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.

Plotting with Julia

The previous lab activity introduced Julia by using it as a desktop calculator to solve a quadratic equation. The resulting sequence of commands were subsequently turned into a program and uploaded for grading. The real-world advantage of converting an interactive session to a non-interactive program is that the resulting computations can then be scheduled to run on a remote server without further input from the user.

Over the years computers have become faster and more powerful while the computer simulations people were interested in have become larger and more involved. As a result many computations of modern scientific and engineering relevance still take a long time. The possibility of being able to go to sleep or do other things while a computer performs a large calculation is important. Therefore, although we restrict the size of our problems in the lab so they finish in the blink of an eye, we consciously model the switch-

ing back and forth between the interactive and non-interactive techniques needed for practical numerical methods.

Since Julia was designed to work well in both interactive and non-interactive contexts, this course can explore both ways of using a computer with relative ease. Neither interactive nor non-interactive methods should be seen as better than the other, but depending on the task at hand one or the other may be more convenient. Knowing both guards against the danger a person might get stuck using an inconvenient mode of operation for their future work, waste time or fail entirely.

The present activity is data visualization. Depending on the computational effort needed to render the data, this task may be better performed interactively or non-interactively. As before we start with interactive techniques and convert our work to a self-contained program at the end.

There are two simple visualization tools we will employ—line graphs and point plots. In particular, the goal in this lab is to graph

$$f(x) = \cos x - 2x \tag{2.1}$$

and then annotate the point $(x_0, f(x_0))$ for $x_0 = 1$.

Note that in an upcomming lab we will use Newton's method to approximate a root of this same function. Thus, the present activity reflects the task of visualizing a problem before setting out to solve it. Note that unlike the guide for the previous lab which gave a step-by-step account of how to solve the exact problem to be graded, we describe here how to plot a similar function so as not so spoil the fun of doing it yourself.

Suppose then instead of (2.1) that we seek to graph

$$f(x) = 5xe^{-x} - 1 \tag{2.2}$$

and annotate the point $(x_0, f(x_0))$ for $x_0 = 2$.

Open the Julia REPL and enter

`f(x)=5*x*exp(-x)-1` and `x0=2`.

We shall now use the Julia `Plots` library to create a graph of the function along with the desired point.

Load the `Plots` library into the REPL by typing `using Plots`. At this point the output on the screen should look like

```
julia> f(x)=5*x*exp(-x)-1
f (generic function with 1 method)

julia> x0=2
2

julia> using Plots

julia>
```

It is possible, especially if using your own personal computer rather than the one in the lab, that an error will occur when trying to load the plotting library. If you see the following on your screen

```
julia> using Plots
ERROR: ArgumentError: Package Plots not found in current path:
- Run `import Pkg; Pkg.add("Plots")` to install the Plots package.

Stacktrace:
 [1] require(into::Module, mod::Symbol)
   @ Base ./loading.jl:893

julia>
```

This may be resolved by installing the `Plots` package and trying again.

To install `Plots` use either the package manager built into the REPL or follow the suggestion provided by the error message. The suggested method with `Pkg.add` can also be used inside a program code whereas the built-in package manager is simpler but only works from within the REPL. As the error message already explains how to use `Pkg.add`, how to use the built-in package manager will be explained henceforth.

To start the built-in package manager press the `]` key. The prompt on the screen should now look like

```
(@v1.6) pkg>
```

Now type `add Plots` and press enter.

Curiously the plotting library is one of the larger packages and has hundreds dependencies. As a result it will take a few minutes to download and install. This is why `Plots` is preloaded onto the lab computers. If you are reading this document ahead of time and plan to use your own laptop in the lab, it would also be reasonable to preload `Plots` before class.

When the package manager is finished the screen should look like

```
Precompiling project...
  131 dependencies successfully precompiled in 204 seconds

(@v1.6) pkg>
```

To exit the built-in package manager press ⟨backspace⟩ until you get back to the `julia>` prompt. At this point it should be possible to type `using Plots` and continue with the lab.

To create a plot specify a range of x-values and map them to the corresponding y-values by typing

`xs=0:0.1:5` and `ys=f.(xs)`

Here `xs` is an iterator for the interval $[0, 5]$ with points spaced 0.1 units apart. The `.` which appears in the expression `f.(xs)` indicate the function should be applied pointwise to each value of `xs`. The Julia documentation describes this operation as a broadcast; however, to me the `.` mnemonically indicates a pointwise mapping. The result `ys` will be a 51-element vector.

