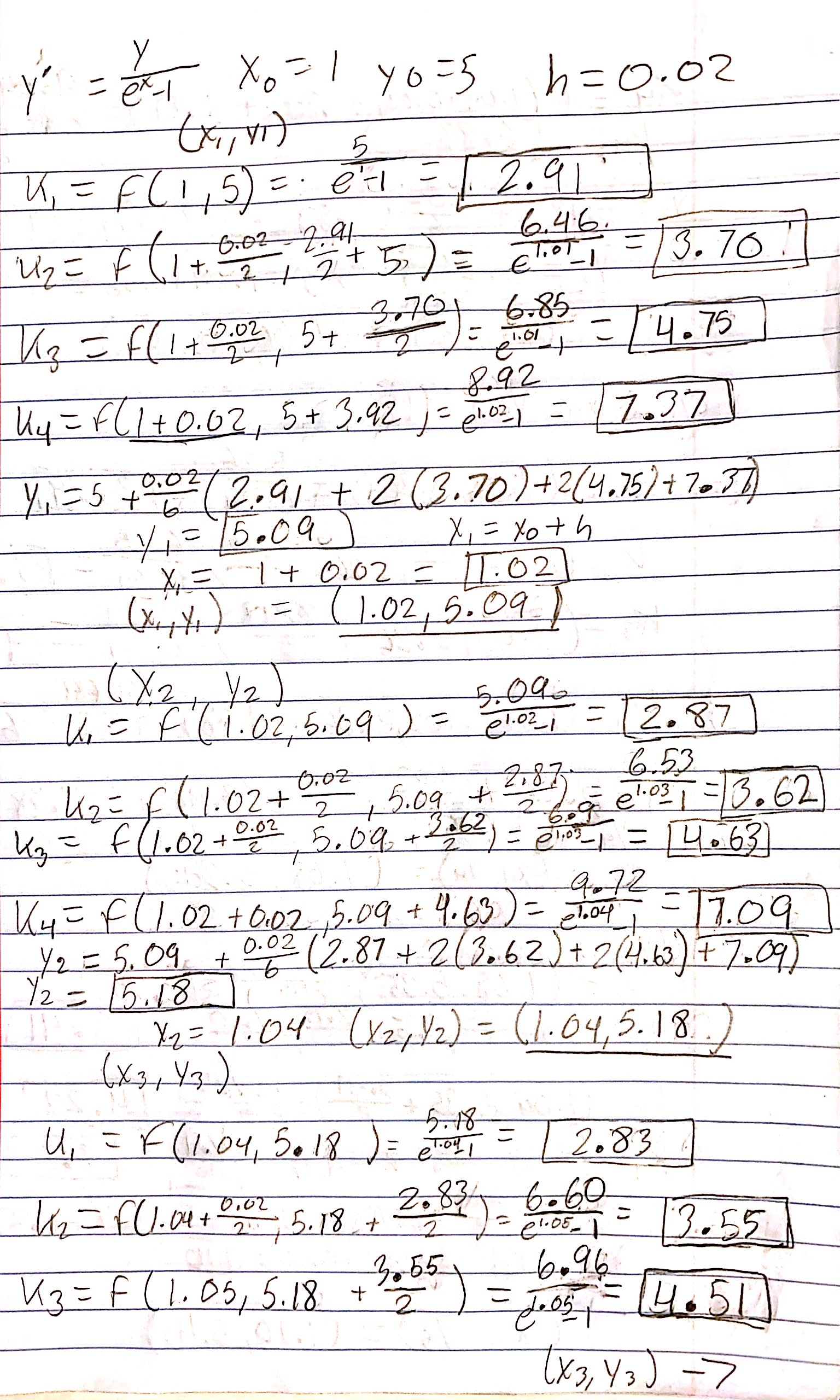
**CST-305: Project 2 – Runge-Kutta-Fehlberg (RKF) for ODE**

