Financial Numerical Recipes in C++.

Bernt Arne Ødegaard

December 10, 2003

Contents

This book is a a discussion of the calculation of specific formulas in finance. The field of finance has seen a rapid development in recent years, with increasing mathematical sophistication. While the formalization of the field can be traced back to the work of Markowitz [1952] on investors mean-variance decisions and Modigliani and Miller [1958] on the capital structure problem, it was the solution for the price of a call option by Black and Scholes [1973], Merton [1973] which really was the starting point for the mathematicalization of finance. The fields of derivatives and fixed income have since then been the main fields where complicated formulas are used. This book is intended to be of use for people who want to both understand and use these formulas, which explains why most of the algorithms presented later are derivatives prices.

This project started when I was teaching a course in derivatives at the University of British Columbia, in the course of which I sat down and wrote code for calculating the formulas I was teaching. I have always found that implementation helps understanding these things. For teaching such complicated material it is often useful to actually look at the implementation of how the calculation is done in practice. The purpose of the book is therefore primarily pedagogical, although I believe all the routines presented are correct and reasonably efficient, and I know they are also used by people to price real options.

To implement the algorithms in a computer language I choose C++. My students keep asking why anybody would want to use such a backwoods computer language, they think a spreadsheet can solve all the worlds problems. I have some experience with alternative systems for computing, and no matter what, in the end you end up being frustrated with higher end "languages", such as *Matlab* og *Gauss* (Not to mention the straitjacket which is is a spreadsheet.) and going back to implementation in a standard language. In my experience with empirical finance I have come to realize that nothing beats knowledge a **real** computer language. This used to be `FORTRAN`, then `C`, and now it is **C++** . All example algorithms are therefore coded in C++.

The manuscript has been sitting on the internet for a number of years, during which it has been visited by a large number of people, to judge by the number of mails I have received about the routines. The present (2003) version mainly expands on the background discussion of the routines, this is much more extensive. I have also added a good deal of introductory material on how to program in C++, since a number of questions make it obvious this manuscript is used by a number of people who know finance but not C++. All the routines have been made to confirm to the new ISO/ANSI C++ standard, using such concepts as namespaces and the standard template library.

The current manscript therefore has various intented audiences. Primarily it is for students of finance who desires to see a complete discussion and implementation of some formula. But the manuscript is also useful for students of finance who wants to learn C++, and for computer scientists who want to understand about the finance algorithms they are asked to implent and embed into their programs.

In doing the implementation I have tried to be as generic as possible in terms of the C++ used, but I have taken advantage of a some of the possibilities the language provides in terms of abstraction and modularization. This will also serve as a lesson in why a *real* computer language is useful. For example I have encapsulated the term structure of interest rate as an example of the use of `classes`.

This is not a textbook in the underlying theory, for that there are many good alternatives. For much of the material the best textbooks to refer to are Hull [2003] and McDonald [2002], which I have used as references, and the notation is also similar to these books.

Chapter 1

On C++ and programming.

Contents

In this chapter I introduce C++ and discuss how to run programs written in C++. This is by no means a complete reference to programming in C++, it is designed to give enough information to understand the rest of the book. This chapter also only discusses a subset of C++, it concentrates on the parts of the language used in the remainder of this book. For really learning C++ a textbook is necessary. I have found Lippman and Lajoie [1998] an excellent introduction to the language. The authorative source on the language is Stroustrup [1997].

1.1 Compiling and linking

To program in C++ one has to first write a separate file with the program, which is then *compiled* into low-level instructions (machine language) and *linked* with libraries to make a complete executable program. The mechanics of doing the compiling and linking varies from system to system, and we leave these details as an exercise to the reader.

1.2 The structure of a C++ program

The first thing to realize about C++ is that it is a strongly typed language. Everything must be declared before it is used, both variables and functions. C++ has a few basic building blocks, which can be grouped into types, operations and functions.

1.2.1 Types

The types we will work with in this book are `bool`, `int`, `long`, `double` and `string`.

Here are some example definitions

```
bool this_is_true=true;
int i = 0;
long j = 123456789;
double pi = 3.1415926535897932384626 43;
string s("this is a string");
```

The most important part of C++ comes from the fact that these basic types can be expanded by use of *classes*, of which more later.

1.2.2 Operations

To these basic types the common mathematical operations can be applied, such as addition, subtraction, multiplication and division:

```
int i = 100 + 50;
int j = 100 - 50;
int n = 100 * 2;
int m = 100 / 2;
```

These operations are defined for all the common datatypes, with exception of the `string` type. Such operations can be defined by the programmer for other datatypes as well.

Increment and decrement In addition to these basic operations there are some additional operations with their own shorthand. An example we will be using often is incrementing and decrementing a variable. When we want to increase the value of one item by one, in most languages this is written:

```
int i=0;
i = i+1;
i = i-1;
```

In `C++` this operation has its own shorthand

```
int i=0;
i++;
i--;
```

While this does not seem intuitive, and it is excusable to think that this operation is not really necessary, it does come in handy for more abstract data constructs. For example, as we will see later, if one defines a `date` class with the necessary operations, to get the next date will simply be a matter of

```
date d(1,1,1995);
d++;
```

These two statements will result in the date in `d` being 2jan95.

1.2.3 Functions and libraries

In addition to the basic mathematical operations there is a large number of additional operations that can be performed on any type. However, these are not parts of the core language, they are implemented as standalone functions (most of which are actually written in `C` or C++). These functions are included in the large *library* that comes with any C++ installation. Since they are not part of the core language they must be defined to the compiler before they can be used. Such definitions are performed by means of the *include* statement.

For example, the mathematical operations of taking powers and performing exponentiation are defined in the mathematical library `cmath`. In the C++ program one will write

```
#include <cmath>
```

`cmath` is actually a file with a large number of function defintions, among which one finds `pow(x,n)` which calculates x^n, and `exp(r)` which calculates e^r. The following programming stub calculates $a = 2^2$ and $b = e^1$.

```
#include <cmath>
double a = pow(2,2);
double b = exp(1);
```

which will give the variables `a` and `b` values of 4 and 2.718281828..., respectively.

1.2.4 Templates and libraries

The use of libraries is not only limited to functions. Also included in the standard library is generic data structures, which can be used on any data type. The example we will be considering the most is the `vector<>`, which defines an array, or vector of variables.

```
#include <vector>
vector<double> M(2);
M[0]=1.0;
M[1]=2.0;
M.push_back(3);
```

This example defines an array with three elements of type double

$$M = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Note some pecularities here. When first defining the vector with the statement

```
vector<double> M(2);
```

we defined an array of 2 elements of type `double`, which we then proceeded to fill with the values 1 and 2. When filling the array we addressed each element directly. Note that in the statement

```
M[0]=1.0;
```

lies one of the prime traps for programmers coming to `C` or C++ from another language. Indexing of arrays starts at zero, not at one. `M[0]` really means the *first* element of the array.
The last statement,

```
M.push_back(3);
```

shows the ability of the programmer of changing the size of the array after it has been defined. `push_back` is a standard operation on arrays which "pushes" the element onto the back of the array, extending the size of the array by one element. Most programming languages do not allow the programmer to specify variable-sized arrays "on the fly." In `FORTRAN` or `Pascal` we would usually have to set a maximum length for each array, and hope that we would not need to exceed that length. The `vector`<> template of C++ gets rid of the programmers need for "bookkeeping" in such array manipulations.

1.2.5 Flow control

To repeat statements several times one will use on of the possibilities for flow control, such as the `for` or `while` constucts. For example, to repeat an operation `n` times one can use the following `for` loop:

```
for (int i=0; i<n; i++) {
   some_operation(i);
};
```

The `for` statement has tree parts. The first part gives the initial condition (`i=0`). The next part the terminal condition (i<n), which says to stop when `i<n` is not fulfilled, which is at the `n`'th iteration. The last part is the increment statement (`i++`), saying what to do in each iteration. In this case the value of `i` is increased by one in each iteration. This is the typical `for` statement. One of the causes of `C`'s reputation for terseness is the possibility of elaborate `for` constructs, which end up being almost impossible to read. In the algorithms presented in this book we will try to avoid any obfuscated `for` statements, and stick to the basic cases.

1.2.6 Input Output

For any program to do anything useful it needs to be able to output its results. Input and output operations is defined in a couple of libraries, `iostream` and `fstream`. The first covers in/output to standard terminals and the second in/output to files.

To write to standard output `cout` (the terminal), one will do as follows:

```
#include <iostream>
cout << "This is a test" << endl;
```

To write to a file "`test.out`", one will do as follows:

```
#include <fstream>
ofstream outf;
outf.open("test.out");
outf << "This is a test" << endl;
outf.clear();
outf.close();
```

1.2.7 Splitting up a program

Any nontrivial program in C++ is split into several pieces. Usually each piece is written as a function which returns a value of a given type. To illustrate we provide a complete example program, shown in **Code 1.1**.

The program defines a function performing the mathematical power operation, `power(x,n)` which calculates x^n through the simple identity $x^n = e^{n \ln(x)}$. This function is then used to calculate and print the first 5 powers of 2.

```
#include <iostream> // input output operations
#include <cmath> // mathematics library
using namespace std; // the above is part of the standard namespace

double power(double x, double n){
    // define a simple power function
    double p = exp(n*log(x));
    return p;
};

int main(){
    for (int n=1;n<6;n++){
        cout << " 2^" << n << " = " << power(2,n) << endl;
    };
};
```

Code 1.1: A complete program

When compiled, linked and run, the program will provide the following output

```
 2^1 = 2
```

```
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
```

1.2.8 Namespaces

To help in building large programs, the concept of a `namespace` was introduced. Namespaces are a means of keeping the variables and functions defined local to the context in which they are used. For now it is necessary to know that any function in the standard C++ library lies in its own namespace, called the standard namespace. To actually access these library functons it is necessary to explicitly specify that one wants to access the standard namespace, by the statement

```
using namespace std;
```

Instead of such a general approach, one can also specify the namespace on an element by element basis, but this is more a topic for specialized C++ texts, for the current purposes we will allow all routines access to the whole standard namespace.

1.3 Extending the language, the `class` concept.

One of the major advances of C++ relative to other programming languages is the programmers ability to extend the language by creating new data types and defining standard operations on these data types. This ability is why C++ is called an object oriented programming language, since much of the work in programming is done by creating *objects*. An object is best though of as a data structure with operations on it defined. How one uses an object is best shown by an example.

1.3.1 `date`, an example class

Consider the abstract concept of a date. A date can be specified in any number of ways. Let us limit ourselves to the Gregorian calendar. 12 august 2003 is a common way of specifying a date. However, it can also be represented by the strings: "`2003/8/12`", "`12/8/2003`" and so on, or by the number of years since 1 january 1900, the number of months since January, and the day of the month (which is how a UNIX programmer will think of it).

However, for most people writing programs the representation of a date is not relevant, they want to be able to enter dates in some abstract way, and then are conserned with such questions as:

- Are two dates equal?
- Is one date earlier than another?
- How many days is it between two dates?

A C++ programmer will proceed to use a *class* that embodies these uses of the concept of a date. Typically one will look around for an extant class which has already implemented this, but we will show a trivial such date class as an example of how one can create a class.

A class is defined in a header file, as shown in code 1.2. A number of things is worth noting here. As internal representation of the date is chosen the three integers `day_`, `month_` and `year_`. This is the data structure which is then manipulated by the various functions defined below.

The functions are used to

- Create a date variable: `date(const int& d, const int& m, const int& y);`
- Functions outputting the date by the three integer functions `day()`, `month()` and `year()`.

```
#include <string>
using namespace std;

class Date {
protected:
    int year_;
    int month_;
    int day_;
public:
    date(const int& d, const& int m, const& int y);

    int day()  const;
    int month() const;
    int year() const;

    void set_day (const int& day );
    void set_month (const int& month );
    void set_year (const int& year );

    date operator ++(); // prefix
    date operator ++(int); // postfix
    date operator --(); // prefix
    date operator --(int); // postfix
};

bool operator == (const date&, const date&); // comparison operators
bool operator != (const date&, const date&);
bool operator < (const date&, const date&);
bool operator > (const date&, const date&);
bool operator <= (const date&, const date&);
bool operator >= (const date&, const date&);
```

Code 1.2: Defining a `date` class

- Functions setting the date `set_day(int)`, `set_month(int)` and `set_year(int)`, which are used by providing an integer as arguments to the function.
- Increment and decrement functions `++` and `--`
- Comparison functions `<`, `<=`, `>`, `>=`, `==` and `!-`.

After including this header file, programmers using such a class will then treat an object of type `date` just like any other.

For exmple,

```
date d(1,1,2001);
++d;
```

would result in the `date` object `d` containing the date 2 january 2001.

Any C++ programmer who want to use this `date` object will only need to look at the header file to know what are the possible functions one can use with a `date` object, and be happy about not needing to know anything about how these functions are implemented. This is the encapsulation part of object oriented programming, all relevant information about the `date` object is specified by the header file. This is the only point of interaction, all details about implementation of the class objects and its functions is not used in code using this object.

1.4 `Const` references

Consider two alternative calls to a function, defined by function calls:

```
some_function(double r);
some_function(const double& r);
```

They both are called by an argument which is a double, and that argument is guaranteed to not be changed in the calling function, but they work differently. In the first case a copy of the variable referenced to in the argument is created for use in the function, but in the second case one uses the same variable, the argument is a *reference* to the location of the variable. The latter is more efficient, in particular when the argument is a large class. However, one worries that the variable referred to is changed in the function, which in most cases one do not want. Therefore the `const` qualifier, it says that the function can not modify its argument. The compiler will warn the programmer if an attempt is made to modify such a variable.

For efficiency, in most of the following routines arguments are therefore given as as constant references.

Chapter 2

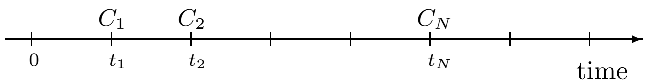
The value of time

Contents

Finance as a field of study is sometimes somewhat flippantly said to deal with the value of two things: *time* and *risk*. While this is not the whole story, there is a deal of truth in it. These are the two issues which is always present. We start our discussion by ignoring risk and only considering the implications of the fact that anybody prefers to get something earlier rather than later, or the value of time.

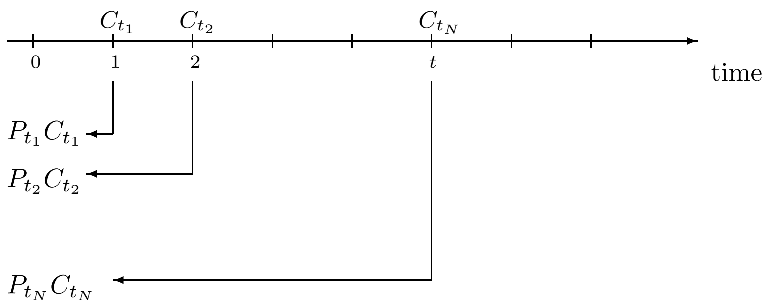
2.1 Present value.

The calculation of present value is one of the basics of finance. The present value is the current value of a stream of future payments. Let C_t be the cash flow at time t. Suppose we have N future cash flows that occur at times $t_1, t_2, \cdots, t_N$.



To find the *present* value of these future cash flows one need a set of prices of future cash flows. Suppose P_t is the price one would pay today for the right to recive one dollar at a future date t. If one knows this set of prices one would calculate the present value as the sum of the preent values of the different elements.

$$PV = \sum_{i=1}^{N} P_{t_i} C_{t_i}$$



However, knowing the set of prices of all future dates is not always feasible. As a first approximation we assume that there is one interest rate which is used for discounting, (this is termed a flat term structure),

and the prices of future payments P_t, which is also called a *discount factor*, is calculated from this interest rate.
The best known example, known as discrete compounding, is calculated as

$$P_t = \frac{1}{(1+r)^t},$$

In this case one would calculate the present value as

$$PV = \sum_{i=1}^{N} \frac{C_{t_i}}{(1+r)^t}$$

The implementation of this calculation is shown in code 2.1

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

double cash_flow_pv_discrete(const vector<double>& cflow_times,
                             const vector<double>& cflow_amounts,
                             const double& r){
    double PV=0.0;
    for ( int t=0; t<cflow_times.size();t++) {
        PV += cflow_amounts[t]/pow(1+r,cflow_times[t]);
    };
    return PV;
};
```

Code 2.1: Present value with discrete compounding

However, such discrete compounding is not the only alternative way to approximate the discount factor. The discretely compounded case assumes that interest is added at discrete points in time (hence the name). However, an alternative assumption is to assume that interest is added continously. If compounding is continous, and r is the interes rate, one would calculate the current price of reciving one dollar at a future date t as

$$P_t = e^{-rt},$$

which implies the following present value calculation:

$$PV = \sum_{i=1}^{N} e^{-rt_i} C_{t_i}$$

This calculation is implemented as shown in code 2.2. In much of what follows we will work with this continously compounded case. There is a number of reasons why, but a prime reason is actually that it is easier to use continously compounded interest than discretely compounded, because it is easier to deal with uneven time periods. Discretely compounded interest is easy to use with evenly spaced cash flows (such as annual cash flows), but harder otherwise.

2.2 Internal rate of return.

The internal rate of return of a set of cash flows is the interest rate that makes the present value of the cash flows equal to zero. Finding an internal rate of return is thus to find a root of of the equation

$$PV(C, t, r) = 0$$

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

double cash_flow_pv( const vector<double>& cflow_times,
                     const vector<double>& cflow_amounts,
                     const double& r){
    double PV=0.0;
    for (int t=0; t<cflow_times.size();t++) {
        PV += cflow_amounts[t] * exp(-r*cflow_times[t]);
    };
    return PV;
};
```

Code 2.2: Present value calculation with continously compounded interest

As any textbook in basic finance, such as Brealey and Myers [1996] or Ross et al. [1996] or will tell, there is a number of problems with using the IRR, most of them stemming from the possibility for more than one interest rate being defined.

