PHYS 352 – Midterm

Due: Tues., Feb. 15, midnight

Submit code solutions and the .png's, .sh's, .plt's requested below. **Source files for your main executables** should be named "midterm_X.c", where "X" corresponds to the question numbers. **Your ODE solver code** should be named euler.c, euler.h, *etc.* and placed in the usual `ode`directory structure. Likewise, **you physics specific code** should live in a corresponding `physics` directory structure. Include your name enclosed in C comment tags (ie: `/*YourName*/`) at the top of each program. Create a zip archive containing all of your files, name it "midterm_YourLastName.zip" (with the appropriate name replacement) and copy it to your `homework` directory by midnight on Tuesday, Feb. 15.

1. **Trade-Offs (10 pt.)**

   (a) Use the euler, rkO2, and rkO4 techniques to solve for the evolution of two radioactive species with coupled decays. Take the time constants of both decays to be $1.0s^{-1}$. Begin with an initial population for the A (B) species of 1000 (0). Explore how the use of finer-grained simulation steps (while keeping the overall time period fixed) impacts the accuracy of your numeric simulation. Consider 6 different time steps :

   $(0.05s, 0.01s, 0.005s, 0.001s, 0.0005s, 0.0001s)$

   Use 100 simulation steps with the time step of $0.05s$, thereby simulating a total period of $5s$. For subsequent time steps, adapt the total number of simulation steps to maintain a $5s$ period. For each of the 6 time steps, plot the maximum relative error of the simulated population of species A (with respect to the analytic solution) *vs* the total number of simulation steps. Include results from each of the 3 ODE methods in a single plot. It will be best to use log scale for both axes. Comment on your results.

   (b) Now investigate the computational cost of finer-grained simulation steps using `perf`. First, be sure to disable the print outs in your code, as these are fairly expensive and can mask underlying performance differences in the ODE algorithms. You can disable by commenting out the printouts, by using a conditional that checks a user-input value, or similar.

   Run your executables through `perf` using the `stat` directive. Do this by issuing the following at the command line :

   ```
   perf stat <your executable> <your arguments>
   ```

   This will produce output similar to :

   ```
   $ perf stat  ~/hello.exe

    Performance counter stats for '/home/khahn/hello.exe':

             1.18 msec task-clock:u             #    0.732 CPUs utilized
                0      context-switches:u       #    0.000 /sec
                0      cpu-migrations:u         #    0.000 /sec
   ```

```
          116      page-faults:u             #   98.002 K/sec
    1,609,096      cycles:u                  #    1.359 GHz
    2,155,237      instructions:u            #    1.34  insn per cycle
      339,786      branches:u                #  287.067 M/sec
       10,428      branch-misses:u           #    3.07% of all branches

  0.001616988 seconds time elapsed

  0.000877000 seconds user
  0.000884000 seconds sys
```

We are interested in the `instructions` count. Plot the maximum relative error for each time step value, for each of the euler, rkO2, and rkO4 methods, *vs* the instruction count . You could additionally plot total simulation steps *vs* the instruction count. How does computational cost compare with simulation granularity in your results?

(c) We have thus far neglected optimization strategies offered by the `C` compiler. Re-run your `perf` studies from the second part of this problem, but now with optimization enabled. Do this by adding the commonly used `-O2` optimization flag to `gcc` when compiling. Note you will have to recompile your `ode` and `physics` libraries with `-O2` as well in order for this to take effect. Compare performance plots for the compiler-optimized studies *vs* your original results, and comment.

2. **Beats (10 pt.)**

We briefly discussed the application of FFTs to beat phenomena in class. Ideally I would have simulated an infinite string for the studies I showed. Instead, I took a short-cut; I simulated a long string with free endpoints, but focused only on a short section in the interior, and only for a short period of time. This allowed me to neglect the effects of reflected waves from the endpoints. A cleaner way to simulate an infinite string is to use periodic boundary conditions that map the endpoints of the simulation to a time-varying sine function.

(a) Implement a time-dependent initialization function :

```
void initialize_with_free_wave(double ** yarr, int tlen, int xlen,
                               double k, double dx,
                               double omega, double dt );
```

This function should set the values of the amplitudes at spatial indices 0 and $xlen - 1$ to a sinusoid : $\sin(kx - \omega t)$. The evolution of amplitudes at spatial points between these indices will be governed by the `propagate` function discussed earlier.

Use the new initialization function to simulate two traveling waves. Take $k_1 = 4$ and $k_2 = 4.4$, and use the same simulation constants from class, *i.e.*

```
const double c  = 300;
const double dx = 0.01;
const double dt = 0.0000333333333;
const double r  = c*dt/dx;
```

Your 2D array should span 800 elements in the time dimension, and 1600 elements in the spatial dimension. Run simulations of the two traveling waves and plot their evolution in an animated gif.

(b) Collect time-series data at the spatial index position 5 for each of the waves. Discard data from the time-series prior to the arrival of the wavefront at that position. Perform FFTs using the time-series data, and compare to the expected frequencies.

(c) As discussed in class, create a superposition by summing the k_1 and k_2 waveforms at each time index. Plot the evolution of this waveform as an animated gif. Perform an FFT on time-series data at spatial index 5, and compare to the results of the individual waves.

(d) Another means of achieving superposition is to use additive periodic boundary conditions. For this, implement a new initialization routine :

```
void initialize_with_free_wave_sum(double ** yarr, int tlen, int xlen,
                                   double k1, double k2, double dx,
                                   double omega1, double omega2, double dt );
```

Now, rather than manually summing wavefunctions at each interior point, allow them to evolve simply given the new boundary conditions. Create an animated gif of the evolution of this waveform, perform a FFT, and compare with your preceding results.