

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 11-3: Extrapolation of evolution operators

In [Lecture → § 11.3.1.1] we have seen how discrete evolution operators can describe single-step methods for the numerical integration of ODEs. This task will study a way to combine discrete evolution operators of known order in order to build a single-step method with increased order. The method can be generalized to an **extrapolation construction** of higher-order single-step methods. These can be used for time-local stepsize control following the policy of [Lecture → § 11.5.2.11].

This problem assumes familiarity with the [Lecture → Section 11.3] and, in particular, the concept of discrete evolution, cf. [Lecture → Def. 11.3.1.5]. It also addresses the adaptive timestepping strategy presented in [Lecture → Section 11.5].

Let Ψ^h define the **discrete evolution** of an order p Runge-Kutta single step method for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subseteq \mathbb{R}^d \rightarrow \mathbb{R}^d$. We define a new discrete evolution operator:

$$\tilde{\Psi}^h := \frac{1}{1-2^p} \left(\Psi^h - 2^p \cdot (\Psi^{h/2} \circ \Psi^{h/2}) \right), \quad (11.3.1)$$

where $\circ$ denotes the composition of mappings.

(11-3.a) ☐ (10 min.) If Ψ^h belongs to the explicit/forward Euler method as introduced in [Lecture → Section 11.2.1], give the explicit formula for $\tilde{\Psi}^h$.

SOLUTION of (11-3.a):

Writing the explicit/forward Euler discrete evolution as $\Psi^h(\mathbf{y}_0) := \mathbf{y}_0 + hf(\mathbf{y}_0)$ (and $p = 1$), we obtain:

$$\begin{aligned} \tilde{\Psi}^h(\mathbf{y}_0) &= \frac{1}{1-2^p} \left(\mathbf{y}_0 + hf(\mathbf{y}_0) - 2^p \cdot \left(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0) + \frac{h}{2}f\left(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0)\right) \right) \right) \\ &= \mathbf{y}_0 + \frac{1}{1-2^p} \left(h(1-2^{p-1})f(\mathbf{y}_0) - 2^{p-1}hf\left(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0)\right) \right) \\ &= \mathbf{y}_0 + \left(hf\left(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0)\right) \right). \end{aligned}$$

This is the evolution operator of the **explicit midpoint method**, a 2-stage explicit Runge-Kutta method, see [Lecture → Eq. (11.4.0.8)].



(11-3.b) ☐ (15 min.) Implement an EIGEN-based C++ function

```
template <class DiscEvlop>
Eigen::VectorXd psitilde(DiscEvlop &&Psi, unsigned int p,
    double h, const Eigen::VectorXd & y0);
```

that returns $\tilde{\Psi}^h \mathbf{y}_0$ when given the underlying Ψ through the functor `Psi`. Objects of type `DiscEvlop` must provide an evaluation operator:

```
Eigen::VectorXd operator() (double h, const Eigen::VectorXd &y);
```

providing the result of $\Psi^h(\mathbf{y})$. For instance, suitable C++ lambda functions satisfy this requirement, see [Lecture → Section 0.3.3].

SOLUTION of (11-3.b):

The formula (11.3.1) can be implemented directly:

C++ code 11.3.2: Solution of (11-3.b).

```

2  template <class DiscEvlOp>
3  Vector psitilde(DiscEvlOp &&Psi, unsigned int p, double h, const Vector &y0) {
4      // TO DO (12-3.b): apply the evolution operator \tilde{\Psi}
5      // with step-size h to the value y0
6      // START
7      return ( Psi(h,y0) - (2<<(p-1))*Psi(h/2., Psi(h/2.,y0)) ) / (1. - (2<<(p-1)));
8      // END
9  }

```



(11-3.c) 🕒 (10 min.) Implement a C++ function

```

template <class DiscEvlOp>
std::vector<Eigen::Vector>
odeintequi(DiscEvlOp &&Psi, double T,
           const Eigen::VectorXd &y0, unsigned int N);

```

that carries out N equidistant steps of size $\tau := T/N$ of a single step method described by the discrete evolution Ψ passed through the functor object Ψ . The function should return the sequence $(y_k)_{k=0}^N$ produced by the single-step method when started with initial value y_0 (given as argument y_0).