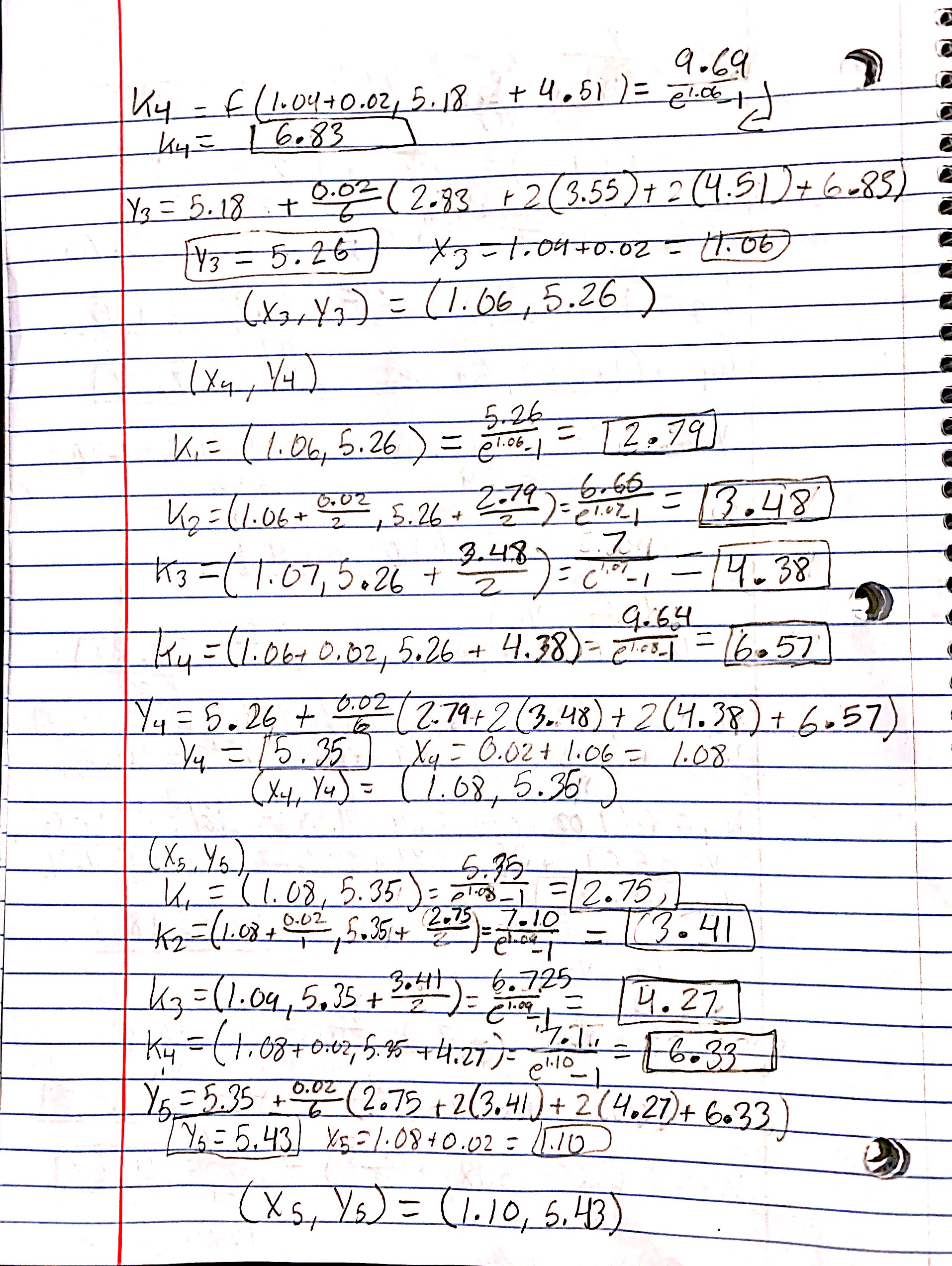
|  |  |  |  |
| --- | --- | --- | --- |
| Method: RUNGE-KUTTA METHOD | | | |
| Problem: | | | |
|  |  | | True Solution |
|  |  |
|  |  |  |  |
|  | 1.02 | 5.09 |  |
|  | 1.04 | 5.18 |  |
|  | 1.06 | 5.26 |  |
|  | 1.08 | 5.35 |  |
|  | 1.10 | 5.43 |  |

