In science and engineering an important goal is to become a skilled practitioner by doing it yourself. The computer labs provide computational experience related to the analytic theory presented in the lectures.

A Two-stage Runge–Kutta Method

This lab is about using Runge–Kutta methods to approximate the solution to the differential equation

$$y' = f(x,y) \qquad \text{where} \qquad y(x_0) = y_0$$

on the interval $[a,b]$ where $x_0 = a$.

Let $x_n = a + nh$ where $h = (b-a)/N$ and N is the number of grid points. Integrate over each of the subintervals $[x_n, x_{n+1}]$ as

$$\int_{x_n}^{x_{n+1}} y'(x)dx = \int_{x_n}^{x_{n+1}} f(x, y(x))dx.$$

Applying the fundamental theorem of calculus on the left and the trapezoid rule on the right yields

$$y(x_{n+1}) - y(x_n) \approx \frac{h}{2}\big(f(x_n, y(x_n)) + f(x_{n+1}, y(x_{n+1}))\big).$$

The above approximation allows one to approximate $y(x_{n+1})$ in terms of $y(x_n)$ by solving for ξ such that

$$\xi = y(x_n) + \frac{h}{2}\big(f(x_n, y(x_n)) + f(x_{n+1}, \xi)\big).$$

and then taking $y(x_{n+1}) \approx \xi$. However, one difficulty is that ξ is defined implicitly and finding it may involve solving a non-linear equation. While this could be accomplished using Newton's method, an explicit iteration can be obtained by replacing ξ on the right by the first order Taylor series

$$y(x_{n+1}) \approx y(x_n) + hy'(x_n) = y(x_n) + hf(x_n, y(x_n)).$$

Letting y_n be the resulting approximation of $y(x_n)$ immediately leads to the numerical scheme

$$y_{n+1} = y_n + \frac{h}{2}\big(f(x_n, y_n) + f(x_{n+1}, y_n + hf(x_n, y_n))\big).$$

The above scheme is known as the RK2 method. Note the nesting of f inside of f on the right. This composition is characteristic of RK methods.

The Lorenz System

The Lorenz system is an autonomous three-dimensional ordinary differential equation of the form

$$\frac{dy}{dx} = f(y)$$

with a given initial condition $y(0) = y_0$ where $y(x)$ is a vector in $\mathbf{R}^3$ and

$$f(y) = \begin{bmatrix} -10y_1 + 10y_2 \\ 28y_1 - y_2 - y_1y_3 \\ y_1y_2 - (8/3)y_3 \end{bmatrix}.$$

Each person will have a different initial condition y_0. Click on the following link to retrieve the values of your initial condition:

`https://fractal.math.unr.edu/~ejolson/466-23/y0/mky0.cgi`

Please do not use anyone else's initial condition for this lab.

To implement the RK2 method described above first write a subroutine to compute the function $f(y)$. In Julia this may be done with the code

```julia
function f(y)
    r=[10*(y[2]-y[1]),
       (28.0-y[3])*y[1]-y[2],
       y[1]*y[2]-(8/3)*y[3]]
    return r
end
```

The RK2 Timestep

In order to use RK2 for practical computation first rewrite it as

$$\begin{aligned} K_1 &= hf(x_n, y_n) \\ K_2 &= hf(x_{n+1}, y_n + K_1) \\ y_{n+1} &= y_n + \tfrac{1}{2}\big(K_1 + K_2\big). \end{aligned}$$

Since the f in the Lorenz system is does not depend on x the RK2 method can be simplified as

$$\begin{aligned} K_1 &= hf(y_n) \\ K_2 &= hf(y_n + K_1) \\ y_{n+1} &= y_n + \tfrac{1}{2}\big(K_1 + K_2\big). \end{aligned}$$

Next, write a subroutine to make one RK2 timestep. Using the coefficients given in the tableaux to compute the k_i yields

```
function rk2(y,h)
    k1=h*f(y)
    k2=h*f(y+k1)
    return y+1/2*(k1+k2)
end
```

Note Julia will use multiple dispatch to compile efficient versions of `rk2` for whatever length vectors appear as `y` in the arguments. The built-in vector notation then makes the code for solving systems of ordinary differential equations appear identical to the code for solving scalar equations.

Plotting the Solution

Our goal is to plot an approximation of the solution's trajectory in phase space for $t \in [0, T]$ where $T = 10$. This will yield a visualization of what has commonly been called the Lorenz butterfly in chaos theory and the study of nonlinear dynamics.

Consider approximating the solution using $N = 20480$ time steps of size $h = T/N$. The result in a sequence of 20480 vectors $y_n \in \mathbf{R}^3$. While the plotting system would likely handle 20480 points without trouble, it's not difficult to imagine lengthier calculations with even more points. Thus, it is reasonable to plot only a subsample of the total timesteps.

