Session_06 activities worksheet

Online handouts: listings for diffeq_test.cpp, diffeq_routines.cpp, eigen_tridiagonal_class.cpp.

The Chapter 6 notes have an introduction to algorithms for integrating differential equations. As part of these activities, we'll go over the basic ideas by example, using the routines in session06.zip, in preparation for looking at "Anharmonic Oscillations" and "Differential Chaos in Phase Space".

Your goals for this Session:

- Take a look at using a C++ class to "encapsulate" the GSL functions in the "eigen_tridiagonal" code.
- Have a first look at parallel processing with OpenMP.
- Run a code to integrate a simple first-order differential equation using Euler's and 4th-order Runge-Kutta algorithms, then modify it so you can analyze the errors.
- Extras: Add code for 2nd-order Runge-Kutta. Adapt the code to treat the 2nd order F=ma problem.

Please work in pairs (more or less). The instructors will bounce around the room and answer questions.

A) Wrapping GSL Functions: eigen_tridiagonal_class

The code eigen_tridiagonal_class.cpp, together with GslHamiltonian.cpp and GslHamiltonian.h, are a rewrite of eigen_tridiagonal.cpp from Session_05. A C++ class called "Hamiltonian" hides all of the GSL function calls from the main program. Minimal changes were made to the code for clarity (so it is not optimal!).

1. Look at the eigen_tridiagonal_class.cpp printout and compare to the eigen_tridiagonal.cpp printout from Session_05. If this is your first (real) exposure to a C++ class (or if you forget what you used to know), there will be confusing aspects. Look at the motivation in the Session_06 notes. A detailed guide to the implementation will be given in the Session_07 notes. For now, identify what has happened to each of the GSL function calls. *In what ways is the new version better? (For experts, what would you do differently?)*

   → The GSL function calls are now in the class in GslHamiltonian file, encapsulated within the class itself. This is better by making the code easier to read and debug, by being separated into different sections based on the shared usage. Instead of having all of the GSL calls in one file with the main function, they are now separated out, which is better for debugging and testing.

2. Verify that the code still works. Try adding a loop over the matrix dimension N. *Does it work?*

   → Yes, the code still works, and it works if you add a loop over matrix dimension N.

3. (Bonus) *What additional functions might you define in the Hamiltonian class? What other classes might you define?*

   →There are maybe a few additional functions I would define that would be useful for the physics. It would be nice to have a function that calculates an expectation value or the energy gap, or maybe a function that loads the harmonic oscillator directly instead of manually doing it. It might be nice to define more classes for the potential so we can swap between different potentials.

B) Introduction to Parallel Processing with OpenMP

OpenMP (not to be confused with Open MPI!) implements parallel processing when there is *shared memory*, as is common these days with multi-core hardware. If you have a dual-core processor that means that in principle you can run your program twice as fast by running two "threads" of your code simultaneously on the two cores. If you have more cores available, in principle the speed scales with the number of cores. One way to achieve this in practice is to use OpenMP. We'll use a simple example written by Chris Orban to illustrate how it works.

1. To do this section on OSC (and only for this section!), please start a new cardinal desktop session at OSC (you can leave the other one open), but this time ask for 8 cores (still only 1 node). Once you are in a terminal on that new desktop session, say `echo $SLURM_NTASKS` to verify that you indeed have access to 8 cores in parallel. If you are using your personal Mac instead use: About This Mac --> System Report --> Hardware to look up the number of cores. *How many cores do you have?*

   → I am not using my personal mac, but it has 14 cores. The OSC has 8 cores, as expected.

2. Look at the program simpson_cosint_openmp.cpp in an editor or in the printed copy. The key openmp features are the omp include statement and the #pragma omp statement, which is an instruction to the compiler. *What part of the program is executed in parallel?*

   → The part of the program executed in parallel is the for loop that actually calculates Simpson's rule.

3. Let's compare using one and two threads. There is a built-in timer in the program, but run the program with time ./simpson_cosint_openmp, which will automatically print some timing info on the cpu time *and* the wall time after you run the program. Compile the program with g++ following the instructions in the comments, and then run it. *Record the "num_time", the CPU time, and the wall time.* (In tcsh, these are the first and third numbers; in bash, these are the second and first numbers.)

   → num_time = 24.4299, CPU time = 0m47.915s, wall time = 0m24.456s.

