MATH60082
Examples Sheet 1
Analytic Pricing

Contents

1.1 Coding up the Cumulative Normal Distribution

Write a function to calculate N(x) the cumulative normal distribution with mean zero and variance one given by the formula

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

1.1.(i) Using a polynomial approximation

Using a polynomial approximation to $N(x)$ (less than single precision!):

$$N(x) = 1 - N'(x)(a_1 k + a_2 k^2 + a_3 k^3), \quad x \geq 0,$$

$$N(x) = 1 - N(-x), \quad x < 0.$$

Here $k = \frac{1}{1+\gamma x}$, $\gamma = 0.33267$, $a_1 = 0.43618$, $a_2 = -0.12017$, and $a_3 = 0.93730$.

First, start with a new project and an empty cpp file.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
```

```
int main()
{
  // do nothing
  return 0;
}
```

download

Compile and check it runs. Looking at the algorithm in the question we see that we need several variables, k, γ, a_1, a_2 and a_3. Declare all those variables along with a default value of $x = 1$.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x=1;
  double gamma=0.33267,a1=0.43618,a2=-0.12017,a3=0.93730;
  double k=1./(1.+gamma*x);
  // do nothing
  return 0;
}
```

download

Compile and check it runs. Now enter the calculation for $N(x)$ with $x \geq 0$ and check it works.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x=1;
  double gamma=0.33267,a1=0.43618,a2=-0.12017,a3=0.93730;
  double k=1./(1.+gamma*x);
  cout << 1.-1./sqrt(8.*atan(1.))*exp(-x*x/2.) *(a1*k+a2*k*k+a3*k*k*k) << endl;
  // do nothing
  return 0;
}
```

download

Compile and check the output, it should be 0.841352. Now you need to make it work when $x < 0$, this means using an textttif statement and changing the value of γ as well as the formula. Check your code works in both cases.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
```

```
int main()
{
  double x=-1;
  double gamma=0.33267,a1=0.43618,a2=-0.12017,a3=0.93730;
  if(x>=0)
  {
    double k=1./(1.+gamma*x);
    cout << 1.-1./sqrt(8.*atan(1.))*exp(-x*x/2.) *(a1*k+a2*k*k+a3*k*k*k) << endl;
  }
  else
  {
    double k=1./(1.-gamma*x);
    cout << 1./sqrt(8.*atan(1.))*exp(-x*x/2.) *(a1*k+a2*k*k+a3*k*k*k) << endl;
  }
  // do nothing
  return 0;
}
```

download

Finally, move your code into a function and test the result. You will need to use the `return` keyword to return the value of the function instead of printing to screen.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

double normalDistribution_poly(double x)
{
  static double one_over_sqrt2=1./sqrt(8.*atan(1.));
  double gamma=0.33267,a1=0.43618,a2=-0.12017,a3=0.93730;
  if(x>=0)
  {
    double k=1./(1.+gamma*x);
    return 1.-one_over_sqrt2*exp(-x*x/2.) *(a1*k+a2*k*k+a3*k*k*k) ;
  }
  else
  {
    double k=1./(1.-gamma*x);
    return  one_over_sqrt2*exp(-x*x/2.) *(a1*k+a2*k*k+a3*k*k*k) ;
  }
}

int main()
{
  for(int x=-3;x<=3;x++)
    cout << "N("<<x<<") = " << normalDistribution_poly(x) << endl;
  // do nothing
  return 0;
}
```

download

The output from this code is:
N(-3) = 0.00135489

N(-2) = 0.0227587
N(-1) = 0.158648
N(0) = 0.500002
N(1) = 0.841352
N(2) = 0.977241
N(3) = 0.998645

1.1.(ii) Using integration

Use Simpson's rule to evaluate the integral defining $N(x)$. This states that when there are values of $y = f(x)$ for values of x at equal intervals h apart, an approximate numerical integration is given by

$$\int_{x=a}^{b} f(x)dx \approx \frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \ldots + 4y_{n-1} + y_n]$$

where $h = x_1 - x_0 = (b-a)/n$. Note this formula involves an even number of intervals, and the (truncation) error diminishes as h^4. You will need around 2000 points or more for machine accuracy, making this a very expensive method.

Again, first we think about the stages of our program. We must choose a method to implement the integration, test and verify it. Once it is then applied to the problem we need to decide how to deal with the concept of infinity in this setting. First let use code up the integration method. To be able to check the method, you should choose a simple function with a known integral so that we can compare accuracy. Start with a new project and an empty cpp file.

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  // do nothing
  return 0;
}
```

download

Compile and check it runs. We are going to implement Simpson's method for integration firstly on the sin function so that we can check the results. The method is given by

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(a) + 2\sum_{j=1}^{n/2-1} f(a+2jh) + 4\sum_{j=1}^{n/2} f(a+(2j-1)h) + f(b)\right] \quad (1)$$