HINT 1 for (11-3.c): You can use `std::vector<Vector>::push_back()` to add elements to the end of a `std::vector`.

SOLUTION of (11-3.c):

Requested is a straightforward implementation of [Lecture $\rightarrow$ Eq. (11.3.1.6)].

C++ code 11.3.3: Solution of Sub-problem (11-3.c).

```

2  template <class DiscEvlOp>
3  std::vector<Vector> odeintequi(DiscEvlOp &&Psi, double T, const Vector &y0,
4                                unsigned int N) {
5      std::vector<Vector> Y;
6      // TO DO (12-3.c): Compute y from time 0 to T using N equidistant time steps
7      // return a std::vector containing all steps y_0, ..., y_N
8      // START
9      double h = T / N;
10     double t = 0.;
11     Y.reserve(N+1);
12     Y.push_back(y0);
13     Vector y = y0;
14
15     while( t < T ) {
16         // append new value
17         y = Psi(h, Y.back());
18         Y.push_back(y);
19     }
20 }

```

```

18 // time stepping
19 t += std::min(T-t, h);
20 }
21 // END
22 return Y;
23 }

```



(11-3.d) (15 min.) [depends on Sub-problem (11-3.c), Sub-problem (11-3.a)]

For the particular scalar initial-value problem (IVP)

$$\dot{y} = 1 + y^2, \quad y(0) = 0, \quad (11.3.4)$$

with exact solution $y(t) = \tan(t)$, determine empirically the order of the single step method induced by $\tilde{\Psi}^h$ from (11.3.1), when Ψ arises from the explicit Euler method, recall Sub-problem (11-3.a). For that purpose, write a C++ function

double testcvpExtrapolatedEuler(void);

that monitors and tabulates the error $|y_N(1) - y_{ex}(1)|$ at final time $T = 1$ for N uniforms steps with $N = 2^q, q = 2, \dots, 12$, also returns an estimate of the rate of algebraic convergence based on linear regression.

HINT 1 for (11-3.d): The solution of the IVP Eq. (11.3.4) is $\tan(t)$.

SOLUTION of (11-3.d):

C++ code 11.3.5: Sub-problem (11-3.d): Implementation of testcvpExtrapolatedEuler()

```

2 double testcvpExtrapolatedEuler(void) {
3
4     double conv_rate;
5     double T = 1.;
6     Vector y0 = Vector::Zero(1);
7     auto f = [] (const Vector &y) -> Vector { return Vector::Ones(1) + y*y; };
8     // TO DO (12-3.d): tabulate the values of the error corresponding to
9     // \tilde{\psi}, where \psi is the explicit Euler method.
10    // return the empirical convergence rate using polyfit.
11    // Hint: first define a lambda for \psi. Then use psitilde to obtain a
12    // suitable input for odeintequi.
13
14    // START
15    auto Psi = [&f] (double h, const Vector &y0) -> Vector {
16        return y0 + h*f(y0);
17    };
18    unsigned p = 1;
19    // lambda corresponding to \tilde{\psi}
20    auto PsiTilde = [&Psi, &f, &p] (double h, const Vector &y0) -> Vector {
21        return psitilde(Psi, p, h, y0);
22    };
23
24    // exact value
25    Vector y_ex1(1);

```

```

26 y_ex1(0) = std::tan(T);
27 // values for convergence study
28 Eigen::ArrayXd err(11);
29 Eigen::ArrayXd N(11);
30
31 std::cout << "Error table for equidistant steps:" << std::endl;
32 std::cout << "N" << "\t" << "Error" << std::endl;
33 for(int i = 0; i < 11; ++i) {
34     N(i) = std::pow(2, i + 2);
35     Vector yT = odeinteqi(PsiTilde, T, y0, N(i)).back();
36     err(i) = (yT - y_ex1).norm();
37     std::cout << N(i) << "\t" << err(i) << std::endl;
38 }
39 // compute fitted rate
40 Vector coeffs = polyfit(N.log(), err.log(), 1);
41 conv_rate = -coeffs(0);
42 // END
43 return conv_rate;
44 }