One way to do this is with two nested loops where the outer loop stores the points to be plotted while the inner loop advances a certain number of time steps to find the next suitable point for plotting. To make the code more straight forward, we place the inner loop in a separate subroutine `solve` that performs `n` steps of size `h`. In particular, we have

```
16 function solve(y0,h,n)
17     yn=copy(y0)
18     for j=1:n
19         yn=rk2(yn,h)
20     end
21     return yn
22 end
```

It's worth mentioning that line 17 copies the initial condition as `yn=copy(y0)` to prevent `y0` from getting overwritten. If instead line 17 appeared as `yn=y0` this would indicate `yn` is a pointer referencing `y0`. In that case any changes to `yn` would also change `y0`. The explicit `copy` avoids this pitfall.

Before writing the outer loop that repeatedly calls `solve` we need to decide how many `rk2` steps should be made between the points we plot. One doesn't want to plot so many points that the plotting library runs slowly or out of memory, nor does one want to plot so few points that the graph no longer appears like a smooth solution to a differential equation.

Given the apparent speed at which the dynamics in the Lorenz equations evolve and the fact that $h = 1/2048$, skipping every $m = 16$ timesteps between plotted points still yields a smooth curve. On the other hand, skipping every 16 timesteps reduces the number of points to plot from $N = 20480$ to $P = 1280$ which results in a graph that is efficient to render.

Code to calculate the relevant parameters used in the loops is

```
24 N=20480
25 T=10
26 h=T/N
27 
28 m=16
29 P=N÷m
```

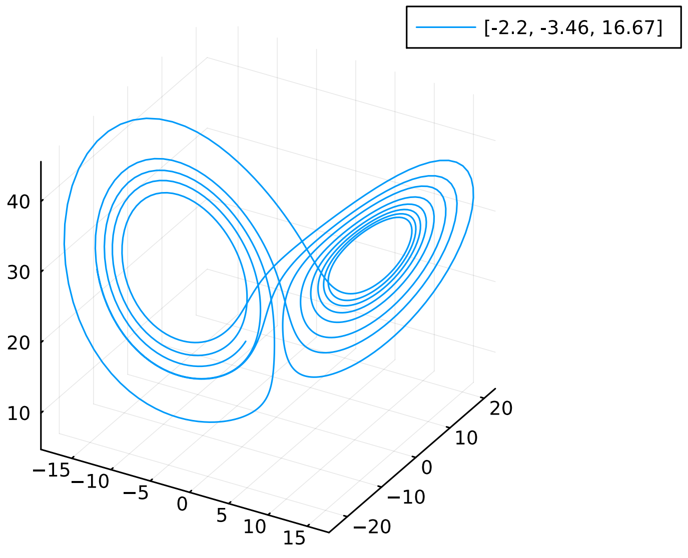
Note that line 29 includes the Unicode integer division operator `÷` rather than the usual `/` which would have resulted in a floating point value. This character can be entered in the Julia REPL by typing `\div` followed by the ⟨tab⟩ key. If you have difficulty typing `÷` into the editor try cut and paste from the REPL using the mouse.

One can initialize the arrays in which to store the points for plotting and write the outer loop as

```
31 X=zeros(P)
32 Y=zeros(P)
33 Z=zeros(P)
34
35 y0=[-2.20, -3.46, 16.67]
36 yj=copy(y0)
37 for j=1:P
38     global yj=solve(yj,h,m)
39     X[j]=yj[1]
40     Y[j]=yj[2]
41     Z[j]=yj[3]
42 end
```

The initial condition `y0` appearing on line 35 reflects the value when I click on the web link mentioned earlier. You will have to change this to your individualized initial condition. Line 38 includes a `global` declaration to resolve the ambiguity between `yj` in the global scope and the possibility of a local version of `yj` inside the scope of the loop.

Finally, to create a graph that looks similar to



plot the output using the `Plots` library.

```
using Plots
plot(X,Y,Z,label="$y0")
savefig("butterfly.pdf")
```

At this point you should have a file called `butterfly.pdf` stored in your working directory. If it looks *exactly* like the above figure, that may mean you forgot to change the initial condition.

Submitting Your Work

Two things should be uploaded for grading:

- A PDF file `lorenz.pdf` containing the code `lorenz.jl` used to generate the graph `butterfly.pdf`.
- The graph `butterfly.pdf` corresponding to your initial condition.

The files `butterfly.pdf` has already been created and should be in the `lab04` subdirectory. The only thing left is to convert `lorenz.jl` and its output into a PDF file for upload. In the lab the commands

```
$ j2pdf -o lorenz.pdf lorenz.jl
```

may be used to produce a file `lorenz.pdf` suitable for uploading. You may check your submission using `evince` to view the PDF files.

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.