If we know that there is one IRR, the following method is probably simplest, bisection. It is an adaption of the bracketing approach discussed in Press et al. [1992], chapter 9. Note that this approach will only find one interest rate, if there is more than one irr, the simplest is always to graph the PV as a function of interest rates, and use that to understand when an investment is a good one.

```
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;
#include "fin_recipes.h"

const double ERROR=-1e30;

double cash_flow_irr(const vector<double>& cflow_times,
                     const vector<double>& cflow_amounts) {
    // simple minded irr function. Will find one root (if it exists.)
    // adapted from routine in Numerical Recipes in C.
    if (cflow_times.size()!=cflow_amounts.size()) return ERROR;
    const double ACCURACY = 1.0e-5;
    const int MAX_ITERATIONS = 50;
    double x1=0.0;
    double x2 = 0.2;

    // create an initial bracket, with a root somewhere between bot,top
    double f1 = cash_flow_pv(cflow_times, cflow_amounts, x1);
    double f2 = cash_flow_pv(cflow_times, cflow_amounts, x2);
    int i;
    for (i=0;i<MAX_ITERATIONS;i++) {
        if ( (f1*f2) < 0.0) { break; }; //
        if (fabs(f1)<fabs(f2)) {
            f1 = cash_flow_pv(cflow_times,cflow_amounts, x1+=1.6*(x1-x2));
        }
        else {
            f2 = cash_flow_pv(cflow_times,cflow_amounts, x2+=1.6*(x2-x1));
        };
    };
    if (f2*f1>0.0) { return ERROR; };
    double f = cash_flow_pv(cflow_times,cflow_amounts, x1);
    double rtb;
    double dx=0;
    if (f<0.0) {
        rtb = x1;
        dx=x2-x1;
    }
    else {
        rtb = x2;
        dx = x1-x2;
    };
    for (i=0;i<MAX_ITERATIONS;i++){
        dx *= 0.5;
        double x_mid = rtb+dx;
        double f_mid = cash_flow_pv(cflow_times,cflow_amounts, x_mid);
        if (f_mid<=0.0) { rtb = x_mid; }
        if ( (fabs(f_mid)<ACCURACY) || (fabs(dx)<ACCURACY) ) return x_mid;
    };
    return ERROR; // error.
};
```

Code 2.3: Estimation of the internal rate of return

2.2.1 Check for unique IRR

If you worry about finding more than one IRR, the code shown in code 2.4 implements a simple check for that. It is only a neccesary condition for a unique IRR, not sufficient, so you may still have a well-defined IRR even if this returns false.

The first test is just to count the number of sign changes in the cash flow. From Descartes rule we know that the number of real roots is one if there is only one sign change. If there is more than one change in the sign of cash flows, we can go further and check the *aggregated* cash flows for sign changes (See Norstrom [1972], or Berck and Sydsæter [1995]).

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

inline int sgn(const double& r){ if (r>=0) {return 1;} else {return -1;}; };

bool cash_flow_unique_irr(const vector<double>& cflow_times,
                          const vector<double>& cflow_amounts) {
    int sign_changes=0; // first check Descartes rule
    for (int t=1;t<cflow_times.size();++t){
        if (sgn(cflow_amounts[t-1]) !=sgn(cflow_amounts[t])) sign_changes++;
    };
    if (sign_changes==0) return false; // can not find any irr
    if (sign_changes==1) return true;

    double A = cflow_amounts[0]; // check the aggregate cash flows, due to Norstrom
    sign_changes=0;
    for (int t=1;t<cflow_times.size();++t){
        if (sgn(A) != sgn(A+=cflow_amounts[t])) sign_changes++;
    };
    if (sign_changes<=1) return true;
    return false;
}
```

Code 2.4: Test for uniqueness of IRR

2.3 Bonds

In this part we look at the treatment of bonds and similar fixed income securities. What distinguishes bonds is that the future payments (of coupon, principal) are decided when the security is issued. We again limit ourselves to the case of a flat term structure, and return to bond pricing with more general term structures later.

2.3.1 Bond Price

The price of a bond is the present value of its future cashflows. If we consider a coupon bond like a US goverment bond (T-Bond), the cash flows look like

$t =$	0	1	2	3	$\cdots$	T
Coupon		C	C	C	$\cdots$	C
Face value						F

The current price of the bond is

$$P_0 = \sum_{t=1}^{T} \frac{C}{(1+r)^t} + \frac{F}{(1+r)^T}$$

with discrete compounding, and

$$P_0 = \sum_{t=1}^{T} e^{-rt} C + e^{-rT} F$$

with continous compounding. The interest rate r is fixed, which means that the term structure is "flat."

Let us look at two versions of the bond price algorithm for the continous case.

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

double bonds_price(const vector<double>& coupon_times,
                   const vector<double>& coupon_amounts,
                   const vector<double>& principal_times,
                   const vector<double>& principal_amounts,
                   const double& r) {
    double p = 0;
    for (int i=0;i<coupon_times.size();i++) {
        p += exp(-r*coupon_times[i])*coupon_amounts[i];
    };
    for (int i=0;i<principal_times.size();i++) {
        p += exp(-r*principal_times[i])*principal_amounts[i];
    };
    return p;
};
```

Code 2.5: Bond price

There are two version of the routine listed, one which is called with both interest and principal vectors (code 2.5) and another (code 2.6) which is simply called with the cashflows. I show both to make one think about the fact that for most purposes the distinction between coupons and principals is not necessary to make, what counts is the cashflows, which is the sum of coupons and principal. There are cases where the distinction is important, for example when taxes are involved. Then we need to keep track of what is interest

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

double bonds_price(const vector<double>& cashflow_times,
                   const vector<double>& cashflows,
                   const double& r) {
    double p=0;
    for (int i=0;i<cashflow_times.size();i++) {
        p += exp(-r*cashflow_times[i])*cashflows[i];
    };
    return p;
};
```

Code 2.6: Bond price

and what is principal. But in the simple cases considered here I stick to the case of one set of cashflows, it makes the routines easier to follow.

Let us also look at the case of discrete (annual) *compounding*, shown in code 2.7.

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

double bonds_price_discrete(const vector<double>& cashflow_times,
                            const vector<double>& cashflows,
                            const double& r) {
    double p=0;
    for (int i=0;i<cashflow_times.size();i++) {
        p +=cashflows[i]/(pow((1+r),cashflow_times[i]));
    };
    return p;
};
```

Code 2.7: Bond price

2.3.2 Yield to maturity.

The yield to maturity is the interest rate that makes the present value of the future coupon payments equal to the current bondprice, that is, for a known price P_0, the yield is the solution y to the equation

$$P_0 = \sum_{t=1}^{T} e^{-yt}C + e^{-yT}F$$

Note that this is the same as finding the internal rate of return, if we include in the cash flows the price of the bond as a negative cash flow "today". The algoritm shown in code 2.8 is simple bisection, we know that the yield is above zero, and find a maximium yield which the yield is below, and then bisect the interval until we are "close enough."

```
#include <cmath>
#include "fin_recipes.h"

double bonds_yield_to_maturity( const vector<double>& cashflow_times,
                                const vector<double>& cashflow_amounts,
                                const double& bondprice) {
  const float ACCURACY = 1e-5;
  const int MAX_ITERATIONS = 200;
  double bot=0, top=1.0;
  while (bonds_price(cashflow_times, cashflow_amounts, top) > bondprice) {
    top = top*2;
  };
  double r = 0.5 * (top+bot);
  for (int i=0;i<MAX_ITERATIONS;i++){
    double diff = bonds_price(cashflow_times, cashflow_amounts,r) - bondprice;
    if (std::fabs(diff)<ACCURACY) return r;
    if (diff>0.0) { bot=r;}
    else          { top=r; };
    r = 0.5 * (top+bot);
  };
  return r;
};
```

Code 2.8: Bond yield

2.3.3 Duration.

The duration of a bond is the "weighted average maturity" of the bond.

$$\text{Duration} = \sum_t \frac{t C_t P_t}{\text{Bond Price}}$$

where

- C_t is the cash flow in period t, and
- P_t is the discount factor, the current price of a discount bond paying \$1 at time t.

There are two versions of the duration calculation, based on how one estimate the bond price. One assumes the current interest rate r and calculates

$$\text{Bond price} = \sum_{t=1}^{T} e^{-rt} C_t$$

which gives the duration calculation shown in code 2.9

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

double bonds_duration(const vector<double>& cashflow_times,
                      const vector<double>& cashflows,
                      const double& r) {
    double S=0;
    double D1=0;
    for (int i=0;i<cashflow_times.size();i++){
        S  +=                      cashflows[i] * exp(-r*cashflow_times[i]);
        D1 += cashflow_times[i] * cashflows[i] * exp(-r*cashflow_times[i]);
    };
    return D1 / S;
};
```

Code 2.9: Bond duration

Alternatively one can calculate the yield to maturity for the bond, and use that in estimating the bond price. This is called *Macaulay Duration* First one calculates y, the yield to maturity, from

$$\text{Bond price} = \sum_{t=1}^{T} e^{-yt} C_t$$

and then use this y in the duration calculation:

$$\text{Macaulay duration} = \sum \frac{t C_t e^{-yt}}{\sum e^{-yt} C_t}$$

Code 2.10

Modified Duration

$$\text{Modified Duration} = \frac{\text{duration}}{\text{yield}}$$

```cpp
#include "fin_recipes.h"

double bonds_duration_macaulay(const vector<double>& cashflow_times,
                               const vector<double>& cashflows,
                               const double& bond_price) {
  double y = bonds_yield_to_maturity(cashflow_times, cashflows, bond_price);
  return bonds_duration(cashflow_times, cashflows, y); // use YTM in duration
};
```

Code 2.10: Calculating the Macaulay duration of a bond

```cpp
#include <vector>
#include "fin_recipes.h"

double bonds_duration_modified (const vector<double>& cashflow_times,
                                const vector<double>& cashflow_amounts,
                                const double& bond_price,
                                const double& r){
    double dur = bonds_duration(cashflow_times, cashflow_amounts, r);
    double y = bonds_yield_to_maturity(cashflow_times, cashflow_amounts, bond_price);
    return dur/(1+y);
};
```

Code 2.11: Modified duration

Bond convexity

Convexity measures the curvature of the approximation done when using duration. It is calculated as

$$\sum_{i=1}^{n} c_i t_i^2 e^{-y t_i}$$

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

double bonds_convexity(const vector<double>& cashflow_times,
                       const vector<double>& cashflow_amounts,
                       const double& y ) {
    double C=0;
    for (int i=0;i<cashflow_times.size();i++){
        double t = cashflow_times[i];
        C+= cashflow_amounts[i] * t * t * exp(-y*t);
    };
    return C;
};
```

Code 2.12: Bond convexity

Chapter 3

The term structure of interest rates and an object lesson

Contents

In this chapter we look at various algorithms that has been used to estimate a "term structure," i e a relation between length of period for investment and interest rate.

The term structure is the current price for a future (time t) payment of one dollar (discount factor). It can also be viewed as the yield on a zero coupon bond paying one dollar at time t. Alternatively one can think about forward interest rates, the yield on borrowing at some future date t_1 and repaying it at a later date t_2. Knowing one of these one also knows the others, since there are one-to-one transforms moving one into the other.

3.1 Term structure calculations

Let us show some useful transformations for moving between these three alternative views of the term structure. . Let $r(t)$ be the yield on a t–period discount bond, and $d(t)$ the discount factor for time t (the current price of a bond that pays \$1 at time t. Then

$$d(t) = e^{-r(t)t}$$

$$r(t) = \frac{-\ln(d(t))}{t}$$

Also, the forward rate for borrowing at time t_1 for delivery at time T is calculated as

$$f_t(t_1, T) = \frac{-\ln\left(\frac{d(T)}{d(t_1)}\right)}{T - t_1} = \frac{\ln\left(\frac{d(t_1)}{d(T)}\right)}{T - t_1}$$

The forward rate can also be calculated directly from yields as

$$f_d(t, t_1, T) = r_d(t, T)\frac{T - t}{T - t_1} - r_d(t, t_1)\frac{t_1 - t}{T - t_1}$$

Note that this assumes continously compounded interest rates.

Code 3.1 shows the implementation of these transformations.

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

double term_structure_yield_from_discount_factor(const double& dfact,
                                                 const double& t) {
    return (-log(dfact)/t);
}

double term_structure_discount_factor_from_yield(const double& r,
                                                 const double& t) {
    return exp(-r*t);
};

double term_structure_forward_rate_from_disc_facts(const double& d_t,
                                                   const double& d_T,
                                                   const double& time) {
    return (log (d_t/d_T))/time;
};

double term_structure_forward_rate_from_yields(const double& r_t1,
                                               const double& r_T,
                                               const double& t1,
                                               const double& T) {
    return (r_T*(T/(T-t1))-r_t1*(t1/T));
};
```

Code 3.1: Term structure transformations

3.2 Using the currently observed term structure.

To just use todays term structure, we need to take the observations of yields that is observed in the market and use these to generate a term structure. The simplest possible way of doing this is to linearly interpolate the currently observable zero coupon yields.

3.2.1 Linear Interpolation.

If we are given a set of yields for various maturities, the simplest way to construct a term structure is by straightforward linear interpolation between the observations we have to find an intermediate time. For many purposes this is "good enough." This interpolation can be on either yields, discount factors or forward rates, we illustrate the case of linear interpolation of spot rates.

Computer algorithm, linear interpolation of yields. Note that the algorithm assumes the yields are ordered in increasing order of time to maturity.

3.3 The term structure as an object

To actually use the term structure one has to specify one of these three alternative views of the term stucuture for all possible future dates. This is of course not feasible, so one will need to have a method for specifying the term structure, such as the linear interpolation above. Next this term structure will have to be made available for use in other contexts. This is perfect for specifying a *class*, so we will use this as the prime example of the uses of a class. One can think of the term structure class as an abstract function that either return a `discount_factor` or a `yield`.

```
#include <vector>
using namespace std;
#include "fin_recipes.h"

double term_structure_yield_linearly_interpolated(const double& time,
                                                  const vector<double>& obs_times,
                                                  const vector<double>& obs_yields) {
    // assume the yields are in increasing time to maturity order.
    int no_obs = obs_times.size();
    if (no_obs<1) return 0;
    double t_min = obs_times[0];
    if (time <= t_min) return obs_yields[0]; // earlier than lowest obs.

    double t_max = obs_times[no_obs-1];
    if (time >= t_max) return obs_yields[no_obs-1]; // later than latest obs

    int t=1; // find which two observations we are between
    while ( (t<no_obs) && (time>obs_times[t])) { ++t; };
    double lambda = (obs_times[t]-time)/(obs_times[t]-obs_times[t-1]);
    // by ordering assumption, time is between t-1,t
    double r = obs_yields[t-1] * lambda + obs_yields[t] * (1.0-lambda);
    return r;
};
```

Code 3.2: Interpolated term structure from spot rates

3.4 Implementing a term structure class

A term structure can thus be abstractly described as a *function* of time. The user of a term structure will not need to know the underlying model of term structures, all that is needed is an interface that specifies functions for

- prices of zero coupon bonds (discount factors).
- yields of zero coupon bonds (spot rates).
- forward rates.

These will for given parameters and term structure models provide all that a user will want to use for calculations.

3.4.1 Base class

```
#ifndef _TERM_STRUCTURE_CLASS_H_
#define _TERM_STRUCTURE_CLASS_H_

class term_structure_class {
public:
    virtual double yield(const double& T) const;
    virtual double discount_factor(const double& T) const;
    virtual double forward_rate(const double&, const double&) const;
};

#endif
```

Code 3.3:

The code for these functions uses algorithms that are described earlier in this chapter for transforming between various views of the term structure. The term structure class merely provide a convenient interface to these algoritms.

```
#include "term_structure_class.h"
#include "fin_recipes.h"

double term_structure_class::forward_rate(const double& t1, const double& T) const{
   double d1 = discount_factor(t1);
   double dT = discount_factor(T);
   return term_structure_forward_rate_from_disc_facts(d1,dT,T-t1);
};

double term_structure_class::yield(const double& t) const{
    return term_structure_yield_from_discount_factor(discount_factor(t),t);
};

double term_structure_class::discount_factor(const double& t) const {
    return term_structure_discount_factor_from_yield(yield(t),t);
};
```

Code 3.4:

Note that the definitions of calculations are circular. Any given *specific* type of term structure has to over-ride at least one of the functions `yield`, `discount_factor` or `forward_rate`.

We next consider two examples of *specific* term structures.

3.4.2 Flat term structure.

The flat term structure overrides both the `yield` member function of the base class.

The only piece of data this type of term structure needs is an interest rate.

```
#ifndef _TERM_STRUCTURE_CLASS_FLAT_
#define _TERM_STRUCTURE_CLASS_FLAT_

#include "term_structure_class.h"

class term_structure_class_flat : public term_structure_class {
private:
    double R_;                              // interest rate
public:
    term_structure_class_flat(const double& r);
    virtual double yield(const double& t) const;
    void set_int_rate(const double& r);
};

#endif
```

Code 3.5:

```
#include "term_structure_class_flat.h"

term_structure_class_flat::term_structure_class_flat(const double& r){ R_ = r; }

double term_structure_class_flat::yield(const double& T) const { if (T>=0) return R_; return 0;};

void term_structure_class_flat::set_int_rate(const double& r) { R_ = r; };
```

Code 3.6:

3.4.3 Interpolated term structure.

The interpolated term structure implemented here uses a set of observations of yields as a basis, and for observations in between observations will interpolate between the two closest. The following only provides implementations of calculation of the yield, for the other two rely on the base class code.

There is some more book-keeping involved here, need to have code that stores observations of times and yields.

```
#ifndef _TERM_STRUCTURE_CLASS_INTERPOLATED_
#define _TERM_STRUCTURE_CLASS_INTERPOLATED_

#include "term_structure_class.h"
#include <vector>
using namespace std;

class term_structure_class_interpolated : public term_structure_class {
private:
    vector<double> times_; // use to keep a list of yields
    vector<double> yields_;
    void clear();
public:
    term_structure_class_interpolated();
    term_structure_class_interpolated(const vector<double>& times, const vector<double>& yields);
    virtual ~term_structure_class_interpolated();
    term_structure_class_interpolated(const term_structure_class_interpolated&);
    term_structure_class_interpolated operator= (const term_structure_class_interpolated&);

    int no_observations() const { return times_.size(); };
    virtual double yield(const double& T) const;
    void set_interpolated_observations(vector<double>& times, vector<double>& yields);
};

#endif
```

Code 3.7:

```
#include "term_structure_class_interpolated.h"
#include "fin_recipes.h"

void term_structure_class_interpolated::clear(){
    times_.erase(times_.begin(), times_.end());
    yields_.erase(yields_.begin(), yields_.end());
};

term_structure_class_interpolated::term_structure_class_interpolated():term_structure_class(){clear();};

term_structure_class_interpolated::term_structure_class_interpolated(const vector<double>& in_times,
                                                                     const vector<double>& in_yields) {
    clear();
    if (in_times.size()!=in_yields.size()) return;
    times_= vector<double>(in_times.size());
    yields_= vector<double>(in_yields.size());
    for (int i=0;i<in_times.size();i++) {
        times_[i]=in_times[i];
        yields_[i]=in_yields[i];
    };
};

term_structure_class_interpolated::~term_structure_class_interpolated(){ clear();};

term_structure_class_interpolated::term_structure_class_interpolated(const term_structure_class_interpolated& term) {
    times_      = vector<double> (term.no_observations());
    yields_     = vector<double> (term.no_observations());
    for (int i=0;i<term.no_observations();i++){
        times_[i]   = term.times_[i];
        yields_[i]  = term.yields_[i];
    };
};

term_structure_class_interpolated
term_structure_class_interpolated::operator= (const term_structure_class_interpolated& term) {
    times_      = vector<double> (term.no_observations());
    yields_     = vector<double> (term.no_observations());
    for (int i=0;i<term.no_observations();i++){
        times_[i]   = term.times_[i];
        yields_[i]  = term.yields_[i];
    };
    return (*this);
};

double term_structure_class_interpolated::yield(const double& T) const {
    return term_structure_yield_linearly_interpolated(T, times_, yields_);
};

void
term_structure_class_interpolated::set_interpolated_observations(vector<double>& in_times,
                                                                 vector<double>& in_yields) {
    clear();
    if (in_times.size()!=in_yields.size()) return;
    times_= vector<double>(in_times.size());
    yields_= vector<double>(in_yields.size());
    for (int i=0;i<in_times.size();i++) {
        times_[i]=in_times[i];
        yields_[i]=in_yields[i];
    };
};
```

Code 3.8:

3.5 Bond calculations using the term structure class

Codes 3.9 and 3.10 illustrates how one would calculate bond prices and duration if one has a term structure class.