Start by declaring the input parameters required by the method, including the number of steps and the integration range, and follow with the variables used inside such as the dummy

integration variable s and the step size h. We should supply default values and initialise variables if required

```
  // number of steps
  int N=10;
  // range of integration
  double a=0,b=1.;
  // local variables
  double s,h,sum=0.;
  // inialise the variables
  h=(b-a)/N;
```

download

Next we need to add the first few terms and the last one to sum, before adding a loop to go over the rest.

```
  // add in the first few terms
  sum = sum + sin(a) + 4.*sin(a+h);
  // and the last one
  sum = sum + sin(b);
```

download

Note that we should really check that N is even as this method relies on that fact. Since we have already taken care of the terms for 0, 1 and N, the loop must run over the terms 2 up to and including N-1. Check your loop runs over the correct values before continuing

```
  // loop over terms 2 up to N-1
  for(int i=1;i<N/2;i++)
  {
    s = a + 2*i*h;
    // check even terms
    cout << 2*i << " " << s << endl;
    s = s + h;
    // check odd terms
    cout << 2*i+1 << " " << s << endl;
  }
```

download

Now add them in and check against the analytical solution.

```
int main()
{
  // number of steps
  int N=10;
  // range of integration
  double a=0,b=1.;
  // local variables
  double s,h,sum=0.;
  // inialise the variables
  h=(b-a)/N;
  // add in the first few terms
```

```
  sum = sum + sin(a) + 4.*sin(a+h);
  // and the last one
  sum = sum + sin(b);
  // loop over terms 2 up to N-1
  for(int i=1;i<N/2;i++)
  {
    s = a + 2*i*h;
    sum = sum + 2.*sin(s);
    s = s + h;
    sum = sum + 4.*sin(s);
  }
  // complete the integral
  sum = h*sum/3.;
  // output results
  cout << sum << " " << 1.-cos(b) << endl;
  return 0;
}
```

download

You should check accuracy by varying the value of N.

Next we are going to change this algorithm to solve the normal distribution. Do this in 2 steps, first go through and change all calls to the sin function by exp(-s*s/2.), then multiply the final result by 1/sqrt(2*pi). The result (for a=0 and b=1) should be 0.341345. Now we have to deal with infinity, so how can we integrate over infinity? For the normal distribution this is quite easy as it is an even function and we know that the integral between 0 and infinity is one half. So simply adding one half to the result of the integration between 0 and b gives the value of N(b) for all b. At this stage your code should look like

```
  // add in the first few terms
  sum = sum + exp(-a*a/2.) + 4.*exp(-(a+h)*(a+h)/2.);
  // and the last one
  sum = sum + exp(-b*b/2.);
  // loop over terms 2 up to N-1
  for(int i=1;i<N/2;i++)
  {
    s = a + 2*i*h;
    sum = sum + 2.*exp(-s*s/2.);
    s = s + h;
    sum = sum + 4.*exp(-s*s/2.);
  }
  // complete the integral
  sum = 0.5 + h*sum/3./sqrt(8.*atan(1.));
  // output results
  cout << sum << endl;
```

download

Now move this algorithm into a function with the definition

```
double normalDistribution(double x)
```

download

The last thing to take account of is what happens when x is either very large and positive or very large and negative. The best thing to do here is use analytical results and return 0 if x is large and negative and 1 if x if large and positive. Don't forget to set the value of b to be x and choose a value of N sufficiently large to maintain accuracy.

```
#include <iostream>
#include <cmath>
using namespace std;

double normalDistribution(double x)
{
  if(x<-10.)return 0.;
  if(x>10.)return 1.;
  // number of steps
  int N=2000;
  // range of integration
  double a=0,b=x;
  // local variables
  double s,h,sum=0.;
  // inialise the variables
  h=(b-a)/N;
  // add in the first few terms
  sum = sum + exp(-a*a/2.) + 4.*exp(-(a+h)*(a+h)/2.);
  // and the last one
  sum = sum + exp(-b*b/2.);
  // loop over terms 2 up to N-1
  for(int i=1;i<N/2;i++)
  {
    s = a + 2*i*h;
    sum = sum + 2.*exp(-s*s/2.);
    s = s + h;
    sum = sum + 4.*exp(-s*s/2.);
  }
  // complete the integral
  sum = 0.5 + h*sum/3./sqrt(8.*atan(1.));
  // return result
  return sum;
}

int main()
{
  cout.precision(16);
  cout << "N(0)=" << normalDistribution(0.) << endl;
  cout << "N(1)=" << normalDistribution(1.) << endl;
  cout << "N(2)=" << normalDistribution(2.) << endl;
  return 0;
}
```

download

Here we needed to choose N=2000 to get close to double precision, meaning that a single function call is incredibly expensive. There are other, more efficient approximations out there. If you are using a compiler that is math library C99 compliant you will have access to the erfc

function, which will give you double precision accuracy using the formula

$$N(x) = \frac{1}{2}\text{erfc}\left(-\frac{x}{\sqrt{2\pi}}\right)$$

1.1.(iii) Using a higher order polynomial

Using a fast and accurate polynomial approximation to $N(x)$ (a double precision approximation, see West, 2005, for more details):

$$N(x) = 1 - \sqrt{2\pi}N'(x)\frac{P(x)}{Q(x)}, \quad 0 \le x \le \frac{10}{\sqrt{2}},$$

