Astro 585 Lab/Homework #1 (Floating Point Arithmetic, Functions) (v1.0)

1. Open an IJulia notebook (if in Osmond computer lab), Julia Studio session (if on your own laptop) and/or the command line Julia repl (i.e., read-eval-print-loop) interface (if you can't make a previous option work). In the Osmond lab, you'll need to follow a fairly complicated set of direction. On your own computer, it's probably easiest to install Julia Studio from http://forio.com/products/julia-studio/download/. You may also want to install IJulia by following instructions at https://github.com/JuliaLang/IJulia.jl. Once you get some method working smoothly, just write one sentence saying what computers and interfaces you got working.

2. Consider an astronomer analyzing data from a large survey or simulation. A common task is to compute the mean value (m) of observations of some quantity (y_i) for N different objects and to provide an estimate of the variance of that quantity (s^2). For example, this occurs often when performing Monte Carlo integrations. In this case, you will investigate some of the potential complications of performing such calculations.

2a) Generate an array of simulated "data" using:

```
srand(42);
N = 10000;
true_mean = 10000.;
y = true_mean+randn(N);
```

The first line seeds a pseudo-random number generator with the value 42 (so that results will be reproducible when run multiple times). Next two lines create the variable N that contain the integer 10000 and the variable true_mean that contains a floating-point value of 10000. The fourth line sets a variable y that contains a 1-D array with N values drawn from a normal distribution (i.e., mean true_mean and variance of unity) using the pseudo-random number generator. Calculate and report the sample mean and sample variance using Julia's built in functions:

```
sample_mean = mean(y);
sample_var = var(y);
(sample_mean , sample_var)
```

2b) By default, Julia uses 64 bits of memory to store each floating point value. To explore the effects of floating point arithmetic, convert the array of y values into arrays of floating point values that use fewer bits to store each value using the following code.

```
y32bit = convert(Array{Float32,1},y);
y16bit = convert(Array{Float16,1},y);
```

Using Julia's built in mean and var function, compute (and report) the sample mean and sample variance for each of these arrays. How large are the differences? Which have the larger errors? Why?

2c) What do you think would happen if we increased N to 10^5? What do you think would happen if we increased true_mean to 10^5 (assuming we revert N to 10^4)? Write down your guesses. (It's ok if they're not right.)

2d) Modify the above code to test your hypotheses. Were your guesses accurate? If not, explain any differences.

2e) What lessons does this exercise illustrate that could be important when writing similar code for your research?

3. For this exercise, you will compute the variance of the above data using multiple algorithms and compare their relative merits. Algebraically, the sample mean is calculated via
$m = 1/N \times \sum_{i=1}^{N} y_i$ and the sample variance can be written two ways
$$s^2 = 1/(N-1) \times \sum_{i=1}^N (y_i-m)^2 = 1/(N-1) \times \left[\left(\sum_{i=1}^N y_i^2 \right) - N m^2 \right]$$.
In this exercise, you will consider how to calculate the sample variance accurately and efficiently.

Functions: It will be useful to write your code as functions. I strongly recommend you develop a habit of writing code in the form of functions (preferably ones that can be printed on one page of paper). Julia provides multiple syntaxes for writing functions, as described here in the Julia manual. (I suggest deferring the subsections on anonymous function and varargs functions.) The following example demonstrates how to write a function, a simple for loop and access elements of an array in Julia:

```
function mean_demo(y::Array)  # the syntax ::Array specifies that this function can only be applied if argument is an array.
  n = length(y);        # get the number of elements in the array y
  sum = zero(y[1]);  # using zero(y[1]) makes sum have the same data type as y[1]
  for i in 1:n          # In Julia and Fortran, arrays start a 1, not 0 (like in C arrays and Python lists)
    sum = sum + y[i];
  end;              # semi-colons are unnecessary, but can be useful when pasting code interactively
  return sum/n;  # return isn't necessary since functions return the last value by default
end
```

This could also be written more succinctly as

```
mean_demo(y::Array) = sum(y)/length(y);
```

Indeed, Julia comes with a function mean() that is written almost identically to this.

3a) Write a function containing a "one-pass" algorithm to calculate the variance using a single loop.
3b) Write a function containing a "two-pass" algorithm to calculate the variance using two loops over the y_i's.
3c) Compare the accuracy of the results using N=10^6 and true_mean=10^6.
3d) What considerations would affect the decision of whether to use the one-pass algorithm or the two-pass algorithm?
3e) Consider a third "online" 1-pass algorithm for calculating the sample variance given below. Under what circumstance would it be a good/poor choice to use?

```
function var_online(y::Array)
  n = length(y);
  sum1 = zero(y[1]);
  mean = zero(y[1]);
  M2 = zero(y[1]);
  for i in 1:n
        diff_by_i = (y[i]-mean)/i;
        mean = mean +diff_by_i;
        M2 = M2 + (i-1)*diff_by_i*diff_by_i+(y[i]-mean)*(y[i]-mean);
  end;
  variance = M2/(n-1);
end
```

4. A common task is to compute the probability of a set of observations under one or more models. For example, consider a set of observations (y_i's), each of which can be assumed to follow a normal distribution centered on the true value (z_i) with a standard deviation of σ_i, so
$$p(y_i | z_i) = e^[(y_i-z_i)^2/(2\sigma_i^2) / \sqrt{2\pi \sigma_i^2 }$$
When the measurement error for each observation is independent and uncorrelated with the other observations, the probability of a combination of measurements is simply the product of the individual probabilities.
For example, in this exercise, we will consider a radial velocity planet search that measures the velocity of a target star with a precision of $\sigma_i = 2$ m/s at each of N_obs well-separated observation times. In the simplest possible model is that the star has no planets and it's true velocity is a constant 0 m/s.

4a) Use the functions srand(), zeros(), ones() and randn() to seed Julia's random number generator with a value of 42, to generate an array of the star's true velocities (assuming it has no planets), generate an array containing the measurement uncertainties, and generate an array containing simulated observations of the star's velocity. (FYI: The syntax to perform element-wise multiplication of two arrays and b is "a .* b" and not simply "a*b".)

4b) Write a function to calculate the probability of a single measurement value, given the true value and measurement uncertainty, assuming Gaussian measurement uncertainties.
(FYI: You can use the built-in functions exp(x) and sqrt(x). Julia makes it easy to define small functions using the syntax: "add3(a,b,c) = a+b+c".)

4c) Write a function to calculate the likelihood of a set of observations, i.e., the probability of an array of measurement values y (first function argument), given arrays of the measurement uncertainties (\sigma; second function argument) and the true values (z; 3rd function argument).

4d) Test your function for N_obs = 100 and N_obs = 600. Are the results plausible? If not, what's going wrong?

4e) Write a function that calculates the log of the likelihood for a set of observations (as in 4c). Compare the results of this function to the log of the results from your function in part 4c. How could your function be generalized to a case with non-Gaussian measurement uncertainties?

4f) What lessons does this exercise illustrate that could be important when writing code for your research?

5. Identify (at least) three ways the following function should be improved:

```
function round_down_to_power_of_ten(x::Real)
  z = 1.0;
  if x >= 1.0;
    while z*10.<=x
      z = z * 10.0;
    end;
  else
    while z >= x
      z = z / 10.0;
    end;
  end;
  return z;
end
```