MATH 312 Numerical Methods Paper

Ross Kogel, Alana Nadolski, Stan Laguerre

December 2023

1 Introduction - outline goals of paper

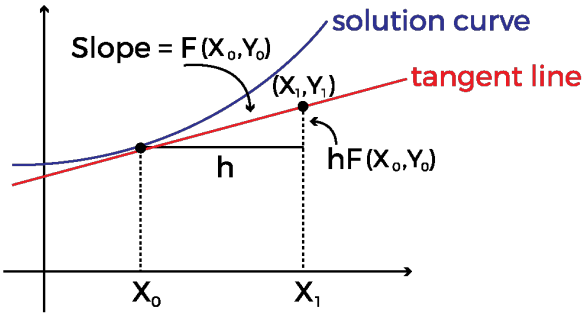
In this paper, we will discuss in depth our Numerical Methods app. First, we will outline the goals of the app and what we hope that the users can gain from it. Next, we will discuss how we built the app and how it works. Finally, we will discuss what we would add to the app if we had more time to work on it.

2 Goals of the App - purpose of the app and what we can gain from it

In this app, we wanted to build an easy interface for people to explore the differences between numerical methods and quickly get an idea what the solutions to their equations look like. We started out with implementing Euler's Method and Heun's Method, but later decided to look into and add the Runge-Kutta method to explore numerical methods further than we did in class.

The Euler Method is a numerical method for approximating solutions to ODEs based on the slope of a particular solution given that we know an initial condition of that particular solution and a function $f(y,t)$ equal to the derivative at any values of y and t. Together with Heun's and the Runge-Kutta numerical methods, it makes up part of the Runge-Kutta family of methods for numerical approximation, which generalizes much of the Euler method into solving higher order ODEs. In the plane of solutions, Euler's method takes the output $f(y,t)$, which has already been established as equal to the derivative, and follows a line with that slope for a given step size of t, giving the approximation for the output of the solution at the end of the step size based on the endpoint of the line with slope $\frac{dy}{dt}$:
$y_{n+1} = y_n + hf(t_n, y_n)$, where h is the step size.

This figure visually represents the execution of Euler's Method on the y versus t solution plane:

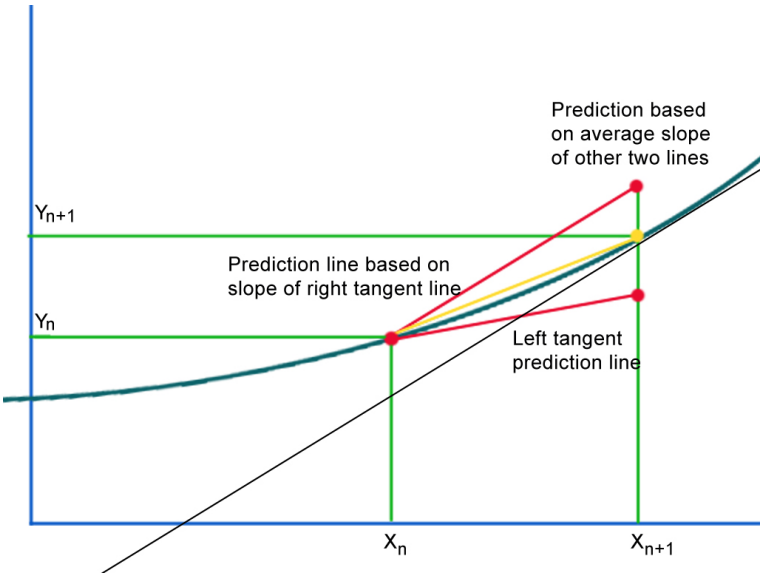


Calcworkshop.com

The Heun method takes the Euler method one step further: since we are working with small step sizes t, the Heun method improves accuracy by assuming that at each small interval in our solution approximation the solution does not switch from concave up to concave down or vice versa; for sufficiently small step sizes this will be true almost all of the time. Since the concavity is unchanging we can predict on a concave up curve that the slope of the tangent line at the end of the interval making up part of our solution approximation is steeper than the slope of the tangent line at the beginning; we can predict something analogous for concave down curves. So we can take the average of the slope at the beginning and end of our interval generated by Euler's method to create a more accurate slope for our given interval (representing the slope across the interval, not just the beginning of it) than the one given to us by Euler's method, creating an improved numerical method:

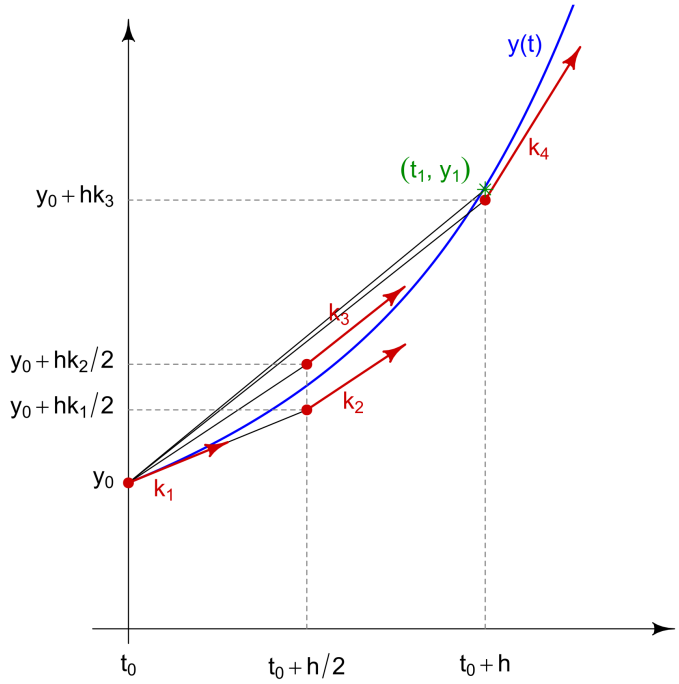
$\hat{y}_{i+1} = y_i + hf(y_i, t_i)$; $y_{i+1} = y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, \hat{y}_{i+1})]$