The above table is the result is from the manual calculations of the Runge Kutta method. The method is as follows:

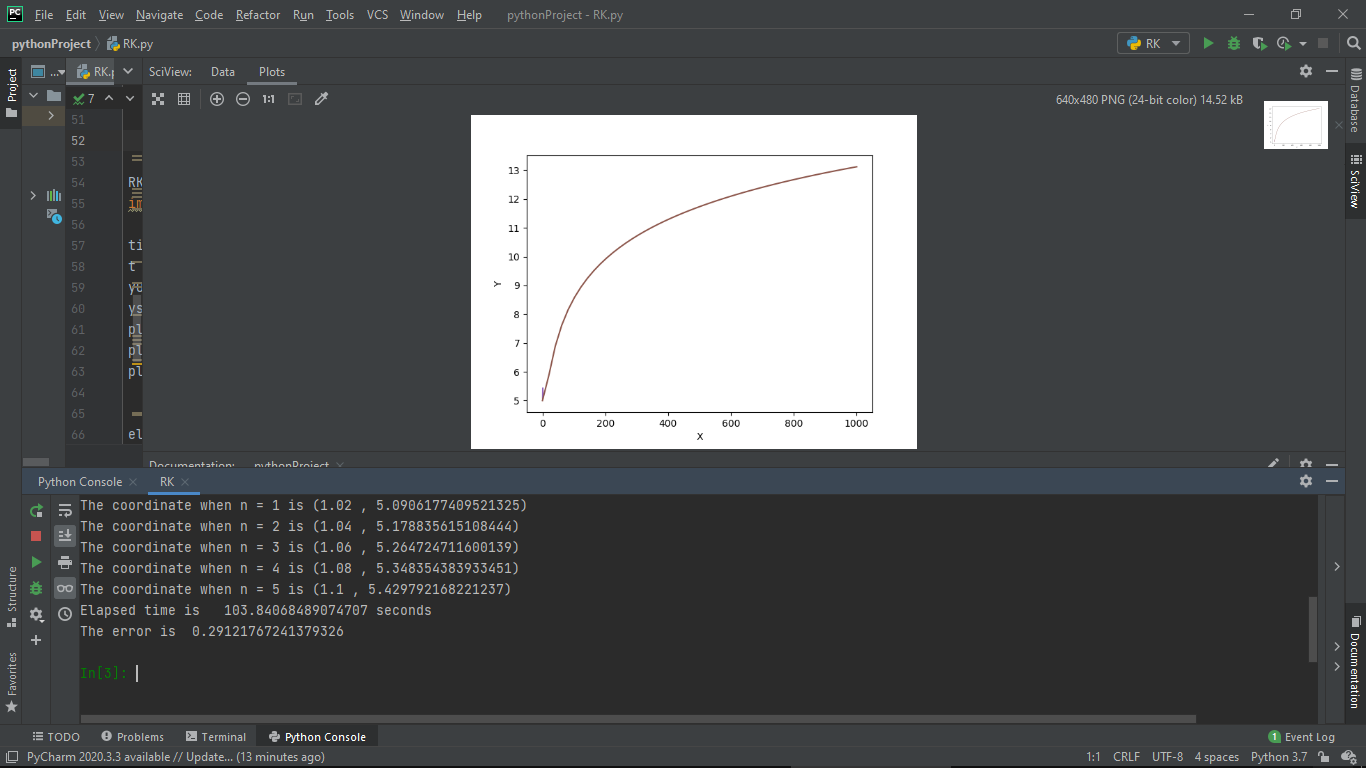
|  |  |  |
| --- | --- | --- |
| k_1 | = | hf(x_n,y_n) |
| k_2 | = | hf(x_n+1/2h,y_n+1/2k_1) |
| k_3 | = | hf(x_n+1/2h,y_n+1/2k_2) |
| k_4 | = | hf(x_n+h,y_n+k_3) |
| y_(n+1) | = | y_n+1/6k_1+1/3k_2+1/3k_3+1/6k_4+O(h^5) |

The screenshots below show my hand calculations up to (X5,Y5) using the method described above.