Then change from two threads to one thread by modifying the omp_set_num_threads command in the code, recompile it, and re-run. *Record the numbers again. Why do you think the CPU time is about the same but the wall time differs? Is the parallelization working?*

→ num_time = 48.0766, CPU time = 0m47.898s, wall time = 0m48.102s. The CPU is time is the same because it relates to the number of calculations needed to integrate the function. With one thread, one core works for 48s for a CPU time of 48s, while two cores have each core working for 24s, for a total CPU time of 24s+24s=48s. The wall time differs because that's the real time that actually passes on our clock, which is 48s for one thread while that's half the time for two threads.

4. Now try all the cores you have access to (this should be 8 on OSC but can vary if you asked for a different number of cores or use your own computer - it should be the answer to question B.1). *How well does the program scale? (E.g., compare (num_time for 1 core)/(num_time for n cores) to n.) Why is it not perfect scaling?*

   →The comparison between num_time of 1 core and 8 cores is ~7.74 while it is ~1.97 for 1 core to 2 cores. This is a near-linear scaling (2 cores decreases wall time by a factor of 2, 8 cores decrease time by a factor of 4). The scaling is not exactly perfect because there are some bottlenecks like thread synchronization and memory bandwidth, for example, which slows down the scaling as the threads "talk" to each other.

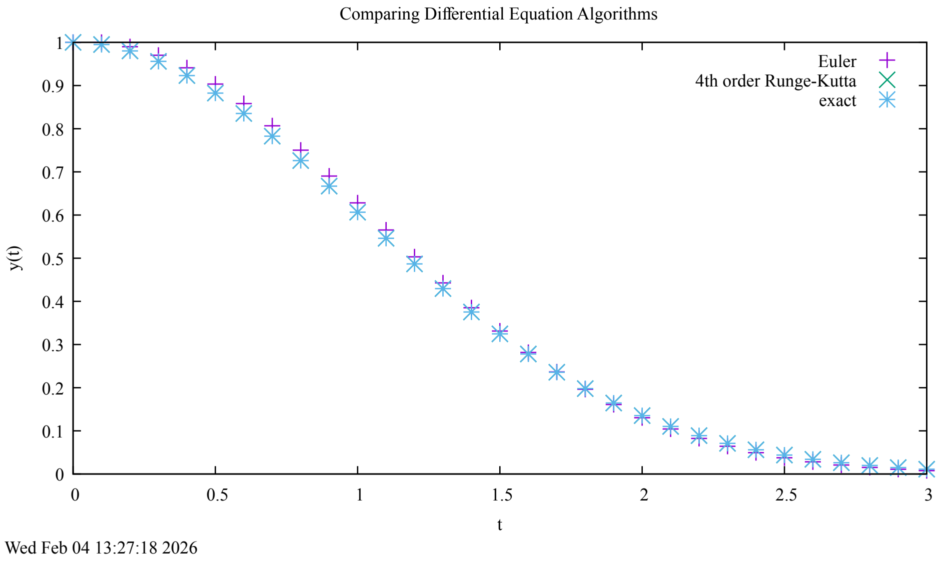
5. If you are on OSC, log out of the desktop session with the multiple cores and go back to your single core session.

C) Integrating a First-Order Differential Equation

The code diffeq_test.cpp calls the differential-equation-solving routines in diffeq_routines.cpp ("euler" and "runge4") to integrate a series of coupled differential equations (but we'll start with a single equation). Functions for the right-hand-side of a first order differential equation (dy/dt = rhs[y,t]) and the exact y(t) [called "exact_answer"] for a specified initial condition are also defined in this file. There is also a header file diffeq_routines.h.

1. Look through the Session_06 Notes for a quick overview of differential equation solving.
2. Use make_diffeq_test to compile and link diffeq_test. Run the program to generate diffeq_test.dat and look at it in an editor. The gnuplot plotfile diffeq_test.plt generates comparison plots of the integrated function from the output in diffeq_test.out. Load this plotfile in gnuplot:
   gnuplot> load "diffeq_test.plt"
   and examine the result. *What can you conclude at this point?*

   →I can conclude that the differential equation integrators get close to the exact result, but there is some deviation from the exact by the Euler method. Looking at the data file and the plot, we can see that RK4 gets us very close to the exact result, so much so that we can barely differentiate them on the plot. The plot is below.

