

The computer labs provide computational experience related to the analytic theory presented in the lectures. The main tool for these exercises is a programming language called Julia designed for the implementation of numerical algorithms that combines the compiled efficiency of C and Fortran with the interactive and notational convenience of MATLAB and Python.

In science and engineering an important goal is to become a skilled practitioner by doing it yourself. To this end the computers in the lab have been provisioned with a Linux programming environment similar to what is deployed on the university high-performance cluster, all other supercomputers worldwide and for most cloud computing. To access Linux please restart the computer using the USB network boot key for this class.

Rather than using the lab equipment it is also possible to freely install Julia on your personal laptop. While using your own computer goes along well with doing it yourself, I will unfortunately be unable to help with any technical problems that might crop up in that case. Even so, I'd recommend trying to install Julia at home, if only to avoid coming in after hours to complete the homework. You may also use your laptop in the lab.

Note that it is possible to forgo Julia and perform all your computations using a different programming language. Although I would be happy to grade assignments completed using such alternatives, my opinion is Julia makes numerical methods much easier than a general-purpose programming language. I am also able to provide more help with Julia.

Symbolic Techniques

Our lab exercises start with solving an ODE ordinary differential equation. Suppose we seek to find $y(t)$ such that

$$y' + 3y = \sin t \quad \text{and} \quad y(0) = 7.$$

Cauchy's variation of parameters, in this case also known as the integrating factor method, tells us to multiply by $\mu(t) = \exp(3t)$ to obtain

$$(\mu y)' = \mu y' + \mu' y = \mu \sin t.$$

Integrating both sides over $[0, t]$ as

$$\int_0^t (\mu(s)y(s))' ds = \int_0^t \mu(s) \sin s ds$$

then yields

$$\mu(t)y(t) - \mu(0)y(0) = \int_0^t \mu(s) \sin(s) ds$$

or that

$$y(t) = \exp(-3t) \left\{ 7 + \int_0^t \exp(3s) \sin(s) ds \right\}.$$

The integral on the right may be solved using integration by parts, a complex line integral or a CAS computer algebra system. Although the CAS built into Julia is not as advanced as programs such as Maple or Mathematica, this drawback is outweighed by the fact that the numerical performance of Julia is far superior. Since most of our work will be of a numeric nature, it is convenient to also do the algebra with Julia. To this end, create a working directory, change to that directory, start Julia and then load the `Symbolics` and `SymbolicNumericIntegration` packages.

```
$ mkdir lab01
$ cd lab01
$ julia
```

```

      _       _
     (_)     | |  _(_) | |
    _ _     | | | | | | | |
   | | | | | | | | | | | |
   | | | | | | | | | | | |
  _/ | \_ ' | | | \_ ' | |
 |_/

Documentation: https://docs.julialang.org
Type "?" for help, "]"? for Pkg help.
Version 1.6.7 (2022-07-19)
Official https://julialang.org/ release

```

```
julia> using Symbolics, SymbolicNumericIntegration
```

Note that these packages have been preinstalled on the lab computers to save time and storage space. If you are using your own personal computer or notebook you will have to download the packages yourself.

After the package is loaded, you may then solve the ODE analytically by entering the following sequence of commands:

```
julia> @variables s
```

```
1-element Vector{Num}:

```

```
s
```

```
julia> mu(t)=exp(3*t)
mu (generic function with 1 method)
```

```
julia> I1=integrate(mu(s)*sin(s))[1]
(3//10)*sin(s)*exp(3s) - (1//10)*cos(s)*exp(3s)
```

```
julia> F1=eval(build_function(I1,s))
#11 (generic function with 1 method)
```

```
julia> y(t)=exp(-3*t)*(7+F1(t)-F1(0))
y (generic function with 1 method)
```

```
julia> y(s)
(7.1 + (3//10)*sin(s)*exp(3s) - (1//10)*cos(s)*exp(3s))*exp(-3s)
```

Euler's Method

As not all ordinary differential equations are solvable using exact symbolic techniques, this course focuses on numeric approximations. Recall the simple scheme to approximate $y(t)$ on the interval $[t_0, T]$ given by

$$y_{n+1} = y_n + hf(t_n, y_n) \quad \text{where} \quad t_n = t_0 + hn \quad \text{and} \quad h = (T - t_0)/N.$$

Here N specifies how many subintervals in which to divide $[t_0, T]$ when performing the approximation and h is the length of those subintervals.