```
#include <vector>
using namespace std;

#include "term_structure_class.h"

double bonds_price(const vector<double>& cashflow_times,
                   const vector<double>& cashflows,
                   const term_structure_class& d) {
    double p = 0;
    for (unsigned i=0;i<cashflow_times.size();i++) {
        p += d.discount_factor(cashflow_times[i])*cashflows[i];
    };
    return p;
};
```

Code 3.9: Pricing a bond with a term structure class

```
#include "term_structure_class.h"
#include <vector>
using namespace std;

double bonds_duration(const vector<double>& cashflow_times,
                      const vector<double>& cashflow_amounts,
                      const term_structure_class& d ) {
    double S=0;
    double D1=0;
    for (unsigned i=0;i<cashflow_times.size();i++){
        S += cashflow_amounts[i] * d.discount_factor(cashflow_times[i]);
        D1 += cashflow_times[i] * cashflow_amounts[i] * d.discount_factor(cashflow_times[i]);
    };
    return D1/S;
};
```

Code 3.10: Calculating a bonds duration with a term structure class

References Shiller [1990] is a good reference on the term structure.

Chapter 4

Futures algoritms.

Contents

In this we discuss algoritms used in valuing futures contracts.

4.1 Pricing of futures contract.

The futures price of an asset without payouts is the future value of the current price of the assset.

$$f_t = e^{r(T-t)} S_t$$

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

double futures_price(const double& S,           // current price of underlying asset
                     const double& r,           // risk free interest rate
                     const double& time_to_maturity) {
    return exp(r*time_to_maturity)*S;
};
```

Code 4.1: Futures price

The program

```
void test_futures_price(){
    double S=100; double r=0.10; double time=0.5;
    cout << " futures price = " << futures_price(S,r, time) << endl;
};
```

provides the output

```
 futures price = 105.127
```

Example 4.1: Futures price

Chapter 5

Binomial option pricing

Contents

Option and other derivative pricing is one of the prime "success stories" of modern finance. An option is a derivative security, the cash flows from the security is a function of the price of some *other* security, typically called the underlying security. A call option is a right, but not obligation, to buy a given quantity of the underlying security at a given price, called the exercise price K, within a certain time interval. A put option is the right, but not obligation, to *sell* a given quantity of the underlying security to an agreed excercise price within a given time interval. If an option can only be exercised (used) at a given date (the time interval is one day), the option is called an European Option. If the option can be used in a whole time period up to a given date, the option is called American.

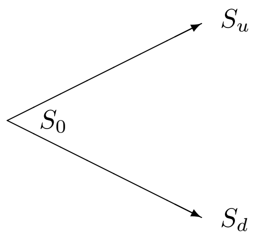
An option will only be used if it is valuable to the option holder. In the case of a call option, this is when the exercise price K is lower than the price one alternatively could buy the underlying security for, which is the current price of the underlying security. Hence, options have never negative cash flows at maturity. Thus, for anybody to be willing to offer an option, they must have a cost when entered into. This cost, or price, is typically called an option *premium.* As notation, let C signify the price of a call option, P the price of a put option and S the price of the underlying security. All of these prices are indexed by time. We typically let 0 be "now" and T the final maturity date of the option. From the definition of the options, it is clear that at their last possible exercise date, the maturity date, they have cash flows.

$$C_T = \max(0, S_T - K)$$

$$P_T = \max(0, K - S_T)$$

The challenge of option pricing is to determine the option premium C_0 and P_0.

All pricing considers that the cashflows from the derivative is a direct function of the price of the underlying security. Pricing can therefore be done *relative* to the price of the underlying security. To price options it is necessary to make assumptions about the probability distribution of movements of the underlying security. We start by considering this in a particularly simple framework, the binomial assumption. The price of the underlying is currently S_0. The price can next period only take on two values, S_u and S_d.



If one can find all possible future "states," an enumeration of all possibilities, one can value a security by constructing artificial "probabilities", called "state price probabilities," which one use to find an artificial expected value of the underlying security, which is then discounted at the risk free interest rate. The binomial framework is particularly simple, since there are only two possible states. If we find the "probability" q of one state, we also find the probability of the other as $(1-q)$. Equation 5.1 demonstrates this calculation for the underlying security.

$$S_0 = e^{-r}(qS_u + (1-q)S_d) \tag{5.1}$$

Now, any derivative security based on this underlying security can be priced using the same "probability" q. The contribution of binomial option pricing is in actually calculating the number q. To do valuation, start by introducing constants u and d implicitly defined by $S_u = uS_0$ and $S_d = dS_0$, and you get a process as illustrated in figure 5.1.

$$\begin{tikzcd}[column sep=large]
 & uS_0 \\
S_0 \arrow[ur] \arrow[dr] & \\
 & dS_0
\end{tikzcd}$$

Figure 5.1: Binomial Tree

and calculate the artifical "probability" q as

$$q = \frac{e^r - d}{u - d}$$

The price of a one-period call option in a binomial framework is shown in formula 5.1 and implemented in code 5.1.

$$C_u = \max(0, S_u - K)$$

$$C_d = \max(0, S_d - K)$$

$$C_0 = e^{-r}\left(qC_u + (1-q)C_d\right)$$

$$q = \frac{e^r - d}{u - d}$$

$S_u = uS_0$ and $S_d = dS_0$ are the possible values for the underlying security next period, u and d are constants, r is the (continously compounded) risk free interest rate and K is the call option exercise price.

Formula 5.1: The single period binomal call option price

The "state price probability" q is found by an assumption of no arbitrage opportunities. If one has the possibility of trading in the underlying security and a risk free bond, it is possible to create a portfolio of these two assets that exactly duplicates the future payoffs of the derivative security. Since this portfolio has the same future payoff as the derivative, the price of the derivative has to equal the cost of the duplicating portfolio. Working out the algebra of this, one can find the expression for q as the function of the up and down movements u and d.

Exercise 1.

The price of the underlying security follows the binomial process

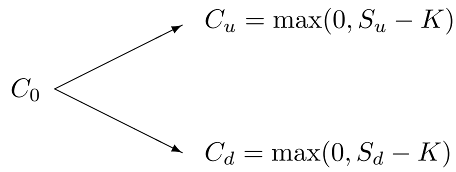
$$\begin{tikzcd}[column sep=large]
 & S_u \\
S_0 \arrow[ur] \arrow[dr] & \\
 & S_d
\end{tikzcd}$$

```
#include <cmath>         // standard mathematical library
#include <algorithm>        // defining the max() operator
using namespace std;

double option_price_call_european_binomial( const double& S, // spot price
                                            const double& X, // exercice price
                                            const double& r, // interest rate (per period)
                                            const double& u, // up movement
                                            const double& d){ // down movement
   double p_up = (exp(r)-d)/(u-d);
   double p_down = 1.0-p_up;
   double c_u = max(0.0,(u*S-X));
   double c_d = max(0.0,(d*S-X));
   double call_price = exp(-r)*(p_up*c_u+p_down*c_d);
   return call_price;
};
```

Code 5.1: Binomial European, one period

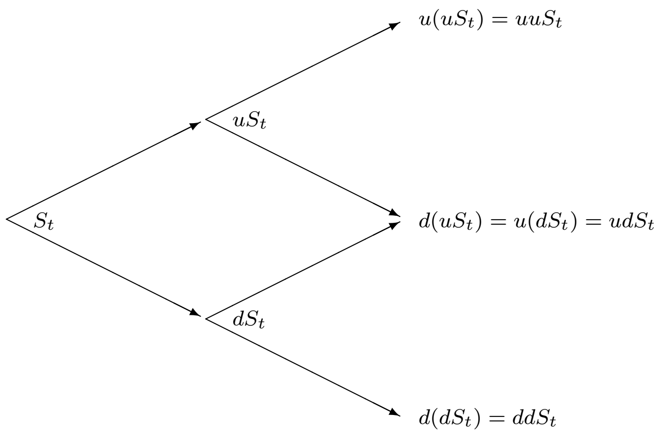
A one period call option has payoffs



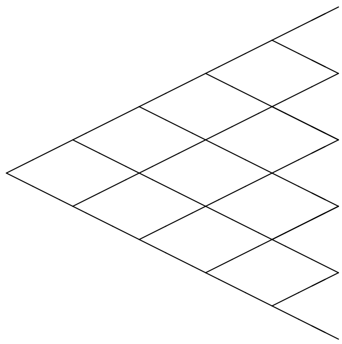
1. Show how one can combine a position in the underlying security with a position in risk free bonds to create a portfolio which exactly duplicates the payoffs from the call.
2. Use this result to show the one period pricing formula for a call option shown in formula 5.1.

5.1 Multiperiod binomial pricing

Of course, an assumption of only two possible future states next period is somewhat unrealistic, but if we iterate this assumption, and assume that every date, there are only two possible outcomes *next* date, but then, for each of these two outcomes, there is two new outcomes, as illustrated in the next figure:

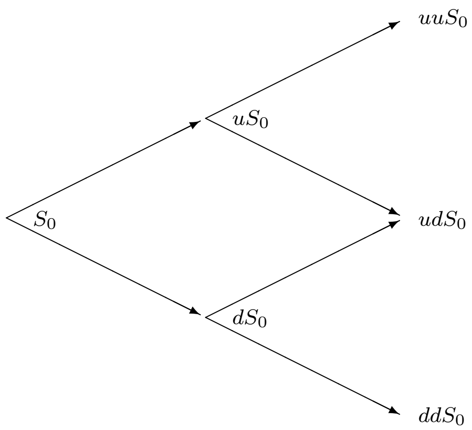


Iterating this idea a few times more, the number of different *terminal* states increases markedly, and we get closer to a realistic distribution of future prices of the underlying at the terminal date. Note that a crucial assumption to get a picture like this is that the factors u and d are the same on each date.

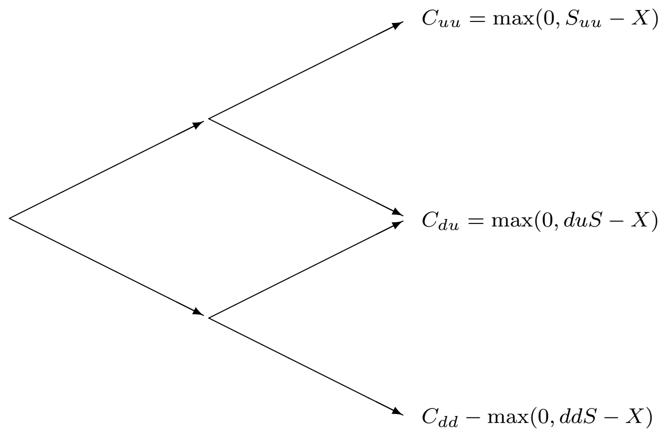


Pricing in a setting like this is done by working backwards, starting at the terminal date. Here we know all the possible values of the underlying security. For each of these, we calculate the payoffs from the derivative, and find what the set of possible derivative prices is *one period before*. Given these, we can find the option one period before this again, and so on. Working ones way down to the root of the tree, the option price is found as the derivative price in the first node.

For example, suppose we have two periods, and price a two period call option with exercise price K.



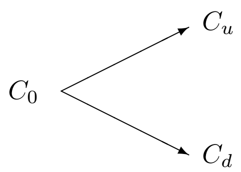
First step: Find terminal payoffs of derivative security:



Next step: Find the two possible call prices at time 1:

$$C_u = e^{-r}(qC_{uu} + (1-q)C_{ud})$$

$$C_d = e^{-r}(qC_{ud} + (1-q)C_{dd})$$



Final step: Using the two possible payoffs at time 1, C_u and C_d, find option value at time 0:

$$C_0 = e^{-r}(qC_u + (1-q)C_d)$$

Thus, binomial pricing really concerns "rolling backward" in a binomial tree, and programming therefore concerns an efficient way of traversing such a tree. The obvious data structure for describing such a tree is shown in code 5.2, where the value in each node is calculated from finding out the number of up and down steps are used to get to the particular node.

Exercise 2.

In terms of computational efficiency the approcach of code 5.2 will not be optimal, since it requires a lot of calls to the `pow()` functional call. More efficient would be to carry out the tree building by doing the multiplication from the previous node, for example the j'th vector is the $j-1$'th vector times u, and then one need to add one more node by multiplying the lowest element by d.

1. Implement such an alternative tree building procedure.

Basing the recursive calculation of a derivative price on a triangular array structure as shown in code 5.2 is the most natural approach, but with some cleverness based on understanding the structure of the binomial tree, we can get away with the more efficienent algorithm that is shown in code 5.3. Note that here we only use one `vector<double>`, not a triangular array as built above.

Exercise 3.

Implement pricing of single and multi period binomial put options.

Further reading The derivation of the single period binomial is e.g. shown in Bossaerts and Ødegaard [2001]. Hull [2003] and McDonald [2002] are standard references.

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

vector< vector<double> > binomial_tree(const double& S0,
                                        const double& u,
                                        const double& d,
                                        const int& no_steps){
   vector< vector<double> > tree;
   for (int i=1;i<=no_steps;++i){
      vector<double> S(i);
      for (int j=0;j<i;++j){
         S[j] = S0*pow(u,j)*pow(d,i-j-1);
      };
      tree.push_back(S);
   };
   return tree;
};
```

Code 5.2: Building a binomial tree

```
#include <cmath>      // standard mathematical library
#include <algorithm>  // defining the max() operator
#include <vector>     // STL vector templates
using namespace std;

double option_price_call_european_binomial(const double& S, // spot price
                                           const double& K, // exercice price
                                           const double& r, // interest rate (per period)
                                           const double& u, // up movement
                                           const double& d, // down movement
                                           const int& no_periods){ // no steps in binomial tree
   double Rinv = exp(-r);                    // inverse of interest rate
   double uu = u*u;
   double p_up = (exp(r)-d)/(u-d);
   double p_down = 1.0-p_up;
   vector<double> prices(no_periods+1); // price of underlying
   prices[0] = S*pow(d, no_periods); // fill in the endnodes.
   for (int i=1; i<=no_periods; ++i) prices[i] = uu*prices[i-1];
   vector<double> call_values(no_periods+1); // value of corresponding call
   for (int i=0; i<=no_periods; ++i) call_values[i] = max(0.0, (prices[i]-K)); // call payoffs at maturity
   for (int step=no_periods-1; step>=0; --step) {
      for (int i=0; i<=step; ++i) {
         call_values[i] = (p_up*call_values[i+1]+p_down*call_values[i])*Rinv;
      };
   };
   return call_values[0];
};
```

Code 5.3: Binomial multiperiod pricing of European call option

Let $S = 100.0$, $K = 100.0$, $r = 0.025$, $u = 1.05$ and $d = 1/u$.

1. Price one and two period European Call options.

The program

```
void test_bin_eur_call_ud (){
    double S = 100.0; double K = 100.0; double r = 0.025;
    double u = 1.05; double d = 1/u;
    cout << " one period european call = "
         << option_price_call_european_binomial(S,K,r,u,d) << endl;
    int no_periods = 2;
    cout << " two period european call = "
         << option_price_call_european_binomial(S,K,r,u,d,no_periods) << endl;

};
```

provides the output

```
 one period european call =  3.64342
 two period european call =  5.44255
```

Chapter 6

Basic Option Pricing, the Black Scholes formula

Contents

The pricing of options and related instruments has been a major breakthrough for the use of financial theory in practical application. Since the original papers of Black and Scholes [1973] and Merton [1973], there has been a wealth of practical and theoretical applications. We will now consider the orginal Black Scholes formula for pricing options, how it is calculated and used. For the basic intuition about option pricing the reader should first read the discussion of the binomial model in the previous chapter, as that is a much better environment for understanding what is actually calculated.

An option is a *derivative security*, its value depends on the value, or price, of some other underlying security, called the *underlying* security.. Let S denote the value, or price, of this underlying security. We need to keep track of what time this price is observed at, so let S_t denote that the price is observed at time t. A call (put) option gives the holder the right, but not the obligation, to buy (sell) some underlying asset at a given price K, called the exercise price, on or before some given date T. If the option is a so called *European* option, it can only be used (exercised) at the maturity date. If the option is of the so called *American* type, it can be used (exercised) at any date up to and including the maturity date T. If exercised at time T, a call option provides payoff

$$C_T = \max(0, S_T - K)$$

and a put option provides payoff

$$P_T = \max(0, K - S_T)$$

The Black Scholes formulas provides analytical solutions for *European* put and call options, options which can only be exercised at the options maturity date. Black and Scholes showed that the additional information needed to price the option is the (continously compounded) risk free interest rate r, the variability of the underlying asset, measured by the standard deviation σ of (log) price changes, and the time to maturity $(T - t)$ of the option, measured in years. The original formula was derived under the assumption that there are no payouts, such as stock dividends, coming from the underlying security during the life of the option. Such payouts will affection option values, as will become apparent later.

6.1 The formula

Formula 6.1 gives the exact formula for a call option, and the calculation of the same call option is shown in code 6.1

$$c = SN(d_1) - Ke^{-r(T-t)}N(d_2)$$

where

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

and

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

Alternatively one can calculate d_1 and d_2 as

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + r(T-t)}{\sigma\sqrt{T-t}} + \frac{1}{2}\sigma\sqrt{T-t}$$

$$d_2 = \frac{\ln\left(\frac{S}{K}\right) + r(T-t)}{\sigma\sqrt{T-t}} - \frac{1}{2}\sigma\sqrt{T-t}$$

S is the price of the underlying security, K the exercise price, r the (continously compounded) risk free interest rate, σ the standard deviation of the underlying asset, t the current date, T the maturity date, $T-t$ the time to maturity for the option and $N(\cdot)$ the cumulative normal distribution.

Formula 6.1: The Black Scholes formula