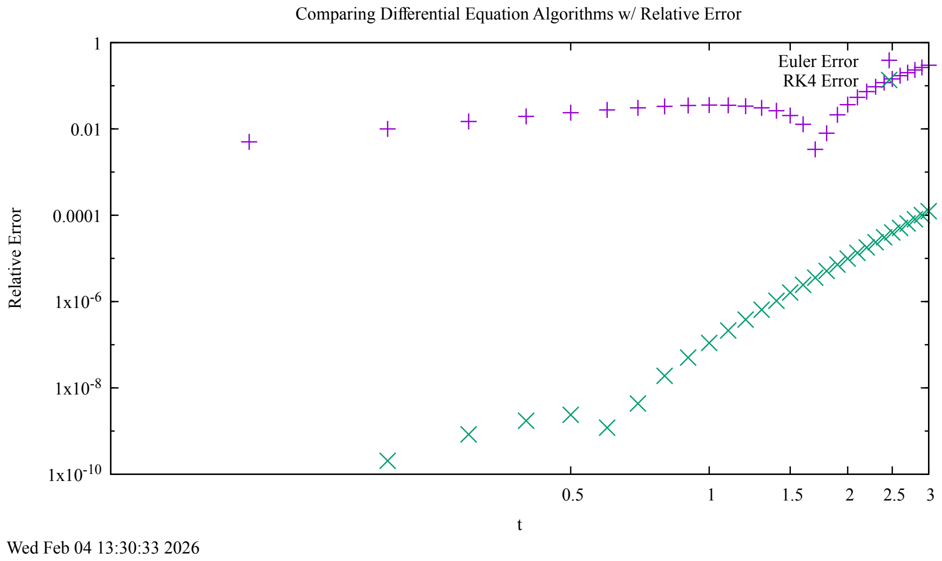


3. Look at the printout for diffeq_test.cpp and diffeq_routines.cpp and compare to the Session_06 notes to figure out what is going on. The codes follow the notation in the notes. At present there is only one equation (first order), so only y[0] is used. *What is the differential equation being integrated?*

   →The specific differential equation being solved is dy/dt = -a*t*y with a=1.0, which corresponds to an analytical solution of y=b*e^(-a*t^2/2).

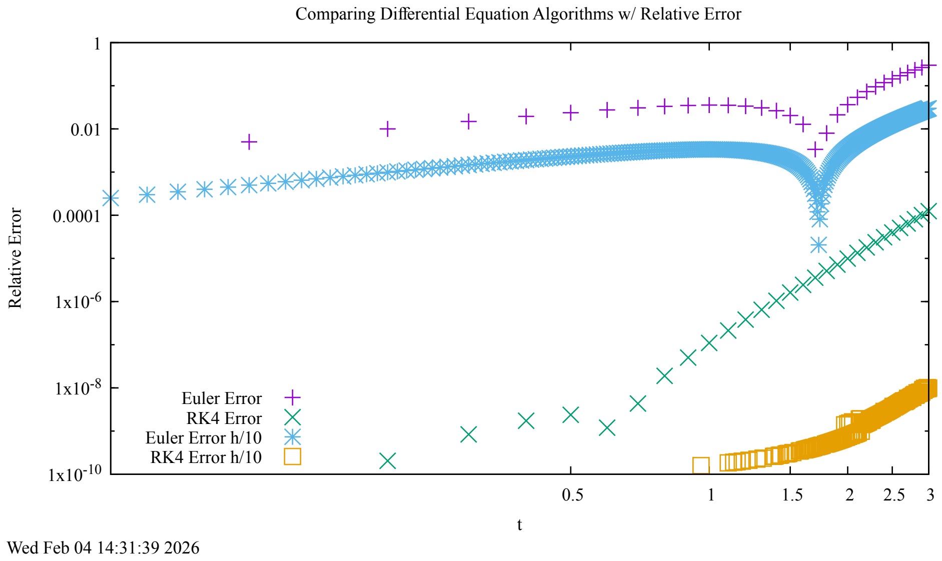
4. Modify the diffeq_test.plt file to plot the **relative** error at each value of t. (Modify the plot file and NOT the program; see the gnuplot handout on plot files for an example of how to do this.) As usual in studying errors, a log-log scale will be useful. The first point at t=0 may get in the way. Use "set xrange [?:?]" in gnuplot (where you fill in the ?'s) to avoid this problem. *What can you say qualitatively about the errors?*

   →The plot is below. Qualitatively, we can see that the relative error for RK4 is many orders of magnitude less than Euler, which we saw from the earlier plot, based on the overlap between RK4 and the exact answer. We can see that both algorithms exhibit an accumulation of error over time as well, with Euler almost approaching 100% relative error. There are also differing interesting dips in the relative error for both functions at different times.