Theoretically, we know that y_n approximates $y(t_n)$ to order $\mathcal{O}(h)$. This means there is a constant c such that the total error

$$E_N = \max \{ \|y_n - y(t_n)\| : n = 0, \dots, N \} \leq ch \quad \text{as} \quad h \rightarrow 0.$$

The goal in this lab is to compute the approximations y_n for different values of N and verify the rate of convergence. Along the way, we shall also estimate the value of c for the specific ODE under consideration.

Computing Techniques

We used Julia as an interactive calculator when solving the ODE using symbolic techniques. One of the advantages of the just-in-time compilation behind the REPL read-evaluate-print loop is that it's also possible to explore numeric algorithms interactively. Once a suitable algorithm has been developed, one can then store the needed sequence of commands in a file. That file is called a program and contains what is referred to as code.

In the practice of scientific computing, once a program is finished it is typically uploaded to a supercomputer or computing cluster and used to solve larger problems. In this class the program will be uploaded for grading along with a copy of the output when solving a smaller problem.

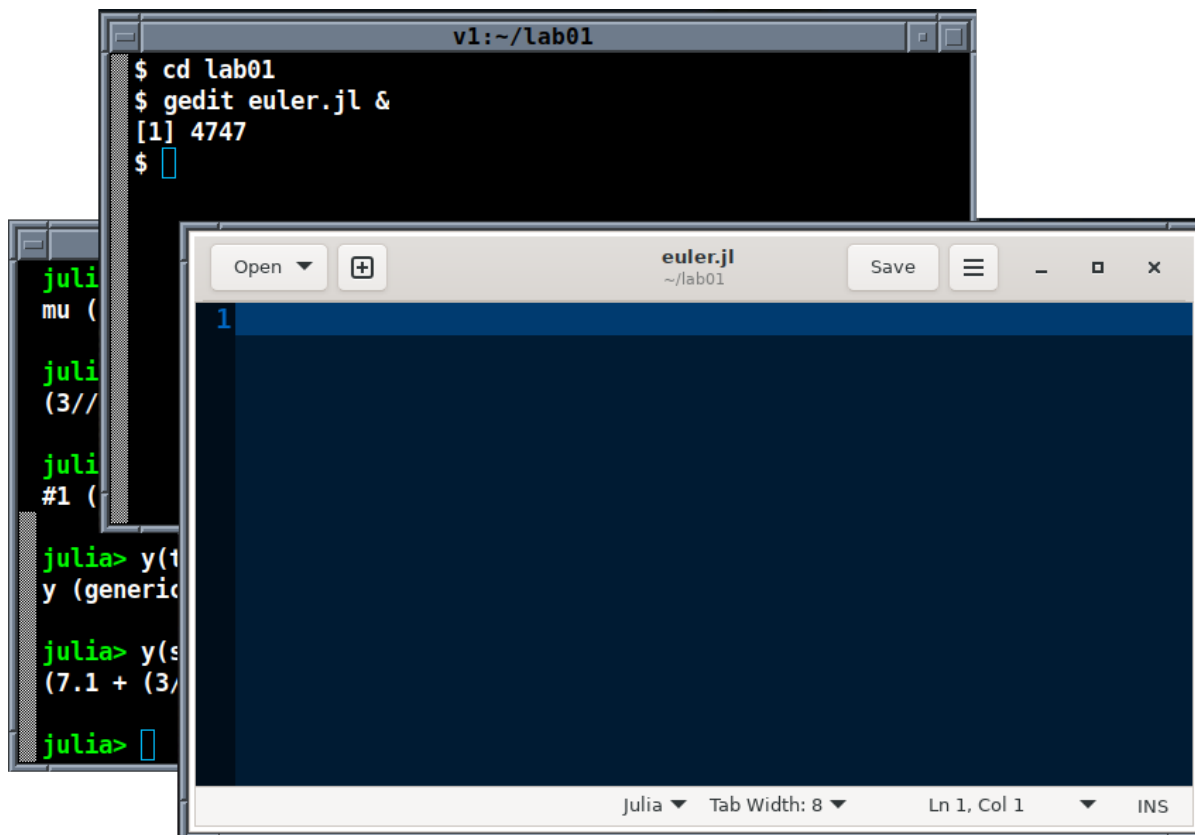
The stages of experimentation, development and finally running large-scale numeric simulations often involve the further task of translating code from one programming language to another as well as optimizing that code for use on a parallel machine. For example, one might experiment using MATLAB or Python and only later convert that code (or parts of that code) to C or Fortran. Although this may seem like too much advertising, I find it revolutionary that desktop computers now have the power to host interactive environments such as Julia with just-in-time compiled efficiency that scales to real-world problem sizes.

