Lab II — Numerical Methods*†

Peter Brinkmann

March 5, 2010

1 Numerical methods

By now, you have seen many examples of initial value problems of the form

$$\frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0. \tag{1}$$

You have also learned some techniques that yield explicit solutions of initial value problems. But in fact for most problems there is little hope of finding an explicit solution; this is where numerical methods come in.

1.1 An example

Let's examine the initial value problem

$$\frac{dx}{dt} = x, \qquad x(0) = 1,$$

in other words $f(t, x) = x$ and $t_0 = 0, x_0 = 1$.

Let $x(t)$ be the solution. We already know how to find the exact solution $x(t) = e^t$ (how?!), but for the sake of argument, let's pretend we don't know much about it and see what we can construct numerically.

*Copyright © 2010, The Triode (iode@math.uiuc.edu). Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is available at `http://www.fsf.org/copyleft/fdl.html`.

†This document is your guide through the second lab session with Iode. It is not a homework assignment, and you do not have to turn in any work.

The first plot of Figure 1 shows the direction field of our initial value problem, with a mark at the point $(0, 1)$. The direction field line segment at this point has slope $f(0, 1) = 1$, and so in the next plot of Figure 1 we show the whole line through $(0, 1)$ with slope 1.

If we step along this line by going across 1 unit, then we have to go up 1 unit also to stay on the line. This brings us to the point $(1, 2)$, as shown in the third plot. This point ought to be reasonably near the exact solution curve, since the exact solution curve also has slope $f(0, 1) = 1$ at the point $(0, 1)$ where we started.

Now let's repeat the process. The line segment at $(1, 2)$ has slope $f(1, 2) = 2$. The fourth plot in Figure 1 shows the whole line through $(1, 2)$ with slope 2. If we step along this line by going across 1 unit, then we have to go up 2 units to stay on the line. This brings us to the point $(2, 4)$, as shown in the fifth plot. Again, this point ought to be reasonably near the exact solution curve.

Repeat the process again to arrive at the point $(3, 8)$ in the seventh plot.

We now have a list of four points $(0, 1), (1, 2), (2, 4)$, and $(3, 8)$ that we expect to lie fairly close to the points $(0, x(0)), (1, x(1)), (2, x(2))$, and $(3, x(3))$ on the exact solution curve.

The eighth plot shows those four points, connected by line segments to give an approximate solution curve. Then the ninth plot shows our approximate solution together with the exact solution, $x(t) = e^t$. While our approximate solution isn't perfect, it's not bad for a first attempt!

This method of obtaining approximate solutions is *Euler's method*. We describe it more generally below, in Section 1.3.

1.2 Numerical methods with Iode

Start the direction fields module of Iode and input the differential equation

$$\frac{dx}{dt} = x,$$

that is, `Relabel variables` so that `t` is the independent variable and `x` is the dependent variable, then `Enter differential equation` and type in `x` for `f(t,x)`.

Then `Change display parameters` so that the t-values range from 0 to 3 and the x-values range from 0 to 10. We are going to consider the initial condition $t_0 = 0, x_0 = 1$. The exact solution of $\frac{dx}{dt} = x$ satisfying this initial