5. Now generate and plot results for a second value of h (your plot should have both values of h, so think about how to best do this). You'll want it use something like 1/10 the value, so it's easy to check the effect (e.g., if the difference goes like h^n, then you'll see 10^n, which is easy to see on a log-log plot). When the local error (for each step h) adds **coherently**, then the "accumulated" or "global" error for a given algorithm at t_f scales like N_f*(local error) = (t_f/h)*(local error). You can verify for Euler's algorithm, for which the local error to be h^2 (see notes and excerpt), that the global error is, in fact, h. *What is the local error for 4th-order Runge-Kutta according to the graph?*

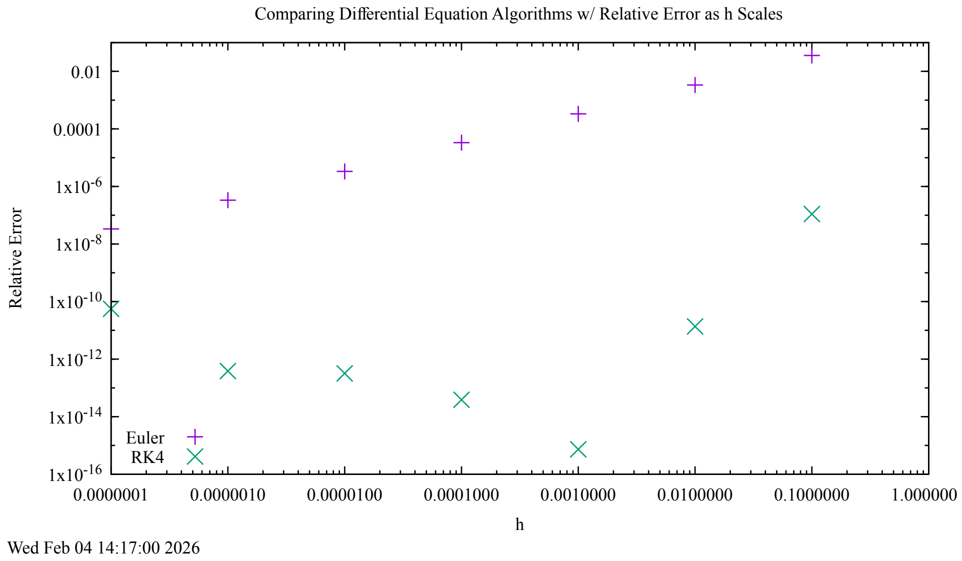
→The plot is below. We see that dividing the step by 10 drops Euler error by a factor of 10 (makes sense as it's first order) and drops RK4 by a factor of 10^4 (makes sense, it's fourth order). This graph shows the global error as a function of time t. To get the local error we take the global error and multiply by (h/tf). We see that the global error for Euler is h and for RK4 is h^4, so by multiplying by h, we see that the local error for the Euler method is h^2 and for RK4 is h^5.



6. Next, check how the accumulated error at one value of t scales with h for the two algorithms. Take t=1, for example. You'll need to modify the code to output the results for y(t=1) for a range of h's (think logarithmic!). Some things to be careful of:
    - Print out enough digits. For small h's, 9 digits is not enough (try 16).
    - The most common problem is printing out y_rk4[0] and exact_answer(t,params_ptr) at two different points. (Note: it is not important that the t used is exactly the same for every h, but it must be the same for the exact and rk4 result for each h.) Look at your output file!

There should be different regions of the error plot, as we've seen before. *Interpret them.*

→The plot is below. For larger values of h (h>0.1), we see that the error is dominated by the algorithms' approximation method, with a straight line indicating a power law. For Euler, the slope appears to be approximately 1, so when h decreases by a factor of 10, the global error decreases by a factor of 10. For RK4, we see the slope appears to be approximately 4, which makes sense for a fourth-order method. The left side is dominated (h<0.1) by round-off error. The number of steps at this h are so large that tiny errors from 16-digit precision accumulate and dominate the total error. The Euler method is so inaccurate that it remains in the discretization-dominated regime for longer than RK4 and doesn't even hit the round-off floor due to bad approximation error.



Given your results, how would you choose a step size for 4th-order Runge-Kutta? (Hint: How do you explain the behavior of the error for small h?)

