QUEEN MARY, UNIVERSITY OF LONDON

MTH6150

Numerical Computing in C and C++

Exercise Sheet 6

---

1. Write a function `cube` that returns the cube of a `double` variable (with return type `double`). Write another function of the same name that returns the cube of an `int` variable (with return type `int`).

   Use both of them in the same program, first with the functions declared and defined before `main`, and then with the function bodies (the definition) after `main`.

   Modify the `int` version to the following:

   ```
   double cube{const int x){
   ```

   What happens now if you try to run a program with this version and the following version at the same time:

   ```
   double cube{const double x){
   ```
   ?

2. Type in the function for the factorial on slide 4 of lecture 6, and run it to find out if it is correct. Add a command within the function to output `n` to the screen and run this code to verify what sequence of numbers the function is called with.

3. Run the example pieces of code on slides 6 to 10 to understand which part of the code variables exist and why the code won't compile in some cases.

4. We have met the function `pow` in `<cmath>`, with $z = x^y$ given by

   ```
   double z = pow(x, y);
   ```

   When y is an integer, it may be more efficient to start with 1 and then repeatedly multiply by x, doing this y times. Implement a function

   ```
   double pow_int(const double x, const int k){
       ...
   }
   ```

   that calculates x^k using a simple `for` loop. Check the numerical results by comparing them with `pow`, and/or with examples where you know the answer already.

5. In question 4, the number of multiplications can bereduced by noting that

$$x^{2m} = (x^m)^2$$
$$x^{2m+1} = x\,(x^m)^2$$

Implement a variant of `pow_int` that takes advantage of this.

One way would be using recursive function calls:

```
double pow_int_rec(const double x, const int k){
    if(k==1){
        ...
    }
    else if(k>=2 && k%2==0){
        const double w = pow_int_rec(x, k/2);
        return w*w;
    }
    else{
        ...
    }
}
```

As before, you can check the numerical results by comparing with `pow`.

Can you see how the computing time to calculate x^k increases as k increases for each of the two methods (in theory, not by trying to time the code)?

6. The probability density function (pdf) of the normal distribution with mean μ and standard deviation σ is given by

$$\varphi(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Write a function `norm_pdf` with three arguments (x,μ,σ) that returns this pdf.

```
double norm_pdf(const double x, const double mu, const double sigma){
    ...
}
```

As a check, two example values are $\varphi(1;\,0,\,1) \approx 0.242$, $\varphi(2;\,1,\,2) \approx 0.176$.

7. The standard normal cumulative distribution function (i.e. for $\mu = 0$, $\sigma = 1$)

$$\Phi(x) = \int_{-\infty}^{x} \varphi(t;0,1)dt = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$$

does not have an explicit closed-form expression. It can be expressed in terms of an infinite series. As an intermediate step

$$\Phi(x) = \frac{1}{2}[1 + g(x\,/\,\sqrt{2})]$$

where $g(w)$ is known as the error function. We also have

$$g(w) = \frac{2}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k w^{2k+1}}{k!(2k+1)}$$

Write a function that calculates an approximation to the error function $g(w)$ using the first n terms from this infinite series:

```
double erff(const double w, const int n)){
    ...
}
```

Then write a function

```
double norm_cdf(const double x, const int n)){
    ...
}
```

that calculates $\Phi(x)$ using the formula for Φ in terms of g, i.e. by using the function `erff` inside the function `norm_cdf`.

To check the results, some standard values that are used for calculation of 90% and 95% confidence intervals are $\Phi(1.64) \approx 0.95$ and $\Phi(1.96) \approx 0.975$. Also, $\Phi(x) = 1 - \Phi(-x)$ for all real x.

Note that the series is very slow to converge for values of x outside the range -2.5 to 2.5, so larger values of n are needed. $n = 10$ or 15 is reasonable inside this range.

Also, see if you can make the function `erff` more efficient by avoiding calculating w^{2k+1} and $k!$ at each loop iteration (by storing certain variables declared before the loop and updating them at each loop iteration).

8. A definite integral can be approximated by the composite trapezoidal rule

$$I = \int_{t_0}^{t_N} f(t)dt \simeq \Delta t\left[\frac{f(t_0)+f(t_N)}{2} + \frac{f(t_1)+f(t_2)}{2} + \ldots + \frac{f(t_{N-1})+f(t_N)}{2}\right]$$
$$= \Delta t\left[\frac{f(t_0)+f(t_N)}{2} + \sum_{k=1}^{N-1} f(t_0 + k\Delta t)\right], \quad \Delta t = \frac{t_N - t_0}{N} > 0$$

on a set of equidistant points $\{t_0, t_1, t_2, \ldots, t_N\}$ for some positive integer N. Use this method to approximate $\Phi(x)$ from question 7, by taking $f(x)$ to be $\varphi(x;\, 0,\, 1)$ from question 6. Try this for various values of N.

Note that $\Phi(0) = 0.5$, so for $x > 0$

$$\Phi(x) = 0.5 + \int_0^x \varphi(t;0,1)dt$$

Also, $\Phi(x) = 1 - \Phi(-x)$ for all real x.

Exercise Sheet 6

Hints for question 7

- Some values for the error function, denoted by $g(w)$ in the question, are:
  $g(1) \approx 0.8427008$
  $g(0.5) \approx 0.5204999$
  Also, $g(-w) = -g(w)$ for all real w.
- Using a loop to calculate a sum: there are simpler examples on exercise sheet 4, question 1b or question 3 (code as a solution to question 3 is on QMplus).
- There is code for the factorial $n!$ on slide 4 of lecture 6.
- In the calculations, possible sources of errors are:
  integer division, so convert `int` to `double`;
  and use round brackets to ensure that the correct terms are included in the denominator of a fraction.