Astro 585 Lab/Homework #2 (Assertions, Benchmarking, Plotting)

Benchmarking. Julia provides several tools for measuring code performance. Perhaps the simplest way is using the @time or @elapsed macro which can be placed prior to a command, like "@time randn(1000)". The @time macro prints the time, but returns the value of the following expression. The @elapsed macro discards the following expression's return value and returns the elapsed time evaluating the expression. For finer control, you can investigate the tic(), toc() and toq() functions (see here in the Julia manual for details).

1. For several common mathematical functions, calculate how many million evaluations your computer performs per second. Try at least three arithmatic operations, two trig functions, and a logarithm. For example,

```
N = 1000000;
println("rand:  ", 1./(@elapsed x = rand(N)));
println(".+:    ",1./(@elapsed x.+x));
```

2. For the text below, we'll consider the log_likelihood function and synthetic data from Lab/HW #1, which was probably very similar to the following.

```
srand(42);
Nobs = 100;
z = zeros(Nobs);
sigma = 2. * ones(Nobs);
y = z + sigma .* randn(Nobs);
normal_pdf(z, y, sigma) = exp(-0.5*((y-z)/sigma)^2)/(sqrt(2*pi)*sigma);

function log_likelihood(y::Array, sigma::Array, z::Array)
  n = length(y);
  sum = zero(y[1]);
  for i in 1:n
   sum = sum + log(normal_pdf(y[i],z[i],sigma[i]));
  end;
  return sum;
end
```

2a) Write a function, calc_time_log_likelihood(Nobs::Int, M::Int = 1), that takes one required argument (the number of observations) and one optional argument (the number of times to repeat the calculation), initializes the y, sigma and z arrays with the same values each call (with the same Nobs), and returns the time required to evaluate the log_likelihood function M times.

2b) Time how long it takes to run log_likelihood using Nobs=100 and M = 1. Time it again. Compare the two. Why was there such a big difference?

2c) If you were only calling this function a few times, then speed wouldn't matter. But if you were calling it with a very large number of observations or many times, then the performance could be important. Plot the run time versus the size of the array. The following example demonstrates how to use the PyPlot module, "comprehensions" to construct arrays easily, and the map function to apply a function to an array of values.

```
using PyPlot
n_list = [ 2^i for i=1:10 ]
elapsed_list = map(calc_time_log_likelihood,n_list)
plot(log10(n_list), log10(elapsed_list), color="red", linewidth=2, marker="+", markersize=12);
xlabel("log N"); ylabel("log (Time/s)");
```

2d) Does the run time increase linearly with the number of observations? If not, try to explain the origin of the non-linearity. What does this imply for how you will go about future timing.

Assertions: Sometimes a programmer calls a function with arguments that either don't make sense or represent a case that the function was not originally designed to handle properly. The worst possible function behavior in such a case is returning an incorrect result without any warning that something bad has happened. Returning an error at the end is better, but can make it difficult to figure out the problem. Generally, the earlier the problem is spotted, the easier it will be to fix the problem. Therefore, good developers often include assertions to verify that the function arguments are acceptable.

2e) Time your current function with Nobs = 100 & M = 100. Then add @assert statements to check that the length of the y, z and sigma arrays match. Retime with the same parameters and compare. What does this imply for your future decisions about how often to insert assert statements in your codes?

2f) Write a version of the log_likelihood and calc_time_likelihood functions that take a function as an argument, so that you could use them for distributions other than the normal distribution. Retime and compare. What does this imply for your future decisions about whether to write generic functions that take another function as an argument?

2g) Write a version of the log_likelihood function that makes use of your knowledge about the normal distribution, so as to avoid calling the exp() function. Retime and compare. How could this influence your future decisions about whether generic or specific functions?

2h) Optional (important for IDL & Python gurus): Rewrite the above function without using any for loops. Retime and compare. What does this imply for choosing whether to write loops or vectorized code?