Note that the assignment to `ys` displays the resulting vector up to as many values as fit on the screen. In cases where the return value fills the screen it is sometimes desirable to suppress printing it. This can be done by placing a semicolon at the end of a command. In the present case

```
ys=f.(xs);
```

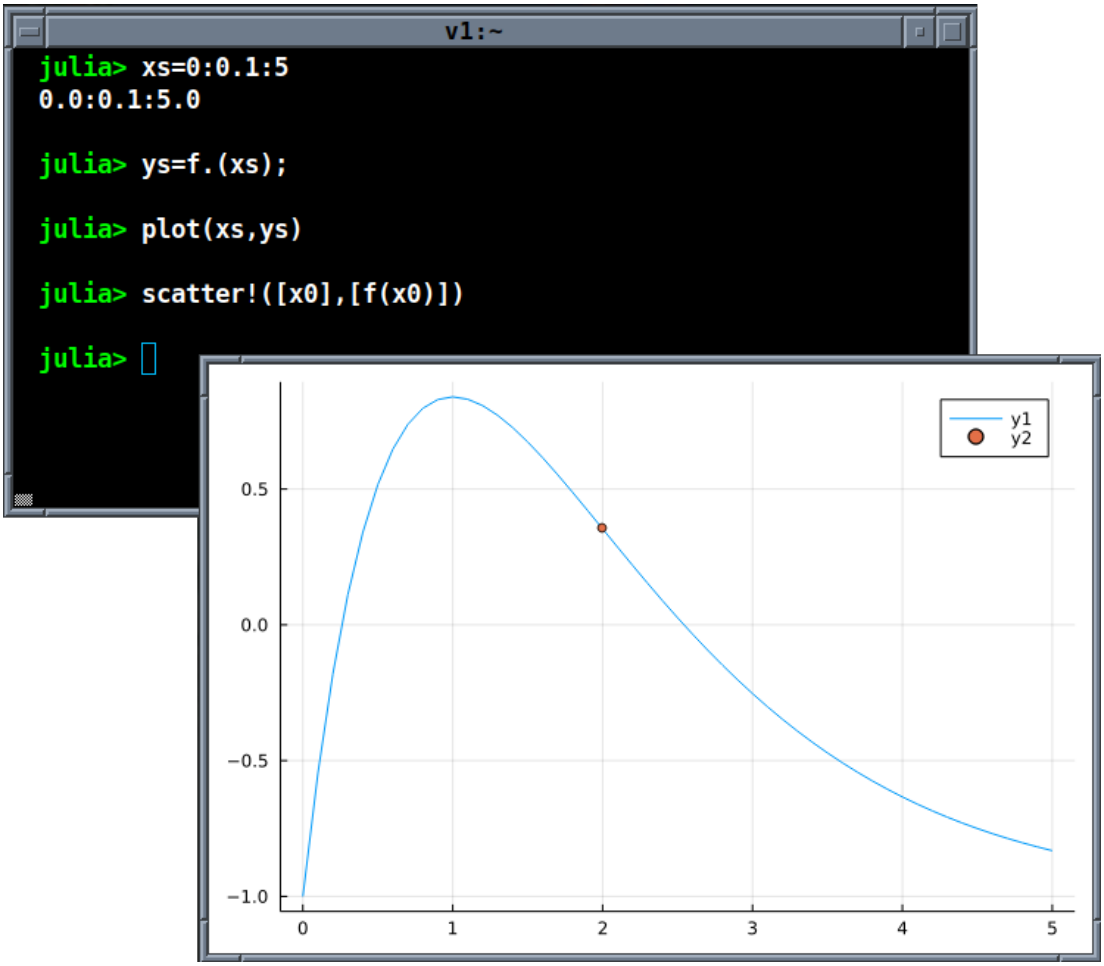
assigns the same 51-element vector to `ys` without filling the screen with the values of that vector.

Now it should be possible to type

`plot(xs,ys)` and `scatter!([x0],[f(x0)])`

to show a graph of the function and the initial approximation for the root. Note that the `!` in `scatter!` tells the plotting library to combine the point plot of $(x_0, f(x_0))$ with the graph of $f(x)$.

The relevant windows on your screen should look like



Graphs are useful for interactive visualization. The one here shows there is a root between 2 and 3 and further suggests that Newton's method starting at $x_0 = 2$ will converge to that root. That will be the topic of the next computing lab. For now we note the benefit of including all kinds of computer-rendered visualizations in a presentation or written report. Thus, we need a way of saving our graph for later use.

Type `savefig("graph02.pdf")` to write the graph to a file. The resulting file `graph02.pdf` can be embedded into many word processors as well as documents like this one prepared using the TEX typesetting system. Such PDF files are also suitable to upload for grading.

Submitting Your Work

For this lab two things should be uploaded for grading:

- A graph of the function $f(x)$ for the problem in equation (2.1) which further illustrates the point at the specified value of x_0.
- A self-contained program that creates the graph.

To help with the second item above and continue this introduction to scientific computing, we do it except for equation (2.2) as follows.

After copying the relevant lines from the REPL into the file `funplot.jl` using a text editor we obtain the program

```
1 f(x)=5*x*exp(-x)-1
2 x0=2
3
4 using Plots
5 xs=0:0.1:5
6 ys=f.(xs)
7 plot(xs,ys)
8 scatter!([x0],[f(x0)])
9 savefig("graph02.pdf")
```

To complete this lab modify lines 1 and 2 in the above program to plot equation (2.1), run it, convert to the source code to PDF format and check the two output files. The results in the terminal should look like

```
$ julia funplot.jl
$ j2pdf -o submit02.pdf funplot.jl
[...omitted output...]
$ evince submit02.pdf &
$ evince graph02.pdf &
```

Due to the complexity of the Julia `Plots` library, it may take a while for the first command to finish. Finally, upload `graph02.pdf` and `submit02.pdf` for grading to the course management system. If using one of the lab computers, please reboot it into Microsoft Windows before leaving.