```

N	$ y_N(1) - y_{ex}(1) $
4	0.0650468
8	0.0209801
16	0.00595508
32	0.00158261
64	0.000407501
128	0.000103355
256	2.60232e-05
512	6.52882e-06
1024	1.63508e-06
2048	4.09129e-07

The empiric convergence rate is 2 ($O(h^2)$), which is evidence that the single-step method $\tilde{\Psi}^h$ from (11.3.1) enjoys order 2.

Conversely, the order of the underlying forward Euler method is $p = 1$.

Hence, the construction from (11.3.1) manages to raise the order of the method by 1.



(11-3.e) (30 min.) In general, the method defined by $\tilde{\Psi}^h$ from (11.3.1) has order $p + 1$. Thus, it can be used for adaptive timestep control and prediction.

Complete the implementation of a function

```

template <class DiscEvlop>
std::pair<std::vector<double>, std::vector<Vector>>
odeintssctrl(DiscEvlop &&Psi, double T,
             const Vector &y0, double h0,
             unsigned int p, double reltol,
             double abstol, double hmin);

```

for the approximation of the solution of an IVP by means of adaptive timestepping based on Ψ and $\tilde{\Psi}$, where Ψ is passed through the argument `Psi`.

Step rejection and stepsize correction and prediction as explained in [Lecture → Section 11.5] is to be employed as in [Lecture → Code 11.5.2.18]. The argument `T` supplies the final time, `y0` the initial state, `h0` an initial stepsize, `p` the order of the discrete evolution Ψ , `reltol` and `abstol` the respective tolerances, and `hmin` a minimal stepsize that will trigger premature termination.

The function should return a vector of tuples $(t_k, \mathbf{y}_k)$, $k = 0, \dots, M$, where $M \in \mathbb{N}$ is the total number

of timesteps, t_k are the knots of the temporal mesh created by the adaptive integrator, and $(y_k)_k$ the sequence of states generated by the single-step method.

HINT 1 for (11-3.e): Just adapt [Lecture $\rightarrow$ Code 11.5.2.18] using the discrete evolutions Ψ and $\tilde{\Psi}$ from (11.3.1) instead of the generic **Psi**low and **Psi**high. ┘

SOLUTION of (11-3.e):

C++ code 11.3.6: Solution of Sub-problem (11-3.e).

```

2  template <class DiscEviOp>
3  std::pair< std::vector<double>, std::vector<Vector> > odeintssctrl(DiscEviOp&&
   Psi, double T, const Vector &y0, double h0, unsigned int p, double reltol,
   double abstol, double hmin) {
4      std::vector<double> t;
5      std::vector<Vector> Y;
6      // TO DO (12-3.e): Compute y from time 0 to T with adaptive time step.
7      // Display a warning if the tolerance cannot be met with minimum
8      // step size hmin. return a pair of vectors containing the times and
9      // the computed values.
10     // START
11     t.push_back(0.);
12     Y.push_back(y0);
13     Vector y = y0;
14
15     double h = h0;
16     while( t.back() < T && h > hmin ) {
17         // psitilde as higher order step
18         Vector y_high = psitilde(Psi, p, std::min(T-t.back(),h), y);
19         Vector y_low = Psi(std::min(T-t.back(),h),y);
20         // estimated error of current step
21         double est = (y_high - y_low).norm();
22         // effective tolerance
23         double tol = std::max(reltol*y.norm(), abstol);
24
25         h = h*std::max(0.5, std::min(2., std::pow(tol/est,1./(p + 1))));
26
27         if( est < tol ) {
28             // accept time step
29             y = y_high;
30             Y.push_back(y_high);
31             t.push_back(t.back() + std::min(T-t.back(), h));
32         }
33     }
34     if (h < hmin) {
35         std::cerr << "Warning: Failure at t = "
36             << t.back()
37             << ". Unable to meet integration tolerances without reducing the step size
   below the smallest value allowed (" << hmin << ") at time t." << std::endl;
38     }
39     // END
40     return std::make_pair(t,Y);
41 }

