurve will need to be reshaped to 2*(N+1).
4  Matrix<double, 2, Dynamic> crookedEggCurve;
5  // To do: (0-2.c)
6  // START
7  Vector2d y0(1., 0.);
8  double T = 4.;
9  auto gradF = [](const Vector2d &x) {
10     double x2 = x(0) * x(0), y2 = x(1) * x(1);
11     double x2y2 = 4. * (x2 + y2);
12     Vector2d grad(x2y2 * x(0) - 3. * x2, x2y2 * x(1) - 3. * y2);
13     return grad;
14 };
15 crookedEggCurve = computeIsolinePoints(gradF, y0, T);
16 // END
17 return crookedEggCurve;
18 }

```



(11-9.d) 🕒 (20 min.) [depends on Sub-problem (11-9.b)]

Again rely on the integrator class `ode45` to implement (in the file `contour.hpp`) a function

```

template<typename FFunctor>
Eigen::Matrix<double, 2, Eigen::Dynamic>
computeIsolinePointsDQ(FFunctor &&F,
                       Eigen::Vector2d y0, double T);

```

that performs the same computation as `computeIsolinePoints()` from Sub-problem (11-9.b). However, this time the argument F merely contains a functor object providing the function F itself.

To obtain an approximation of $\text{grad } F$ rely on an approximation by **symmetric difference quotients**, e.g.,

$$\frac{\partial F}{\partial x_1}(x) \approx \frac{F(x + \begin{bmatrix} h \\ 0 \end{bmatrix}) - F(x - \begin{bmatrix} h \\ 0 \end{bmatrix})}{2h} \quad \text{for some } h > 0.$$

Choose a suitable fixed $h > 0$ and explain your choice in a comment in your code.

HINT 1 for (11-9.d): Of course, you may start with copying what you implemented for `computeIsolinePoints()`. └

SOLUTION of (11-9.d):

In [Lecture → Ex. 1.5.4.7] we learned that the evaluation of difference quotients is affected by cancellation for very small span h . In order to curb cancellation it is recommended to use $h \approx \sqrt{\text{EPS}}$, where `EPS`

is the machine precision: $\text{EPS} \approx 2 \cdot 10^{-16}$ for double precision floating point arithmetic, type **double**, see [Lecture → Code 1.5.3.13].

The implementation of this subproblem just boils down to augmenting the code of `computeIsolinePoints()` by another lambda function supplying an approximation of $\text{grad } F(x)$ based on two symmetric difference quotients in x_1 and x_2 -directions.

C++ code 11.9.8: Sub-problem (11-9.b): The function `computeIsolinePointsDQ()`

```

2  template <typename FFuncor>
3  Eigen::Matrix<double, 2, Eigen::Dynamic>
4  computeIsolinePointsDQ(FFuncor &&F, Eigen::Vector2d y0, double T) {
5      // States will need to be reshaped to 2*(N+1).
6      Matrix<double, 2, Dynamic> States;
7      // To do: (0-2.d)
8      // START
9      // Get gradient of F by symmetric difference quotients with
10     // a span given by the root of the machine precision in order
11     // to curb cancellation as much as possible.
12     double epsrt = std::sqrt(std::numeric_limits<double>::epsilon());
13     auto gradF = [epsrt, F](const Vector2d &x) {
14         Vector2d grad, dx, dy;
15         dx << epsrt, 0.;
16         dy << 0., epsrt;
17         grad(0) = (F(x + dx) - F(x - dx)) / (2 * epsrt);
18         grad(1) = (F(x + dy) - F(x - dy)) / (2 * epsrt);
19         return grad;
20     };
21     // The remainder as in computeIsolinePoints()
22     auto rhs = [gradF](const Vector2d &x) {
23         Vector2d gradFx = gradF(x);
24         Vector2d out(-gradFx(1), gradFx(0)); // rotate
25         out /= gradFx.norm();                // scale
26         return out;
27     };
28
29     ode45<Vector2d> integrator(rhs);
30     integrator.options.rtol = 0.00001;
31     integrator.options.atol = 1e-9;
32
33     std::vector<std::pair<Vector2d, double>> sol = integrator.solve(y0, T);
34     int N = sol.size() - 1;
35     States = MatrixXd::Zero(2, N + 1);
36     for (int i = 0; i <= N; i++) {
37         States.col(i) = sol[i].first;
38     }
39     // END
40     return States;
41 }

```



End Problem 11-9 , 65 min.