Using a computer to work with ODEs

Manuela Girotti

MATH 345 Differential Equations

Contents

1 Numerical solution of first-order equation – Euler's method

Before we dig into numerically solving differential equations using ready-to-use packages provided by MATLAB®, it is worth understanding a little bit how a computer actually calculate the solution.

The basic step in converting a differential equation into something that can be worked numerically is to replace derivatives with **algebraic approximations**. Given a function $t \mapsto y(t)$, the obvious thing to do to approximate the derivative of f is to use the standard difference quotient:

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) \approx \frac{y(t+h) - y(t)}{h}$$

where $h \in \mathbb{R}_+$ is thought of as small (in the limit $h \to 0$ we get the actual derivative, if it exists) and it is known as **time step**.

Notice that there are multiple ways in which one might work with such difference quotient; for example, here are two:

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) \approx \frac{y(t-h) - y(t)}{h}, \qquad \frac{\mathrm{d}y}{\mathrm{d}t}(t) \approx \frac{y\left(t+\frac{h}{2}\right) - y\left(t-\frac{h}{2}\right)}{h}$$

The first rule is called "forward difference", the second "backward difference" and the third "midpoint rule".

If one knows the value of y at time t_0, one can get an approximation for the value of y at time $t_0 + h$ by

$$y(t_0 + h) = h\frac{\mathrm{d}y}{\mathrm{d}t}(t_0) + y(t_0)$$

(remember the Mean Value Theorem!); then, the value at time $t_0 + 2h$ by

$$y(t_0 + 2h) = h\frac{\mathrm{d}y}{\mathrm{d}t}(t_0 + h) + y(t_0 + h)$$

and so on. It can be, of course, repeated as many times as necessary, provided one has values for the derivatives.

If y is the solution of a first-order ODE (in normal form)

$$\frac{\mathrm{d}y}{\mathrm{d}t} = F(t, y),$$

then one indeed does have the values for the derivatives:

$$\begin{aligned} y(t_0 + h) &= hF(t_0, y(t_0)) + y(t_0) \\ y(t_0 + 2h) &= hF(t_0 + h, y(t_0 + h)) + y(t_0 + h) \\ &\vdots \end{aligned}$$

(we can assume that F is sufficiently smooth such that for each pair of values (t, y) there is a definite value of F and that small changes of t, y will be accompanied by only a small variation in F).

Graphically, this means that if a point (t, y) in the (t, y)-plane is chosen, then the derivative $\frac{\mathrm{d}y}{\mathrm{d}t}$ at that point (t, y) is equal to $F(t, y)$; in other words, $\frac{\mathrm{d}y}{\mathrm{d}t} = F(t, y)$ assigns a direction to a point of the (t, y)-plane.

Suppose we have the initial condition $y(t_0) = y_0$. Take the point (t_0, y_0) in the (t, y)-plane as P_0 and draw a straight line through P_0 with slope $F(t_0, y_0)$ and let it intersect the line $t = t_0 + h$: we have found a new "starting point" $P_1 = (t_0 + h, y_1)$.

At P_1 draw a straight line with slope $F(t_0 + h, y_1)$ and let it intersect $t = t_0 + 2h$ at P_2. Continuing this way, we construct a polygonal curve $P_0P_1P_2\ldots$ which has the slope prescribed by the differential equation at the points $t_0, t_0 + h, t_0 + 2h, \ldots$. If h is sufficiently small, this turns out to be a good approximation for the actual solution curve.

The procedure can be defined iteratively as follows: given the initial conditions (t_0, y_0),

$$\begin{aligned} t_{n+1} &= t_n + h \\ y_{n+1} &= y_n + hF(t_n, y_n) \end{aligned}$$

This approximate technique is called **Euler's method** (see the MATLAB® file `Euler.m`).

Although Euler's method is simple to use, it is not particularly efficient. However, it is the prototype for a suite of more advanced numerical methods of solving a differential equation by a step-by-step process (for example, the Runge-Kutta method explained in the textbook, Section 1.16).

MATLAB® has several built-in functions that make use of these more sophisticated methods. One of the most commonly used of these is `ode45` (or `ode15s` for stiff problems), which uses adaptive step size to ensure an accurate solution, so there is no need to for the user to specify the step size h each time.