$$N(x) = 1 - N'(x)\frac{1}{F_4(x)}, \quad \frac{10}{\sqrt{2}} < x \le 37,$$

$$N(x) = 1, \quad x > 37,$$

$$N(x) = 1 - N(-x), \quad x < 0.$$

Here

$$P(x) = \sum_{i=0}^{6} a_i x^i, \qquad Q(x) = \sum_{i=0}^{7} b_i x^i$$

and the coefficients are:

a_0	220.206867912376	b_0	440.413735824752
a_1	221.213596169931	b_1	793.826512519948
a_2	112.079291497871	b_2	637.333633378831
a_3	33.912866078383	b_3	296.564248779674
a_4	6.37396220353165	b_4	86.7807322029461
a_5	0.700383064443688	b_5	16.064177579207
a_6	0.0352624965998911	b_6	1.75566716318264
		b_7	0.0883883476483184

For the function $F_4(x)$ use the recurrence relation

$$F_0(x) = x + \frac{13}{20}$$

$$F_i(x) = x + \frac{5-i}{F_{i-1}(x)} \qquad \text{for } i = 1, 2, 3, 4$$

Unfortunately if you are using Visual Studio it is not yet implemented (might yet be in VS2013) so we need another method. One such method is described in West (2005) and they give a VB code implementation of the method. We are going to follow that method closely. Start with your empty program, I have included some benchmark values in comments to check against

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  // Benchmark values of N(x)
  // N(0) = 0.5
  // N(1) = 0.841344746068543
  // N(2) = 0.977249868051821
  return 0;
}
```

download

Compile and check the code runs with no output.

The algorithm is given by the following expressions

$$N(x) = 1 - \sqrt{2\pi}N'(x)\frac{P(x)}{Q(x)}, \quad 0 \leq x \leq \frac{10}{\sqrt{2}},$$

$$N(x) = 1 - N'(x)\frac{1}{F(x)}, \quad \frac{10}{\sqrt{2}} < x \leq 37,$$

$$N(x) = 1, \quad x > 37,$$

$$N(x) = 1 - N(-x), \quad x < 0.$$

where P, Q and F are polynomials, more details can be found in the examples sheet (click here). First declare and assign some of the variables needed for the calculation

```cpp
  // calculate \sqrt{2\pi} upfront once
  double RT2PI = sqrt(4.0*acos(0.0));
  // calculate 10/\sqrt{2} upfront once
  double SPLIT = 10./sqrt(2);
  // enter coefficients for the polynomials P and Q
  double a[] =
      {220.206867912376,221.213596169931,112.079291497871,33.912866078383,6.37396220353165,0.700
      e-02};
  double b[] =
      {440.413735824752,793.826512519948,637.333633378831,296.564248779674,86.7807322029461,16.0
      e-02};

  // calculate the value of N at x=1
  double x=1.;
```

download

where we have chosen $x = 1$ to test. Since we know what x is we don't need to consider the 3 cases. Now calculate the polynomial expressions using a nested multiplication (important for accuracy) and we get

```cpp
  double NDash = exp(-x*x/2.0)/RT2PI;
  double Px = (((((a[6]*x + a[5])*x + a[4])*x + a[3])*x + a[2])*x + a[1])*x + a[0];
```

```
  double Qx = ((((((b[7]*x + b[6])*x + b[5])*x + b[4])*x + b[3])*x + b[2])*x + b
      [1])*x + b[0];
  cout << Px << " " << Qx << " " << NDash << endl;
  // output is 594.522 2272.83 0.241971
```

download

the output you should expect is shown. We can now estimate the value of $N(x)$

```
  cout.precision(16);
  cout << 1 - exp(-x*x/2.0)*Px/Qx << endl;
  // output is 0.841344746068543
```

download

You now need to include the different cases, such as negative x, etc, and then move everything into a function. After some testing you should arrive at an algorithm that looks something like:

```
#include <iostream>
#include <cmath>
using namespace std;
/*
 *    A * * fast* and accurate polynomial approximation to $N(x)$ (double precision
    ):
 *    $$ N(x)= 1-\sqrt{2\pi}N'(x)\frac{P(x)}{Q(x)}, \quad 0 \leq x\leq \frac{10}{\
    sqrt{2}},$$
 *    $$ N(x)= 1-N'(x)\frac{1}{F_4(x)}, \quad \frac{10}{\sqrt{2}} < x\leq 37,$$
 *    $$ N(x)= 1, \quad x > 37,$$
 *    $$N(x)=1-N(-x),\quad x<0.$$
 *    Here
 *    $$
 *    P(x) = \sum_{i=0}^6 a_i x^i, \quad \quad Q(x) = \sum_{i=0}^7 b_i x^i
 *    $$
 *    and the coefficients are:
 *    \begin{tabular}{lc||lc}
 *    \hline
 *    $a_0$ & 220.206867912376& $b_0$&440.413735824752 \\
 *    $a_1$ & 221.213596169931& $b_1$&793.826512519948 \\
 *    $a_2$ & 112.079291497871& $b_2$& 637.333633378831\\
 *    $a_3$ & 33.912866078383& $b_3$&296.564248779674 \\
 *    $a_4$ & 6.37396220353165& $b_4$&86.7807322029461 \\
 *    $a_5$ & 0.700383064443688& $b_5$& 16.064177579207\\
 *    $a_6$ & 0.035262496599891 1& $b_6$&1.75566716318264 \\
 *    & & $b_7$& 0.0883883476483184\\
 *    \end{tabular}
 *    For the function $F_4(x)$ we use the recurrence relation
 *    $$
 *    F_0(x) = x + \frac{13}{20}
 *    $$
 *    $$
 *    F_{i}(x) = x + \frac{5-i}{F_{i-1}(x)} \quad \quad \text{for }i=1,2,3,4
 *    $$
 *
 *    On entry x is a real value, and we return double precision estimate of the
    cummulative normal distribution
 */
```

