## Due Friday, October 16, 17:30

**IMPORTANT: Guidelines for solution sets:**

- You can have MATLAB on your own personal computer for free, just go and download it at the IT website. Alternatively, MATLAB is installed in the PCs in computer labs and other common places, but those computers can be slow. Let me know if you have trouble.
- Once on MATLAB, generate a folder `phys414_ps01_surname_name` and do all your work in there.
- Put all the MATLAB functions we ask you to create in this folder. They should work as instructed if we run them on our own within the folder.
- Your actual solution set aside from the functions themselves (e.g. calculations, verbal explanations, plots) should be in a single pdf file `SolutionSet01_surname_name.pdf`. If there is a short verbal explanation part in the question, you can put it in the caption of the plot or as text in the plot (use the `text()` function).
- Your code should be well-commented. A stranger should be able to follow what you are doing throughout, and you should clearly explain what the function does in the beginning. TAs will take off points for not doing this! However, if we ask you for an explanation in a problem, always provide it separately in your solution set file, do not assume we will see the explanation within your comments.
- Your code should follow the style and design guidelines we discuss in class, proper style is as important as comments. TAs will take off points for not doing this! We will besically follow Google C++ style whose naming conventions on variables and functions you should adopt.
- Compress the folder with all of the above files into `phys414_ps01_surname_name.zip`, and upload it on Blackboard (where you download this file).

These guidelines will be similar for the coming problem sets unless we tell you otherwise. Good luck.

## Problem 1

(a) Assume that the numbers we work on are always in the interval $1 > n > \frac{1}{2}$. Explain what kind of digital representation would be ideal in such a case. What would be our roundoff error for 64 bit double precision in this ideal representation?

(b) Implement the function $f(x) = 1 - \frac{\sqrt{1+x^2}}{1+\frac{1}{2}x^2}$ as

```
function y = func0101b(x)
```

on MATLAB *such that its input and output are vectors* (not just numbers). Plot its values on a log-log scale between $10^{-10} < x < 1$. Make sure your precision never drops below $\sim 10^{-8}$, and explain how you do this. *Hint: Taylor expansion*.

(c) Repeat the previous part for $f(x) = \cot x^2 - \frac{1}{x^2}$, name the function `func0101c`. *Hint: You can find various series expansions and other properties of functions on Mathematica and its documentation.*

## Problem 2

Write the following MATLAB function as described in the comment

```
function [r1 r2] = quadRootsNaive(a,b,c)
% QUADROOTSNAIVE Finds the two roots of a qudratic equation ax^2+bx+c=0
% using the discriminant formula.
```

You can assume that the roots are real, hence you do not need to check for the sign of the discriminant explicitly. Check `quadRootsNaive` for $a = 1,\ b = 200,\ c = 1.5 \times 10^{-15}$, and see that there is tremendous loss of significance. Now write a better version of the root finding function, `function [r1 r2] = quadRoots(a,b,c)`, which addresses the above problem. Explain how you avoided the problem of the naive version. *Hint: Remember my advise related to Wikipedia in the syllabus.*

## Problem 3

In this problem, we aim to analyze the roundoff error in the sum of a large number of finite precision numbers. Such operations are ubiquitous in all parts of computational sciences, so you should be aware of some fundamental issues (even though we can ignore them in many cases). We will use the vector-based nature of MATLAB to generate our list of numbers to be summed.

(a) All the elements of a vector `vec` can be summed using `sum(vec)`. Complete the following function as described in the comment (can be done in a single line)

```
function diff = sumDiff(vec)
% SUMDIFF Sums all the elements in a given vector first from beginning to
% the end, and then in the reverse order, and finally takes the absulate
% value of the difference of these sums. This is an estimate of the
% cumulative roundoff error in the sum of the numbers.
```

The output vanishes in infinite precision, but it gives an estimate of the error in our sum when we have double precision floating point arithmetic as usual. Give `sumDiff` a few tries to see this yourself.

Note that if we are extremely lucky, or in special occasions like where all the elements of the vector are the same, the above difference can still vanish. We are not after such rare instances, we want to see the "average" error for summing many numbers. The logic is clear, generate large lists of random numbers and add them in different orders to see an *estimate* of the roundoff error.

For a given integer $n$, generate a vector with random elements of similar magnitude using `vec = rand(1,n)` (use `help rand` if you want to see what it exactly does). Find the error in the sum of the elements of `vec` using `sumDiff(vec)`. To obtain an estimate of the error in summing $n$ numbers of order unity, repeat this many times and take the average of `sumDiff(v)` using

```
function [vec_sum] = sumError(n)
% SUMERROR Sums n random numbers generated by rand, in two different
% orders, and calculates the difference to estimate the error in the sum.
% Averages such errors for numTrial=100 cases to get a general estimate of
% the error in the sum of n numbers of similar values.
```

Now write the following function

```
function ps0103(n_vec)
% PS0103 Main function for solving PHYS414/514 PS01 problem 4. For
% each element n of the integer vector n_vec, it calculates the roundoff
% error in summing n randomly generated numbers uniformly chosen from the
% interval [0 1]. Then these errors are plotted w.r.t. n. This can be used
% to infer the dependence of the error on n.
```

As in the description, it should repeat the previous paragraph for each of the elements in the integer vector `nvec`, and then plot the error vs $n$ for up to $n = 10^4$. Have the plot in your solution set.

You should observe in the plot that the average error grows with $n$. Explain this result, and show how the growth depends on $n$, i.e. exponentially, linearly, power law etc. *Hint: Random walk.*

What is the point of all this? $10^{-16}$ precision might look enough for anything at first, but this should demonstrate that even such tiny roundoff error can accumulate if your dataset is large enough.

## Problem 4

Implement a new summation algorithm to replace `sum(vec)` with

```
function [vec_sum] = sumKahan(vec)
% SUMKAHAN Sums all the elements in vec using Kahan summation to achieve
% roundoff error independent of the length of the vector.
```

which uses *Kahan summation*. Do you wonder what the heck *Kahan summation* is? Follow the hint from Problem 3.

Modify `ps0103()` to generate a second plot with this method (keep the parts from part (a)), and similarly see how the roundoff error changes with vector size. Use linear axes rather than loglog since you will probably see the roundoff error estimate to vanish.

Did we achieve roundoff error-free summation? Kahan summation indeed reduces the roundoff error dramatically, but it cannot get rid of it. Remember that what we plot is an "estimate" of the roundoff error in the sum, and the estimation does not work here. Think why, but no points for this.

Kahan summation reduces the relative error in the sum to the standard level of double precision ($\sim 10^{-16}$), much better than the growing relative error we had before, but there is still roundoff error. However when we use Kahan summation, changing the ordering of the summation gives the same small roundoff error, hence the difference vanishes. This difference was our estimate of the roundoff error in the sum when it changed with order, but it is not a good estimate in this case.

## Optional for those who enjoy coding (no points, bonus or otherwise)

You can actually switch to single precision on MATLAB using the `single()` command. Make your random vectors single-precision, and see what changes.

Another thing you can investigate is *pairwise summation*, which is a simpler method than Kahan summation that dramatically reduces the relative errors in numerical summation. It is slightly worse than Kahan summation, and the relative error grows with the number of summands, but only logarithmically. Try to see if you can actually see this logarithmic trend.

If you repeat Problem 3 in Python using NumPy, you will not get similar results, because `numpy.sum` uses pairwise summation as the default method to sum all elements in a vector. It may help to know such details about your programming platform.