Numerical solution of ordinary differential equations: Euler's and Heun's method

Anne Kværnø, André Massing

Oct 3, 2021

The Python codes for this note are given in `ode.py`.

1 Introduction: Whetting your appetite

The topic of this note is the numerical solution of systems of ordinary differential equations (ODEs). This has been discussed in previous courses, see for instance the web page Differensialligninger from Mathematics 1, as well as in Part 1 of this course, where the Laplace transform was introduced as a tool to solve ODEs analytically.

Before we present the first numerical methods to solve ODEs, we quickly look at a number of examples which hopefully will will serve as test examples throughout this topic.

1.1 Scalar first order ODEs

A scalar, first-order ODE is an equation on the form

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0,$$

where $y'(t) = \frac{dy}{dt}$. The *inital condition* $y(t_0) = y_0$ is required for a unique solution.

Notice.

It is common to use the term *initial value problem (IVP)* for an ODE for which the inital value $y(t_0) = y_0$ is given, and we only are interested in the solution for $t > t_0$. In these lecture notes, only initial value problems are considered.

Example 1.1. *Population growth and decay processes.*

One of the simplest possible IVP is given by

$$y'(t) = \lambda y(t), \quad y(t_0) = y_0. \tag{1}$$

In this case, the righ-hand side f is simpy given by

$$f(t, y) := \lambda y,$$

so f is not *explicitly* depending on t.

For $\lambda > 0$ this equation can present a simple model for the growth of some population, e.g., cells, humans, animals, with unlimited resources (food, space etc.). The constant λ then corresponds to the *growth rate* of the population.

Negative $\lambda < 0$ typically appear in decaying processes, e.g., the decay of a radioactive isotopes, where λ is then simply called the *decay constant*.

The analytical solution to (1) is

$$y(t) = y_0 e^{\lambda(t-t_0)} \tag{2}$$

and will serve us at several occasions as reference solution to assess the accuracy of the numerical methods to be introduced.

Example 1.2. *Time-dependent coefficients.*

Given the ODE

$$y'(t) = -2ty(t), \quad y(0) = y_0.$$

for some given initial value y_0.

Now, the righ-hand side $f(t, y)$ is simpy given by

$$f(t, y) := -2ty(t).$$

The general solution of the ODE is

$$y(t) = Ce^{-t^2},$$

where C is a constant. To determine the constant C, we use the initial condition $y(0) = y_0$ yielding the solution

$$y(t) = y_0 e^{-t^2}.$$

1.2 Systems of ODEs

A system of m ODEs are given by

$$\begin{aligned} y_1' &= f_1(t, y_1, y_2, \ldots, y_m), & y_1(t_0) &= y_{1,0} \\ y_2' &= f_2(t, y_1, y_2, \ldots, y_m), & y_2(t_0) &= y_{2,0} \\ &\vdots & &\vdots \\ y_m' &= f_m(t, y_1, y_2, \ldots, y_m), & y_m(t_0) &= y_{m,0} \end{aligned}$$

or more compactly by

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

where we use boldface to denote vectors in $\mathbb{R}^m$,

$$\mathbf{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_m(t) \end{pmatrix}, \qquad \mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} f_1(t, y_1, y_2, \ldots, y_m), \\ f_2(t, y_1, y_2, \ldots, y_m), \\ \vdots \\ f_m(t, y_1, y_2, \ldots, y_m), \end{pmatrix}, \qquad \mathbf{y}_0 = \begin{pmatrix} y_{1,0} \\ y_{2,0} \\ \vdots \\ y_{m,0} \end{pmatrix}.$$

Example 1.3. *Lotka-Volterra equation.*

The Lotka-Volterra equation is a system of two ODEs describing the interaction between preys and predators over time. The system is given by

$$\begin{aligned} y'(t) &= \alpha y(t) - \beta y(t) z(t) \\ z'(t) &= \delta y(t) z(t) - \gamma z(t) \end{aligned}$$

where x denotes time, $y(t)$ describes the population of preys and $z(t)$ the population of predators. The parameters α, β, δ and γ depends on the populations to be modeled.

Example 1.4. *Spreading of diseases.*

Motivated by the ongoing corona virus pandemic, we consider a (simple!) model for the spreading of an infectious disease, which goes under the name SIR model.