This figure visually represents the execution of Heun's Method on the y versus t solution plane:



The Runge-Kutta fourth-order method makes even finer adjustments to more accurately capture all the information contained inside one interval of the solution based on the ODE and the initial condition by taking a weighted average of four quantities k_1, k_2, k_3, k_4: k_1 is the slope at the beginning of the interval calculated with Euler's method; k_2 is the slope at the midpoint of the line on the interval created using k_1; k_3 is the slope at the midpoint of the line on the interval created using k_2; and finally k_4 is the slope at the endpoint of the line calculated using k_3. In this way we have a more nuanced representation of what might be happening at different points along the interval using different slopes, weighting the ones at the midpoints more heavily since they're more likely to represent the behavior that's actually going on for most of the interval. Notationally we compute Runge-Kutta using
$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$; $k_1 = f(t_n, y_n)$; $k_2 = f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2})$; $k_3 = f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2})$; $k_4 = f(t_n + h, y_n + hk_3)$

This figure visually represents the execution of the Runge-Kutta Method on the y versus t solution plane:



3 Building the App

To build this app, we utilized the Shiny package in R. This package has built in methods for programming reactive outputs based on the inputs a user gives, making the building blocks of this app simple. We used the functions for Euler's and Heun's methods that were provided for us in class by Professor O'Loughlin, and modeled our own function for the Runge-Kutta method off of those. Below is our code for the RK4 function with a system of 2 equations.

```
RungeKuttaSystemTable<-function(Fxyt=function(x,y,t){},
Gxyt=function(x,y,t){},x0,y0,a,b,n){
  x<-0
  y<-0
  t<-seq(a,b,by=(b-a)/(n-1))
  y[1]<-y0
  x[1]<-x0
  h <- (b-a)/(n-1) #step size
  for(i in 2:n){
    k1<-Fxyt(x[i-1], y[i-1],t[i-1])
    l1<-Gxyt(x[i-1], y[i-1],t[i-1])
```

```
    k2<-Fxyt(x[i-1]+((h*k1)/2), y[i-1]+((h*l1)/2),
    t[i-1]+(h/2))
    l2<-Gxyt(x[i-1]+((h*k1)/2), y[i-1]+((h*l1)/2),
    t[i-1]+(h/2))

    k3<-Fxyt(x[i-1]+((h*k2)/2), y[i-1]+((h*l2)/2),
    t[i-1]+(h/2))
    l3<-Gxyt(x[i-1]+((h*k2)/2), y[i-1]+((h*l2)/2),
    t[i-1]+(h/2))

    k4<-Fxyt(x[i-1]+(h*k3), y[i-1]+(h*l3),
    t[i-1]+h)
    l4<-Gxyt(x[i-1]+(h*k3), y[i-1]+(h*l3),
    t[i-1]+h)

    x[i]<-x[i-1]+(h/6)*(k1+2*k2+2*k3+k4)
    y[i]<-y[i-1]+(h/6)*(l1+2*l2+2*l3+l4)
  }
  plot(x,y,col="red",pch=20,cex=1.5)
  points(x[1],y[1],col="blue",pch=20,cex=2.5)
  recordPlot()

  table<-data.frame(t,x,y)
  return(table)
}
```

This function works by making vectors for x, y, and t and then iterating through each step and adding the new prediction to both x and y. In our app, we pass calls to this function directly with the inputs on the right when the "Runge-Kutta Method" button is selected. The app works similarly with the other two methods.

For the UI of the app, we put all of the inputs on the right of the screen and all of the outputs on the left of the screen. We made the design decision for the user to have to scroll down to see all of the plots so that we could have the xt and yt plot on top of each other for easy comparison as they both have a t-axis on the same scale.

4 Conclusion - summary + one more month

If we had more time to work on this project, there are lots of things that we could add. One big thing would be to fix the multiple methods display so that you can see all of the plots when the solutions go in drastically different directions. Another idea would be to add more methods to the app. One that we considered adding that we could look into is the implicit Euler's method. Additionally, it would be helpful if we could make a way for the app to display

the values of x and y for a given t range, since that is a function we had to use frequently when doing homework involving numerical methods.

This was an enjoyable project to do because of its combination of more in-depth understanding of numerical analysis and coding. Though the troubleshooting was challenging, the end result is a more accessible calculator for many people working with essential ODEs to use to understand what's actually going on in many solution fields of differential equations. We got a strong intuition for the numerical methods needed, especially the Runge-Kutta family, and we developed a far better appreciation of writing numerical methods in R, something that will inevitably help us when coding more functions in R in the future, or building on our intuition in numerical analysis. All in all, developing this app was a strong way to solidify our comfort working with numerical methods.