As we begin implementing Euler's method in Julia, it should be emphasized that no matter how powerful our software tools, the main advantage of Euler's method is simplicity not numerical efficiency. In particular, the low-order $\mathcal{O}(h)$ convergence (as well as poor stability properties) makes Euler's method unsuitable for many problems.

At this point the Julia REPL that was used to find the exact solution of the ODE should already be running in one window on the desktop. Interactive computing is great for experimentation and visualization, but not so much when performing a sequence of long-running numerical calculations.

Since we shall be writing a program, open an editor window with an application such as `gedit` or `pluma`. On the lab computers this can be done by first creating another terminal window and then typing `cd lab01` followed by `gedit euler.jl &` at the prompt. Note carefully the `&` typed at the end of the `gedit` command above. This is necessary so the terminal window will continue to accept commands while the editor is running.

Not typing the `&` can lead to confusion. If this happens close the editor window and try again. The relevant windows on the screen might look like



We shall switch back and forth between the editor and the REPL while writing our program. Many Linux window managers—including the one in the lab—make extensive use of mouse click or drag operations performed while pressing the `<alt>` key. As the resulting user interface is slightly different than Microsoft or Apple, here are a few hints to get started:

- The keyboard types into the window containing the mouse pointer whether that window is on top or not.
- To raise a window hold the `<alt>` key and then left click with the mouse anywhere in the window you want to raise.
- To lower a window hold the `<alt>` key and then right click with the mouse anywhere in the window you want to raise.
- To move a window hold the `<alt>` key and drag with either the right or left mouse button anywhere in the window you want to move.

Note there are many window-manager choices on Linux. Not all work the same, many are customizable and some are specially intended to be more similar to either Microsoft or Apple.

A Numerical Approximation

Our numerical work begins by identifying the function $f(t, y)$ in the differential equation

$$y' + 3y = \sin t$$

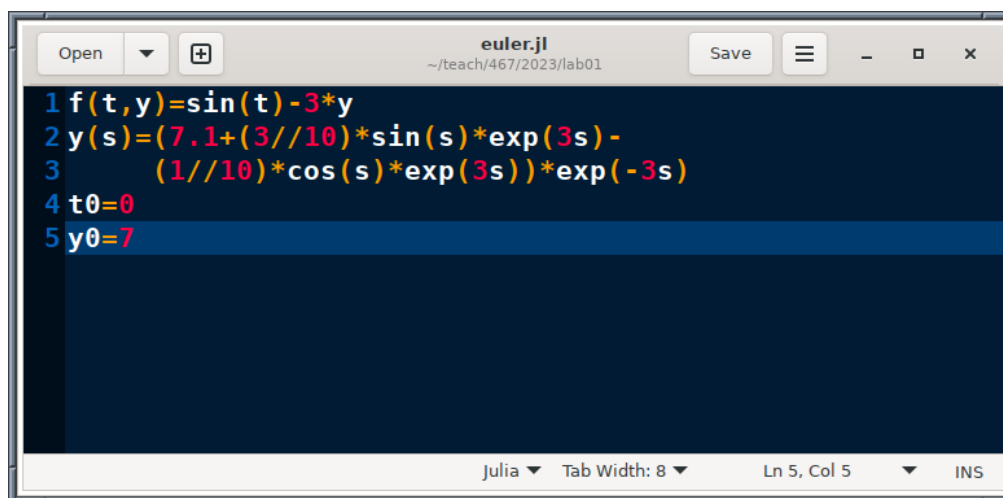
previously solved symbolically. Since $y' = f(t, y)$ subtracting $3y$ from both sides yields that

$$f(t, y) = \sin t - 3y.$$

The Julia commands to define such a function are

```
f(t,y)=sin(t)-3*y
```

Enter this line the editor along with the definition of the exact solution found earlier and the initial condition. Save the file using $\langle \text{ctrl} \rangle$ -S or the mouse menu. The editor window should now look like



```
euler.jl
~/teach/467/2023/lab01
Save
1 f(t,y)=sin(t)-3*y
2 y(s)=(7.1+(3//10)*sin(s)*exp(3s)-
3   (1//10)*cos(s)*exp(3s))*exp(-3s)
4 t0=0
5 y0=7
julia Tab Width: 8 Ln 5, Col 5 INS
```