The SIR models divides the population into three population classes, namely

S(t): number individuals **susceptible** for infection,

I(t): number **infected** individuals, capable of transmitting the disease,

R(t): number **removed** individuals who cannot be infected due death or to immunity after recovery

The model is of the spreading of a disease is based on moving individual from S to I and then to R. A short derivation can be found in [1, Ch. 4.2]. The final ODE system is given by

$$S' = -\beta SI \tag{3}$$
$$I' = \beta SI - \gamma I \tag{4}$$
$$R' = \gamma I, \tag{5}$$

where β denotes the infection rate, and γ the removal rate.

1.3 Higher order ODEs

An initial value ODE of order m is given by

$$u^{(m)} = f(t, u, u', \ldots, u^{(m-1)}), \qquad u(t_0) = u_0, \quad u'(t_0) = u_0', \quad \ldots, \quad u^{(m-1)}(t_0) = u_0^{(m-1)}.$$

Here $u^{(1)} = u'$ and $u^{(m+1)} = \frac{du^{(m)}}{dx}$, for $m > 0$.

Example 1.5. .

Van der Pol's equation is a second order differential equation, given by:

$$u^{(2)} = \mu(1 - u^2)u' - u, \qquad u(0) = u_0, \quad u'(0) = u_0'.$$

where $\mu > 0$ is some constant. As initial values $u_0 = 2$ and $u_0' = 0$ are common choices.

Later in the note we will see how such equations can be rewritten as a system of first order ODEs. Systems of higher order ODEs can be treated similarly.

2 Euler's method

Now we turn to our first numerical method, namely Euler's method, known from Mathematics 1. We quickly review two alternative derivations, namely one based on *numerical differentiation* and one on *numerical integration*.

2.1 Derivation of Euler's method

Euler's method is the simplest example of a so-called **one step method (OSM)**. Given the IVP

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0,$$

and some final time T, we want to compute to an approximation of $y(t)$ on $[t_0, T]$.

We start from t_0 and choose some (usually small) time step size τ_0 and set the new time $t_1 = t_0 + \tau_0$. The goal is to compute a value y_1 serving as approximation of $y(t_1)$.

To do so, we Taylor expand the exact (but unknown) solution $y(t_0 + \tau)$ around x_0:

$$y(t_0 + \tau) = y(t_0) + \tau y'(t_0) + \frac{1}{2}\tau^2 y''(t_0) + \cdots .$$

Assume the step size τ to be small, such that the solution is dominated by the first two terms. In that case, these can be used as the numerical approximation in the next step:

$$y(t_0 + \tau) \approx y(t_0) + \tau y'(t_0) = y_0 + \tau f(t_0, y_0)$$

giving

$$y_1 = y_0 + \tau_0 f(t_0, y_0).$$

Now we can repeat this procedure and choose the next (possibly different) time step τ_1 and compute a numerical approximation y_2 for $y(t)$ at $t_2 = t_1 + \tau_1$ by setting

$$y_2 = y_1 + \tau_1 f(t_1, y_1).$$

The idea is to repeat this procedure until we reached the final time T resulting in the following

Recipe: Euler's method.

Given a function $f(t, y)$ and an initial value (t_0, y_0).

- Set $t = t_0$.
- `while` $t < T$:

  Choose step-size τ_k

  $y_{k+1} := y_k + \tau_k f(t_k, y_k)$

  $t_{k+1} := t_k + \tau_k$

  $t := t_{k+1}$

So we can think of the Euler method as a method which approximates the continuous but unknown solution $y(t) : [t_0, T] \to \mathbb{R}$ by a discrete function $y_\Delta : \{t_0, t_1, \ldots, t_{N_t}\}$ such that $y_\Delta(t_k) := y_k \approx y(t_k)$.

How to choose τ_k? The simplest possibility is to set a maximum number of steps $N_{\max} = N_t$ and then to chose a *constant time step* $\tau = (T - t_0)/N_{\max}$ resulting in $N_{\max} + 1$ equidistributed points. Later we will also learn, how to choose the *time step adaptively*, depending on the solution's behavior.

Numerical solution between the nodes.