Note 1. If one types `odeexamples` at the MATLAB® prompt, you will be given a list of examples of differential equations.

Example 1 – The Hill equation.

$$y' = -\frac{Qy^p}{K + y^p}$$

is called the Hill equation and arises in enzymatic reaction. The equation is difficult to solve in general for y as an explicit function of t.

Fix parameters $Q = 1$, $K = 5$ and $p = 3.3$ and calculate/plot an approximate solution from $t = 0$ to $t = 10$ with initial condition $y(0) = 5$. The function `Euler` needs to know how to calculate $\frac{\mathrm{d}y}{\mathrm{d}t}$ for any values of t and y. We must therefore provide MATLAB® a function `Hill` that does this.

To solve the DE using Euler's method with say 10 steps, we need to enter the following commands:

```
>> t0=0; t1 = 10; y0 = 5;
>> [time1, y1] = Euler(@Hill, [t0,t1], y0, 10);
```

The first argument `@Hill` in the list of inputs for the function `Euler` is called a function handle and tells `Euler` the name of the function to use to evaluate the right-hand side of the differential equation.

Alternatively, we can use the built-in MATLAB® function `ode45`:

```
>> [time2, y2] = ode45(@Hill, [t0,t1], y0);
```

(note that we don't need to specify the number of steps to take).

We can now compare the graphs:

```
>> plot(time1, y1, 'o-', time2, y2, 'x-')
>> axis([t0 t1 0 y0])
>> legend('Euler', 'ode45')
>> xlabel('t')
>> ylabel('y(t)')
```

Clearly, the solution calculated by `ode45` is more accurate than the one calculated using `Euler`.

Example 2. Consider the separable ODE

$$y' = -\frac{ty}{2 - y}$$

As before, we set a time interval

```
>> t = linspace(0,5);
```

and initial condition

```
>> y0 = 1;
```

and we create a function `ex_separable` that will tell `Euler` or `ode45` which differential equation we want to solve numerically.

Example 3 – The skydiver. Consider a skydiver jumping from a plane. Using Newton's laws of force balance, the governing equation is

$$y'' = -g + \frac{\rho}{m}\left(y'\right)^2$$

This is a second order ODE. In order to use our solvers we first need to transform it into a first order equation

$$\begin{cases} y' = v \\ v' = -g + \frac{\rho}{m}v^2 \end{cases}$$

We are now dealing with a system of ODEs, but the numerical integration is the same (the output `y_sol` will be a structure array).

```
>> t = [0 20];
>> y0 = [500 0];
>> ysol = ode45(@parachute, t, y0);

>> figure(1)
>> subplot(2,1,1)
>> plot(ysol.x,ysol.y(1,:))
>> ylabel('height [m]');
>> xlabel('time[s]');

>> subplot(2,1,2)
>> plot(ysol.x,ysol.y(2,:))
>> ylabel('velocity [m/s]');
>> xlabel('time[s]');
```

2 Symbolic computation

Although primarily designed for numerical calculations, MATLAB® can also be implemented to perform computations using symbolic algebra. This feature can be sued to obtain analytical solutions to some differential equations.

MATLAB® symbolic representation of the derivative $\frac{\mathrm{d}y}{\mathrm{d}t}$ is simply `Dy`; similarly, $\frac{\mathrm{d}^2y}{\mathrm{d}t^2}$ is `D2y` and so on. The command for solving a differential equation is `dsolve`. The basic syntax is

```
>> y_sol = dsolve('ode', 'ic1', 'ic2', ..., 'var')
```

where `ode` is the equation to be solved, `ic1, ic2, ...` are the initial conditions and `var` is the independent variable (usually the time t).

The initial conditions may be omitted and in this case MATLAB® will attempt to find the general solution. If the independent variable is omitted, it will be assumed that it is `t`.

Example 1. To solve the separable equation

$$y' = \frac{2t}{(1+t^2)y}$$

the appropriate command is

```
>> y_sol = dsolve('Dy=2*t/((1+t^2)*y)')
```

This will produce a two-dim vector containing the two branches of the solution (notice the constant of integration `C1`).

If we have a specific initial condition, say $y(0) = -1$, then we have

```
>> y_sol = dsolve('Dy=2*t/((1+t^2)*y)', 'y(0)=1')
```

and the solution will be clearly only one of the two branch with an identified constant of integration.