```
#include <cmath>          // mathematical C library
#include "normdist.h"     // the calculation of the cumularive normal distribution

double option_price_call_black_scholes(const double& S, // spot (underlying) price
                                       const double& K,  // strike (exercise) price,
                                       const double& r,   // interest rate
                                       const double& sigma, // volatility
                                       const double& time) { // time to maturity
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    double c = S*N(d1) - K*exp(-r*time)*N(d2);
    return c;
};
```

Code 6.1: Price of European call option using the Black Scholes formula

Let us price a call option. The option matures 6 months from now, at which time the holder of the option can recive one unit of the underlying security by paying the exercise price of $K = 50$. The current price of the underlying security is $S = 50$. The volatility of the underlying security is given as $\sigma = 30\%$. The current risk free interest rate (with continous compounding) for six month borrowing is 10%.
To calculate this we use the Black Scholes formula with inputs $S = 50$, $K = 50$, $r = 0.10$, $\sigma = 0.3$ and $(T - t) = 0.5$.
The program

```
void test_option_price_call_black_scholes(){
    double S = 50; double K = 50; double r = 0.10;
    double sigma = 0.30; double time=0.50;
    cout << " Black Scholes call price = ";
    cout << option_price_call_black_scholes(S, K , r, sigma, time) << endl;
};
```

provides the output

```
 Black Scholes call price = 5.45325
```

Example 6.1: Example using the Black Scholes formula

Exercise 4.

The Black Scholes price for a put option is:

$$p = Ke^{-r(T-t)}N(-d_2) - SN(-d_1)$$

where d_1 and d_2 are as for the call option:

$$d_1 = \frac{\ln\left(\frac{S}{X}\right) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

and S is the price of the underlying secrutity, K the exercise price, r the (continously compounded) risk free interest rate, σ the standard deviation of the underlying asset, $T-t$ the time to maturity for the option and $N(\cdot)$ the cumulative normal distribution.

1. Implement this formula.

6.2 Understanding the why's of the formula

To get some understanding of the Black Scholes formula and why it works will need to delve in some detail into the mathematics underlying its derivation. It does not help that there are a number of ways to prove the Black Scholes formula, depending on the setup. As it turns out, two of these ways are important to understand for computational purposes, the original Black Scholes continous time way, and the "limit of a binomial process" way of Cox et al. [1979].

6.2.1 The original Black Scholes analysis

The primary assumption underlying the Black Scholes analyis concerns the stochastic process governing the price of the underlying asset. The price of the underlying asset, S, is assumed to follow a geometric Brownian Motion process, conveniently written in either of the shorthand forms

$$dS = \mu S dt + \sigma S dZ$$

or

$$\frac{dS}{S} = \mu dt + \sigma dZ$$

where μ and σ are constants, and Z is Brownian motion.

Using Ito's lemma, the assumption of no arbitrage, and the ability to trade continuously, Black and Scholes showed that the price of *any* contingent claim written on the underlying must solve the *partial differential equation* (6.1).

$$\frac{\partial f}{\partial S} rS + \frac{\partial f}{\partial t} + \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2 = rf \tag{6.1}$$

For any *particular* contingent claim, the terms of the claim will give a number of *boundary conditions* that determines the form of the pricing formula.

The pde given in equation (6.1), with the boundary condition $c_T = \max(0, S_T - K)$ was shown by Black and Scholes to have an analytical solution of functional form shown in the Black Scoles formula 6.1.

6.2.2 The limit of a binomial case

Another is to use the limit of a binomial process [Cox et al., 1979]. The latter is particularly interesting, as it allows us to link the Black Scholes formula to the binomial, allowing the binomial framework to be used as an approximation.

6.2.3 The representative agent framework

A final way to show the BS formula to assume a representative agent and lognormality as was done in Rubinstein [1976].

6.3 Partial derivatives.

In trading of options, a number of partial derivatives of the option price formula is important.

6.3.1 Delta

The first derivative of the option price with respect to the price of the underlying security is called the *delta* of the option price. It is the derivative most people will run into, since it is important in hedging of options.

$$\frac{\partial c}{\partial S} = N(d_1)$$

Code 6.2 shows the calculation of the delta for a call option.

```
#include <cmath>
#include "normdist.h"

double option_price_delta_call_black_scholes(const double& S, // spot price
                                             const double& K, // Strike (exercise) price,
                                             const double& r, // interest rate
                                             const double& sigma, // volatility
                                             const double& time){ // time to maturity
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt) + 0.5*sigma*time_sqrt;
    double delta = N(d1);
    return delta;
};
```

Code 6.2: Calculating the delta of the Black Scholes call option price

6.3.2 Other Derivatives

The remaining derivatives are more seldom used, but all of them are relevant. All of them are listed in formula 6.3.2.

The calculation of all of these partial derivatives for a call option is shown in code 6.3

Delta (Δ)

$$\Delta = \frac{\partial c}{\partial S} = N(d_1)$$

Gamma (Γ)

$$\frac{\partial^2 c}{\partial S^2} = \frac{n(d_1)}{S\sigma\sqrt{T-t}}$$

Theta (Θ) (careful about which of these you want)

$$\frac{\partial c}{\partial (T-t)} = Sn(d_1)\frac{1}{2}\sigma\frac{1}{\sqrt{T-t}} + rKe^{-r(T-t)}N(d_2)$$

$$\frac{\partial c}{\partial t} = -Sn(d_1)\frac{1}{2}\sigma\frac{1}{\sqrt{T-t}} - rKe^{-r(T-t)}N(d_2)$$

Vega

$$\frac{\partial c}{\partial \sigma} = S\sqrt{T-t}n(d_1)$$

Rho (ρ)

$$\frac{\partial c}{\partial r} = K(T-t)e^{-r(T-t)}N(d_2)$$

S is the price of the underlying security, K the exercise price, r the (continously compounded) risk free interest rate, σ the standard deviation of the underlying asset, t the current date, T the maturity date and $T-t$ the time to maturity for the option. $n(\cdot)$ is the normal distribution function $\left(n(z) = \frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}z^2}\right)$ and $N(\cdot)$ the cumulative normal distribution $\left(N(z) = \int_{-\infty}^{z} n(t)dt\right)$.

Formula 6.2: Partial derivatives of the Black Scholes call option formula

```
#include <cmath>
#include "normdist.h"
using namespace std;

void option_price_partials_call_black_scholes( const double& S, // spot price
                                               const double& K, // Strike (exercise) price,
                                               const double& r, // interest rate
                                               const double& sigma, // volatility
                                               const double& time, // time to maturity
                                               double& Delta, // partial wrt S
                                               double& Gamma, // second prt wrt S
                                               double& Theta, // partial wrt time
                                               double& Vega, // partial wrt sigma
                                               double& Rho){ // partial wrt r
  double time_sqrt = sqrt(time);
  double d1 = (log(S/K)+r*time)/(sigma*time_sqrt) + 0.5*sigma*time_sqrt;
  double d2 = d1-(sigma*time_sqrt);
  Delta = N(d1);
  Gamma = n(d1)/(S*sigma*time_sqrt);
  Theta =- (S*sigma*n(d1))/(2*time_sqrt) - r*K*exp( -r*time)*N(d2);
  Vega = S * time_sqrt*n(d1);
  Rho  = K*time*exp(-r*time)*N(d2);
};
```

Code 6.3: Calculating the partial derivatives of a Black Scholes call option

Consider the same call option as in the previous example. The option matures 6 months from now, at which time the holder of the option can recive one unit of the underlying security by paying the exercise price of $K = 50$. The current price of the underlying security is $S = 50$. The volatility of the underlying security is given as $\sigma = 30\%$. The current risk free interest rate (with continous compounding) for six month borrowing is 10%. To calculate the partial derivatives we therefore use inputs $S = 50$, $K = 50$, $r = 0.10$, $\sigma = 0.3$ and $(T - t) = 0.5$.
The program

```
void test_black_scholes_partials_call(){
    cout << " Option price partial derivatives, call option using Black Scholes " << endl;
    double S = 50; double K = 50; double r = 0.10;
    double sigma = 0.30; double time=0.50;
    double Delta, Gamma, Theta, Vega, Rho;
    option_price_partials_call_black_scholes(S,K,r,sigma, time,
                                             Delta, Gamma, Theta, Vega, Rho);
    cout << " Delta = " << Delta << endl;
    cout << " Gamma = " << Gamma << endl;
    cout << " Theta = " << Theta << endl;
    cout << " Vega = " << Vega << endl;
    cout << " Rho = " << Rho << endl;
};
```

provides the output

```
 Option price partial derivatives, call option using Black Scholes
  Delta = 0.633737
  Gamma = 0.0354789
  Theta = -6.61473
  Vega  = 13.3046
  Rho   = 13.1168
```

Example 6.2: Example caclulating partial derivatives using the Black Scholes formula

6.3.3 Implied Volatility.

In calculation of the option pricing formulas, in particular the Black Scholes formula, the only unknown is the standard deviation of the underlying stock. A common problem in option pricing is to find the implied volatility, given the observed price quoted in the market. For example, given c_0, the price of a call option, the following equation should be solved for the value of σ

$$c_0 = c(S, K, r, \sigma, T - t)$$

Unfortunately, this equation has no closed form solution, which means the equation must be numerically solved to find σ. What is probably the algorithmic simplest way to solve this is to use a binomial search algorithm, which is implemented in the following. We start by bracketing the sigma by finding a high sigma that makes the BS price higher than the observed price, and then, given the bracketing interval, we search for the volatility in a systematic way. Code 6.4 shows such a calculation.

```
#include <cmath>
#include "fin_recipes.h"

double option_price_implied_volatility_call_black_scholes_bisections(const double& S,
                                                                      const double& X,
                                                                      const double& r,
                                                                      const double& time,
                                                                      const double& option_price){
    double sigma_low=0.0001; // check for arbitrage violations:
    double price = option_price_call_black_scholes(S,X,r,sigma_low,time);
    if (price>option_price) return 0.0; // if price at almost zero volatility greater than price, return 0

    // simple binomial search for the implied volatility.
    // relies on the value of the option increasing in volatility
    const double ACCURACY = 1.0e-5; // make this smaller for higher accuracy
    const int MAX_ITERATIONS = 100;
    const double HIGH_VALUE = 1e10;
    const double ERROR = -1e40;

    // want to bracket sigma. first find a maximum sigma by finding a sigma
    // with a estimated price higher than the actual price.
    double sigma_high=0.3;
    price = option_price_call_black_scholes(S,X,r,sigma_high,time);
    while (price < option_price) {
        sigma_high = 2.0 * sigma_high; // keep doubling.
        price = option_price_call_black_scholes(S,X,r,sigma_high,time);
        if (sigma_high>HIGH_VALUE) return ERROR; // panic, something wrong.
    };
    for (int i=0;i<MAX_ITERATIONS;i++){
        double sigma = (sigma_low+sigma_high)*0.5;
        price = option_price_call_black_scholes(S,X,r,sigma,time);
        double test = (price-option_price);
        if (fabs(test)<ACCURACY) { return sigma; };
        if (test < 0.0) { sigma_low = sigma; }
        else { sigma_high = sigma; }
    };
    return ERROR;
};
```

Code 6.4: Calculation of implied volatility of Black Scholes using bisections

Instead of this simple bracketing, which is actually pretty fast, and will (almost) always find the solution, we can use the Newton–Raphson formula for finding the root of an equation in a single variable. The general description of this method starts with a function $f()$ for which we want to find a root.

$$f(x) = 0.$$

The function $f()$ needs to be differentiable. Given a first guess x_0, iterate by

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

until

$$|f(x_i)| < \epsilon$$

where ϵ is the desired accuracy.[1]
In our case

$$f(x) = c_{obs} - c_{BS}(\sigma)$$

and, each new iteration will calculate

$$\sigma_{i+1} = \sigma_i + \frac{c_{obs} - c_{BS}(\sigma_i)}{-\frac{\partial c_{BS}()}{\partial \sigma}}$$

Code 6.5 shows the calculation of implied volatility using Newton-Raphson.

```
#include "fin_recipes.h"
#include "normdist.h"
#include <cmath>

double option_price_implied_volatility_call_black_scholes_newton(const double& S,
                                                                 const double& X,
                                                                 const double& r,
                                                                 const double& time,
                                                                 const double& option_price) {
  double sigma_low = 1e-5;                 // check for arbitrage violations:
  double price = option_price_call_black_scholes(S,X,r,sigma_low,time);
  if (price > option_price) return 0.0; // if price at almost zero volatility greater than price, return 0

  const int MAX_ITERATIONS = 100;
  const double ACCURACY = 1.0e-4;
  double t_sqrt = sqrt(time);

  double sigma = (option_price/S)/(0.398*t_sqrt); // find initial value
  for (int i=0;i<MAX_ITERATIONS;i++){
    price = option_price_call_black_scholes(S,X,r,sigma,time);
    double diff = option_price -price;
    if (fabs(diff)<ACCURACY) return sigma;
    double d1 = (log(S/X)+r*time)/(sigma*t_sqrt) + 0.5*sigma*t_sqrt;
    double vega = S * t_sqrt * n(d1);
    sigma = sigma + diff/vega;
  };
  return -99e10; // something screwy happened, should throw exception
};
```

Code 6.5: Calculation of implied volatility of Black Scholes using Newton-Raphson

Note that to use Newton-Raphson we need the derivative of the option price. For the Black-Scholes formula this is known, and we can use this. But for pricing formulas like the binomial, where the partial derivatives are not that easy to calculate, simple bisection is the preferred algorithm.

[1]For further discussion of the Newton-Raphson formula and bracketing, a good source is chapter 9 of Press et al. [1992]

Consider the same call option as in the previous examples. The option matures 6 months from now, at which time the holder of the option can recive one unit of the underlying security by paying the exercise price of $K = 50$. The current price of the underlying security is $S = 50$. The current risk free interest rate (with continous compounding) for six month borrowing is 10%. To calculate we therefore use inputs $S = 50$, $K = 50$, $r = 0.10$ and $(T - t) = 0.5$.
We are now told that the current option price is $C = 2.5$. The implied volatility is the σ which, input in the Black Scholes formula with these other inputs, will produce an option price of $C = 2.5$.
The program

```
void test_black_scholes_implied_volatility(){
    double S = 50; double K = 50; double r = 0.10; double time=0.50;
    double C=2.5;
    cout << " Black Scholes implied volatility using Newton search = ";
    cout << option_price_implied_volatility_call_black_scholes_newton(S,K,r,time,C) << endl;
    cout << " Black Scholes implied volatility using bisections = ";
    cout << option_price_implied_volatility_call_black_scholes_bisections(S,K,r,time,C) << endl;
};
```

provides the output

```
 Black Scholes implied volatility using Newton search = 0.0500427
 Black Scholes implied volatility using bisections = 0.0500414
```

Example 6.3: Example finding implied volatility using the Black Scholes formula

Chapter 7

Warrants

Contents

A warrant is an option-like security on equity, but it is issued by the same company which has issued the equity, and when a warrant is exercised, a *new* stock is issued. This new stock is issued at a the warrant strike price, which is lower than the current stock price (If it wasn't the warrant would not be exercised.) Since the new stock is a a fractional right to all cashflows, this stock issue *waters out*, or *dilutes*, the equity in a company. The degree of dilution is a function of how many warrants are issued.

7.1 Warrant value in terms of assets

Let K be the strike price, n the number of shares outstanding and m the number of warrants issues. Assume each warrant is for 1 new share, and let A_t be the current asset value of firm. Suppose all warrants are exercised simultaneously. Then the assets of the firm increase by the number of warrants times the strike price of the warrant.

$$A_t + mK,$$

but this new asset value is spread over more shares, since each exercised warrant is now an equity. The assets of the firm is spread over all shares, hence each new share is worth:

$$\frac{A_t + mK}{m + n}$$

making each exercised warrant worth:

$$\frac{A_t + mK}{m + n} - K = \frac{n}{m + n}\left(\frac{A_t}{n} - K\right)$$

If we knew the current value of assets in the company, we could value the warrant in two steps:

1. Value the option using the Black Scholes formula and $\frac{A_t}{n}$ as the current stock price.
2. Multiply the resulting call price with $\frac{n}{m+n}$.

If we let W_t be the warrant value, the above arguments are summarized as:

$$W_t = \frac{n}{n + m} C_{BS}\left(\frac{A}{n}, K, \sigma, r, (T - t)\right),$$

where $C_{BS}(\cdot)$ is the Black Scholes formula.

7.2 Valuing warrants when observing the stock value

However, one does not necessarily observe the asset value of the firm. Typically one only observes the equity value of the firm. If we let S_t be the current stock price, the asset value is really:

$$A_t = nS_t + mW_t$$

Using the stock price, one would value the warrant as

$$W_t = \frac{n}{n+m} C_{BS}\left(\frac{nS_t + mW_t}{n}, K, \sigma, r, (T-t)\right)$$

or

$$W_t = \frac{n}{n+m} C_{BS}\left(S_t + \frac{m}{n}W_t, K, \sigma, r, (T-t)\right)$$

Note that this gives the value of W_t as a function of W_t. One need to solve this equation numerically to find W_t.

The numerical solution for W_t is done using the Newton-Rhapson method. Let

$$g(W_t) = W_t - \frac{n}{n+m} C_{BS}\left(S_t + \frac{m}{n}W_t, K, \sigma, r, (T-t)\right)$$

Starting with an initial guess for the warrant value W_t^o, the Newton-Rhapson method is that one iterates as follows

$$W_t^i = W_t^{i-1} - \frac{g(W_t^{i-1})}{g'(W_t^{i-1})},$$

where i signifies iteration i, until the criterion function $g(W_t^{i-1})$ is below some given accuracy ϵ. In this case

$$g'(W_t) = 1 - \frac{m}{m+n} N(d_1)$$

where

$$d_1 = \frac{\ln\left(\frac{S_t + \frac{m}{n}W_t}{K}\right) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

An obvious starting value is to set calculate the Black Scholes value using the current stock price, and multiplying it with $\frac{m}{m+n}$.

Code 7.1 implements this calculation.

7.3 Readings

McDonald [2002] and Hull [2003] are general references. A problem with warrants is that exercise of all warrants simultaneously is not necessarily optimal.

Press et al. [1992] discusses the Newton-Rhapson method for root finding.

