C for Science - 2011 - Practical Exercise #2

1. Edit your linear equation solver from the previous exercise to solve the quadratic equation:

$$A[2]x^2 + A[1]x + A[0] = 0.$$

The program should consist of:

(a) A `main` function which:

i. Reads in three coefficients of the quadratic.

ii. Calls a function `quad_roots` which solves the quadratic.

iii. Notifies the user which class the roots fall in and displays them in the appropriate form.

(b) A function `quad_roots` with the following declaration:

```
int quad_roots(double * A, double * roots)
```

The function should return the following values

`-3`	when `A[2] = A[1] = 0, A[0]` $\neq$ `0`, not an equation.
`-2`	when `A[0] = A[1] = A[2] = 0`, no meaningful solution.
`-1`	when `A[2] = 0, A[1]` $\neq$ `0`, a linear equation with one solution `roots[0]`.
`0`	for two different real roots (`A[1]`^{2} `> 4A[2]A[0]`), `roots[0]` and `roots[1]`.
`1`	for complex conjugate roots (`A[1]`^{2} `< 4A[2]A[0]`), `roots[0]` should contain the real part and `roots[1]` the imaginary part.
`2`	for the case with two equal real roots `A[1]`^{2} `= 4A[0]A[2]`, `roots[0] = roots[1]`.

Separate your program into two files, one containing `main` and the other containing `quad_roots`. The program should be tested on the data set:

i. `A[2] = 1, A[1] = 3, A[0] = 2,`

ii. `A[2] = 1, A[1] = 2, A[0] = 3,`

iii. `A[2] = 4, A[1] = -4, A[0] = 1,`

iv. `A[2] = 0, A[1] = 2, A[0] = 1.`

2. The C Standard header file `<float.h>` contains information about the various floating point numbers used in the C system such as largest and smallest numbers (in magnitude). One useful quantity is the smallest number that can be added to `1.0` and cause a difference to the value (this is known as ϵ).

(a) Locate and open the `<float.h>` used by your compiler and extract the following constants `FLT_MAX`, `FLT_MIN` and `FLT_EPSILON`.

(b) What are the corresponding values for `double` types?

(c) Given the following doubles:

```
double a = 1.0 + DBL_EPSILON;
double b = (1.0 + DBL_EPSILON / 2.0) + DBL_EPSILON / 2.0;
double c = 1.0 + (DBL_EPSILON / 2.0 + DBL_EPSILON / 2.0);
```

Print out `1.0-a`, `1.0-b` and `1.0-c`, using the `%g` format specifier, is this what you expect? We see even the process of summing up numbers can produce problems. One famous algorithm to sum up numbers accurately is *Kahan Summation* (look this up!).

3. *Combatting rounding errors:* Returning to our quadratic solver, if `A[1]`$^2 \gg$ `4A[0]A[2] > 0`, then the calculation of one of the roots (`-A[1]` $\pm\sqrt{\cdot}$)/`2A[2]` involves a subtraction of nearly equal quantities, and thus a loss of accuracy. ('Adding' two numbers of similar magnitude and different sign also results in an accuracy loss!). When possible, an alternative formulation should be sought which avoids this subtraction.

   One useful property of quadratics is that the roots satisfy the following expression:

   ```
   root[0]*root[1] = A[0]/A[2]
   ```

   Thus a more accurate way to calculate both roots of the quadratic is to choose the sign $\pm$ such that (`-A[1]` $\pm\sqrt{\cdot}$)/`2A[2]` is not a subtraction, and then compute the other root from the equation above.

   For an example we consider how high a mobile telephone tower must be in order to transmit to a certain horizon. The relationship between the height of the tower h, the distance to the horizon d and the radius of the Earth $R = 6350$km is:

   $$h^2 + 2hR = d^2$$

   (a) Solve the above equation for values of d of 1 cm, 1 meter, 100 meters, 10 km and 1000km; using `quad_sol`.

   (b) Modify `quad_sol` to avoid subtraction when computing roots.

   (c) Compare the modified `quad_sol` to the original, when do differences occur?