2i) Optional (potenetially useful for IDL & Python gurus): Try to improve the performance using either the Devectorize and/or NumericExpressions modules. Retime and compare. Does this change your conclusions from part 2h? If so, how?

3. Consider integrating the equations of motion for a test particle orbiting a star fixed at the origin. In 2D, there are two 2nd order ODEs:

d^2 r_x/dt^2 = a_x = -GM r_x/r^3
d^2 r_y/dt^2 = a_y = -GM r_y/r^3,

where r = sqrt(r_x^2+r_y^2) and a_x and a_y are the accelerations in the x and y directions. Numerically, it is generally easier to integrate 1st order ODEs, so we transform these into:

d v_x/dt = a_x = -GM r_x/r^3
d v_y/dt = a_y = -GM r_y/r^3
d r_x/dt = v_x
d r_y/dt = v_y

Euler's method is a simplistic algorithm for integrating a set of 1st order ODEs with state stored in a vector x:
r(t_n+1) = r(t_n) + dt * dr/dt|_(t_n)
v(t_n+1) = v(t_n) + dt * dv/dt|_(t_n)

3a) Write a function integrate_euler!(state,dt, duration) that integrates a system (described by state) using Euler's method and steps of size dt for an interval of time given by duration. It would probably be wise to write at least two additional functions. Have your function return a two dimensional array containing a list of the states of the system during your integration

3b) Use the above code to integrate a system with an initial state of (r_x, r_y, v_x, v_y) = (1, 0, 0, 1) for roughly 3 orbits (i.e., 3*2pi time units if you set GM=1) using approximately 200 time steps per orbit. Inspect the final state and comment on the accuracy of the integration.

3c) Plot the trajectory of the test particle (e.g., x vs y). Is the calculated behavior accurate?

3d) Write a new function integrate_leapfrog!(state,dt, duration) that is analogous to integrate_euler, but uses the "leapfrog" or modified Euler's integration method:
r(t_n+1/2) = r(t_n) + dt/2 * dr/dt|_(t_n)
v(t_n+1) = v(t_n) + dt * dv/dt|_(t_n+1/2)
r(t_n+1) = r(t_n+1/2) + dt/2 * dr/dt|_(t_n+1/2)

3e) Repeat the integration from part 3b, but using the integrate_leapfrog code. Inspect the final end state and make a similar plot. Describe the main difference in the results and explain on what basis you would choose one over the other.

3f) Time how long your code for a similar integration of 100 orbits. How long would it take to integrate for 4.5*10^9 orbits (e.g., age of Earth)?

3g) Repeat the integration from part 3e, but using a larger dt. Inspect the final end state. How much less accurate is the integration that used a larger dt?

3h) Make a log-log plot of the accuracy of the integration versus the integration duration for durations spanning at least a few orders of magnitude. Estimate the slope (by eye is fine). What are the implications for the accuracy of a long-term integration?

3i) List a few ideas for how you could accelerate the integrations. Comment on how likely each of your ideas is to make a significant difference. (If you try it out, report the new time for 100 orbits, so we can compare.) Discuss any drawbacks of each idea to accelerate the integrations.

4. Consider a modern laptop CPU with 4 GB (=4*2^30) of usable memory and capable performing 20 Gflops/second. Assume it uses 8 bytes of memory to store each floating point number (“double precision”). [Based on Oliveira & Stewart’s Writing Scientific Software, Chapter 5, Problem #6.]

4a) What is the largest matrix that the above computer could fit into its available memory at one time?

4b) If we use LU factorization to solve a linear system, estimate how long would it take to solve the maximum size linear system that would fit into memory at once. You may assume the computation is maximally efficient, the computer reaches peak performance and the LU decomposition requires ~(⅔)n^3 flops.

4c) For a modern laptop, does memory or compute time limit the size of system that can be practically solved with LU decomposition?

4d) For real life problems, what other considerations are likely to limit performance?

4e) How could one practically solve even larger systems?