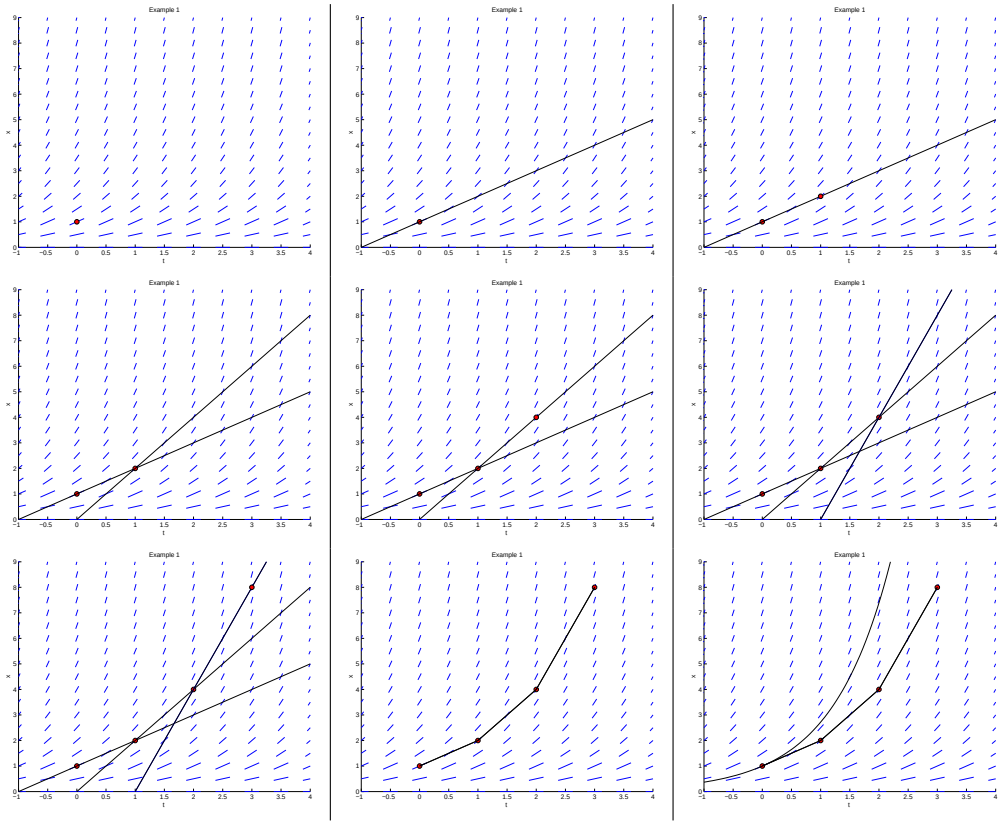

Figure 1: The plots for Section 1

condition is $x = e^t$, so plot $x = \exp(t)$ using `Plot arbitrary function` (in the `Equation` menu).

[*Alternative.* Use the `Solution method` button to choose the `Exact` solution method, then enter $t_0 = 0$ and $x_0 = 1$ and click the `Plot solution` button. This alternative will work provided your Matlab installation includes the Symbolic Toolbox.]

Let's see how Euler's method performs, compared to the exact solution. Change the `Solution method` to `Euler`, and enter step size 1. Now plot a numerical solution by clicking on the point $(0, 1)$ (or else enter $t_0 = 0$ and $x_0 = 1$ and click the `Plot solution` button). Check that your resulting Euler plot looks similar to the ninth plot in Figure 1.

How does the Euler numerical solution compare to the exact solution?

Jot down your observations on the last page of this lab!

e.g. At which t-values does the graph of the numerical solution have corners? Does the numerical solution graph get closer to the exact solution as t increases, or does it get further away?

Reduce the step size to 0.5 and plot another Euler numerical solution with the same initial conditions. How does the solution change, now that you have reduced the step size? Repeat this with step sizes 0.2 and 0.1. Jot down your observations on the last page.

Finally, change to the `Runge-Kutta` solution method with step size 0.5. Runge–Kutta is a more sophisticated numerical method than Euler. Plot a Runge–Kutta numerical solution with the same initial conditions. (First choose a different color for this plot, to make it easier to follow.) What do you see? How does the accuracy of Runge–Kutta with step size 0.5 compare to Euler's method with step size 0.5? to Euler with step size 0.1?

Again jot down your observations on the last page.

1.3 Euler's method

Take a *uniformly spaced* sequence $t_0, t_1, t_2, \ldots$, which means there is a *step size* $h > 0$ such that $t_0 + h = t_1$, and $t_1 + h = t_2$, and so on. (In Section 1, the step size was 1.) Let $x(t)$ be a solution of Equation 1 satisfying the initial condition $x(t_0) = x_0$. We want to compute a sequence of x-values $x_1, \ldots, x_n$ such that x_k is reasonably close to $x(t_k)$, for each $k = 0, 1, \ldots, n$. Here's how we do it, based on our example in Section 1...

The initial point (t_0, x_0) lies on the graph of $x(t)$, and the graph has slope $f(t_0, x_0)$ at that point. So if we just follow the tangent line and step across by h and up by $hf(t_0, x_0)$, then the new point should still be fairly close to the graph. That is, if we step across and up to the point

$$t_1 = t_0 + h$$
$$x_1 = x_0 + h\, f(t_0, x_0)$$

then the point (t_1, x_1) should be fairly close to the graph of the solution.

[*Check:* The slope of the line from (t_0, x_0) to (t_1, x_1) equals

$$\frac{x_1 - x_0}{t_1 - t_0} = \frac{h\, f(t_0, x_0)}{h} = f(t_0, x_0),$$

which is exactly the slope of the direction field line segment at (t_0, x_0).]