```
#include "fin_recipes.h"
#include "normdist.h"
#include <cmath>

double warrant_price_adjusted_black_scholes(const double& S, // current stock price
                                            const double& K, // strike price
                                            const double& r, // interest rate
                                            const double& sigma, // volatility
                                            const double& time, // time to maturity
                                            const double& m, // number of warrants outstanding
                                            const double& n){ // number of shares outstanding
    const double epsilon=0.00001;
    double time_sqrt = sqrt(time);
     double w = (n/(n+m))*option_price_call_black_scholes(S,K,r,sigma,time);
    double g = w-(n/(n+m))*option_price_call_black_scholes(S+(m/n)*w,K,r,sigma,time);
    while (fabs(g)>epsilon) {
        double d1 = (log((S+(m/n))/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
        double gprime = 1-(m/n)*N(d1);
        w=w-g/gprime;
        g = w-(n/(n+m))*option_price_call_black_scholes(S+(m/n)*w,K,r,sigma,time);
    };
    return w;
};
```

Code 7.1: Adjusted Black Scholes value for a Warrant

A stock is currently priced at $S = 48$. Consider warrants on the same company with exercise price $K = 40$ and time to maturity of six months. The company has $n = 10000$ shares outstanding, and has issued $m = 1000$ warrants. The current (continously compounded) risk free interest rate is 8%. Determine the current warrant price.

The program

```
void test_warrant_price_adjusted_black_scholes(){
    double S = 48; double K = 40; double r = 0.08; double sigma = 0.30;
    double time = 0.5; double m = 1000; double n = 10000;
    double w = warrant_price_adjusted_black_scholes(S,K,r,sigma, time, m, n);
    cout << " warrant price = " << w << endl;
};
```

provides the output

```
 warrant price = 10.142
```

Example 7.1: Example warrant pricing

Chapter 8

Extending the Black Scholes formula

Contents

8.1 Adjusting for payouts of the underlying.

For options on other financial instruments than stocks, we have to allow for the fact that the underlying may have payouts during the life of the option. For example, in working with commodity options, there is often some storage costs if one wanted to hedge the option by buying the underlying.

8.1.1 Continous Payouts from underlying.

The simplest case is when the payouts are done continuously. To value an European option, a simple adjustment to the Black Scholes formula is all that is needed. Let q be the *continuous payout* of the underlying commodity.

Call and put prices for European options are then given by formula 8.1, which are implemented in code 8.1.

$$c = Se^{-q(T-t)}N(d_1) - Ke^{-r(T-t)}N(d_2)$$

where

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + (r - q + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

S is the price of the underlying secrutity, K the exercise price, r the risk free interest rate, q the (continous) payout and σ the standard deviation of the underlying asset, t the current date, T the maturity date, $T-t$ the time to maturity for the option and $N(\cdot)$ the cumulative normal distribution.

Formula 8.1: Analytical prices for European call option on underlying security having a payout of q

Exercise 5.

The price of a put on an underlying security with a continous payout of q is:

$$p = Ke^{-r(T-t)}N(-d_2) - Se^{-q(T-t)}N(-d_1)$$

1. Implement this formula.

```
#include <cmath>        // mathematical library
#include "normdist.h"     // this defines the normal distribution
using namespace std;

double option_price_european_call_payout( const double& S, // spot price
                                          const double& X, // Strike (exercise) price,
                                          const double& r, // interest rate
                                          const double& q, // yield on underlying
                                          const double& sigma, // volatility
                                          const double& time) { // time to maturity
    double sigma_sqr = pow(sigma,2);
    double time_sqrt = sqrt(time);
    double d1 = (log(S/X) + (r-q + 0.5*sigma_sqr)*time)/(sigma*time_sqrt);
    double d2 = d1-(sigma*time_sqrt);
    double call_price = S * exp(-q*time)* N(d1) - X * exp(-r*time) * N(d2);
    return call_price;
};
```

Code 8.1: Option price, continous payout from underlying

8.1.2 Dividends.

A special case of payouts from the underlying security is stock options when the stock pays dividends. When the stock pays dividends, the pricing formula is adjusted, because the dividend changes the value of the underlying.

The case of continuous dividends is easiest to deal with. It corresponds to the continuous payouts we have looked at previously. The problem is the fact that most dividends are paid at discrete dates.

European Options on dividend-paying stock.

To adjust the price of an European option for known dividends, we merely subtract the present value of the dividends from the current price of the underlying asset in calculating the Black Scholes value.

```
#include <cmath>        // mathematical library
#include <vector>
#include "fin_recipes.h"    // define the black scholes price

double option_price_european_call_dividends( const double& S,
                                             const double& K,
                                             const double& r,
                                             const double& sigma,
                                             const double& time_to_maturity,
                                             const vector<double>& dividend_times,
                                             const vector<double>& dividend_amounts ) {
    double adjusted_S = S;
    for (int i=0;i<dividend_times.size();i++) {
        if (dividend_times[i]<=time_to_maturity){
            adjusted_S -= dividend_amounts[i] * exp(-r*dividend_times[i]);
        };
    };
    return option_price_call_black_scholes(adjusted_S,K,r,sigma,time_to_maturity);
};
```

Code 8.2: European option price, dividend paying stock

The program

```
void test_black_scholes_with_dividends(){
    double S = 100.0; double K = 100.0;
    double r = 0.1;   double sigma = 0.25;
    double time=1.0;
    double dividend_yield=0.05;
    vector<double> dividend_times; vector<double> dividend_amounts;
    dividend_times.push_back(0.25); dividend_amounts.push_back(2.5);
    dividend_times.push_back(0.75); dividend_amounts.push_back(2.5);
    cout << " european stock call option with contininous dividend = "
         << option_price_european_call_payout(S,K,r,dividend_yield,sigma,time) << endl;
    cout << " european stock call option with discrete dividend = "
         << option_price_european_call_dividends(S,K,r,sigma,time,dividend_times,dividend_amounts) << endl;
};
```

provides the output

```
 european stock call option with contininous dividend = 11.7344
 european stock call option with discrete dividend =  11.8094
```

8.2 American options.

American options are much harder to deal with than European ones. The problem is that it may be optimal to use (exercise) the option before the final expiry date. This optimal exercise policy will affect the value of the option, and the exercise policy needs to be known when solving the pde. There is therefore no general analytical solutions for American call and put options. There is some special cases. For American call options on assets that do not have any payouts, the American call price is the same as the European one, since the optimal exercise policy is to not exercise. For American Put is this not the case, it may pay to exercise them early. When the underlying asset has payouts, it may also pay to exercise the option early. There is one known known analytical price for American call options, which is the case of a call on a stock that pays a known dividend *once* during the life of the option, which is discussed next. In all other cases the American price has to be approximated using one of the techniques discussed in later chapters: Binomial approximation, numerical solution of the partial differential equation, or another numerical approximation.

8.2.1 Exact american call formula when stock is paying one dividend.

When a stock pays dividend, a call option on the stock may be optimally exercised just before the stock goes ex-dividend. While the general dividend problem is usually approximated somehow, for the special case of one dividend payment during the life of an option an analytical solution is available, due to Roll–Geske–Whaley.

If we let S be the stock price, K the exercise price, D_1 the amount of dividend paid, t_1 the time of dividend payment, T the maturity date of option, we denote the time to dividend payment $\tau_1 = T - t_1$ and the time to maturity $\tau = T - t$.

A first check of early exercise is:

$$D_1 \leq K\left(1 - e^{-r(T-t_1)}\right)$$

If this inequality is fulfilled, early exercise is not optimal, and the value of the option is

$$c(S - e^{-r(t_1-t)}D_1, K, r, \sigma, (T-t))$$

where $c(\cdot)$ is the regular Black Scholes formula.

If the inequality is not fulfilled, one performs the calculation shown in formula 8.2 and implemented in code 8.3

$$C = (S - D_1 e^{-r(t_1-t)}) \left(N(b_1) + N(a_1, -b_1, \rho)\right) + K e^{-r(T-t))} N(a_2, -b_2, \rho) - (K - D_1) e^{-r(t_1-t)} N(b_2)$$

where

$$\rho = -\sqrt{\frac{(t_1 - t}{T - t}}$$

$$a_1 = \frac{\ln\left(\frac{S - D_1 e^{-q(\tau_1}}{K}\right) + (r + \frac{1}{2}\sigma^2)\tau}{\sigma\sqrt{\tau}}$$

$$a_2 = a_1 - \sigma\sqrt{T - t}$$

$$b_1 = \frac{\ln\left(\frac{S - D_1 e^{-r(t_1-t)}}{\bar{S}}\right) + (r + \frac{1}{2}\sigma^2)(t_1 - t)}{\sigma\sqrt{(t_1 - t)}}$$

$$b_2 = b_1 - \sigma\sqrt{T - t}$$

and $\bar{S}$ solves

$$c(\bar{S}, t_1) = \bar{S} + D_1 - K$$

S is the price of the underlying secrutity, K the exercise price, r the risk free interest rate, D_1 is the dividend amount and σ the standard deviation of the underlying asset, t the current date, T the maturity date, $T - t$ the time to maturity for the option and $N(\cdot)$ the cumulative normal distribution. $N()$ with one argument is the univariate normal cumulative distribution. $N()$ with three arguments is the bivariate normal distribution with the correlation between the two normals given as the third arguement.

Formula 8.2: Roll–Geske–Whaley price of american call option paying one fixed dividend

Exercise 6.

The Black approximation to the price of an call option paying a fixed dividend is an approximation to the value of the call. Suppose the dividend is paid as some date t_1 before the maturity date of the option T. Blacks approximation calculates the value of two European options using the Black Scholes formula. One with expiry date equal to the ex dividend date of the options. Another with expiry date equal to the option expiry, but the current price of the underlying security is adjusted down by the amount of the dividend.

1. Implement Black's approximation.

```cpp
#include <cmath>
#include "normdist.h" // define the normal distribution functions
#include "fin_recipes.h" // the regular black sholes formula

double option_price_american_call_one_dividend(const double& S,
                                               const double& K,
                                               const double& r,
                                               const double& sigma,
                                               const double& tau,
                                               const double& D1,
                                               const double& tau1){
  if (D1 <= K* (1.0-exp(-r*(tau-tau1)))) // check for no exercise
    return option_price_call_black_scholes(S-exp(-r*tau1)*D1,K,r,sigma,tau);
  const double ACCURACY = 1e-6; // decrease this for more accuracy
  double sigma_sqr = sigma*sigma;
  double tau_sqrt = sqrt(tau);
  double tau1_sqrt = sqrt(tau1);
  double rho = - sqrt(tau1/tau);

  double S_bar = 0; // first find the S_bar that solves c=S_bar+D1-K
  double S_low=0; // the simplest: binomial search
  double S_high=S; // start by finding a very high S above S_bar
  double c = option_price_call_black_scholes(S_high,K,r,sigma,tau-tau1);
  double test = c-S_high-D1+K;
  while ( (test>0.0) && (S_high<=1e10) ) {
    S_high *= 2.0;
    c = option_price_call_black_scholes(S_high,K,r,sigma,tau-tau1);
    test = c-S_high-D1+K;
  };
  if (S_high>1e10) { // early exercise never optimal, find BS value
    return option_price_call_black_scholes(S-D1*exp(-r*tau1),K,r,sigma,tau);
  };
  S_bar = 0.5 * S_high; // now find S_bar that solves c=S_bar-D+K
  c = option_price_call_black_scholes(S_bar,K,r,sigma,tau-tau1);
  test = c-S_bar-D1+K;
  while ( (fabs(test)>ACCURACY) && ((S_high-S_low)>ACCURACY) ) {
    if (test<0.0) { S_high = S_bar; }
    else { S_low = S_bar; };
    S_bar = 0.5 * (S_high + S_low);
    c = option_price_call_black_scholes(S_bar,K,r,sigma,tau-tau1);
    test = c-S_bar-D1+K;
  };
  double a1 = (log((S-D1*exp(-r*tau1))/K) +( r+0.5*sigma_sqr)*tau) / (sigma*tau_sqrt);
  double a2 = a1 - sigma*tau_sqrt;
  double b1 = (log((S-D1*exp(-r*tau1))/S_bar)+(r+0.5*sigma_sqr)*tau1)/(sigma*tau1_sqrt);
  double b2 = b1 - sigma * tau1_sqrt;
  double C = (S-D1*exp(-r*tau1)) * N(b1) + (S-D1*exp(-r*tau1)) * N(a1,-b1,rho)
    - (K*exp(-r*tau))*N(a2,-b2,rho) - (K-D1)*exp(-r*tau1)*N(b2);
  return C;
};
```

Code 8.3: Option price, Roll–Geske–Whaley call formula for dividend paying stock

The program

```
void test_rgw_price_am_call_div(){
    double S = 100.0; double K = 100.0;
    double r = 0.1;   double sigma = 0.25;
    double tau = 1.0; double tau1 = 0.5;
    double D1 = 10.0;
    cout << " american call price with one dividend = "
         << option_price_american_call_one_dividend(S,K,r,sigma,tau,D1, tau1)<< endl;
};
```

provides the output

```
american call price with one dividend = 10.0166
```

Example 8.1: Example of pricing of option on stock paying one dividend during the life of the option

8.3 Options on futures

8.3.1 Black's model

For an European option written on a futures contract, we use an adjustment of the Black Scholes solution, which was developed in Black [1976]. Essentially we replace S_0 with $e^{-r(T-t)r}F$ in the Black Scholes formula, and get the formula shown in 8.3 and implemented in code 8.4.

$$c = e^{-r(T-t)}\left(FN(d_1) - KN(d_2)\right)$$

where

$$d_1 = \frac{\ln\left(\frac{F}{K}\right) + \frac{1}{2}\sigma^2(T-t)}{\sigma\sqrt{T-t}}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

F is the futures price, K is the exercise price, r the risk free interest rate, σ the volatility of the futures price, and $T-t$ is the time to maturity of the option (in years).

Formula 8.3: Blacks formula for the price of an European Call option with a futures contract as the underlying security

```
#include <cmath>                    // mathematics library
#include "normdist.h"               // normal distribution
using namespace std;

double futures_option_price_call_european_black( const double& F, // futures price
                                                 const double& K, // exercise price
                                                 const double& r, // interest rate
                                                 const double& sigma, // volatility
                                                 const double& time){ // time to maturity
    double sigma_sqr = sigma*sigma;
    double time_sqrt = sqrt(time);
    double d1 = (log (F/K) + 0.5 * sigma_sqr * time) / (sigma * time_sqrt);
    double d2 = d1 - sigma * time_sqrt;
    return exp(-r*time)*(F * N(d1) - K * N(d2));
};
```

Code 8.4: Price of European Call option on Futures contract

The program

```
void test_futures_option_price_black(){
    double F = 50.0; double K = 45.0;
    double r = 0.08; double sigma = 0.2;
    double time=0.5;
    cout << " european futures call option = "
         << futures_option_price_put_european_black(F,K,r,sigma,time) << endl;
};
```

provides the output

```
european futures call option = 0.851476
```

Example 8.2: Pricing of Futures option using the Black formula

Exercise 7.

The Black formula for a put option on a futures contract is

$$p = e^{-r(T-t)} \left(KN(-d_2) - FN(-d_1)\right)$$

where the varibles are as defined for the call option.

1. Implement the put option price.

8.4 Foreign Currency Options

Another relatively simple adjustment of the Black Scholes formula occurs when the underlying security is a currency exchange rate (spot rate). In this case one adjusts the Black-Scholes equation for the interest-rate differential.

Let S be the spot exchange rate, and now let r be the domestic interest rate and r_f the foreign interest rate. σ is then the volatility of changes in the exchange rate. The calculation of the price of an European call option is then shown in formula 8.4 and implented in code 8.5.

$$c = Se^{-r_f(T-t)}N(d_1) - Ke^{-r(T-t)}N(d_2)$$

where

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r - r_f + \frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

S is the spot exchange rate and K the exercise price. r is the domestic interest rate and r_f the foreign interest rate. σ is the volatility of changes in the exchange rate. $T-t$ is the time to maturity for the option.

Formula 8.4: European currency call

```
#include <cmath>
#include "normdist.h" // define the normal distribution function

double currency_option_price_call_european( const double& S, // exchange_rate,
                                            const double& X, // exercise,
                                            const double& r, // r_domestic,
                                            const double& r_f, // r_foreign,
                                            const double& sigma, // volatility,
                                            const double& time){ // time to maturity
    double sigma_sqr = sigma*sigma;
    double time_sqrt = sqrt(time);
    double d1 = (log(S/X) + (r-r_f+ (0.5*sigma_sqr)) * time)/(sigma*time_sqrt);
    double d2 = d1 - sigma * time_sqrt;
    return S * exp(-r_f*time) * N(d1) - X * exp(-r*time) * N(d2);
};
```

Code 8.5: European Futures Call option on currency

The program

```
void test_currency_option_european_call(){
    double S = 50.0;  double K = 52.0;
    double r = 0.08;  double rf=0.05;
    double sigma = 0.2; double time=0.5;
    cout << " european currency call option = "
         << currency_option_price_call_european(S,K,r,rf,sigma,time) << endl;
};
```

provides the output

```
european currency call option = 2.22556
```

Example 8.3: Pricing a foreign currency call option

Exercise 8.

The price for an european put for a currency option is

$$p = Ke^{-r(T-t)}N(-d_2) - Se^{-r_f(T-t)}N(-d_1)$$

1. Implement this formula.

8.5 Perpetual puts and calls

A *perpetal* option is one with no maturity date, it is inifinitely lived. Of course, only American perpetual options make any sense, European perpetual options would probably be hard to sell.[1] For both puts and calls analytical formulas has been developed. We consider the price of an American call, and discuss the put in an exercise. Formula 8.5 gives the analytical solution.

$$C^p = \frac{K}{h_1 - 1}\left(\frac{h_1 - 1}{h_1}\frac{S}{K}\right)^{h_1}$$

where

$$h_1 = \frac{1}{2} - \frac{r-q}{\sigma^2} + \sqrt{\left(\frac{r-q}{\sigma^2} - \frac{1}{2}\right)^2 + \frac{2r}{\sigma^2}}$$

S is the current price of the underlying security, K is the exercise price, r is the risk free interest rate, q is the dividend yield and σ is the volatility of the underlying asset.

Formula 8.5: Price for a perpetual call option

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

double option_price_american_perpetual_call(const double& S,
                                            const double& K,
                                            const double& r,
                                            const double& q,
                                            const double& sigma){
    double sigma_sqr=pow(sigma,2);
    double h1 = 0.5- ((r-q)/sigma_sqr);
    h1 += sqrt(pow(((r-q)/sigma_sqr-0.5),2)+2.0*r/sigma_sqr);
    double pric=(K/(h1-1.0))*pow(((h1-1.0)/h1)*(S/K),h1);
    return pric;
};
```

Code 8.6: Price for an american perpetual call option

The program

```
void test_option_price_perpetual_american_call(){
    double S=50.0; double K=40.0;
    double r=0.05; double q=0.02;
    double sigma=0.05;
    double price = option_price_american_perpetual_call(S,K,r,q,sigma);
    cout << " perpetual call price = " << price << endl;
};
```

provides the output

```
 perpetual call price = 19.4767
```

Example 8.4: Example of pricing of perpetual call

Exercise 9.