At first we have only an approximation of $y(t)$ at the $N_t + 1$ nodes $y_\Delta : \{t_0, t_1, \ldots, t_{N_t}\}$. If we want to evaluate the numerical solution between the nodes, a natural idea is to extend the discrete solution linearly between each pair of time nodes t_k, t_{k+1}. This is compatible with the way the numerical solution can be plotted, namely by connected each pair (t_k, y_k) and (t_{k+1}, y_{k+1}) with straight lines.

Also, in order to compute an approximation at the next point t_{k+1}, Euler's method only needs to know f, τ_k and the solution y_k at the *current* point t_k, but not at earlier points $t_{k-1}, t_{k-2}, \ldots$ Thus Euler's method is an prototype of a so-called **One Step Method (OSM)**. We will formalize this concept later.

Interpretation: Euler's method via forward difference operators.

After rearranging terms, we can also interpret the computation of an approximation $y_1 \approx y(t_1)$ as replacing the derivative $y'(t_0) = f(t_0, y_0)$ with a **forward difference operator**

$$f(t_0, y_0) = y'(t_0) \approx \frac{y(t_1) - y(t_0)}{\tau}$$

Thus *Euler's method replace the differential quotient by a difference quotient.*

Alternative derivation via numerical integration. First we recall that for a function $f : [a, b] \to \mathbb{R}$, we can approximate its integral $\int_a^b f(t)\,\mathrm{d}t$ by a *very simple* quadrature rule of the form

$$\int_a^b f(t)\,\mathrm{d}t \approx (b - a)f(a). \tag{6}$$

Notice.

Recall that the degree of exactness of the previous quadrature rule was 0.

Turning to our IVP, we know formally integrate the ODE $y'(t) = f(t, y(t))$ on the time interval $I_k = [t_k, t_{k+1}]$ and then applying the quadrature rule (6) leading to

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} y'(t)\,\mathrm{d}t = \int_{t_k}^{t_{k+1}} f(t, y(t))\,\mathrm{d}t \approx \underbrace{(t_{k+1} - t_k)}_{\tau_k} f(t_k, y(t_k))$$

Sorting terms gives us back Euler's method

$$y(t_{k+1}) \approx y(t_k) + \tau_k f(t_k, y(t_k)).$$

2.2 Implementation of Euler's method

Euler's method can be implemented in only a few lines of code:

```
def explicit_euler(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        ys.append(y + dt*f(t, y))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

Let's test Euler's method with the simple IVP given in Example 1.1.

```
t0, T = 0, 1
y0 = 1
lam = 1
Nmax = 4

# rhs of IVP
f = lambda t,y: lam*y

# Compute numerical solution using Euler
ts, ys_eul = explicit_euler(y0, t0, T, f, Nmax)

# Exact solution to compare against
```

```
y_ex = lambda t: y0*np.exp(lam*(t-t0))
ys_ex = y_ex(ts)

# Plot it
plt.plot(ts, ys_ex)
plt.plot(ts, ys_eul, 'ro-')
plt.legend(["$y_{ex}$", "y" ])
```

Plot the solution for various N_t, say $N_t = 4, 8, 16, 32$ against the exact solution given in Example 1.1.

Exercise 1: Error study for the Euler's method

We observed that the more we decrease the constant step size τ (or increase $N_{\max}$), the closer the numerical solution gets to the exact solution.

Now we ask you to quantify this. More precisely, write some code to compute the error

$$\max_{i \in \{0,\dots,N_{\max}\}} |y(t_i) - y_i|$$

for $N_{\max} = 4, 8, 16, 32, 64, 128$. How does the error reduces if you double the number of points?

We observe that for a uniform/constant time step, if we double the number of steps/reduce the time step by 2, the error reduces also by a factor of 2.

```
# Insert your code here.
```

```
def error_study(y0, t0, T, f, Nmax_list, solver, y_ex):
    max_errs = []
    for Nmax in Nmax_list:
        ts, ys = solver(y0, t0, T, f, Nmax)
        ys_ex = y_ex(ts)
        errors = ys - ys_ex
        max_errs.append(np.abs(errors).max())
        print("For Nmax = {:3}, max ||y(t_i) - y_i||= {:.3e}".format(Nmax,max_errs[-1]))
    print("The computed error reduction rates are")
    max_errs = np.array(max_errs)
    print(max_errs[:-1]/max_errs[1:])