```
double normalDistribution(double x)
{
  // calculate \sqrt{2\pi} upfront once
  static const double RT2PI = sqrt(4.0*acos(0.0));
  // calculate 10/\sqrt{2} upfront once
  static const double SPLIT = 10./sqrt(2);
  static const double a[] =
      {220.206867912376,221.213596169931,112.079291497871,33.912866078383,6.37396220353165,0.700
      e-02};
  static const double b[] =
      {440.413735824752,793.826512519948,637.333633378831,296.564248779674,86.7807322029461,16.0
      e-02};

  const double z = fabs(x);
  // Now N(x) = 1 - N(-x) = 1-\sqrt{2\pi}N'(x)\frac{P(x)}{Q(x)}
  //  so N(-x) = \sqrt{2\pi}N'(x)\frac{P(x)}{Q(x)}
  // now let \sqrt{2\pi}N'(z)\frac{P(x)}{Q(z)} = Nz
  // Therefore we have
  //      Nxm = N(x) = \sqrt{2\pi}N'(z)\frac{P(x)}{Q(z)} = Nz if x<0
  //      Nxp = N(x) = 1 - \sqrt{2\pi}N'(z)\frac{P(x)}{Q(z)} = 1-Nz if x>=0
  double Nz = 0.0;

  // if z outside these limits then value effectively 0 or 1 for machine precision
  if(z<=37.0)
  {
    // NDash = N'(z) * sqrt{2\pi}
    const double NDash = exp(-z*z/2.0)/RT2PI;
    if(z<SPLIT)
    {
      // here Pz = P(z) is a polynomial
      const double Pz = (((((a[6]*z + a[5])*z + a[4])*z + a[3])*z + a[2])*z + a[1])
          *z + a[0];
      // and Qz = Q(z) is a polynomial
      const double Qz = ((((((b[7]*z + b[6])*z + b[5])*z + b[4])*z + b[3])*z + b
          [2])*z + b[1])*z + b[0];
      // use polynomials to calculate N(z)  = \sqrt{2\pi}N'(x)\frac{P(x)}{Q(x)}
      Nz = RT2PI*NDash*Pz/Qz;
    }
    else
    {
      // implement recurrence relation on F_4(z)
      const double F4z = z + 1.0/(z + 2.0/(z + 3.0/(z + 4.0/(z + 13.0/20.0))));
      // use polynomials to calculate N(z), note here that Nz = N' / F
      Nz = NDash/F4z;
    }
  }

  //
  return x>=0.0 ? 1-Nz : Nz;
}

int main()
{
  cout.precision(16);
  cout << "N(0)=" << normalDistribution(0.) << endl;
  cout << "N(1)=" << normalDistribution(1.) << endl;
```

```
  cout << "N(2)=" << normalDistribution(2.) << endl;
  return 0;
}
```

download

1.1.(iv) Using a inbuilt functions

Use the built in cmath function `erfc` to create an approximation to the normal distribution. Use the relationship

$$N(x) = \frac{1}{2}\text{erfc}\left(-\frac{x}{\sqrt{2}}\right).$$

First, start with a new project and an empty cpp file.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  // do nothing
  return 0;
}
```

download

Compile and check it runs. Now we can use the `erfc` function to calculate $N(x)$. Output your calculation with $x = 1$.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x=1.;
  cout << 0.5*erfc(-x/sqrt(2.)) << endl;
  return 0;
}
```

download

The output should be 0.841345. Put this into a function for ease of use.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
```

```
double normalDistribution_builtin(double x)
{
  return 0.5*erfc(-x/sqrt(2.));
}

int main()
{
  for(int x=-3;x<=3;x++)
    cout << "N("<<x<<") = " << normalDistribution_builtin(x) << endl;
}
```

download

1.1.(v) Comparing efficiency

Create a table of values to compare each of the approximations (i)-(iv). Which of the calculations is most accurate? Create a loop calling each function 1 million times, which of the function is the most efficient?