[1]Such options would be like the classical April fools present, a perpetual zero coupon bond...

The price for a perpetual american put is

$$P^p = \frac{K}{1-h_2}\left(\frac{h_2-1}{h_2}\frac{S}{K}\right)^{h_2}$$

where

$$h_2 = \frac{1}{2} - \frac{r-q}{\sigma^2} - \sqrt{\left(\frac{r-q}{\sigma^2} - \frac{1}{2}\right)^2 + \frac{2r}{\sigma^2}}$$

1. Implement the calculation of this formula.

8.6 Readings

Hull [2003] and McDonald [2002] are general references. A first formulation of an analytical call price with dividends was in Roll [1977]. This had some errors, that were partially corrected in Geske [1979], before Whaley [1981] gave a final, correct formula. See Hull [2003] for a textbook summary. Black [1976] is the original development of the futures option. The original formulations of European foreign currency option prices are in Garman and Kohlhagen [1983] and Grabbe [1983]. The price of a perpetual put was first shown in Merton [1973]. For a perpetual call see McDonald and Siegel [1986]. The notation for perpetual puts and calls follows the summary in [McDonald, 2002, pg. 393].

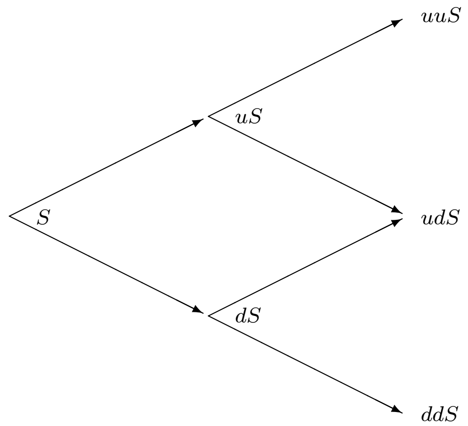
Chapter 9

Option pricing with binomial approximations

Contents

9.1 Introduction

We have shown binomial calculations given an up and down movement in chapter 5. However, binomial option pricing can also be viewed as an *approximation* to a continuous time distribution by judicious choice of the constants u and d. To do so one has to ask: Is it possible to find a parametrization (choice of u and d) of a binomial process



which has the same time series properties as a (continous time) process with the same mean and volatility? There is actually any number of ways of constructing this, hence one uses one degree of freedom on imposing that the nodes *reconnect*, by imposing $u = \frac{1}{d}$.

To value an option using this approach, we specify the number n of periods to split the time to maturity $(T - t)$ into, and then calculate the option using a binomial tree with that number of steps.

Given S, X, r, σ, T and the number of periods n, calculate

$$\Delta t = \frac{T-t}{n}$$

$$u = e^{\sigma\sqrt{\Delta t}}$$

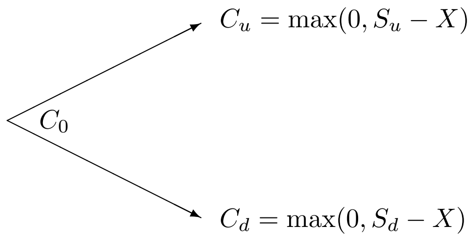
$$d = e^{-\sigma\sqrt{\Delta t}}$$

We also redefine the "risk neutral probabilities"

$$R = e^{r\Delta t}$$

$$q = \frac{R-d}{u-d}$$

To find the option price, will "roll backwards:" At node t, calculate the call price as a function of the two possible outcomes at time $t+1$. For example, if there is one step,



find the call price at time 0 as

$$C_0 = e^{-r}(qC_u + (1-q)C_d)$$

With more periods one will "roll backwards" as discussed in chapter 5

9.2 Pricing of options in the Black Scholes setting

Consider options on underlying securities not paying dividend.

9.2.1 European Options

For European options, binomial trees are not that much used, since the Black Scholes model will give the correct answer, but it is useful to see the construction of the binomial tree without the checks for early exercise, which is the American case.

The computer algorithm for a binomial in the following merits some comments. There is only one vector of call prices, and one may think one needs two, one at time t and another at time $t+1$. (Try to write down the way you would solve it before looking at the algorithm below.) But by using the fact that the branches reconnect, it is possible to get away with the algorithm below, using one less array. You may want to check how this works. It is also a useful way to make sure one understands binomial option pricing.

9.2.2 American Options

An American option differs from an European option by the exercise possibility. An American option can be exercised at any time up to the maturity date, unlike the European option, which can only be exercised at maturity. In general, there is unfortunately no analytical solution to the American option problem, but in some cases it can be found. For example, for an American call option on non-dividend paying stock, the American price is the same as the European call.

```
#include <cmath>        // standard mathematical library
#include <algorithm>        // defining the max() operator
#include <vector>       // STL vector templates
using namespace std;

double option_price_call_european_binomial( const double& S, // spot price
                                            const double& X, // exercice price
                                            const double& r, // interest rate
                                            const double& sigma, // volatility
                                            const double& t, // time to maturity
                                            const int& steps){ // no steps in binomial tree
   double R = exp(r*(t/steps));        // interest rate for each step
   double Rinv = 1.0/R;                // inverse of interest rate
   double u = exp(sigma*sqrt(t/steps)); // up movement
   double uu = u*u;
   double d = 1.0/u;
   double p_up = (R-d)/(u-d);
   double p_down = 1.0-p_up;
   vector<double> prices(steps+1); // price of underlying
   prices[0] = S*pow(d, steps); // fill in the endnodes.
   for (int i=1; i<=steps; ++i) prices[i] = uu*prices[i-1];
   vector<double> call_values(steps+1); // value of corresponding call
   for (int i=0; i<=steps; ++i) call_values[i] = max(0.0, (prices[i]-X)); // call payoffs at maturity
   for (int step=steps-1; step>=0; --step) {
      for (int i=0; i<=step; ++i) {
         call_values[i] = (p_up*call_values[i+1]+p_down*call_values[i])*Rinv;
      };
   };
   return call_values[0];
};
```

Code 9.1: Option price for binomial european

It is in the case of American options, allowing for the possibility of early exercise, that binomial approximations are useful. At each node we calculate the value of the option as a function of the next periods prices, and then check for the value exercising of exercising the option now

Code 9.2 illustrates the calculation of the price of an American call. Actually, for this particular case, the american price will equal the european.

```
#include <cmath>              // standard mathematical library
#include <algorithm>          // defines the max() operator
#include <vector>             // STL vector templates
using namespace std;

double option_price_call_american_binomial( const double& S, // spot price
                                            const double& X, // exercice price
                                            const double& r, // interest rate
                                            const double& sigma, // volatility
                                            const double& t, // time to maturity
                                            const int& steps) { // no steps in binomial tree
   double R = exp(r*(t/steps));          // interest rate for each step
   double Rinv = 1.0/R;                  // inverse of interest rate
   double u = exp(sigma*sqrt(t/steps)); // up movement
   double uu = u*u;
   double d = 1.0/u;
   double p_up = (R-d)/(u-d);
   double p_down = 1.0-p_up;
   vector<double> prices(steps+1); // price of underlying
   vector<double> call_values(steps+1); // value of corresponding call

   prices[0] = S*pow(d, steps); // fill in the endnodes.
   for (int i=1; i<=steps; ++i) prices[i] = uu*prices[i-1];
   for (int i=0; i<=steps; ++i) call_values[i] = max(0.0, (prices[i]-X)); // call payoffs at maturity
   for (int step=steps-1; step>=0; --step) {
      for (int i=0; i<=step; ++i) {
         call_values[i] = (p_up*call_values[i+1]+p_down*call_values[i])*Rinv;
         prices[i] = d*prices[i+1];
         call_values[i] = max(call_values[i],prices[i]-X);       // check for exercise
      };
   };
   return call_values[0];
};
```

Code 9.2: Binomial option price american opstion

The program

```
void test_binomial_approximations_option_pricing(){
   double S = 100.0; double K = 100.0;
   double r = 0.1;   double sigma = 0.25;
   double time=1.0;
   int no_steps = 100;
   cout << " european call = "
        << option_price_call_european_binomial(S,K,r,sigma,time,no_steps)
        << endl;
   cout << " american call = "
        << option_price_call_american_binomial(S,K,r,sigma,time,no_steps)
        << endl;
};
```

provides the output

```
 european call = 14.9505
 american call = 14.9505
```

Example 9.1: Option pricing using binomial approximations

9.2.3 Estimating partials.

It is always necessary to calculate the partial derivatives as well as the option price.

The binomial methods gives us ways to approximate these as well. How to find them in the binomial case are described in Hull [2003]. The code below is for the non–dividend case.

Delta , the derivative of the option price with respect to the underlying.

```
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;

double option_price_delta_american_call_binomial(const double& S,
                                                 const double& X,
                                                 const double& r,
                                                 const double& sigma,
                                                 const double& t,
                                                 const int& no_steps){ // steps in binomial
   vector<double> prices (no_steps+1);
   vector<double> call_values (no_steps+1);
   double R = exp(r*(t/no_steps));
   double Rinv = 1.0/R;
   double u = exp(sigma*sqrt(t/no_steps));
   double d = 1.0/u;
   double uu= u*u;
   double pUp = (R-d)/(u-d);
   double pDown = 1.0 - pUp;
   prices[0] = S*pow(d, no_steps);
   int i;
   for (i=1; i<=no_steps; ++i) prices[i] = uu*prices[i-1];
   for (i=0; i<=no_steps; ++i) call_values[i] = max(0.0, (prices[i]-X));
   for (int CurrStep=no_steps-1 ; CurrStep>=1; --CurrStep) {
      for (i=0; i<=CurrStep; ++i) {
         prices[i] = d*prices[i+1];
         call_values[i] = (pDown*call_values[i]+pUp*call_values[i+1])*Rinv;
         call_values[i] = max(call_values[i], prices[i]-X);       // check for exercise
      };
   };
   double delta = (call_values[1]-call_values[0])/(S*u-S*d);
   return delta;
};
```

Code 9.3: Delta

Other hedge parameters.

```
#include <cmath>
#include <algorithm>
#include "fin_recipes.h"

void option_price_partials_american_call_binomial(const double& S, // spot price
                                                  const double& X, // Exercise price,
                                                  const double& r, // interest rate
                                                  const double& sigma, // volatility
                                                  const double& time, // time to maturity
                                                  const int& no_steps, // steps in binomial
                                                  double& delta, // partial wrt S
                                                  double& gamma, // second prt wrt S
                                                  double& theta, // partial wrt time
                                                  double& vega, // partial wrt sigma
                                                  double& rho){ // partial wrt r
    vector<double> prices(no_steps+1);
    vector<double> call_values(no_steps+1);
    double delta_t =(time/no_steps);
    double R = exp(r*delta_t);
    double Rinv = 1.0/R;
    double u = exp(sigma*sqrt(delta_t));
    double d = 1.0/u;
    double uu= u*u;
    double pUp = (R-d)/(u-d);
    double pDown = 1.0 - pUp;
    prices[0] = S*pow(d, no_steps);
    for (int i=1; i<=no_steps; ++i) prices[i] = uu*prices[i-1];
    for (int i=0; i<=no_steps; ++i) call_values[i] = max(0.0, (prices[i]-X));
    for (int CurrStep=no_steps-1; CurrStep>=2; --CurrStep) {
        for (int i=0; i<=CurrStep; ++i) {
            prices[i] = d*prices[i+1];
            call_values[i] = (pDown*call_values[i]+pUp*call_values[i+1])*Rinv;
            call_values[i] = max(call_values[i], prices[i]-X);       // check for exercise
        };
    };
    double f22 = call_values[2];
    double f21 = call_values[1];
    double f20 = call_values[0];
    for (int i=0;i<=1;i++) {
        prices[i] = d*prices[i+1];
        call_values[i] = (pDown*call_values[i]+pUp*call_values[i+1])*Rinv;
        call_values[i] = max(call_values[i], prices[i]-X);       // check for exercise
    };
    double f11 = call_values[1];
    double f10 = call_values[0];
    prices[0] = d*prices[1];
    call_values[0] = (pDown*call_values[0]+pUp*call_values[1])*Rinv;
    call_values[0] = max(call_values[0], S-X);    // check for exercise on first date
    double f00 = call_values[0];
    delta = (f11-f10)/(S*u-S*d);
    double h = 0.5 * S * ( uu - d*d);
    gamma = ( (f22-f21)/(S*(uu-1)) - (f21-f20)/(S*(1-d*d)) ) / h;
    theta = (f21-f00) / (2*delta_t);
    double diff = 0.02;
    double tmp_sigma = sigma+diff;
    double tmp_prices = option_price_call_american_binomial(S,X,r,tmp_sigma,time,no_steps);
    vega = (tmp_prices-f00)/diff;
    diff = 0.05;
    double tmp_r = r+diff;
    tmp_prices = option_price_call_american_binomial(S,X,tmp_r,sigma,time,no_steps);
    rho = (tmp_prices-f00)/diff;
};
```

Code 9.4: Hedge parameters

The program

```
void test_binomial_approximations_option_price_partials(){
    double S = 100.0; double K = 100.0;
    double r = 0.1;   double sigma = 0.25;
    double time=1.0; int no_steps = 100;

    double delta, gamma, theta, vega, rho;
    option_price_partials_american_call_binomial(S,K,r, sigma, time, no_steps,
                                                  delta, gamma, theta, vega, rho);
    cout << " Call price partials " << endl;
    cout << " delta = " << delta << endl;
    cout << " gamma = " << gamma << endl;
    cout << " theta = " << theta << endl;
    cout << " vega = " << vega << endl;
    cout << " rho = " << rho << endl;
};
```

provides the output

```
 Call price partials
  delta = 0.699792
  gamma = 0.0140407
  theta = -9.89067
  vega  = 34.8536
  rho   = 56.9652
```

Example 9.2: Option price partials using binomial approximations

9.3 Adjusting for payouts for the underlying

The simplest case of a payout is the similar one to the one we saw in the Black Scholes case, a continous payout of y.

```
#include <cmath>            // standard mathematical library
#include <algorithm>        // defines the max() operator
#include <vector>           // STL vector templates
using namespace std;

double option_price_call_american_binomial( const double& S, // spot price
                                            const double& X, // exercice price
                                            const double& r, // interest rate
                                            const double& y, // continous payout
                                            const double& sigma, // volatility
                                            const double& t, // time to maturity
                                            const int& steps) { // no steps in binomial tree
   double R = exp(r*(t/steps));          // interest rate for each step
   double Rinv = 1.0/R;                  // inverse of interest rate
   double u = exp(sigma*sqrt(t/steps)); // up movement
   double uu = u*u;
   double d = 1.0/u;
   double p_up = (exp((r-y)*(t/steps))-d)/(u-d);
   double p_down = 1.0-p_up;
   vector<double> prices(steps+1); // price of underlying
   vector<double> call_values(steps+1); // value of corresponding call

   prices[0] = S*pow(d, steps); // fill in the endnodes.
   for (int i=1; i<=steps; ++i) prices[i] = uu*prices[i-1];
   for (int i=0; i<=steps; ++i) call_values[i] = max(0.0, (prices[i]-X)); // call payoffs at maturity
   for (int step=steps-1; step>=0; --step) {
      for (int i=0; i<=step; ++i) {
         call_values[i] = (p_up*call_values[i+1]+p_down*call_values[i])*Rinv;
         prices[i] = d*prices[i+1];
         call_values[i] = max(call_values[i],prices[i]-X);      // check for exercise
      };
   };
   return call_values[0];
};
```

Code 9.5: Binomial option price with continous payout

9.4 Pricing options on stocks paying dividends using a binomial approximation

9.4.1 Checking for early exercise in the binomial model.

If the underlying asset is a stock paying dividends during the maturity of the option, the terms of the option is not adjusted to reflect this cash payment, which means that the option value will reflect the dividend payments.

In the binomial model, the adjustment for dividends depend on whether the dividends are discrete or proportional.

9.4.2 Proportional dividends.

For proportional dividends, we simply multiply with an adjustment factor the stock prices at the ex–dividend date, the nodes in the binomial tree will "link up" again, and we can use the same "rolling back" procedure.

9.4.3 Discrete dividends

The problem is when the dividends are constant dollar amounts.

In that case the nodes of the binomial tree do not "link up," and the number of branches increases dramatically, which means that the time to do the calculation is increased.