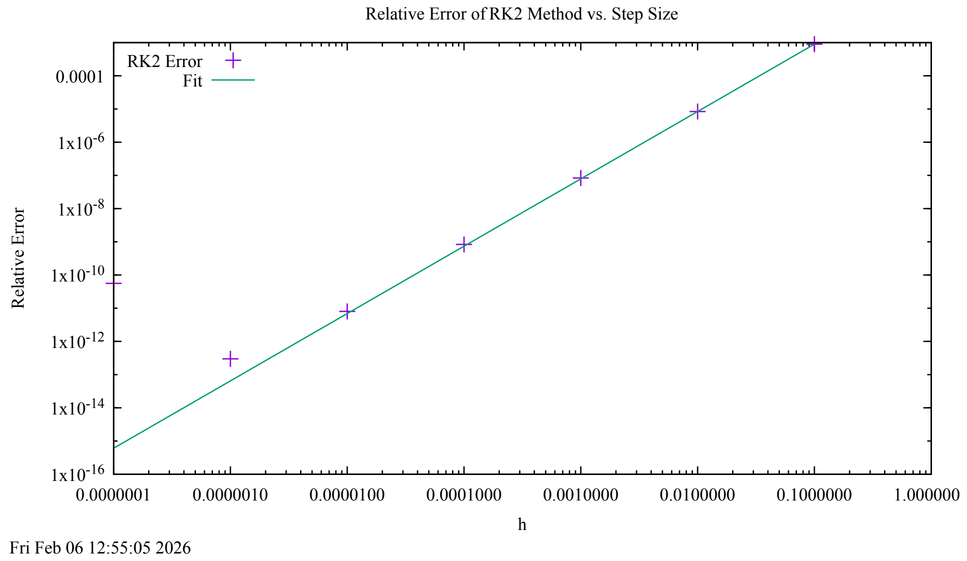
→Given the plot, a step size for RK4 would be about 10^-3. Making h any smaller or larger than this increases relative error, and you aren't gaining accuracy. We can explain the behavior of the error for small h as the transition from discretization error to round-off error, as stated prior.

D) Extra: 2nd Order Runge-Kutta Algorithm

This exercise is intended to verify that you understand the meaning of the Runge-Kutta algorithms and how they are implemented in C++. *(Most likely for a future Homework.)*

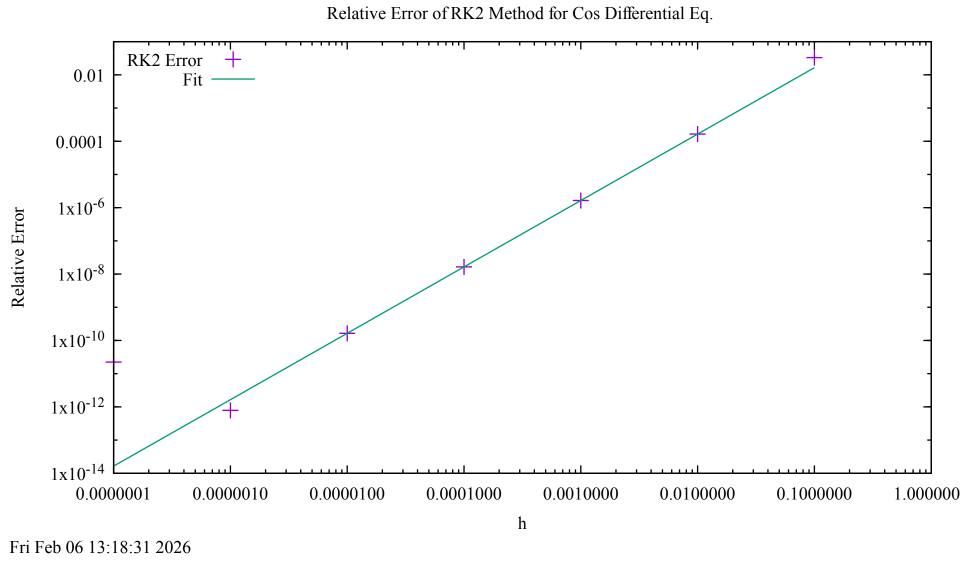
1. Add a third diffeq routine to the code, which implements the 2nd-order Runge-Kutta algorithms described in the Session_06 notes.
2. Check the scaling of the error with h, i.e., is the error proportional to a power of h?. *What is the power?*

   →Below is a plot on how the error scales with h. The error after fitting is proportional to h^2, which is what we expect given that RK2 is a second order method.