Nmax_list = [4, 8, 16, 32, 64, 128]
error_study(y0, t0, T, f, Nmax_list, explicit_euler, y_ex)
```

3 Heun's method

Solution. Before we start to look at more exciting examples, we will derive a one step method which is more accurate then Euler's method. Note that Euler's method can be interpreted as being based on a quadrature rule with degree of exactness equal to 0. Let's try to use a better quadrature rule!

Again, we start from the *exact representation*, but this time we use the trapezoidal rule, which has degree of exactness equal to 1, yielding

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} f(t, y(t))\,\mathrm{d}t \approx \frac{\tau_k}{2}\left(f(t_{k+1}, y(t_{k+1})) + f(t_k, y(t_k))\right)$$

This suggest to consider the scheme

$$y_{k+1} - y_k = \frac{\tau_k}{2}\left(f(t_{k+1}, y_{k+1}) + f(t_k, y(k))\right)$$

But note that starting from y_k, we cannot immediately compute y_{k+1} as it appears also in the expression $f(t_{k+1}, y_{k+1})$! This is an example of an **implicit method**!

To turn this scheme into an explicit scheme, the idea is now to approximate y_{k+1} appearing in f with an explicit Euler step:

$$y_{k+1} = y_k + \frac{\tau_k}{2}\left(f\big(t_{k+1}, y_k + \tau_k f(t_k, y_k)\big) + f(t_k, y_k)\right).$$

Observe that we have now nested evaluations of f. This can be best arranged by computing the nested expression in stages, first the inner one and then the outer one. This leads to the following recipe.

Recipe: Heun's method.

Given a function $f(t, y)$ and an initial value (t_0, y_0).

- Set $t = t_0$.
- `while` $t < T$:

  Choose τ_k

  Compute stage $k_1 := f(t_k, y_k)$

  Compute stage $k_2 := f(t_k + \tau_k, y_k + \tau_k k_1)$

  $y_{k+1} := y_k + \frac{\tau_k}{2}(k_1 + k_2)$

  $t_{k+1} := t_k + \tau_k$

  $t := t_{k+1}$

The function `heun` is implemented in `ode.py`:

```
def heun(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        k1 = f(t,y)
        k2 = f(t+dt, y+dt*k1)
        ys.append(y + 0.5*dt*(k1+k2))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

Exercise 2: Comparing Heun with Euler

a) Redo the Example 1.1 with Heun, and plot both the exact solution, y_{eul} and y_{heun} for $N_t = 4, 8, 16, 32$.

```
# Insert code here.
```

```
t0, T = 0, 1
y0 = 1
lam = 1
Nmax = 4

# rhs of IVP
f = lambda t,y: lam*y

# Compute numerical solution using Euler and Heun
ts, ys_eul = explicit_euler(y0, t0, T, f, Nmax)
ts, ys_heun = heun(y0, t0, T, f, Nmax)
```

```
# Exact solution to compare against
y_ex = lambda t: y0*np.exp(lam*(t-t0))
ys_ex = y_ex(ts)

# Plot it
plt.plot(ts, ys_ex)
plt.plot(ts, ys_eul, 'ro-')
plt.plot(ts, ys_heun, 'b+-')
plt.legend(["$y_{ex}$", "$y$ Euler", "$y$ Heun" ])
```

b) Redo Exercise 1 with Heun.

```
# Insert code here.
```

```
Nmax_list = [4, 8, 16, 32, 64, 128]
error_study(y0, t0, T, f, Nmax_list, heun, y_ex)
```

We observe that for Heun's method and a uniform/constant time step, if we double of the number of steps/reduce the time step by 2, the error reduces by a factor of 4.

4 Applying Heun's and Euler's method

Solution.

Example 4.1. *The Lotka-Volterra equation revisited.*

Solve the Lotka-Volterra equation

$$y'(t) = \alpha y(t) - \beta y(t)z(t)$$
$$z'(t) = \delta y(t)z(t) - \gamma z(t)$$

In this example, use the parameters and initial values

$$\alpha = 2, \quad \beta = 1, \quad \delta = 0.5, \quad \gamma = 1, \qquad y_{1,0} = 2, \quad y_{2,0} = 0.5.$$

User Euler's method to solve the equation over the interval $[0, 20]$, and use $\tau = 0.02$. Try also other step sizes, e.g. $\tau = 0.1$ and $\tau = 0.002$.