The algorithm presented here implements this case, with no linkup, by constructing a binomial tree up to the ex-dividend date, and then, at the terminal nodes of that tree, call itself with one less dividend payment, and time to maturity the time remaining at the ex-dividend date. Doing that calculates the value of the option at the ex-dividend date, which is then compared to the value of exercising just before the ex-dividend date. It is a cute example of using recursion in simplifying calculations, but as with most recursive solutions, it has a cost in computing time. For large binomial trees and several dividends this procedure will take a long time.

```
#include <cmath>
#include <algorithm>
#include <vector>
#include "fin_recipes.h"
#include <iostream>

double option_price_call_american_proportional_dividends_binomial(const double& S,
                                                                  const double& X,
                                                                  const double& r,
                                                                  const double& sigma,
                                                                  const double& time,
                                                                  const int& no_steps,
                                                                  const vector<double>& dividend_times,
                                                                  const vector<double>& dividend_yields) {
    int no_dividends=dividend_times.size();
    if (no_dividends == 0) {
        return option_price_call_american_binomial(S,X,r,sigma,time,no_steps); // price w/o dividends
    };
    double delta_t = time/no_steps;
    double R = exp(r*delta_t);
    double Rinv = 1.0/R;
    double u = exp(sigma*sqrt(delta_t));
    double uu= u*u;
    double d = 1.0/u;
    double pUp = (R-d)/(u-d);
    double pDown = 1.0 - pUp;
    vector<int> dividend_steps(no_dividends); // when dividends are paid
    for (int i=0; i<no_dividends; ++i) {
        dividend_steps[i] = (int)(dividend_times[i]/time*no_steps);
    };
    vector<double> prices(no_steps+1);
    vector<double> call_prices(no_steps+1);
    prices[0] = S*pow(d, no_steps);
    for (int i=0; i<no_dividends; ++i) { prices[0]*=(1.0-dividend_yields[i]); };
    for (int i=1; i<=no_steps; ++i) { prices[i] = uu*prices[i-1]; }; // terminal tree nodes
    for (int i=0; i<=no_steps; ++i) call_prices[i] = max(0.0, (prices[i]-X));
    for (int step=no_steps-1; step>=0; --step) {
        for (int i=0;i<no_dividends;++i) { // check whether dividend paid
            if (step==dividend_steps[i]) {
                for (int j=0;j<=step;++j) {
                    prices[j]*=(1.0/(1.0-dividend_yields[i]));
                };
            };
        };
        for (int i=0; i<=step; ++i) {
            prices[i] = d*prices[i+1];
            call_prices[i] = (pDown*call_prices[i]+pUp*call_prices[i+1])*Rinv;
            call_prices[i] = max(call_prices[i], prices[i]-X);        // check for exercise
        };
    };
    return call_prices[0];
};
```

Code 9.6: Binomial option price of stock option where stock pays proportional dividends

```
#include <cmath>
#include <vector>
#include "fin_recipes.h"
#include <iostream>
double option_price_call_american_discrete_dividends_binomial(const double& S,
                                                              const double& K,
                                                              const double& r,
                                                              const double& sigma,
                                                              const double& t,
                                                              const int& steps,
                                                              const vector<double>& dividend_times,
                                                              const vector<double>& dividend_amounts) {
    int no_dividends = dividend_times.size();
    if (no_dividends==0) return option_price_call_american_binomial(S,K,r,sigma,t,steps);// just do regular
    int steps_before_dividend = (int)(dividend_times[0]/t*steps);
    const double R = exp(r*(t/steps));
    const double Rinv = 1.0/R;
    const double u = exp(sigma*sqrt(t/steps));
    const double d = 1.0/u;
    const double pUp = (R-d)/(u-d);
    const double pDown = 1.0 - pUp;
    double dividend_amount = dividend_amounts[0];
    vector<double> tmp_dividend_times(no_dividends-1); // temporaries with
    vector<double> tmp_dividend_amounts(no_dividends-1); // one less dividend
    for (int i=0;i<(no_dividends-1);++i){
        tmp_dividend_amounts[i] = dividend_amounts[i+1];
        tmp_dividend_times[i] = dividend_times[i+1] - dividend_times[0];
    };
    vector<double> prices(steps_before_dividend+1);
    vector<double> call_values(steps_before_dividend+1);
    prices[0] = S*pow(d, steps_before_dividend);
    for (int i=1; i<=steps_before_dividend; ++i) prices[i] = u*u*prices[i-1];
    for (int i=0; i<=steps_before_dividend; ++i){
        double value_alive
            = option_price_call_american_discrete_dividends_binomial(prices[i]-dividend_amount,K, r, sigma,
                                                                     t-dividend_times[0],// time after first dividend
                                                                     steps-steps_before_dividend,
                                                                     tmp_dividend_times,
                                                                     tmp_dividend_amounts);
        call_values[i] = max(value_alive,(prices[i]-K)); // compare to exercising now
    };
    for (int step=steps_before_dividend-1; step>=0; --step) {
        for (int i=0; i<=step; ++i) {
            prices[i] = d*prices[i+1];
            call_values[i] = (pDown*call_values[i]+pUp*call_values[i+1])*Rinv;
            call_values[i] = max(call_values[i], prices[i]-K);
        };
    };
    return call_values[0];
};
```

Code 9.7: Binomial option price of stock option where stock pays discrete dividends

The program

```
void test_binomial_approximations_option_price_dividends(){
    double S = 100.0; double K = 100.0;
    double r = 0.10;   double sigma = 0.25;
    double time=1.0;
    int no_steps = 100;
    double d=0.02;
    cout << " call price with continuous dividend payout = "
         << option_price_call_american_binomial(S,K,r,d,sigma,time,no_steps) << endl;
    vector<double> dividend_times; vector<double> dividend_yields;
    dividend_times.push_back(0.25); dividend_yields.push_back(0.025);
    dividend_times.push_back(0.75); dividend_yields.push_back(0.025);
    cout << " call price with proportial dividend yields at discrete dates = "
         << option_price_call_american_proportional_dividends_binomial(S,K,r,sigma,time,no_steps,
                                                              dividend_times, dividend_yields)
         << endl;

    vector<double> dividend_amounts;
    dividend_amounts.push_back(2.5);
    dividend_amounts.push_back(2.5);
    cout << " call price with proportial dividend amounts at discrete dates = "
         << option_price_call_american_discrete_dividends_binomial(S,K,r,sigma,time,no_steps,
                                                              dividend_times, dividend_amounts)
         << endl;
};
```

provides the output

```
 call price with continuous dividend payout = 13.5926
 call price with proportial dividend yields at discrete dates = 11.8604
 call price with proportial dividend amounts at discrete dates = 12.0233
```

Example 9.3: Binomial pricing with dividends

9.5 Option on futures

For American options, because of the feasibility of early exercise, the binomial model is used to approximate the option value.

```
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;

double futures_option_price_call_american_binomial(const double& F, // price futures contract
                                                   const double& X, // exercise price
                                                   const double& r, // interest rate
                                                   const double& sigma, // volatility
                                                   const double& time, // time to maturity
                                                   const int& no_steps) { // number of steps
   vector<double> futures_prices(no_steps+1);
   vector<double> call_values (no_steps+1);
   double t_delta= time/no_steps;
   double Rinv = exp(-r*(t_delta));
   double u = exp(sigma*sqrt(t_delta));
   double d = 1.0/u;
   double uu= u*u;
   double pUp = (1-d)/(u-d); // note how probability is calculated
   double pDown = 1.0 - pUp;
   futures_prices[0] = F*pow(d, no_steps);
   int i;
   for (i=1; i<=no_steps; ++i) futures_prices[i] = uu*futures_prices[i-1]; // terminal tree nodes
   for (i=0; i<=no_steps; ++i) call_values[i] = max(0.0, (futures_prices[i]-X));
   for (int step=no_steps-1; step>=0; --step) {
      for (i=0; i<=step; ++i) {
         futures_prices[i] = d*futures_prices[i+1];
         call_values[i] = (pDown*call_values[i]+pUp*call_values[i+1])*Rinv;
         call_values[i] = max(call_values[i], futures_prices[i]-X); // check for exercise
      };
   };
   return call_values[0];
};
```

Code 9.8: Option on futures

The program

```
void test_binomial_approximations_futures_options(){
    double F = 50.0; double K = 45.0;
    double r = 0.08; double sigma = 0.2;
    double time=0.5;
    int no_steps=100;
    cout << " european futures call option = "
         << futures_option_price_call_american_binomial(F,K,r,sigma,time,no_steps) << endl;
};
```

provides the output

```
european futures call option = 5.74254
```

Example 9.4: Futures option price

9.6 Foreign Currency options

For American options, the usual method is approximation using binomial trees, checking for early exercise due to the interest rate differential.

```
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;

double currency_option_price_call_american_binomial(const double& S,
                                                    const double& K,
                                                    const double& r,
                                                    const double& r_f,
                                                    const double& sigma,
                                                    const double& time,
                                                    const int& no_steps) {
    vector<double> exchange_rates(no_steps+1);
    vector<double> call_values(no_steps+1);
    double t_delta= time/no_steps;
    double Rinv = exp(-r*(t_delta));
    double u = exp(sigma*sqrt(t_delta));
    double d = 1.0/u;
    double uu= u*u;
    double pUp = (exp((r-r_f)*t_delta)-d)/(u-d); // adjust for foreign int.rate
    double pDown = 1.0 - pUp;
    exchange_rates[0] = S*pow(d, no_steps);
    int i;
    for (i=1; i<=no_steps; ++i) {
        exchange_rates[i] = uu*exchange_rates[i-1]; // terminal tree nodes
    }
    for (i=0; i<=no_steps; ++i) call_values[i] = max(0.0, (exchange_rates[i]-K));
    for (int step=no_steps-1; step>=0; --step) {
        for (i=0; i<=step; ++i) {
            exchange_rates[i] = d*exchange_rates[i+1];
            call_values[i] = (pDown*call_values[i]+pUp*call_values[i+1])*Rinv;
            call_values[i] = max(call_values[i], exchange_rates[i]-K); // check for exercise
        };
    };
    return call_values[0];
};
```

Code 9.9: Binomial Currency Option

The program

```
void test_binomial_approximations_currency_options(){
    double S = 50.0;  double K = 52.0;
    double r = 0.08;  double rf=0.05;
    double sigma = 0.2; double time=0.5;

    int no_steps = 100;
    cout << " european currency option call = "
         << currency_option_price_call_american_binomial(S,K,r,rf,sigma,time,no_steps) << endl;
};
```

provides the output

```
european currency option call = 2.23129
```

Example 9.5: Currency option price

9.7 References

The original source for binomial option pricing was the paper by Cox et al. [1979]. Good textbook discussions are in Cox and Rubinstein [1985], Bossaerts and Ødegaard [2001] and Hull [2003].

Exercise 10.

Consider an European call option on non-dividend paying stock, where $S = 100$, $K = 100$, $\sigma = 0.2$, $(T - t) = 1$ and $r = 0.1$.

1. Calculate the price of this option using Black Scholes
2. Calculate the price using a binomial approximation, using 10, 100 and 1000 steps in the approximation.
3. Discuss what are sources of differences in the estimated prices.

Chapter 10

Finite Differences

Contents

10.1 Explicit Finite differences

The method of choice for any engineer given a differential equation to solve is to numerically approximate it using a finite difference scheme, which is to approximate the continous differential equation with a discrete *difference* equation, and solve this difference equation.

10.2 European Options.

For European options we do not need to use the finite difference scheme, but we show how one would find the european price for comparison purposes. We show the case of an explicit finite difference scheme in code 10.1. A problem with the explicit version is that it may not converge for certain combinations of inputs.

```
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;

double option_price_put_european_finite_diff_explicit(const double& S,
                                                      const double& X,
                                                      const double& r,
                                                      const double& sigma,
                                                      const double& time,
                                                      const int& no_S_steps,
                                                      const int& no_t_steps) {
    double sigma_sqr = sigma*sigma;
    unsigned int M;        // need M = no_S_steps to be even:
    if ((no_S_steps%2)==1) { M=no_S_steps+1; } else { M=no_S_steps; };
    double delta_S = 2.0*S/M;
    vector<double> S_values(M+1);
    for (unsigned m=0;m<=M;m++) { S_values[m] = m*delta_S; };
    int N=no_t_steps;
    double delta_t = time/N;

    vector<double> a(M);
    vector<double> b(M);
    vector<double> c(M);
    double r1=1.0/(1.0+r*delta_t);
    double r2=delta_t/(1.0+r*delta_t);
    for (unsigned int j=1;j<M;j++){
        a[j] = r2*0.5*j*(-r+sigma_sqr*j);
        b[j] = r1*(1.0-sigma_sqr*j*j*delta_t);
        c[j] = r2*0.5*j*(r+sigma_sqr*j);
    };
    vector<double> f_next(M+1);
    for (unsigned m=0;m<=M;++m) { f_next[m]=max(0.0,X-S_values[m]); };
    double f[M+1];
    for (int t=N-1;t>=0;--t) {
        f[0]=X;
        for (unsigned m=1;m<M;++m) {
            f[m]=a[m]*f_next[m-1]+b[m]*f_next[m]+c[m]*f_next[m+1];
        };
        f[M] = 0;
        for (unsigned m=0;m<=M;++m) { f_next[m] = f[m]; };
    };
    return f[M/2];
};
```

Code 10.1: Explicit finite differences calculation of european put option

10.3 American Options.

We now compare the American versions of the same algoritms, the only difference being the check for exercise at each point. Code 10.2 shows the code for an american put option.

```
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;

double option_price_put_american_finite_diff_explicit( const double& S,
                                                       const double& X,
                                                       const double& r,
                                                       const double& sigma,
                                                       const double& time,
                                                       const int& no_S_steps,
                                                       const int& no_t_steps) {
    double sigma_sqr = sigma*sigma;

    int M;          // need M = no_S_steps to be even:
    if ((no_S_steps%2)==1) { M=no_S_steps+1; } else { M=no_S_steps; };
    double delta_S = 2.0*S/M;
    vector<double> S_values(M+1);
    for (int m=0;m<=M;m++) { S_values[m] = m*delta_S; };
    int N=no_t_steps;
    double delta_t = time/N;

    vector<double> a(M);
    vector<double> b(M);
    vector<double> c(M);
    double r1=1.0/(1.0+r*delta_t);
    double r2=delta_t/(1.0+r*delta_t);
    for (int j=1;j<M;j++){
        a[j] = r2*0.5*j*(-r+sigma_sqr*j);
        b[j] = r1*(1.0-sigma_sqr*j*j*delta_t);
        c[j] = r2*0.5*j*(r+sigma_sqr*j);
    };
    vector<double> f_next(M+1);
    for (int m=0;m<=M;++m) { f_next[m]=max(0.0,X-S_values[m]); };
    vector<double> f(M+1);
    for (int t=N-1;t>=0;--t) {
        f[0]=X;
        for (int m=1;m<M;++m) {
            f[m]=a[m]*f_next[m-1]+b[m]*f_next[m]+c[m]*f_next[m+1];
            f[m] = max(f[m],X-S_values[m]); // check for exercise
        };
        f[M] = 0;
        for (int m=0;m<=M;++m) { f_next[m] = f[m]; };
    };
    return f[M/2];
};
```

Code 10.2: Explicit finite differences calculation of american put option

Readings Brennan and Schwartz [1978] is one of the first finance applications of finite differences. Section 14.7 of Hull [1993] has a short introduction to finite differences. Wilmott et al. [1994] is an exhaustive source on option pricing from the perspective of solving partial differential equations.

The program

```
void test_explicit_finite_differences(){
    double S = 50.0;
    double K = 50.0;
    double r = 0.1;
    double sigma = 0.4;
    double time=0.4167;
    int no_S_steps=20;
    int no_t_steps=11;
    cout << " explicit finite differences, european put price = ";
    cout << option_price_put_european_finite_diff_explicit(S,K,r,sigma,time,no_S_steps,no_t_steps)
         << endl;
    cout << " explicit finite differences, american put price = ";
    cout << option_price_put_american_finite_diff_explicit(S,K,r,sigma,time,no_S_steps,no_t_steps)
         << endl;
};
```

provides the output

```
 explicit finite differences, european put price = 4.03667
 explicit finite differences, american put price = 4.25085
```

Example 10.1: Explicit finite differences

Chapter 11

Option pricing by simulation

Contents

We now consider using Monte Carlo methods to estimate the price of an European option, and let us first consider the case of the "usual" European Call, which is priced by the Black Scholes equation. Since there is already a closed form solution for this case, it is not really necessary to use simulations, but we use the case of the standard call for illustrative purposes.

At maturity, a call option is worth

$$c_T = \max(0, S_T - X)$$

At an earlier date t, the option value will be the expected present value of this.

$$c_t = E[PV(\max(0, S_T - X)]$$

Now, an important simplifying feature of option pricing is the "risk neutral result," which implies that we can treat the (suitably transformed) problem as the decision of a risk neutral decision maker, if we also modify the expected return of the underlying asset such that this earns the risk free rate.

$$c_t = e^{-r(T-t)} E^*[\max(0, S_T - X)],$$

where $E^*[\cdot]$ is a transformation of the original expectation. One way to estimate the value of the call is to simulate a large number of sample values of S_T according to the assumed price process, and find the estimated call price as the average of the simulated values. By appealing to a law of large numbers, this average will converge to the actual call value, where the rate of convergence will depend on how many simulations we perform.

11.1 Simulating lognormally distributed random variables

Lognormal variables are simulated as follows. Let $\tilde{x}$ be normally distributed with mean zero and variance one. If S_t follows a lognormal distribution, then the one-period-later price S_{t+1} is simulated as

$$S_{t+1} = S_t e^{\left(r - \frac{1}{2}\sigma^2\right) + \sigma\tilde{x}},$$

or more generally, if the current time is t and terminal date is T, with a time between t and T of $(T-t)$,

$$S_T = S_t e^{\left(r - \frac{1}{2}\sigma^2\right)(T-t) + \sigma\sqrt{T-t}\tilde{x}}$$

Simulation of lognormal random variables is illustrated by code 11.1.

```
#include <cmath>
using namespace std;
#include "normdist.h"

double simulate_lognormal_random_variable(const double& S, // current value of variable
                                          const double& r, // interest rate
                                          const double& sigma, // volatitily
                                          const double& time) { // time to final date
   double R = (r - 0.5 * pow(sigma,2) )*time;
   double SD = sigma * sqrt(time);
   return S * exp(R + SD * random_normal());
};
```

Code 11.1: Simulating a lognormally distributed random variable

11.2 Pricing of European Call options

For the purposes of doing the Monte Carlo estimation of the price if an European call

$$c_t = e^{-r(T-t)}E[\max(0, S_T - X)],$$

note that here one merely need to simulate the terminal price of the underlying, S_T, the price of the underlying at any time between t and T is not relevant for pricing. We proceed by simulating lognormally distributed random variables, which gives us a set of observations of the terminal price S_T. If we let $S_{T,1}, S_{T,2}, S_{T,3}, \ldots S_{T,n}$ denote the n simulated values, we will estimate $E^*[\max(0, S_T - X)]$ as the average of option payoffs at maturity, discounted at the risk free rate.

$$\hat{c}_t = e^{-r(T-t)} \left(\sum_{i=1}^{n} \max\left(0, S_{T,i} - X\right) \right)$$

Code 11.2 shows the implementation of a Monte Carlo estimation of an European call option.

```
#include <cmath> // standard mathematical functions
#include <algorithm> // define the max() function
using namespace std;
#include "normdist.h" // definition of random number generator

double
option_price_call_european_simulated( const double& S, // price of underlying
                                      const double& X, // exercise price
                                      const double& r, // risk free interest rate
                                      const double& sigma, // volatility of underlying
                                      const double& time, // time to maturity (in years)
                                      const int& no_sims){ // number of simulations
    double R = (r - 0.5 * pow(sigma,2))*time;
    double SD = sigma * sqrt(time);
    double sum_payoffs = 0.0;
    for (int n=1; n<=no_sims; n++) {
        double S_T = S* exp(R + SD * random_normal());
        sum_payoffs += max(0.0, S_T-X);
    };
    return exp(-r*time) * (sum_payoffs/double(no_sims));
};
```

Code 11.2: European Call option priced by simulation

11.3 Hedge parameters

It is of course, just as in the standard case, desirable to estimate hedge parameters as well as option prices. We will show how one can find an estimate of the option *delta*, the first derivative of the call price with respect to the underlying security: $\Delta = \frac{\partial c_t}{\partial S}$. To understand how one goes about estimating this, let us recall that the first derivative of a function f is defined as the limit