A star before the name `euler.jl` means means you haven't saved the file. Make sure to save the file each time before loading it into Julia.

Now, switch to Julia and type `include("euler.jl")` in the REPL. If all goes well, the output should look like

```
julia> include("euler.jl")
7
```

Note the 7 which is returned represents the result of the final line `y0=7` in the program. The `print`, `println` and `display` functions can be used to output intermediate results while the program is running.

Let's use eight steps of Euler's method to approximate the solution $y(t)$ on the interval $[0, 1]$. After defining T , N and h , create two array tn and yn to hold the results of the calculation. Initialize the arrays and then make a loop to compute tn and yn . The code in the editor should now look like

```

1 f(t,y)=sin(t)-3*y
2 y(s)=(7.1+(3//10)*sin(s)*exp(3s)-
3     (1//10)*cos(s)*exp(3s))*exp(-3s)
4 t0=0
5 y0=7
6
7 T=1
8 N=8
9 h=(T-t0)/N
10 tn=zeros(N+1)
11 yn=zeros(N+1)
12
13 yn[1]=y0
14 tn[1]=t0
15 for n=1:N
16     tn[n+1]=t0+h*n
17     yn[n+1]=yn[n]+h*f(tn[n],yn[n])
18 end

```

Run the new code by typing `include("euler.jl")` in the REPL. If there are errors, first check that the file in the editor has been saved. After this, it may help to cut and paste lines from the editor into the reply using the mouse to debug them. Cut and paste is a bit weird.

- In the terminal use `<ctrl>-<shift>-c` to copy and `<ctrl>-<shift>-v` to paste.
- In the editor `<ctrl>-c` to copy and `<ctrl>-v` to paste.

The inconsistency is because `<ctrl>-c` traditionally means stop the running program in the terminal window and still does.

Now, suppose the program had a typo so line 10 read as `tn=zerops(N+1)` instead. The result in the REPL would be a stack trace that looks like

```
julia> include("euler.jl")
```

```
ERROR: LoadError: UndefVarError: zerops not defined
```

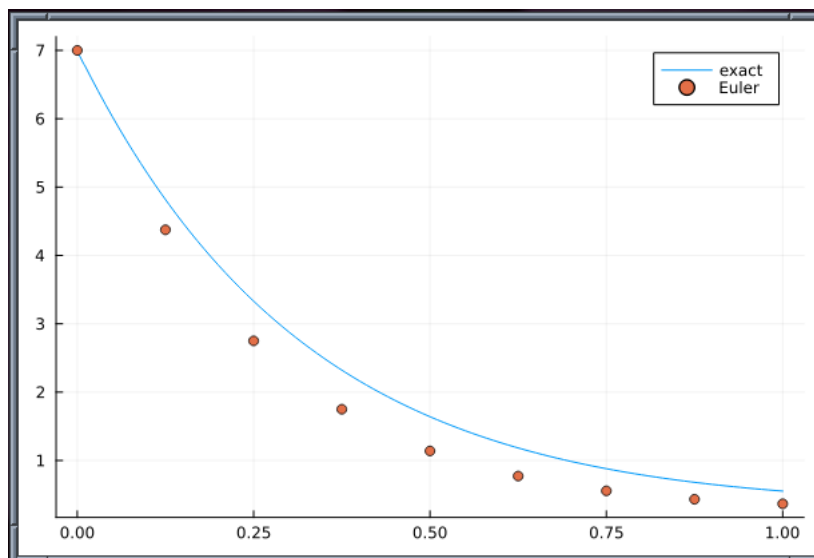
```
Stacktrace:
```

```
[1] top-level scope
      @ ~/teach/467/2023/lab01/euler.jl:10
[2] include(fname::String)
      @ Base.MainInclude ./client.jl:444
[3] top-level scope
      @ REPL[42]:1
```

```
in expression starting at /home/ejolson/teach/467/2023/lab01/euler.jl:10
```

That's a lot of output for a single-character typo; however, upon closer inspection the message clearly indicates what's wrong. Because of the misspelling `zerops` is not defined. There is no such function. The top level of the stack trace indicates the error occurred in on line 10 in `euler.jl` and indeed that is where we put the typo in this example.