Here we use an example code as the start point. This code time how long $N = 10^7$ sin function evaluations take (plus the one or two additions/checks to run the loop). This can give you a measure of how long functions take **relative** to each other. For functions that take longer to evaluate you will need to lower the value of N.

```
#include <iostream>
#include <iomanip>
#include <chrono>
#include <cmath>
using namespace std;

int main()
{
  int N=10000000;
  // get start time
  auto start = std::chrono::steady_clock::now();
  // code in here is timed
  double sum=0.;
  for(int i=0;i<N;i++)
  {
    sum = sum + sin(i);
  }
  cout << sum << endl;
  // get finish time
  auto finish = std::chrono::steady_clock::now();
  // convert into real time in seconds
  auto elapsed = std::chrono::duration_cast<std::chrono::duration<double> >(finish
      - start);
  // output values
  cout << " Total time elapsed for ";
  cout << N << " calculations is " << elapsed.count() << endl;
}
```

download

Try adding in the *normalDistribution* codes from above in place of the the sin function and see how long they each take – which one is fastest? Make sure you change the build environment to **Release** when checking code efficiency. The code here implements this for the polynomial approximation, note that we evaluate the function over the interval $[-5, 5]$ to make it a fair test.

```
#include <iostream>
#include <iomanip>
#include <chrono>
#include <cmath>
using namespace std;

double normalDistribution_poly(double x)
{
  double one_over_sqrt2pi = 1. / sqrt(8.*atan(1.));
  double gamma = 0.33267;
  double a1 = 0.43618;
  double a2 = -0.12017;
  double a3 = 0.93730;
  if (x >= 0)
  {
    double k = 1 / (1. + gamma*x);
    return 1. - one_over_sqrt2pi*exp(-x*x*0.5)*(a1*k + a2*k*k + a3*k*k*k);
  }
  else
  {
    double k = 1 / (1. - gamma*x);
    return one_over_sqrt2pi*exp(-x*x*0.5)*(a1*k + a2*k*k + a3*k*k*k);
  }
}

int main()
{
  int N=1000000;
  // get start time
  auto start = std::chrono::steady_clock::now();
  // code in here is timed
  double sum=0.;
  for(int i=0;i<N;i++)
  {
    for(int x=-5;x<=5;x++)
      sum = sum + normalDistribution_poly(x);
  }
  cout << sum << endl;
  // get finish time
  auto finish = std::chrono::steady_clock::now();
  // convert into real time in seconds
  auto elapsed = std::chrono::duration_cast<std::chrono::duration<double> >(finish
      - start);
  // output values
  cout << " Total time elapsed for ";
  cout << 11.*N << " calculations is " << elapsed.count() << endl;
```

```
}
```

download

1.2 Black Scholes

Write a C++ code to calculate the option values for a call $C(t = 0, S)$, and a put $P(t = 0, S)$ using the Black-Scholes formula:

$$C(t = 0, S) = SN(d_1) - Xe^{-r(T-t)}N(d_2), \tag{2}$$

$$P(t = 0, S) = Xe^{-r(T-t)}N(-d_2) - SN(-d_1) \tag{3}$$

with

$$d_1 = \frac{\log(S/X) + (r + \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}} \tag{4}$$

$$d_2 = \frac{\log(S/X) + (r - \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}} \tag{5}$$

You can use the cumulative distribution function $N(x)$ from earlier on. So rather than starting with an empty project copy and paste the code from the previous project into your new project, you can find a version of the code on my website (click here). Now we need to create a function for the call option, so we need to think about what arguments need to be supplied and what are the local variables to the function. Obviously we need to supply the asset price, current time both of which may vary, and the parameters for the function are the strike price, interest rate, volatility and maturity, and we want to return a real number as the answer. Inside the function, we will need to calculate the value of d1 and d2. This leads to a function definition of the form

```
double callOptionPrice(double S,double t,double X,double r,double sigma,double T)
{
  double d1;
  double d2;
  return 0;
}
```

download

You **must** place this function header in between the main function and the normalDistribution function, since we will be calling "normalDistribution" inside "callOptionPrice" and then "callOptionPrice" inside "main". Next we can simply fill in the mathematical formulas as follows

```
double callOptionPrice(double S,double t,double X,double r,double sigma,double T)
{
  double d1=(log(S/X) + (r+sigma*sigma/2.)*(T-t))/(sigma*sqrt(T-t));
  double d2=(log(S/X) + (r-sigma*sigma/2.)*(T-t))/(sigma*sqrt(T-t));
  return normalDistribution(d1)*S - normalDistribution(d2)*X*exp(-r*(T-t));
```

```
}

int main()
{
  cout << "Call Option Price = " << callOptionPrice(1,0,1,0.05,0.2,1) << endl;
  return 0;
}
```

download

Now we need to test the function under different settings. There are obviously going to be problem in each of the following cases:

- $S = 0$
- $\sigma = 0$
- $t = T$

since all will result in undefined mathematical values given the way that d1 and d2 are calculated. However, returning a value of plus or minus infinity for d1 and d2 does not result in an undefined value for the cumulative normal distribution since the function returns finite values from infinite limits. Using our knowledge of what happens to this function we can go through each case and decide what are the appropriate values to return.

Case S=0

In this situation we must make use of the boundary condition of the problem, which is that the option value is worthless if the asset value is zero. The only slight caveat here is that you should check whether S is smaller than a very small number *relative* to the strike price rather than comparing it with zero. In code you add the following line to your function before the calculations.

```
if(S<1.e-14*X)return 0.;
```

download

By executing a return the function will stop and return the value without executing any more code.

