*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?)*

**It does not takes many lines of code to find the eigenvalues and eigenvectors, in other words, the newer version is more efficient**

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

**Yes it does**

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

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.)?*

**My computer has 6 cores**

1. 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?*
2. #pragma omp parallel for
3. for (i = 0; i < numi; i++)
4. {
5. k[i] = kmin + i\*(kmax-kmin)/(numi-1);
6. result[i] = simpsons\_rule (100001, k[i], lower\_limit, upper\_limit,
7. &my\_integrand);
8. }
9. end = omp\_get\_wtime();     // get the end time

**The above part was the one executed in parallel**

1. 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 = 9.99635 s**

**Cpu\_time = 19.718 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\_time = 19.1601 s**

**Cpu\_time = 19.125 s**

**The parallelization is working because when 2 threads are run, the wall time is shorter, therefore showing efficiency in running the code**

1. 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?*

**The num\_time goes down as the number of threads increases. It’s not perfect scaling because the gaps between the num\_time is not even as the number of threads increases**

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 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?*

**The graph has a sigmoid figure**

1. 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?*

**The differential equation being integrated is the Euler method and the 4th order Runge-Kutta Differential Equation**

1. 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 4th order Runge-Kutta equation and the Euler method have negative optimal y-values**

1. 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 hn, then you'll see 10n, 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 tf scales like Nf\*(local error) = (tf/h)\*(local error). You can verify for Euler's algorithm, for which the local error to be h2 (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?*

**According to the graph, the local error would be around 0.3**

1. 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 Euler method has the error dying down in sort of a parabola-like pattern, while the 4th-order Runge-Kutta has sort of a linear looking-slope**

*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?)

**I would choose a step size that is larger than 0.01 because it would provide a much better picture on how the points scale up/down**

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

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: d2x/dt2 + (k/m)x = 0]. *What is the general solution in terms of the initial position and velocity?*
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.
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?*