Repeating this reasoning, we obtain (t_2, x_2) by letting

$$t_2 = t_1 + h$$
$$x_2 = x_1 + h\, f(t_1, x_1)$$

The general rule of this type is the *Euler update formula*:

$$\boxed{x_{i+1} = x_i + h\, f(t_i, x_i)}, \tag{2}$$

which tells us how to compute x_1 once we know x_0, and then how to compute x_2 once we know x_1, and so on.

Graphically, the Euler points (t_i, x_i) are the "corners" in the Euler approximate solution.

1.4 Approximation error in Euler's method

To deepen your understanding of the Euler method, enter the following differential equation into Iode:

$$\frac{dx}{dt} = \sin(x - t^2/4 - 3)$$

with display parameter $-3 < t < 3$ and $-3 < x < 3$. Plot a numerical solution using the Euler method in Iode, for step size 1 and initial condition $x(-3) = -1$. Also plot the Runge–Kutta solution with step size 0.1, which we will regard as being the true solution curve (because the error is known to be very small when using the Runge–Kutta method).

Now answer the questions at the bottom of the last page.

```
>> tc=0:0.2:1
tc =

   0.00000   0.20000   0.40000   0.60000   0.80000   1.00000

>> xc=sin(pi*tc)
xc =

   0.00000   0.58779   0.95106   0.95106   0.58779   0.00000

>> plot(tc,xc)
>> length(tc)
ans = 6
>> tc(2)
ans = 0.20000
>> tc(3)
ans = 0.40000
```

Figure 2: Representing and plotting functions with Matlab

2 Programming Euler's method

In Project II you will implement your own numerical method. So you need to understand how Matlab and Octave represent numerical solutions internally. The code in Figure 2 illustrates some of the main points, and you can type the code yourself at the prompt in the command window of Matlab or Octave. (But you will need to quit out of Iode before doing so, or else the plots might not show up correctly.)

The first command, tc=0:0.2:1, creates a vector of t-values, ranging from 0 to 1 with step size 0.2, i.e., it creates the vector $(0, 0.2, 0.4, 0.6, 0.8, 1)$. The second command, xc=**sin**(**pi**∗tc), creates a vector of x-values by evaluating the function $\sin(\pi t)$ at all the entries of the vector tc. The third command, **plot**(tc,xc), interprets tc as a list of coordinates on the horizontal axis and xc as a list of corresponding coordinates on the vertical axis. It plots all six points (t, x) and connects adjacent points with straight line segments. The resulting graph looks like a rough approximation of a part of a sine curve. We can obtain a better picture by decreasing the step size: if you replace the first line by tc=0:0.05:1; and repeat the remaining two steps, then you'll see

For each i, compute:

$$h = t_{i+1} - t_i$$
$$k_1 = f(t_i, x_i)$$
$$x_{i+1} = x_i + h\,k_1, \quad \text{and then}$$

append x_{i+1} to the vector of x-values

```
1  function xc=euler(fs,x0,tc);
2      x=x0;
3      xc=[x0];
4      for i=1:(length(tc)-1)
5          h=tc(i+1)-tc(i);
6          k1=feval(fs,tc(i),x);
7          x=x+h*k1;
8          xc=[xc,x];
9      end;
```

Figure 3: Euler's method and Iode's implementation of it

a plot that looks very much like a piece of a sine curve.[1]

The last few commands in Figure 2 show how to access certain information about the vector tc, such as its length (i.e., the number of entries), and the values of its individual entries, numbered from 1 to 6.

Remark 1. The above way of representing a function numerically as a vector of t-values and a vector of x-values fits beautifully into our discussion of numerical methods: the Euler method takes an initial value x_0 and a vector $t_0, t_1, \ldots, t_n$ of t-values, and computes a vector $x_0, x_1, \ldots, x_n$ of x-values.

Now we are ready to inspect Iode's implementation of Euler's method. Use the `Open` menu item in the Matlab main window (not the Iode window!) to open the file `euler.m`.

Figure 3 shows the contents of `euler.m`, without the comment lines (which begin with a percentage sign). We'll go through it line by line.

Line 1 defines a new function called `euler`. When this function is called, it expects to receive three parameters; the parameter fs represents the function $f(t, x)$ from our differential equation, x0 is the initial x-value, and tc is a vector of t-coordinates, like we have seen before (in particular the first entry of `tc` is t_0). Line 1 also indicates that this function will return a value in the variable xc, which will turn out to be the desired vector of x-coordinates computed by Euler's method.

