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.

Newton's Method

The first computer lab introduced Julia by using it as a desktop calculator while the second explored the convergence of Newton's method using arbitrary precision arithmetic. The next activity is to use Newton's method approximate solutions to systems of nonlinear equations.

Consider finding the root to $f(x) = 0$ where $f\colon \mathbf{R}^n \to \mathbf{R}^n$. Newton's method for systems can be described mathematically as the iterative scheme

$$x^{(k+1)} = x^{(k)} - [Df(x^{(k)})]^{-1} f(x^{(k)})$$

where $x^{(0)} \in \mathbf{R}^n$ is an initial guess and each $x^{(k)}$ is an improved approximation of that solution. Here $Df(x) \in \mathbf{R}^{n\times n}$ is the matrix derivative given as the Jacobian matrix for f by

$$Df(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}.$$

The goal today is for you to write a program that approximates the solution of an individualized system of $n = 3$ nonlinear equations in three variables. Your system may be obtained by clicking on the following link:

```
https://fractal.math.unr.edu/~ejolson/466-24/newt3d/newt3d.cgi
```

Please do not use anyone else's system of equations for this lab.

The following discussion concerns the system which appears when I click the above link. That system is different than what you will obtain when you click the same link. To finish this lab please repeat these same steps but for own individualized system of equations.

Upon clicking on the link, I obtained

Your function f(x,y,z) in Julia-compatible code is

```
f1(x,y,z) = -3x^(-3)*y^(-3)*z^3 + x*y^(-3)*z^(-2) - 0.9812 + x^3*z
f2(x,y,z) = -x^2*z^(-1) + 3x*z^2 + 2.405
f3(x,y,z) = -2x*z^3 + z + 23.38 - 2x^(-1)*y^2*z^(-1) + 3x^(-1)*y^2*z
f(x,y,z) = [f1(x,y,z), f2(x,y,z), f3(x,y,z)]
```

and the initial guess x0 is

```
x0 = [1.155, -3.768, 0.4616]
```

Now, open the Julia REPL and enter the above formula and initial condition.

Let's check how close the initial approximation is to the root by plugging it in. Since `f(x,y,z)` is a function of three scalars rather than a 3-vector we need the splat operator `...` to evaluate $f(x^{(0)})$. The result

```
julia> f(x0...)
3-element Vector{Float64}:
  -0.36771284790703074
   0.25330118402218327
 -12.623310707923679
```

appears to show that the initial guess is not very good. Fortunately, $x^{(0)}$ has been tested ahead of time and it's known Newton's method will converge.

One of the complications of Newton's method is that the algorithm requires the derivative $Df(x)$ to perform the iterations. While it might appear necessary to find the derivative by hand, in the 1960s computer algebra systems began to appear that were capable of finding derivatives 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 $Df(x)$ as it runs. In general this type of activity is called metaprogramming. 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. If using your laptop 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> f1(x,y,z) = -0.9812 + x^3*z - 3x^(-3)*y^(-3)*z^3 + x
*y^(-3)*z^(-2)
f1 (generic function with 1 method)

julia> f2(x,y,z) = 2.405 - x^2*z^(-1) + 3x*z^2
f2 (generic function with 1 method)

julia> f3(x,y,z) = 3x^(-1)*y^2*z - 2x*z^3 + z + 23.38 - 2x^
(-1)*y^2*z^(-1)
f3 (generic function with 1 method)

julia> f(x,y,z) = [f1(x,y,z), f2(x,y,z), f3(x,y,z)]
f (generic function with 1 method)

julia> x0 = [1.155, -3.768, 0.4616]
3-element Vector{Float64}:
  1.155
 -3.768
  0.4616

julia> f(x0...)
3-element Vector{Float64}:
  -0.36771284790703074
   0.25330118402218327
 -12.623310707923679
```

```
julia> using Symbolics

julia>
```

Define the symbolic variables `x`, `y` and `z` by typing

```
@variables x,y,z
```

The derivative $Df(t)$ may now be found by typing

```
Dfsym=Symbolics.jacobian(f(x,y,z),[x,y,z])
```

The screen should now look like

```
julia> @variables x,y,z
3-element Vector{Num}:
 x
 y
 z

julia> Dfsym=Symbolics.jacobian(f(x,y,z),[x,y,z])
3×3 Matrix{Num}:
 ((1 / y)^3)*((1 / z)^2) + 3z*(x^2) + 9(z^3)*((1 / x)^2)*(1
 / (x^2))*((1 / y)^3)  …  x^3 + (-2x*((1 / y)^3)*(1 / (z^2)
)) / z - 9(z^2)*((1 / x)^3)*((1 / y)^3)