At this point it is useful to visualize the solution and its approximation. This can be done interactively using the `Plots` package. To obtain



enter the following into the REPL:

```
julia> using Plots
```

```
julia> ts=0:0.01:1
```



```
0.0:0.01:1.0
```

```
julia> plot(ts,y.(ts),label="exact")
```

```
julia> scatter!(tn,yn,label="Euler")
```

Note that `ts` is an iterator that is used for plotting the exact solution. The graph can be saved to a PDF by typing `savefig("approx.pdf")` that is suitable for inclusion in word-processing documents as well as uploading as part of the work for this lab. The output should look like

```
julia> savefig("approx.pdf")
```

and the graph displayed on the screen will disappear.

That completes the first half of this lab project. To finish we now verify the order of convergence and estimate the value of c appearing in the error bound numerically.

Study of Convergence

So far we've computed an approximation of the differential equation using eight time steps and visually compared that approximation to the exact solution. Next add an additional loop to compute the error E_N for different values of N and then estimate the value of c such that $E_N \leq ch$.

After some changes we arrive at

```
1 f(t,y)=sin(t)-3*y
2 y(s)=(7.1+(3//10)*sin(s)*exp(3s)-
3     (1//10)*cos(s)*exp(3s))*exp(-3s)
4 t0=0
5 y0=7
6
7 T=1
8 J=10
9 EN=zeros(J)
10 HN=zeros(J)
11 for j=1:J
```

```

12     N=2^(j+2)
13     h=(T-t0)/N
14     tn=zeros(N+1)
15     yn=zeros(N+1)
16
17     yn[1]=y0
18     tn[1]=t0
19     for n=1:N
20         tn[n+1]=t0+h*n
21         yn[n+1]=yn[n]+h*f(tn[n],yn[n])
22     end
23     HN[j]=h
24     EN[j]=maximum(abs.(yn-y.(tn)))
25 end
26
27 scatter(HN,EN,scale=:log10,
28         legend=:topleft,label="EN")
29 plot!(HN,4*HN,label="4*h")

```

Note the additional loop starting at line 11 along with the computation of the error in line 24. The error E_N versus h is automatically plotted in lines 27 and 29 using a logarithmic scale. This is because we have taken N to be powers of 2 in order to cover a wide range of step sizes. Now type `savefig("error.pdf")` in the REPL to save the error graph.

The value of c was found to be 4 using guess and check. A better way would be to perform a least squares fit. Feel free to solve for c using least squares for extra credit. If you decide to do this, please upload your extra credit work under `lab01ec` on the course management system.

Submitting Your Work

Three things should be uploaded for grading:

- A PDF listing of final computer program.
- The graph `approx.pdf` of the approximation.
- The graph `error.pdf` for the error analysis.

The files `approx.pdf` and `error.pdf` have already been created and should be in the `lab01` subdirectory. The only thing left is to convert `euler.jl` into a PDF file for upload. In the lab the command

```
j2pdf -o euler.pdf euler.jl
```

may be used to produce a file `euler.pdf` suitable for uploading.

On Microsoft suitable files for upload can be generated by printing `quadratic.jl` and `quadratic.out` to PDF from within Notepad. For example, if `code01.pdf` corresponds to the printout of `quadratic.jl` and `run01.pdf` to `quadratic.out`, then upload both `code01.pdf` and `run01.pdf` for grading. A similar technique will work for Apple.

Subsequent lab assignments will require more independent work but follow a similar sequence of steps to produce files suitable for upload that can be viewed and annotated by the grader.

Before leaving don't forget to close the applications open on your desktop and logout. Exit the Julia REPL by typing `<ctrl>-d` and then `<ctrl>-d` again to close the terminal. The editor has a menu at the top. If using one of the lab computers, please reboot it into Microsoft Windows.