In order to plot the solution of the Cauchy problem, we need to generate vectors containing actual t and y values. In order to do this, we first declare `t` as a symbolic variable

```
>> syms t
```

Then we define a vector of t values to plot (say, $t \in [0, 10]$ with steps 0.1) and a corresponding vector of y values:

```
>> t_plot = [0:0.1:10];
>> y_plot = subs(y_sol, t, t_plot);

>> plot(t_plot, y_plot)
```

Example 2 – Hill equation. If we try to solve the Hill equation symbolically, we get a warning from MATLAB® telling us that the solution can only be found in implicit form (besides the trivial solution $y(t) = 0$).

Numerical techniques are needed to obtain explicit y values. This may be done either by solving the implicit solution equation

$$t + \frac{y}{Q} - \frac{Ky^{1-p}}{Q(p-1)} + C = 0$$

by a root finding method, such as Newton's method (see the next section) for each value of t required, or by directly solving the original ODE numerically.

Example 3 – Bessel's equation. We already know the Bessel's equation

$$t^2y'' + ty' + (t^2 - \nu^2)y = 0$$

We could write this as a system of first-order equations. Alternatively, we can try to solve it directly:

```
>> y_sol = dsolve('t^2*D2y+t*Dy+(t^2-v^2)*y=0')
```

which will return the solution as

```
>> y_sol = C1*besselj(v,t) + C2*bessely(v,t)
```

here `besselj` and `bessely` are special functions called Bessel functions of first and second kind and they are our two linearly independent solutions.

MATLAB® will happily plot these solutions (as done in the previous examples) if given a value of ν and suitable initial conditions: for example, $\nu = 1$, $y(0) = 0$ and $y'(0) = 1$.

Example 4 - Heaviside function and Dirac's Delta. ODEs with Heaviside functions $H_\gamma(t)$ and Delta functions $\delta_T(t)$ as external forces can also be solved by MATLAB® (without recurring to the Laplace transform) with the use of `dirac(t)` and `heaviside(t)`.

3 Newton's method

The Newton's method is a method for finding roots of a given function: i.e. given a function $g(x)$ defined on a certain interval $[a, b]$, find the values $\eta \in [a, b]$ such that $g(\eta) = 0$.

The Newton's method principle is the same as the one we have seen for the Picard's iteration method. Recall that we used Picard's iteration method to find the solution of the Cauchy problem

$$\begin{cases} y' = F(t, y) \\ y(0) = y_0 \end{cases}$$

(and the prove that the solution exists).

The method was essentially a **fix point method**: finding a solution of the Cauchy problem above was equivalent to finding a function $y(t)$ such that

$$y(t) = y_0 + \int_0^t F(s, y(s))\mathrm{d}s \qquad \text{(Volterra integral equation);}$$

if we call $\mathcal{L}(t, z)$ the right-hand side of the equation, then we are indeed looking for a point (i.e. a solution y) such that $y = \mathcal{L}(t, y)$.

To achieve this goal, we built a sequence of "approximating solutions"

$$\begin{aligned} &y_0(t) = y_0 \\ &y_n(t) = \mathcal{L}(t, y_{n-1}(t)) \qquad \forall\ n \in \mathbb{N} \end{aligned}$$

and we proved (with some additional hypothesis on F) that $y_n(t) \to y(t)$ (there is a limit as $n \nearrow +\infty$) and $y(t)$ was indeed the solution of the Cauchy problem that we were looking for.

The general theory of a Fix Point method is the following: give a function $f(x)$ defined on some interval $[a, b]$, find a point $\eta \in [a, b]$ such that $\eta = f(\eta)$.

To find η,

1. try an initial guess $x_0 \in [a, b]$ and build a sequence $\{x_n\}$ in the following way:

$$x_n = f(x_{n-1});$$

2. prove that this sequence admits a limit: $x_n \to \eta$ as $n \nearrow +\infty$, for some $\eta \in [a, b]$;

3. prove that η is indeed a fix point: $\eta = f(\eta)$.

The following theorem guarantees that this method works!