   (-2x) / z + 3(z^2)
                  (x^2) / (z^2) + 6x*z
                   (-3z*(y^2)) / (x^2) - 2(z^3) - z*((-2(y
^2)) / ((x^2)*(z^2)))                  1 + (3(y^2)) / x - 6
x*(z^2) - x*((-2(y^2)) / ((x^2)*(z^2)))
```

The formatting is admittedly poor and the ellipses indicate that some of the output has been omitted. However, the computer has done the differentiation correctly and 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(x,y,z)="*string(Dfsym)
```

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(x,y,z)="*string(Dfsym)
 Df(x,y,z)=Num[((1 / y)^3)*((1 / z)^2) + 3z*(x^2) + 9(z^3)*
((1 / x)^2)*(1 / (x^2))*((1 / y)^3) 9(z^3)*((1 / x)^3)*((1
/ y)^2)*(1 / (y^2)) - 3x*((1 / y)^2)*(1 / (y^2))*((1 / z)^2
) x^3 + (-2x*((1 / y)^3)*(1 / (z^2))) / z - 9(z^2)*((1 / x)
^3)*((1 / y)^3); (-2x) / z + 3(z^2) 0 (x^2) / (z^2) + 6x*z;
 (-3z*(y^2)) / (x^2) - 2(z^3) - z*((-2(y^2)) / ((x^2)*(z^2)
)) (-4y) / (x*z) + (6y*z) / x 1 + (3(y^2)) / x - 6x*(z^2) -
 x*((-2(y^2)) / ((x^2)*(z^2)))]

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

It should be pointed out metaprogramming in Julia needs to be done at the global scope rather than inside a function. This reason for this has to do with the just-in-time compiler and what's called the world-age problem. The main point, however, is that symbolic differentiation avoids any errors that might occur when finding a derivative by hand.

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

```
xn = xn - Df(xn...) \ f(xn...)
```

as many times as needed to approximate the solution.

In the REPL this looks like

```
julia> xn=copy(x0)
3-element Vector{Float64}:
  1.155
 -3.768
  0.4616

julia> xn=xn-Df(xn...)\f(xn...)
3-element Vector{Num}:
  1.3135088947604612
 -3.734528792976971
  0.5079657717780771

julia> xn=xn-Df(xn...)\f(xn...)
3-element Vector{Num}:
  1.299477798069595
 -3.496119553399677
  0.49982960484484773

julia> xn=xn-Df(xn...)\f(xn...)
3-element Vector{Num}:
  1.300005467488836
 -3.4998237583470253
  0.5000023177811371

julia> xn=xn-Df(xn...)\f(xn...)
3-element Vector{Num}:
  1.3000054447010043
 -3.499824561987055
  0.5000022728764185
```

Note that the up-arrow key followed by ⟨enter⟩ was repeatedly pressed to iterate the recurrence. 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  # newton3d.jl -- Perform five iterations of Newton's method
2  f1(x,y,z) = -3x^(-3)*y^(-3)*z^3 +
3              x*y^(-3)*z^(-2) - 0.9812 + x^3*z
4  f2(x,y,z) = -x^2*z^(-1) + 3x*z^2 + 2.405
5  f3(x,y,z) = -2x*z^3 + z + 23.38 -
6              2x^(-1)*y^2*z^(-1) + 3x^(-1)*y^2*z
7  f(x,y,z) = [f1(x,y,z), f2(x,y,z), f3(x,y,z)]
8
9  x0 = [1.155, -3.768, 0.4616]
10
11 using Symbolics
12 @variables x,y,z
13 Dfsym=Symbolics.jacobian(f(x,y,z),[x,y,z])
14 as="Df(x,y,z)="*string(Dfsym)
15 eval(Meta.parse(as))
16
17 xn=copy(x0)
18 for n=1:5
19     global xn=xn-Df(xn...)\f(xn...)
20     println("n=",n)
21     display(xn)
22     println()
23 end
```

The formula for `f1` has been split with a dangling binary operator so it continues from line 2 to 3. The formula for `f3` has been split between lines

5 and 6 in a similar way. When I first learned Julia, I found the need for `global` in line 19 surprising because it was not needed in the REPL. 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 newton3d.jl
n=1
3-element Vector{Num}:
  1.3135088947604612
 -3.734528792976971
  0.5079657717780771
n=2
3-element Vector{Num}:
  1.299477798069595
 -3.496119553399677
  0.49982960484484773
n=3
3-element Vector{Num}:
  1.300005467488836
 -3.4998237583470253
  0.5000023177811371
n=4
3-element Vector{Num}:
  1.3000054447010043
 -3.499824561987055
  0.5000022728764185
n=5
3-element Vector{Num}:
  1.3000054447010159
 -3.4998245619868174
  0.5000022728764241
```

To finish this lab modify lines 2 through 9 in `newton3d.jl` to solve the correct equation starting with the correct value of x_0. Run the program placing the output in `newton3d.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 newton3d.jl >newton3d.out
$ j2pdf -o submit05.pdf newton3d.jl newton3d.out
```

Upload `submit05.pdf` for grading to the course management system. Please reboot the lab computer into Microsoft Windows before leaving.