Case sigma=0

This case is very similar to when t=T. Depending on the sign of the numerator in the calculation for d1 and d2 the function will either return zero or the asset minus the discounted strike price. In code this looks like

```
if(sigma*sigma*fabs(T-t)<1.e-14)
  {
    if(S<X*exp(-r*(T-t)))return 0.;
    else return S-X*exp(-r*(T-t));
  }
```

download

Case t=T

Finally if t and T are almost equal then we are at maturity and we return the payoff. The code might look like

```
  if(fabs(T-t)<1.e-14)
  {
    if(S<X)return 0.;
    else return S-X;
  }
```

download

On adding each of these parts to the code you should test and validate each part using a range of parameters. Note here that another case that could cause problems is if $t > T$. There is no sensible value that can be returned in this case so if you add in a check for it you would be looking to exit the program if this happened or at least print a warning to the screen. Adding this in is left as an exercise. Your final code should look like

```
#include <iostream>
#include <cmath>
using namespace std;

double normalDistribution(double x)
{
  static const double RT2PI = sqrt(4.0*acos(0.0));
  static const double SPLIT = 10./sqrt(2);
  static const double a[] =
      {220.206867912376,221.213596169931,112.079291497871,33.912866078383,6.37396220353165,0.700
      e-02};
  static const double b[] =
      {440.413735824752,793.826512519948,637.333633378831,296.564248779674,86.7807322029461,16.0
      e-02};

  const double z = fabs(x);
  double Nz = 0.0;

  // if z outside these limits then value effectively 0 or 1 for machine precision
  if(z<=37.0)
  {
    // NDash = N'(z) * sqrt{2\pi}
    const double NDash = exp(-z*z/2.0)/RT2PI;
    if(z<SPLIT)
    {
      const double Pz = (((((a[6]*z + a[5])*z + a[4])*z + a[3])*z + a[2])*z + a[1])
          *z + a[0];
      const double Qz = ((((((b[7]*z + b[6])*z + b[5])*z + b[4])*z + b[3])*z + b
          [2])*z + b[1])*z + b[0];
      Nz = RT2PI*NDash*Pz/Qz;
    }
    else
    {
      const double F4z = z + 1.0/(z + 2.0/(z + 3.0/(z + 4.0/(z + 13.0/20.0))));
      Nz = NDash/F4z;
```

```
    }
  }
  return x>=0.0 ? 1-Nz : Nz;
}

// return the value of a call option using the black scholes formula
double callOptionPrice(double S,double t,double X,double r,double sigma,double T)
{
  if(fabs(T-t)<1.e-14) // check if we are at maturity
  {
    if(S<X)return 0.;
    else return S-X;
  }
  if((T-t)<=-1.e-14)return 0.; // option expired
  if(X<1.e-14*S)return S-X*exp(-r*(T-t)); // check if strike << asset then exercise
      with certainty
  if(S<1.e-14*X)return 0.; // check if asset << strike then worthless
  if(sigma*sigma*(T-t)<1.e-14) // check if variance very small then no diffusion
  {
    if(S<X*exp(-r*(T-t)))return 0.;
    else return S-X*exp(-r*(T-t));
  }
  // calculate option price
  double d1=(log(S/X) + (r+sigma*sigma/2.)*(T-t))/(sigma*sqrt(T-t));
  double d2=(log(S/X) + (r-sigma*sigma/2.)*(T-t))/(sigma*sqrt(T-t));
  return normalDistribution(d1)*S - normalDistribution(d2)*X*exp(-r*(T-t));
}

int main()
{
  cout << "Call Option Price = " << callOptionPrice(1,0,1,0.05,0.2,1) << endl;
  return 0;
}
```

download

Similarly for a put option we would have

```
#include <iostream>
#include <cmath>
using namespace std;

double normalDistribution(double x)
{
  static const double RT2PI = sqrt(4.0*acos(0.0));
  static const double SPLIT = 10./sqrt(2);
  static const double a[] =
      {220.206867912376,221.213596169931,112.079291497871,33.912866078383,6.37396220353165,0.700
      e-02};
  static const double b[] =
      {440.413735824752,793.826512519948,637.333633378831,296.564248779674,86.7807322029461,16.0
      e-02};

  const double z = fabs(x);
  double Nz = 0.0;

  // if z outside these limits then value effectively 0 or 1 for machine precision
```

```
  if(z<=37.0)
  {
    // NDash = N'(z) * sqrt{2\pi}
    const double NDash = exp(-z*z/2.0)/RT2PI;
    if(z<SPLIT)
    {
      const double Pz = (((((a[6]*z + a[5])*z + a[4])*z + a[3])*z + a[2])*z + a[1])
          *z + a[0];
      const double Qz = ((((((b[7]*z + b[6])*z + b[5])*z + b[4])*z + b[3])*z + b
          [2])*z + b[1])*z + b[0];
      Nz = RT2PI*NDash*Pz/Qz;
    }
    else
    {
      const double F4z = z + 1.0/(z + 2.0/(z + 3.0/(z + 4.0/(z + 13.0/20.0))));
      Nz = NDash/F4z;
    }
  }
  return x>=0.0 ? 1-Nz : Nz;
}

// return the value of a put option using the black scholes formula
double putOptionPrice(double S,double t,double X,double r,double sigma,double T)
{
  if(fabs(T-t)<1.e-14) // check if we are at maturity
  {
    if(S<X)return X-S;
    else return 0;
  }
  if((T-t)<=-1.e-14)return 0.; // option expired
  if(X<1.e-14*S)return 0.; // check if strike << asset then exercise with certainty
  if(S<1.e-14*X)return X*exp(-r*(T-t)) - S; // check if asset << strike then
      worthless
  if(sigma*sigma*(T-t)<1.e-14) // check if variance very small then no diffusion
  {
    if(S<X*exp(-r*(T-t)))return X*exp(-r*(T-t)) - S;
    else return 0.;
  }
  // calculate option price
  double d1=(log(S/X) + (r+sigma*sigma/2.)*(T-t))/(sigma*sqrt(T-t));
  double d2=(log(S/X) + (r-sigma*sigma/2.)*(T-t))/(sigma*sqrt(T-t));
  return normalDistribution(-d2)*X*exp(-r*(T-t)) - normalDistribution(-d1)*S ;
}

int main()
{
  cout << "Put Option Price = " << putOptionPrice(1,0,1,0.05,0.2,1) << endl;
  return 0;
}
```