Line 2 initializes the value of the variable x to be x_0. The variable x always contains our current numerical approximation x_i. Line 3 creates the

[1] If you end a line with a semicolon, Matlab will not print the result of the operation. In particular it is wise to use a semicolon when the result is a vector with many entries, like in this case, and you don't want to clutter up the screen with lots of numbers.

vector xc that will contain our numerical approximations; initially, it only contains the value x0. In particular, the first entry of xc is x0, i.e., we have tc(1)=t0 and xc(1)=x0.

Line 4 is the beginning of a loop that lets the variable i range over all numbers from 1 to the length of the vector tc minus one. The body of this loop, lines 5–8, constitutes the Euler update step and will be executed for each value of i. Line 5 computes the step size h by computing the difference between the $(i+1)$-st entry of tc and the i-th entry of tc.[2] The expression tc(i) stands for the i-th entry of tc, which we regard as the current t-value.

Now, the variable x contains the numerical approximation x_i of the solution at the current point t_i, and Line 6 computes the slope of the solution at this point by evaluating the function given by fs at the point with coordinates tc(i) and x. The variable k1 contains the result of this slope computation. Then Line 7 computes the Euler approximate solution value at the next point tc(i+1). Line 8 appends this value to the vector of x-values, and that's it!

Remark 2. In Iode Project II, when you write your own numerical routine for the Improved Euler Method, you can keep the framework of `euler.m`. You only need to change Line 1 in Figure 3, and then Lines 6–7 which compute the update formula.

Remark 3 (Optional: for advanced users)**.** The module `euler.m` can be used without the Iode interface. Figure 4 shows an example of this. The initial value problem in this example is

$$\frac{dx}{dt} = t\sin(x), \quad x(0) = 1,$$

and the Matlab code in Figure 4 computes and plots an Euler solution with step size 0.1 on the interval $[0, 2]$ of t-values.

The first line in the figure creates a new function `fn`, with the first parameter being a string containing the expression for the function (`'t*sin(x)'`), and the remaining two parameters being the names of the independent and dependent variables.[3]

[2] We have only discussed Euler's method for uniformly spaced t-values, and so you may be wondering why the program computes the value of h at every step. The answer is that we want our numerical routine to work even if the vector tc is *not* uniformly spaced.

[3] Some versions of Octave do not know inline functions, but the distribution of Iode comes with an approximation of Matlab's inline feature that works well enough for the purposes of Iode.

```
>> fn=inline('t*sin(x)','t','x');
>> tc=0:0.1:2;
>> xc=euler(fn,1,tc);
>> plot(tc,xc);
```

Figure 4: An example of `euler.m` in action on its own. To try this, first make sure you started Matlab or Octave from your Iode directory, so that it can find the file `euler.m`.

Mathematical expressions in Matlab, Octave

For simple expressions, we use the usual keyboard characters:

`2*x`	means $2x$,
`(x^3-1)/6`	means $(x^3 - 1)/6$,
`pi`	means π.

Built-in functions.

`exp(x)`	exponential, e^x		
`log(x)`	natural logarithm, $\ln x$		
`log10(x)`	base 10 logarithm, $\log_{10} x$		
`abs(x)`	absolute value, $\lvert x \rvert$		
`sqrt(x)`	square root, $\sqrt{x}$		
`sign(x)`	signum function, which equals $\begin{cases} +1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$		
`sin(x)`		`sinh(x)`	
`cos(x)`	trigonometric	`cosh(x)`	hyperbolic
`tan(x)`	functions	`tanh(x)`	trigonometric
`cot(x)`	(x in radians)	`coth(x)`	functions

Example 1.

`sin(exp(y))^4`	means $\sin^4(e^y)$,
`acos(exp(1)^(-1))`	means $\arccos(e^{-1})$.

A Observations (not for handing in)

- Notes for Euler's method, step size 1

- Notes for Euler's method, step size 0.5

- Notes for Euler's method, step size 0.2

- Notes for Euler's method, step size 0.1

- Notes for Runge–Kutta, step size 0.5

- Notes for $dx/dt = \sin(x - t^2/4 - 3)$ (see Section 1.4):

  on your screen plot of the direction field, locate the **Euler points** $(t_0, x_0), (t_1, x_1), (t_2, x_2), \ldots$. Graphically, these are the "corner" points in the Euler approximate solution.

  which of these Euler points lie on the true solution curve?

  is the slope of the Euler segment starting at $t = t_i$ the same as the slope of the true solution curve at $t = t_i$? why or why not?

  is the Euler approximate solution always below the true solution curve?

  does the error (height difference) between the Euler approximate solution and the true solution get bigger with every step to the right?

  Contrast that observation with what happened in Figure 1 for the differential equation $\frac{dx}{dt} = x$.