```

(11-3.f) (20 min.) [depends on Sub-problem (11-3.e)]

Implement a C++ function

```
void solveTangentIVP (void)
```

that relies on `odeintssctrl` from Sub-problem (11-3.e) to compute the solution of the IVP (11.3.4) up to time $T = 1.5$ with the following data: $h_0 = 1/100$, $reltol = 10e-4$, $abstol = 10e-6$, $hmin = 10e-5$. Finally, the function should use `MATPLOTLIBCPP` to plot the approximated solution and store the plot in the file `tangent.png`. Do not forget to add axis labels.

SOLUTION of (11-3.f):

C++ code 11.3.7: Implementation of `solveTangentIVP()`

```

2  void solveTangentIVP(void)
3  {
4      auto f = [] (const Vector &y) -> Vector { return Vector::Ones(1) + y*y; };
5      Vector y0 = Vector::Zero(1);
6      // TO DO (12-3.f): run the adaptive integration algorithm and plot the
7      // resulting values of y(t).
8      // Hint: you might use a loop to convert a std::vector<Vector> into a
9      // std::vector<double>, since each Vector has size 1
10
11     // START
12     double T = 1.5;
13     unsigned p = 1;
14     double h0 = 1./100.;
15
16     auto Psi = [&f] (double h, const Vector &y0) -> Vector { return y0 + h*f(y0); };
17     //run adaptive algorithm
18     std::pair<std::vector<double>, std::vector<Vector>> vec_pair
19         = odeintssctrl(Psi, T, y0, h0, p, 10e-4, 10e-6, 10e-5);
20     std::vector<double> t = vec_pair.first;
21     std::vector<Vector> Y = vec_pair.second;
22     // convert type for plot
23     std::vector<double> y(Y.size());
24     for (unsigned i = 0; i < Y.size(); ++i) {
25         y[i] = Y[i](0);
26     }
27
28     // plotting results
29     plt::figure();
30     plt::plot(t, y, "+", {"label", "approx IVP"});
31     // exact solution
32     Vector x = Vector::LinSpaced(100,0.0,T);
33     Vector exact = x.array().tan();
34     plt::plot(x, exact, {"label", "tangent"});
35     plt::legend();
36
37     plt::title("Approximate vs exact solution");
38     plt::xlabel("t");
39     plt::ylabel("y");
40     plt::grid("True");
41
42     plt::savefig("./cx_out/tangent.png");
43     // END
44 }
```

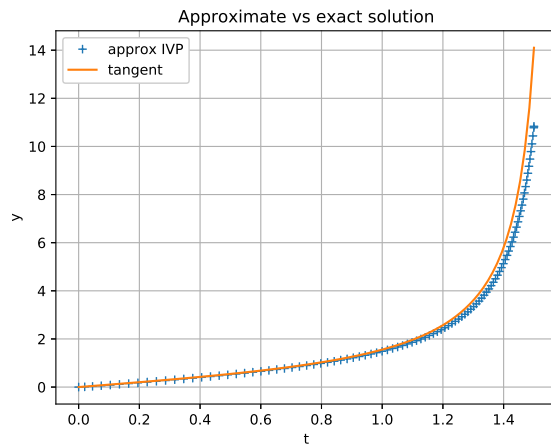


Fig. 100

- ◁ ▷ As in [Lecture → Ex. 11.5.1.4] the adaptive timestep control manages to detect the blow-up of the solution and reduces the timestep when approach the time $t_* = \pi/2$ from which the solution ceases to exist.



End Problem 11-3 , 100 min.