Astro 585 Lab/Homework #4 (Performance Comparisons 1/2: Profiling, Loops vs Vectorize vs Map, Branching)

1. Profiling

Julia comes with a sampling (or statistical) profiler. To profile a function, simply preface the function call with the @profile macro. You can do this for multiple function calls to profile a multi-line section of code or to build up statistics. To see the results run Profile.print(). To clear the profile statistics, run Profile.clear(). For more information about how to interpret the results, see the Profiling section of the Julia manual.

1a) Pick at least two non-trivial functions to profile. If you've already know of function(s) that you'll be using for your class project, then I'd suggest that you profile them. If you haven't, then I'll suggest that you profile two implementations of the leapfrog integrator from Lab 2. (Alternatively, you could also use the log_likelihood function from Lab 2). You'll find my implementation in HW4_leapfrog.jl. For the second implementation, you can use either your implementation or an alternative implementation that I've prepared (based on a student's submission) in HW4_leapfrog_student.jl. If you use your own implementation, be sure to test it, so it gives nearly identical results to mine.

1b) For each functions to be profiled, inspect the code and try to predict which 2-4 lines of code are taking the most time. Estimate what fraction of the time, you think those lines will take.

1c) Profile each function. Make sure that the profiler has collected enough samples that the results are statistically useful (say at least a thousand samples). If you need to increase the number of samples, you could: 1) call your function with a longer integration time/larger dataset, 2) call your function several times (without clearing the profiler between calls), or 3) change the frequency at which the profiler samples your code (using Profile.init()).

1d) For each of the functions profiled above, identify the portion(s) of your code the computer is spending most of it's time. Explain any differences from your predictions.

1e) Is there anything you could do to improve the performance of the sections of code identified above? If so, what? How much of a difference do you think it would make in the overall code run time? Given the likely effort required, would it be worth your time to try to speed-up that section of the code?

1f) Pick at least one idea for optimizing the code to try (whatever you think is most likely to help the performance with a reasonable amount of coding). Implement your idea. Test that it still gives the same answer.

1g) Benchmark and profile the code for your attempted optimization. Is there a difference big enough that it's consistently faster or slower than the original implementation? Has the most time consuming part of the code changed? Or stayed the same? Describe your best hypothesis for explaining the behavior you observe.

2. Loop vs "Vectorized" vs Map vs MapReduce vs "Devectorized"

In many applications, there is some computation that is repeated many, many times with slightly different measurements/initial conditions/model parameters. There are several ways to tell the computer to perform the calculation many times, each with it's own advantages and disadvantages. In this exercise, you will compare multiple approaches for evaluating an integral over a given region.

2a. Write a function to evaluate a univariate integrand, f(x). For starters, we can use the standard normal pdf, exp(-0.5*x^2)/sqrt(2pi).

2b. Write a function to evaluate the integral, \int_a^b dx f(x) using a for loop.
[Optionally, what happens if you add @parallel before the for loop (without adding any extra processors)?]

For parts 2Ab-e of the items below, the functions should take four parameters, a function, the minimum and maximum limits of integration and an optional specifying the number of function evaluations. I'll suggest that you start with simple approximation to the function, (b-a)/N * \sum_i=1^N f(a + i*(b-a)/(N+1)).

For all parts of this problem, remember to run at least one test that your function is working.
It may be useful to use the identify \int_{-a}^{a} dx exp(-0.5*x^2)/sqrt(2pi) = erf(a/sqrt(2))
Benchmark each of the function using 10^4 function evaluations and 10^8 function evaluations (use a combination of a large of evaluations of f(x) inside your function and/or multiple evaluations of the integral), so the run times are at least a tenth of a second. (I found at least ~10^8 total integrand evaluations were needed on my laptop.)

2c. Write a function to evaluate the integral, \int_a^b dx f(x) using vector notation.

2d. Write a function to evaluate the integral, \int_a^b dx f(x) using the map and reduce functions.

2e. Write a function to evaluate the integral, \int_a^b dx f(x) using the mapreduce function.

2f. Write a function to evaluate the integral, \int_a^b dx exp(-0.5*x^2)/sqrt(2pi) using the @devec macro from the Devectorize package. (Hint: For this part, you will need to hardwire the integrand, so you'll drop the function argument to your integrate function.)

2g. Benchmark each of the implementations. Use a combination of a large number of evaluations of f(x) inside your function and/or repeatedly calling the integration function, so the run times are at least a tenth of a second.

2h. For at least two of the best performing implementations, profile the code. Which operations are taking the most time? Discuss the implications for the prospects for further improving the code's performance.

2i. Repeat parts 2b-2e using a more complex function as the integrand.

3. Effect of Branching

Consider the following four functions (traid, triad_twist1, triad_twist2, triad_twist3).
[Based on problem 2.1 from IHPC4SE]

3a. Before you run them, predict which of the triad_twist functionns will perform best and how that will compare to the performance of the how the traid function. Consider eight cases, allowing for either: a) small length vectors (<10^3) and b) large vectors (>10^6) and for the array c to contain: 1) approximately equal number of positive and negative values, 2) all positive values, 3) all negative values, and 4) ~90% positive values.

3b. Benchmark and profile them.

3c. Note any differences from your predictions and reality.

3d. What's your best explanation for the behavior?

```
function triad(b::Vector, c::Vector, d::Vector)
  assert(length(b)==length(c)==length(d))
  a = similar(b)
  for i in 1:length(a)
     a[i] = b[i] + c[i] * d[i]
  end
  return a
end

function triad_twist1(b::Vector, c::Vector, d::Vector)
  assert(length(b)==length(c)==length(d))
  a = similar(b)
  for i in 1:length(a)
     if c[i]<0.
       a[i] = b[i] - c[i] * d[i]
     else
       a[i] = b[i] + c[i] * d[i]
     end
  end
  return a
end

function triad_twist2(b::Vector, c::Vector, d::Vector)
  assert(length(b)==length(c)==length(d))
  a = similar(b)
  for i in 1:length(a)
     if c[i]<0.
       a[i] = b[i] - c[i] * d[i]
     end
  end
  for i in 1:length(a)
     if c[i]>0.
       a[i] = b[i] + c[i] * d[i]
     end
  end
  return a
end

function triad_twist3(b::Vector, c::Vector, d::Vector)
  assert(length(b)==length(c)==length(d))
  a = similar(b)
  for i in 1:length(a)
     cc = abs(c[i])
     a[i] = b[i] + cc * d[i]
  end
  return a
end
```