Theorem 2 (Banach-Caccioppoli Contraction or Fix Point Theorem). *If $f, f' \in C^0([a,b])$ with $|f'(x)| < 1$ for all $x \in [a,b]$ and if the sequence $\{x_n = f(x_{n-1})\}$ is such that $x_n \in [a,b]$ for all n, then*

$$\lim_{n\to+\infty} x_n = \eta \qquad \text{and} \qquad \eta = f(\eta)$$

$\{x_n\}$ has a limit $\eta \in [a,b]$ and η is a fix point.

Note 3. The theorem only says that there exists **at least** 1 fix point, but there could be more than one. Also, this method will certainly converge to **a** fixed point, but we don't really know which one.

To fix this in application, we usually need to restrict our interval $[a,b]$ in a "clever" way, using some information about f that will allow us to identify only one fix point and exactly the one we are looking for.

4 Exercises

Exercise 1 - First order ODEs. Find the general solution of

1. $(1-t^2)y' - ty = (1-t^2)^{\frac{1}{2}}$
2. $y' = (t-4)e^4 t + ty$
3. $tww' = t^4 + w^2$, by putting $w^2 = y$

In each case, write a MATLAB® program to verify your solution by solving the differential equation numerically with an initial condition $y(0) = 1$, and plotting the numerical and the analytical solutions on the same graph over the time interval $I = (0, 10)$.

Exercise 2 - Spring-mass-dashpot system. Consider the spring-mass-dashpot system

$$my'' + cy' + ky = F(t)$$

where $m, c, k \in \mathbb{R}_+$; m is the mass of the object which is hanging on a spring, c is the damping constant and k is the spring constant. $F(t)$ is the (optional) external force acting on the system.

Using MATLAB®, find the general solution and plot the numerical solution of the differential equation in the following cases, where you can assume initial conditions $y(0) = 1$ and $y'(0) = 1$:

1. *Free vibrations:* $m = 1$, $c = 0$, $k = 2$, $F(t) = 0$; $I = (0, 10)$.
2. *Damped free vibrations:* $m = 1$, $F(t) = 0$; $I = (0, 10)$.
   (a) overdamped: $c = 7$, $k = 1$; $I = (0, 5)$.
   (b) critically damped: $c = \frac{1}{2}$, $k = \frac{1}{16}$; $I = (0, 20)$.
   (c) underdamped: $c = \frac{1}{2}$, $k = 20$; $I = (0, 10)$.

3. *Damped forced vibrations:* $m = 1$, $c = 1$, $k = 5$, $F(t) = \cos(\pi t)$; $I = (0, 10)$
4. *Forced free vibrations:* $m = 1$, $c = 0$, $k = 4$, $F(t) = 10\cos(2t)$; $I = (0, 40)$
   (notice that $\sqrt{k} = 2$ as in the argument of the external force: what happens for large times? This is the case that we discussed in class about the Tacoma bridge disaster)

You are free to variate those parameters and see how the solution looks like.

Note 4. It is possible to create only one function for the spring equation that accepts m, c, k and $F(t)$ as parameters. In this case, the syntax for numerically solving the equation is the following: the function $F(t)$ is defined by the $n \times 1$ vector F evaluated at times Ft

```
>> Ft = linspace(0,20); F = cos(pi*Ft);
>> y_sol = ode45(@(t,y)spring(t,y,m,c,k,Ft, F), t, y0)
```

within the `spring` function you need to write a line of code that interpolates F to obtain the value of the time-dependent terms at the specified time. When you then run `ode45`, you need to specify which of the many parameters are your time t and solution y as shown above.

Note 5. In case the direct plot doesn't look very well defined, you can use the following commands

```
>> tt = linspace(0,20,250);
>> y_smooth = deval(y_sol, tt);
>> plot(tt, y_smooth(1,:)); plot(tt, y_smooth(2,:));
```

(use `linspace` to generate 250 points in the interval $[0, 20]$ and evaluate the solution at these points using `deval`).

References

[1] D.S. Jones, M.J. PLank, B.D. Sleeman, *Differential Equations and Mathematical Biology*, 2nd edition, CRC Press, 2010.

[2] Notes by A.D. Lewis, "Introduction to differential equations (for smart kids)", for the course MTHE 237 (Differential Equations for Engineering Science), Queens University, Kingston (ON), Canada, 2017.