download

Other Options

For a binary call we get

```
#include <iostream>
#include <cmath>
using namespace std;

double normalDistribution(double x)
{
  static const double RT2PI = sqrt(4.0*acos(0.0));
  static const double SPLIT = 10./sqrt(2);
  static const double a[] =
      {220.206867912376,221.213596169931,112.079291497871,33.912866078383,6.37396220353165,0.700
      e-02};
  static const double b[] =
      {440.413735824752,793.826512519948,637.333633378831,296.564248779674,86.7807322029461,16.0
      e-02};

  const double z = fabs(x);
  double Nz = 0.0;

  // if z outside these limits then value effectively 0 or 1 for machine precision
  if(z<=37.0)
  {
    // NDash = N'(z) * sqrt{2\pi}
    const double NDash = exp(-z*z/2.0)/RT2PI;
    if(z<SPLIT)
    {
      const double Pz = (((((a[6]*z + a[5])*z + a[4])*z + a[3])*z + a[2])*z + a[1])
          *z + a[0];
      const double Qz = ((((((b[7]*z + b[6])*z + b[5])*z + b[4])*z + b[3])*z + b
          [2])*z + b[1])*z + b[0];
      Nz = RT2PI*NDash*Pz/Qz;
    }
    else
    {
      const double F4z = z + 1.0/(z + 2.0/(z + 3.0/(z + 4.0/(z + 13.0/20.0))));
      Nz = NDash/F4z;
    }
  }
  return x>=0.0 ? 1-Nz : Nz;
}

// return the value of a put option using the black scholes formula
double binaryCallOptionPrice(double S,double t,double X,double r,double sigma,
    double T)
{
  if(fabs(T-t)<1.e-14) // check if we are at maturity
  {
    if(S>X)return 1;
    else return 0;
  }
  if((T-t)<=-1.e-14)return 0.; // option expired
  if(X<1.e-14*S)return exp(-r*(T-t)); // check if strike << asset then exercise
      with certainty
  if(S<1.e-14*X)return 0.; // check if asset << strike then worthless
  if(sigma*sigma*(T-t)<1.e-14) // check if variance very small then no diffusion
  {
    if(S>X*exp(-r*(T-t)))return exp(-r*(T-t));
```

```
    else return 0.;
  }
  // calculate option price
  double d2=(log(S/X) + (r-sigma*sigma/2.)*(T-t))/(sigma*sqrt(T-t));
  return normalDistribution(d2)*exp(-r*(T-t));
}

int main()
{
  cout << "Binary Call Option Price = " << binaryCallOptionPrice(1,0,1,0.05,0.2,1)
     << endl;
  return 0;
}
```