3. Try a different differential equation, such as dy/dt = 2*cos(2pi*t). *How do the errors compare to your first equation?*

→Below is the plot for the new differential equation. We still get that the error scales as h^2. The general error is higher than the previous plot, which makes sense because the cosine function has larger higher-order derivatives (due to 2pi multipliers), leading to a larger truncation error value, even though the order of convergence remains the same.



E) Extra: Integrating a 2nd-Order Differential Equation

Second-order differential equations are treated by writing them as two coupled 1st-order equations, as described in the Session_06 notes. We'll try this out for a simple example, which we'll generalize later to look at chaotic behavior. *(Most likely pushed to Session 7.)*

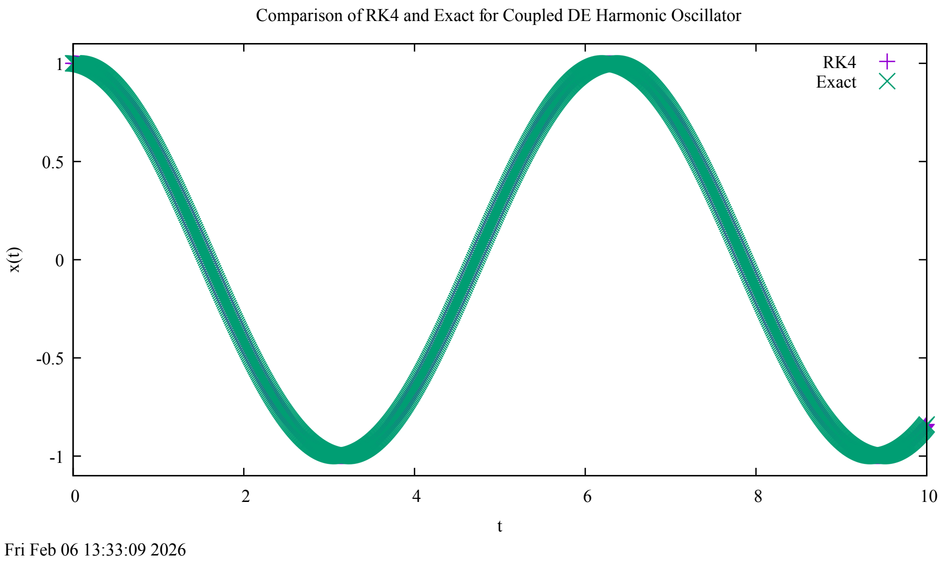
1. Consider the differential equation for a simple, undriven harmonic oscillator [e.g., a mass m on a spring with constant k: $d^2x/dt^2 + (k/m)x = 0$]. *What is the general solution in terms of the initial position and velocity?*

   →The general solution in terms of the initial position x0 and the initial velocity v0 is given by x(t) = x0*cos(w0*t) + (v0/w0)*sin(w0*t), where w0 is the natural frequency given by sqrt(k/m).

2. *Rewrite the differential equation as two coupled 1st-order equations [for x and v=dx/dt]*. Use units in which the oscillator mass and spring constant are equal to one. Take the initial position to be 1.0 and the initial velocity to be zero.

   →The two coupled first order equations would be the following result. Let y0=x and y1=v=dx/dt. Then dy0/dt = y1=v and dy1/dt=-w0^2*y1. If we are taking the natural units given, then this becomes dy0/dt=y1 and dy1/dt=-y0, with y0(0)=1.0 and y1(0)=0.0.

3. Modify a copy of diffeq_test.cpp to use the runge4 subroutine to calculate this oscillator for times t=0 to t=10. You'll need to change N, modify rhs to consider both i==0 and i==1, put the exact answer in exact_answer, and change the limits of t appropriately.
4. Plot the result and the exact result for comparison. *What do you conclude?*
   →We see from the plot below that the exact result and the RK4 are effectively on top of each other. This means the method does a good job of solving the coupled system and that the relative error should be small. The method does a good job of minimizing error even at t=10.0, which was not necessarily true for the other functions we solved for.



F) Reflection

Write down which of the six learning goals on the syllabus this worksheet addressed for you and how:

→This worksheet helped with many of the learning goals. Alongside being able to write clear computer code and learning more tools for computational solutions, I now better understand the limitations of computational solutions over time steps and step sizes. I also have a better understanding how to implement DEs and coupled ODEs of physics concepts directly into computer code (instead of using some wrapper or library).