

## 480/905: Activities 6

*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 and answer questions.

### 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 5. 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 5. 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 6 notes. A detailed guide to the implementation will be given in the Session 7 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?)*

GSL Allocations & workspaces in constructor. Set element is same as before, except you don't need to give pointer. Eigen stuff gets eigen stuff & sorts. Get gets the things. The class essentially just cleans up GSL's messiness, by putting all necessary calls for an action inside the class. Honestly I notice nothing terrible about the class, it should run pretty well, what could be improved? Memory stuff? Either way it is a good "one step" process to cleaning up sections of the code.

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

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

The class already does everything needed for `eigen_tridiagonal`... so what to add? I'd add a general way to save the hamiltonian matrix and eigenvalues/vectors so you could access them later because they can be very useful, but other than that idk.



# 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. Your computer almost certainly has multiple cores. *How many cores are there (hint: Google is your friend if it's not immediately clear how to get this information from your computer.)?* 6 Cores.  
12 processors.

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?*

Only the for loop on line 79. It looks like it should also be reasonably parallel. Does the integral for multiple frequencies & amplitudes. (X)

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 = 6.62333 s

CPU Time = 13.156 s

Wall-time = 6.661 s

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 - 12.8665

cpu - 12.844

Wall - 12.902

Yeah it is working. CPU time adds everything from every "working" CPU. It is cumulative.

4. Now try to use all the cores you can. *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?*

8.6711x for 12 Processors.

It takes effort to decide how to divvy up the job and gather the results, in simple terms.



## 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 ( $dy/dt = \text{rhs}[y,t]$ ) 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 Activities 6 Background Notes for a quick overview of differential equation solving.
2. Download and unzip `session06.zip`.
3. 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:

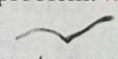
```
gnuplot> load "diffeq_test.plt"
```

and examine the result. *What can you conclude at this point?*

4 RK is pretty much exact. Looks like it is a gaussian.

4. Look at the printout for `diffeq_test.cpp` and `diffeq_routines.cpp` and compare to the Activities 6 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?*

$$\frac{dy}{dt} = -xty$$

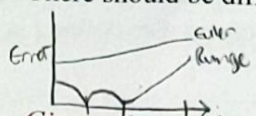
5. Modify the `diffeq_test.plt` file to plot the **relative** error at each value of  $t$ . (Modify the plot file and NOT the program; see the gnuplot handout on plot files for an example of how to do this.) As usual in studying errors, a log-log scale will be useful. The first point at  $t=0$  may get 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?* Both kinds have a  shape, but 4RK is way better. 4RK has a local minimum at  $t \approx 0.6$ , Euler around  $t \approx 1.6$ .
6. 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  $N_f \cdot (\text{local error}) = (t_f/h) \cdot (\text{local error})$ . 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 accumulated error is  $\propto h^4$ , so the local error of 4RK must be  $h^5$ , or  $\mathcal{O}(h^5)$ ?
7. 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).



10-14

- 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.*



Euler error scales consistently. RK4 scales down, then hits a minimum where it is dominated by some kind of calculative 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  $h$ ?)*

An adaptive method for Runge-Kutta is very common, but if you want to stay static, choose an  $h$  that gives off reasonable error at the point of greatest change in the RHS.

### Extra: 2nd Order Runge-Kutta Algorithm

$$y = ax + b \quad x = (\log(a))y + \log(b) \\ y = ae^{bx} \quad b = h^2 \quad y = bh^a$$

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 assignment.*)

1. Add a third diffeq routine to the code, which implements the 2nd-order Runge-Kutta algorithms described in the Activities 6 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?*

Scales  $\propto h^2$

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

$y = \frac{1}{2\pi} \sin(2\pi t) \quad y_0 = 0$   
error is more consistently, but watch out because exact = 0 at  $t=1$ !

### 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 Activities 6 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 if few people reach this today.*)

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?*

$$x = x_0 \cos(\sqrt{\frac{k}{m}} t) + v_0 \sin(\sqrt{\frac{k}{m}} t) \\ x'' = -\frac{k}{m} x$$

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.

$$\frac{dx}{dt} = v \quad \frac{dv}{dt} = -x \quad x(0) = 1.0 \quad v(0) = 0$$



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?*

The result is pretty much perfect. ~~Isn't~~ Isn't YRK  
just great usually?