NB! In this case, the exact solution is not known. What is known is that the solutions are periodic and positive. Is this the case here? Check for different values of τ.

```
# Reset plotting parameters
plt.rcParams.update({'figure.figsize': (12,6)})
```

```
# Define rhs
def lotka_volterra(t, y):
    # Set parameters
    alpha, beta, delta, gamma = 2, 1, 0.5, 1
    # Define rhs of ODE
    dy = np.array([alpha*y[0]-beta*y[0]*y[1],
                   delta*y[0]*y[1]-gamma*y[1]])
    return dy

t0, T = 0, 20            # Integration interval
y0 = np.array([2, 0.5])  # Initital values

# Solve the equation
tau = 0.002
```

```
Nmax = int(20/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_eul = explicit_euler(y0, t0, T, lotka_volterra, Nmax)

# Plot results
plt.plot(ts, ys_eul)
plt.xlabel('t')
plt.legend(['$y_0(t)$ - Euler', '$y_1(t)$ - Euler'],
        loc="upper right" )
```

Exercise 3: Solving the Lotka-Volterra system using Heun's method

Redo the last example with Heun's method and compare the solutions generated by Euler's and Heun's method.

4.1 Higher order ODEs

How can we numerically solve higher order ODEs using, e.g., Euler's or Heun's method?

Given the m-th order ODE

$$u^{(m)}(t) = f\big(t, u(t), u'(x), \ldots, u^{(m-1)}\big).$$

For a unique solution, we assume that the initial values

$$u(t_0), u'(t_0), u''(t_0), \ldots, u^{(m-1)}(t_0)$$

are known.

Such equations can be written as a system of first order ODEs by the following trick: Let

$$y_1(x) = u(x), \quad y_2(x) = u'(x), \quad y_3(x) = u^{(2)}(x), \quad \ldots \quad , y_m(x) = u^{(m-1)}(x)$$

such that

$$\begin{aligned} y_1' &= y_2, & y_1(a) &= u(a) \\ y_2' &= y_3, & y_2(a) &= u'(a) \\ &\vdots & &\vdots \\ y_{m-1}' &= y_m, & y_{m-1}(a) &= u^{(m-2)}(a) \\ y_m' &= f(t, y_1, y_2, \ldots, y_{m-1}, y_m), & y_m(a) &= u^{(m-1)}(a) \end{aligned}$$

which is nothing but a system of first order ODEs, and can be solved numerically exactly as before.

Exercise 4: Numerical solution of Van der Pol's equation

Recalling Example 1.5, the Van der Pol oscillator is described by the second order differential equation

$$u'' = \mu(1 - u^2)u' - u, \qquad u(0) = u_0, \quad u'(0) = u_0'.$$

It can be rewritten as a system of first order ODEs:

$$\begin{aligned} y_1' &= y_2, & y_1(0) &= u_0, \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1, & y_2(0) &= u_0'. \end{aligned}$$

```
# Define the ODE
def f(t, y):
    mu = 2
    dy = np.array([y[1],
                mu*(1-y[0]**2)*y[1]-y[0] ])
    return dy

# Set initial time, stop time and initial value
t0, T - 0, 20
y0 = np.array([2,0])

# Solve the equation using Euler and plot
tau = 0.1
Nmax = int(20/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_eul  = explicit_euler(y0, t0, T, f, Nmax)
plt.plot(ts,ys_eul);

# Solve the equation using Heun
tau = 0.1
Nmax = int(20/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_heun = heun(y0, t0, T, f, Nmax)
plt.plot(ts,ys_heun);

plt.xlabel('x')
plt.title('Van der Pols ligning')
plt.legend(['y1 - Euler','y2 - Euler', 'y1 - Heun','y2 - Heun'],loc='upper right');
```

a) Let $\mu = 2$, $u(0) = 2$ and $u'(0) = 0$ and solve the equation over the interval $[0, 20]$, using the explicit Euler and $\tau = 0.1$. Play with different step sizes, and maybe also with different values of μ.

b) Repeat the previous numerical experiment with Heun's method. Try to compare the number of steps you need to perform with Euler vs Heun to obtain visually the "same" solution. (That is, you measure the difference of the two numerical solutions in the "eyeball norm".)

```
# Insert code here.
```

References

[1] S. Linge and H. P. Langtangen. *Programming for Computations - Python*. Springer, 1 edition, 2016.