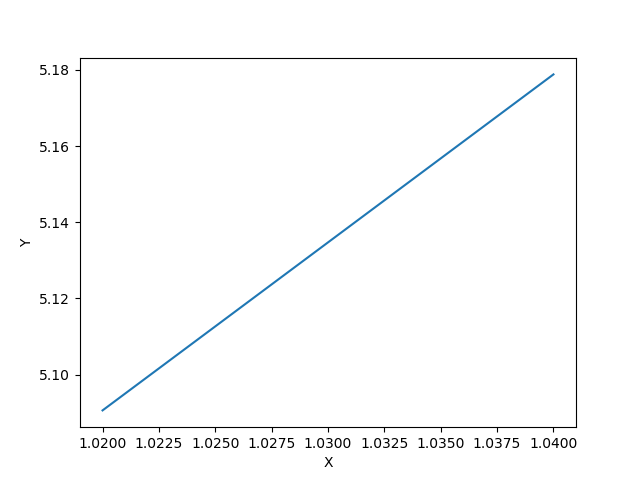


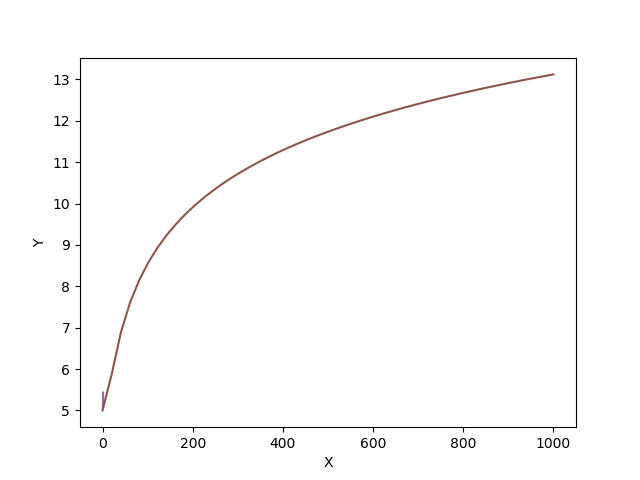


All necessary screenshots and the flowchart will be submitted along with the code and this document in a zipped file. The numpy odeint method was used after converting the method into a python program. After coding the RK method into a loop and graphing the result alongside the solution gained from odeint, I saw a massive overlap in the majority of the graphs which made the display of the curves have a purple color.



The first graph was the graph of the five points calculated from my RX method and the red graph was the solution gained from odeint.





The method also outputs each point as it as calculated from the loop (from x1 to x5). The computation time for the calculation of the 2000th point from ode took an average of 18 seconds. Depending on the execution of the program and the current background processes, the time it would for calculation would fluctuate. However, throughout a majority of the test runs I did (about 80), the execution would take 18 seconds. The error I calculated was about 0.3, this value was obtained by dividing the difference between the first five y-values given from odeint and the five values given from my Runge Kutta method. I then took the average of the points, found the difference, and divided it by the true value given from odeint. To reduce the error, I could possibly do even more steps and calculate more coordinates to compare to odeint. However, the one thing I would specifically change about my method is to implement the function defined dy\_dx in the beginning of the program directly into the loop. I had originally planned to do that, but I tried to keep it as close to the procedure I did by hand as possible. Had I been more proficient in python, I could have taken the time and written a recursive method that would call after the completion of each coordinate. If I were to do that however, the execution time and memory usage would surely increase and my program would not be as efficient. Similarly, to the method I had to take by hand, I had to continuously compare my code to my written calculations and there was a point I had to completely redo the by hand calculations because I made a very impactful error in K1 of the first coordinate. This project has showed me the accuracy of the Runge Kutta method and how straight forward and at the same time meticulous the method can be when being used to solve differential equations. I also learned new imports for python such as time and statistics (I did not use statistics because its implementation was not intuitive and ended up being unnecessary). Most importantly, I learned it is important it is to have the correct computations for the Runge Kutta method to preserve the answers accuracy.