download

and for a binary put

```
#include <iostream>
#include <cmath>
using namespace std;

double normalDistribution(double x)
{
  static const double RT2PI = sqrt(4.0*acos(0.0));
  static const double SPLIT = 10./sqrt(2);
  static const double a[] =
      {220.206867912376,221.213596169931,112.079291497871,33.912866078383,6.37396220353165,0.700
      e-02};
  static const double b[] =
      {440.413735824752,793.826512519948,637.333633378831,296.564248779674,86.7807322029461,16.0
      e-02};

  const double z = fabs(x);
  double Nz = 0.0;

  // if z outside these limits then value effectively 0 or 1 for machine precision
  if(z<=37.0)
  {
    // NDash = N'(z) * sqrt{2\pi}
    const double NDash = exp(-z*z/2.0)/RT2PI;
    if(z<SPLIT)
    {
      const double Pz = (((((a[6]*z + a[5])*z + a[4])*z + a[3])*z + a[2])*z + a[1])
          *z + a[0];
      const double Qz = ((((((b[7]*z + b[6])*z + b[5])*z + b[4])*z + b[3])*z + b
          [2])*z + b[1])*z + b[0];
      Nz = RT2PI*NDash*Pz/Qz;
    }
    else
    {
      const double F4z = z + 1.0/(z + 2.0/(z + 3.0/(z + 4.0/(z + 13.0/20.0))));
      Nz = NDash/F4z;
    }
  }
  return x>=0.0 ? 1-Nz : Nz;
}
```

```
// return the value of a put option using the black scholes formula
double binaryPutOptionPrice(double S,double t,double X,double r,double sigma,double
    T)
{
  if(fabs(T-t)<1.e-14) // check if we are at maturity
  {
    if(S<X)return 1;
    else return 0;
  }
  if((T-t)<=-1.e-14)return 0.; // option expired
  if(X<1.e-14*S)return 0.; // check if strike << asset then exercise with certainty
  if(S<1.e-14*X)return exp(-r*(T-t)); // check if asset << strike then worthless
  if(sigma*sigma*(T-t)<1.e-14) // check if variance very small then no diffusion
  {
    if(S<X*exp(-r*(T-t)))return exp(-r*(T-t));
    else return 0.;
  }
  // calculate option price
  double d2=(log(S/X) + (r-sigma*sigma/2.)*(T-t))/(sigma*sqrt(T-t));
  return normalDistribution(-d2)*exp(-r*(T-t));
}

int main()
{
  cout << "Binary Put Option Price = " << binaryPutOptionPrice(1,0,1,0.05,0.2,1) <<
      endl;
  return 0;
}
```

download

1.3 Coursework Example

A trader has asked you to price the value of the call option $C(S,t)$ at time $t=0$ according to the standard Black-Scholes formula, where $T=1$, $X=1$, $r=0.05$ and $\sigma=0.2$. Write a program to calculate C and output the results to screen. You must generate four columns of data:

- the value of S;
- the value of d_1;
- the value of d_1;
- the value of $C(S,t=0)$.

Output each of the values when the stock price is

$$S \in \{0.8, 0.9, 1, 1.1, 1.2\}.$$

You should use a `for` loop to generate the data.

We could do this with the call option code above, but to make outputting d_1 and d_2 easier lets just calculate them again. First, open up a new project and copy in the code from earlier on with normal distribution, get it by (clicking here). Then simply start by editing the main function so that has variables for all the required inputs (S, t, T, X, r and σ), and then calculate d_1 and $N(d_1)$.

```
int main()
{
  double S=1.,t=0.,X=1.,r=0.05,sigma=0.2,T=1.;
  double d_1=(log(S/X) + (r+sigma*sigma/2.)*(T-t))/sigma/sqrt(T-t);
  cout << "d_1 = " << d_1 << endl;
  cout << "N(d_1) = " << normalDistribution(d_1) << endl;
  return 0;
}
```

download

Once you have checked this result, do the same for d_2 and $N(d_2)$ then calculate the call option price

```
int main()
{
  double S=1.,t=0.,X=1.,r=0.05,sigma=0.2,T=1.;
  double d_1=(log(S/X) + (r+sigma*sigma/2.)*(T-t))/sigma/sqrt(T-t);
  double d_2=(log(S/X) + (r-sigma*sigma/2.)*(T-t))/sigma/sqrt(T-t);
  cout << "d_1 = " << d_1 << endl;
  cout << "N(d_1) = " << normalDistribution(d_1) << endl;
  cout << "d_2 = " << d_2 << endl;
  cout << "N(d_2) = " << normalDistribution(d_2) << endl;
  cout << "C = " << S*normalDistribution(d_1) - X*exp(-r*(T-t))*normalDistribution(
      d_2) << endl;
  return 0;
}
```

download

If you are happy everything is working then run a loop to calculate values for different values of S, something like this should work

```
int main()
{
  double S=1.,t=0.,X=1.,r=0.05,sigma=0.2,T=1.;

  for(int i=8;i<=12;i++)
  {
          S = i*0.1;
          cout << S;
          double d_1=(log(S/X) + (r+sigma*sigma/2.)*(T-t))/sigma/sqrt(T-t);
          double d_2=(log(S/X) + (r-sigma*sigma/2.)*(T-t))/sigma/sqrt(T-t);
          cout << " " << d_1;
          cout << " " << d_2;
```

```
            cout << " " << S*normalDistribution(d_1) - X*exp(-r*(T-t))*
                normalDistribution(d_2) << endl;
    }
    return 0;
}
```

download

Now you have your solution, export this manually (from terminal output) or using a filestream in c++ so that you can create a table. Your final result should look something like this:-

S	d_1	d_2	$C(S,t)$
0.8	-0.765718	-0.965718	0.0185942
0.9	-0.176803	-0.376803	0.0509122
1	0.35	0.15	0.104506
1.1	0.826551	0.626551	0.17663
1.2	1.26161	1.06161	0.26169

Table 1: A table showing the value of a call option for different values of the underlying.

References

West, Graeme. Better approximations to cumulative normal functions. *Wilmott Magazine*, 9: 70–76, 2005. URL `https://lyle.smu.edu/~aleskovs/emis/sqc2/accuratecumnorm.pdf`.