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.

Newton's Method

The first computer lab introduced Julia by using it as a desktop calculator while the second explored some graphing capabilities. The next activity is to approximate solutions to $f(x) = 0$ with Newton's method. Note that

- Newton's method is one of the most important algorithms used in computation due to its fast rate of convergence.
- As it is covered in an introductory calculus course, the familiarity continues our gentle introduction to scientific computing.

After throwing any theory about convergence or the rate of convergence out the window, Newton's method can be described mathematically as the iterative scheme

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where x_0 is an initial guess for the solution and each x_n is an improved approximation of that solution.

The goal today is for you to write a program that approximates the solution near $x_0 = 1$ to the equation

$$\cos x = 2x \tag{3.1}$$

using Newton's method. Upon setting $f(x) = \cos x - 2x$ note this is the same function that was graphed in the previous lab.

How to approximate the solutions of $f(x) = 0$ will now be described in step-by-step details. As in the previous lab we focus on a similar example so as not so spoil the fun of doing it yourself.

Suppose then instead of (3.1) that we seek to approximate the solution near $x_0 = 2$ to the equation

$$5xe^{-x} = 1 \tag{3.2}$$

using Newton's method.

Note that the roots of the function $f(x) = 5xe^{-x} - 1$ correspond to the solutions of equation (3.2). Now, open the Julia REPL and enter

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

One of the complications of Newton's method is that the algorithm requires the derivative $f'(x)$ to perform the iterations. While it might appear necessary for a scientist or engineer to find the derivative by hand, in the 1960s computer algebra systems began to appear that were capable of finding derivates using the rules of calculus. Today, such computer algebra systems are widely available. One is built in to Julia.

Symbolic Differentiation

Julia allows us to write a program that automatically creates a new function `df(x)` corresponding to the derivative $f'(x)$ as it runs. In general this type of activity is called metaprograming. Before writing a full program for Newton's method we will try out some metaprogramming in the REPL.

Type `using Symbolics` to load the Julia computer algebra system into the REPL. As with `Plots` you may need to install the library before proceeding. This may be done with the built-in package manager or the `Pkg.add` command.

After everything is installed and working 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 Symbolics

julia>
```

Define a symbolic variable `t` by typing `@variables t` followed by

```
D(g)=expand_derivatives(Differential(t)(g))
```

to take the derivative with respect to `t`. The derivative $f'(t)$ may now be found by typing `D(f(t))`. The screen should now look like

```
julia> using Symbolics

julia> @variables t
1-element Vector{Num}:
 t

julia> D(g)=expand_derivatives(Differential(t)(g))
D (generic function with 1 method)

julia> D(f(t))
5exp(-t) - 5t*exp(-t)
```

What is left is to use the metaprogramming feature of Julia to convert this symbolic expression into executable code. In particular, we turn the algebraic expression for the derivative into a string and append another string to make a function definition. To do this type

```
as="df(t)="*string(D(f(t)))
```

Finally, evaluate the string by typing

```
eval(Meta.parse(as))
```

At this point `df` has been created and the screen should look like

```
julia> as="df(t)="*string(D(f(t)))
"df(t)=5exp(-t) - 5t*exp(-t)"

julia> eval(Meta.parse(as))
df (generic function with 1 method)
```

Although the above procedure is admittedly a bit cryptic, having the built-in functionality to find derivatives is definitely easier than copying code in and out of a separate computer algebra system like Maple or Mathematica. Symbolic differentiation further avoids any errors that might occur when finding a derivative by hand.

To test Newton's method we set `xn=x0` and compute

```
xn=xn-f(xn)/df(xn)
```

as many times as needed to approximate the solution.

In the REPL this looks like

```
julia> xn=x0
2

julia> xn=xn-f(xn)/df(xn)
2.5221887802138703

julia> xn=xn-f(xn)/df(xn)
2.5425691666820014

julia> xn=xn-f(xn)/df(xn)
2.5426413568569757

julia> xn=xn-f(xn)/df(xn)
```

```
2.5426413577735265

julia> xn=xn-f(xn)/df(xn)
2.5426413577735265
```

Note that the up-arrow key followed by ⟨enter⟩ was repeatedly pressed to iterate the recurence. As the displayed value didn't change in the last iteration we conclude the method has converged to all available digits.

Submitting Your Work

For this lab two things should be uploaded for grading:

- A program that performs five iterations of Newton's method.
- The output from running that program.

To help with the items above, we describe the steps needed to complete them for the example equation (3.2) in detail.

After copying the relevant lines from the REPL and adding a loop to repeat the Newton iteration we obtain the program

```
 1 # newton.jl -- Perform five iterations of Newton's method
 2 f(x)=5*x*exp(-x)-1
 3 x0=2
 4 
 5 using Symbolics
 6 @variables t
 7 D(g)=expand_derivatives(Differential(t)(g))
 8 as="df(t)="*string(D(f(t)))
 9 eval(Meta.parse(as))
10 
11 xn=x0
12 for n=1:5
13     global xn=xn-f(xn)/df(xn)
14     println("n=",n,", xn=",xn)
15 end
```

When I first learned Julia, I found the need for `global` in line 19 surprising. While variables in the global scope are assumed for convenience inside of loops in the REPL, to help avoid unintentional errors they need to be explicitly declared in a Julia program.

Test the program by running it. The output should be

```
$ julia newton.jl
n=1, xn=2.5221887802138703
n=2, xn=2.5425691666820014
n=3, xn=2.5426413568569757
n=4, xn=2.5426413577735265
n=5, xn=2.5426413577735265
```

To finish this lab modify lines 2 and 3 in `newton.jl` to solve the correct equation starting with the correct value of x_0. Run the program placing the output in `newton.out` and finally convert everything to Postscript format. A transcript of the commands needed to prepare the program and output for submission look like

```
$ julia newton.jl >newton.out
$ j2pdf -o submit04.pdf newton.jl newton.out
```

Upload `submit04.pdf` for grading to the course management system. Please reboot the lab computer into Microsoft Windows before leaving.