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Thinking of $f(S)$ as the option price formula $c_t = f(S; X, r, \sigma, (T-t))$, we see that we can evaluate the option price at two different values of the underlying, S and $S+q$, where q is a small quantity, and estimate the option delta as

$$\widehat{\Delta} = \frac{f(S+q) - f(S)}{q}$$

In the case of Monte Carlo estimation, it is very important that this is done by using the same sequence of random variables to estimate the two option prices with prices of the underlying S and $S+q$. Code 11.3 implements this estimation of the option delta. One can estimate other hedge parameters in a simular way.

```
#include <cmath> // standard mathematical functions
#include <algorithm> // define the max() function
using namespace std;
#include "normdist.h" // definition of random number generator

double option_price_delta_call_european_simulated(const double& S,
                                                   const double& X,
                                                   const double& r,
                                                   const double& sigma,
                                                   const double& time,
                                                   const int& no_sims){
    double R = (r - 0.5 * pow(sigma,2))*time;
    double SD = sigma * sqrt(time);
    double sum_payoffs = 0.0;
    double sum_payoffs_q = 0.0;
    double q = S*0.01;
    for (int n=1; n<=no_sims; n++) {
        double Z = random_normal();
        double S_T = S* exp(R + SD * Z);
        sum_payoffs += max(0.0, S_T-X);
        double S_T_q = (S+q)* exp(R + SD * Z);
        sum_payoffs_q += max(0.0, S_T_q-X);
    };
    double c = exp(-r*time) * ( sum_payoffs/no_sims);
    double c_q = exp(-r*time) * ( sum_payoffs_q/no_sims);
    return (c_q-c)/q;
};
```

Code 11.3: Estimate Delta of European Call option priced by Monte Carlo

11.4 More general payoffs. Function prototypes

The above shows the case for a call option. If we want to price other types of options, with different payoffs we could write similar routines for every possible case. But this would be wasteful, instead a bit of thought allows us to write option valuations for any kind of option whose payoff depend on the value of the underlying at maturity, only. Let us now move toward a generic routine for pricing derivatives with Monte Carlo. This relies on the ability of C++ to write subroutines which one call with *function prototypes*, i.e. that in the call to to the subroutine/function one provides a function instead of a variable. Consider pricing of standard European put and call options. At maturity each option only depend on the value of the underlying S_T and the exercise price X through the relations

$$C_T = \max(S_T - X, 0)$$

$$P_T = \max(X - S_T, 0)$$

Code 11.4 shows two C++ functions which calculates this.

```
#include <algorithm>
using namespace std;

double payoff_call(const double& price, const double& X){
    return max(0.0,price-X);
};

double payoff_put (const double& price, const double& X) {
    return max(0.0,X-price);
};
```

Code 11.4: Payoff call and put options

The interesting part comes when one realises one can write a generic simulation routine to which one provide one of these functions, or some other function describing a payoff which only depends on the price of the underlying and some constant. Code 11.5 shows how this is done.

```
#include <cmath>
using namespace std;
#include "fin_recipes.h"

double
derivative_price_simulate_european_option_generic(const double& S, // price of underlying
                                                  const double& X, // used by user provided payoff function
                                                  const double& r, // risk free interest rate
                                                  const double& sigma, // volatility
                                                  const double& time, // time to maturity
                                                  double payoff(const double& price, const double& X),
                                                  // user provided function
                                                  const int& no_sims) { // number of simulations to run
    double sum_payoffs=0;
    for (int n=0; n<no_sims; n++) {
       double S_T = simulate_lognormal_random_variable(S,r,sigma,time);
       sum_payoffs += payoff(S_T,X);
    };
    return exp(-r*time) * (sum_payoffs/no_sims);
};
```

Code 11.5: Generic simulation pricing

Note the presence of the line

```
double payoff(const double& price, const double& X),
```

in the subroutine call. When this function is called, the calling program will need to provide a function to put there, such as the Black Scholes example above. Code 11.6 shows a complete example of how this is done. Running the program in code 11.6 results in the output:

```
#include "fin_recipes.h"
#include <algorithm>
#include <iostream>
using namespace std;

double payoff_european_call(const double& price, const double& X){ return max(0.0,price-X); };
double payoff_european_put (const double& price, const double& X) { return max(0.0,X-price); };

int main(){
    double S      = 100.0;
    double X      = 100.0;
    double r      = 0.1;
    double sigma = 0.25;
    double time  = 1.0;
    int no_sims   = 50000;
    cout << "Black Scholes call option price = "
         << option_price_call_black_scholes(S,X,r,sigma,time)
         << endl;
    cout << "Simulated call option price = "
         << derivative_price_simulate_european_option_generic(S,X,r,sigma,time,payoff_european_call,no_sims)
         << endl;
    cout << "Black Scholes put option price = "
         << option_price_put_black_scholes(S,X,r,sigma,time)
         << endl;
    cout << "Simulated put option price = "
         << derivative_price_simulate_european_option_generic(S,X,r,sigma,time,payoff_european_put,no_sims)
         << endl;
};
```

Code 11.6: Simulating Black Scholes values using the generic routine

```
Simulated call option price = 14.995
Black Scholes call option price = 14.9758
Simulated put option price = 5.5599
Black Scholes put option price = 5.45954
```

As we see, even with as many as 50,000 simuations, the option prices estimated using Monte Carlo still differs substantially from the "true" values.

11.5 Improving the efficiency in simulation

There are a number of ways of "improving" the implementation of Monte Carlo estimation such that the estimate is closer to the true value.

11.5.1 Control variates.

One is the method of control variates. The idea is simple. When one generates the set of terminal values of the underlying security, one can value several derivatives using the same set of terminal values. What if one of the derivatives we value using the terminal values is one which we have an analytical solution to? For example, suppose we calculate the value of an at the money European call option using both the (analytical) Black Scholes formula and Monte Carlo simulation. If it turns out that the Monte Carlo estimate overvalues the option price, we think that this will also be the case for other derivatives valued using the same set of simulated terminal values. We therefore move the estimate of the price of the derivative of interest downwards.

Thus, suppose we want to value an European put and we use the price of an at the money European call as the control variate. Using the same set of simulated terminal values $S_{T,i}$, we estimate the two options using

Monte Carlo as:

$$\hat{p}_t = e^{-r(T-t)} \left(\sum_{i=1}^{n} \max(0, X - S_{T,i}) \right)$$

$$\hat{c}_t = e^{-r(T-t)} \left(\sum_{i=1}^{n} \max(0, S_{T,i} - X) \right)$$

We calculate the Black Scholes value of the call $\hat{c}_t^{bs}$, and calculate p_t^{cv}, the estimate of the put price with a control variate adjustment, as follows

$$\hat{p}_t^{cv} = \hat{p}_t + (c_t^{bs} - \hat{c}_t)$$

One can use other derivatives than the at-the-money call as the control variate, the only limitation being that it has a tractable analytical solution.

Code 11.7 shows the implementation of a Monte Carlo estimation using an at-the-money European call as the control variate.

```
#include <cmath>
using namespace std;
#include "fin_recipes.h"

double
derivative_price_simulate_european_option_generic_with_control_variate(const double& S,
                                                                        const double& X,
                                                                        const double& r,
                                                                        const double& sigma,
                                                                        const double& time,
                                                                        double payoff(const double& S,
                                                                                      const double& X),
                                                                        const int& no_sims) {
    double c_bs = option_price_call_black_scholes(S,S,r,sigma,time);// price an at the money Black Scholes call
    double sum_payoffs=0;
    double sum_payoffs_bs=0;
    for (int n=0; n<no_sims; n++) {
        double S_T= simulate_lognormal_random_variable(S,r,sigma,time);
        sum_payoffs += payoff(S_T,X);
        sum_payoffs_bs += payoff_call(S_T,S); // simulate at the money Black Scholes price
    };
    double c_sim = exp(-r*time) * (sum_payoffs/no_sims);
    double c_bs_sim = exp(-r*time) * (sum_payoffs_bs/no_sims);
    c_sim += (c_bs-c_bs_sim);
    return c_sim;
};
```

Code 11.7: Generic with control variate

11.5.2 Antithetic variates.

An alternative to using control variates is to consider the method of *antithetic* variates. The idea behind this is that Monte Carlo works best if the simulated variables are "spread" out as closely as possible to the true distribution. Here we are simulating unit normal random variables. One property of the normal is that it is symmetric around zero, and the median value is zero. Why don't we enforce this in the simulated terminal values? An easy way to do this is to first simulate a unit random normal variable Z, and then use both Z and $-Z$ to generate the lognormal random variables. Code 11.8 shows the implementation of this idea. Boyle [1977] shows that the efficiency gain with antithetic variates is not particularly large. There are other ways of ensuring that the simulated values really span the whole sample space, sometimes called "pseudo Monte Carlo."

```
#include "fin_recipes.h"
#include "normdist.h"
#include <cmath>
using namespace std;

double
derivative_price_simulate_european_option_generic_with_antithetic_variate(const double& S,
                                                                          const double& X,
                                                                          const double& r,
                                                                          const double& sigma,
                                                                          const double& time,
                                                                          double payoff(const double& S,
                                                                                        const double& X),
                                                                          const int& no_sims) {
    double R = (r - 0.5 * pow(sigma,2) )*time;
    double SD = sigma * sqrt(time);
    double sum_payoffs=0;
    for (int n=0; n<no_sims; n++) {
        double x=random_normal();
        double S1 = S * exp(R + SD * x);
        sum_payoffs += payoff(S1,X);
        double S2 = S * exp(R + SD * (-x));
        sum_payoffs += payoff(S2,X);
    };
    return exp(-r*time) * (sum_payoffs/(2*no_sims));
};
```

Code 11.8: Generic with antithetic variates

11.5.3 Example

Let us see how these improvements change actual values. We use the same numbers as in code 11.6, but add estimation using control and antithetic variates. Code 11.9 shows a complete example of how this is done.

```
#include "fin_recipes.h"
#include <algorithm>
#include <iostream>
using namespace std;

double payoff_european_call(const double& price, const double& X){ return max(0.0,price-X); };
double payoff_european_put (const double& price, const double& X) { return max(0.0,X-price); };

int main(){
    double S     = 100.0;
    double X     = 100.0;
    double r     = 0.1;
    double sigma = 0.25;
    double time  = 1.0;
    int no_sims  = 50000;
    cout << "Black Scholes call option price = "
         << option_price_call_black_scholes(S,X,r,sigma,time)
         << endl;
    cout << "Simulated call option price = "
         << derivative_price_simulate_european_option_generic(S,X,r,sigma,time,
                                                              payoff_european_call,no_sims)
         << endl;
    cout << "Simulated call option price, CV = "
         << derivative_price_simulate_european_option_generic_with_control_variate(S,X,r,sigma,time,
                                                              payoff_european_call,no_sims)
         << endl;
    cout << "Simulated call option price, AV = "
         << derivative_price_simulate_european_option_generic_with_antithetic_variate(S,X,r,sigma,time,
                                                              payoff_european_call,no_sims)
         << endl;
    cout << "Black Scholes put option price = "
         << option_price_put_black_scholes(S,X,r,sigma,time)
         << endl;
    cout << "Simulated put option price = "
         << derivative_price_simulate_european_option_generic(S,X,r,sigma,time,payoff_european_put,no_sims)
         << endl;
    cout << "Simulated put option price, CV = "
         << derivative_price_simulate_european_option_generic_with_control_variate(S,X,r,sigma,time,
                                                              payoff_european_put,no_sims)
         << endl;
    cout << "Simulated put option price, AV = "
         << derivative_price_simulate_european_option_generic_with_antithetic_variate(S,X,r,sigma,time,
                                                              payoff_european_put,no_sims)
         << endl;
};
```

Code 11.9: Simulating Black Scholes values using the generic Monte Carlo routines, with efficiency improvents

Running this program results in the output:

```
Black Scholes call option price = 14.9758
Simulated call option price     = 14.995
Simulated call option price, CV = 14.9758
Simulated call option price, AV = 14.9919
Black Scholes put option price  = 5.45954
Simulated put option price      = 5.41861
Simulated put option price,  CV = 5.42541
```

```
Simulated put option price,  AV = 5.46043
```

11.6 More exotic options

These generic routines can also be used to price other options. Any European option that only depends on the terminal value of the price of the underlying security can be valued. Consider the *binary* options discussed by e.g. Hull [2003]. An *cash or nothing call* pays a fixed amount Q if the price of the asset is above the exercise price at maturity, otherwise nothing. An *asset or nothing call* pays the price of the asset if the price is above the exercise price at maturity, otherwise nothing. Both of these options are easy to implement using the generic routines above, all that is necesary is to provide the payoff functions as shown in code 11.10.

```
double payoff_cash_or_nothing_call(const double& price, const double& X){
    double Q=1;
    if (price>=X) return Q;
    return 0;
};

double payoff_asset_or_nothing_call(const double& price, const double& X){
    if (price>=X) return price;
    return 0;
};
```

Code 11.10: Payoff binary options

Now, many exotic options are not simply functions of the terminal price of the underlying security, but depend on the evolution of the price from "now" till the terminal date of the option. For example options that depend on the average of the price of the underlying (Asian options). For such cases one will have to simulate the whole path. We will return to these cases in the chapter on pricing of exotic options.

Further Reading Boyle [1977] is a good early source on the use of the Monte Carlo technique for pricing derivatives. Simulation is also covered in Hull [2003].

11.7 Exercises

Exercise 11.

Consider the pricing of an European Call option as implemented in code 11.2, and the generic formula for pricing with Monte Carlo for European options that only depend on the terminal value of the underlying security, as implemented in code 11.5.

Note the difference in the implementation of the lognormal simulation of terminal values. Why can one argue that the first implementation is more efficient than the other?

Chapter 12

Approximations

Contents

There has been developed some useful *approximations* to various specific options. It is of course American options that are approximated. The particular example we will look at, is a general quadratic approximation to American call and put prices.

12.1 A quadratic approximation to American prices due to Barone–Adesi and Whaley.

We now discuss an approximation to the option price of an American option on a commodity, described in Barone-Adesi and Whaley [1987] (BAW).[1] The commodity is assumed to have a continuous payout b. The starting point for the approximation is the (Black-Scholes) stochastic differential equation valid for the value of any derivative with price V.

$$\frac{1}{2}\sigma^2 S^2 V_S S + bSV_S - rV + V_t = 0 \tag{12.1}$$

Here V is the (unknown) formula that determines the price of the contingent claim. For an European option the value of V has a known solution, the adjusted Black Scholes formula. For American options, which may be exercised early, there is no known analytical solution.

To do their approximation, BAW decomposes the American price into the European price and the early exercise premium

$$C(S,T) = c(S,T) + \varepsilon_C(S,T)$$

Here ε_C is the early exercise premium. The insight used by BAW is that ε_C must *also* satisfy the same partial differential equation. To come up with an approximation BAW transformed equation (12.1) into one where the terms involving V_t are neglible, removed these, and ended up with a standard linear homeogenous second order equation, which has a known solution.

The functional form of the approximation is shown in formula 12.1.

In implementing this formula, the only problem is finding the critical value S^*. This is the classical problem of finding a root of the equation

$$g(S^*) = S^* - X - c(S^*) - \frac{S^*}{q_2}\left(1 - e^{(b-r)(T-t)} N\left(d_1(S^*)\right)\right) = 0$$

This is solved using Newton's algorithm for finding the root. We start by finding a first "seed" value S_0. The next estimate of S_i is found by:

$$S_{i+1} = S_i - \frac{g()}{g'}$$

[1] The approximation is also discussed in Hull [2003].

$$C(S,T) = \begin{cases} c(S,T) + A_2 \left(\frac{S}{S^*}\right)^{q_2} & \text{if} \quad S < S^* \\ S - X & \text{if} \quad S \geq S^* \end{cases}$$

where

$$q_2 = \frac{1}{2}\left(-(N-1) + \sqrt{(N-1)^2 + \frac{4M}{K}}\right)$$

$$A_2 = \frac{S^*}{q_2}\left(1 - e^{(b-r)(T-t)} N\left(d_1(S^*)\right)\right)$$

$$M = \frac{2r}{\sigma^2},\ N = \frac{2b}{\sigma^2},\ K(T) = 1 - e^{-r(T-t)}$$

and S^* solves

$$S^* - X = c\left(S^*, T\right) + \frac{S^*}{q_2}\left(1 - e^{(b-r)(T-t)} N\left(d_1(S^*)\right)\right)$$

Formula 12.1: The functional form of the Barone Adesi Whaley approximation to the value of an American call

At each step we need to evaluate $g()$ and its derivative $g'()$.

$$g(S) = S - X - c(S) - \frac{1}{q_2} S\left(1 - e^{(b-r)(T-t)} N(d_1)\right)$$

$$g'(S) = (1 - \frac{1}{q_2})\left(1 - e^{(b-r)(T-t)} N(d_1)\right) + \frac{1}{q_2}(e^{(b-r)(T-t)} n(d_1)) \frac{1}{\sigma\sqrt{T-t}}$$

where $c(S)$ is the Black Scholes value for commodities. Code 12.1 shows the implementation of this formula for the price of a call option.

Exercise 12.

The Barone-Adesi – Whaley insight can also be used to value a put option, by approximating the value of the early exercise premium. For a put option the approximation is

$$P(S) = \begin{matrix} p(S,T) + A_1 \left(\frac{S}{S^{**}}\right)^{q_1} & \text{if} \quad S > S^{**} \\ X - S & \text{if} \quad S \leq S^{**} \end{matrix}$$

$$A_1 = -\frac{S^{**}}{q1}(1 - e^{(b-r)(T-t)} N(-d_1(S^{**}))$$

One again solves iteratively for S^{**}, for example by Newton's procedure, where now one would use

$$g(S) = X - S - p(S) + \frac{S}{q_1}\left(1 - e^{(b-r)(T-t)} N(-d_1)\right)$$

$$g'(S) = (\frac{1}{q_1} - 1)\left(1 - e^{(b-r)(T-t)} N(-d_1)\right) + \frac{1}{q_1} e^{(b-r)(T-t)} \frac{1}{\sigma\sqrt{T-t}} n(-d_1)$$

1. Implement the calculation of the price of an American put option using the BAW approach.

```
#include <cmath>
#include <algorithm>
using namespace std;
#include "normdist.h"         // normal distribution
#include "fin_recipes.h"   // define other option pricing formulas

const double ACCURACY=1.0e-6;

double option_price_american_call_approximated_baw( const double& S,
                                                    const double& X,
                                                    const double& r,
                                                    const double& b,
                                                    const double& sigma,
                                                    const double& time) {
    double sigma_sqr = sigma*sigma;
    double time_sqrt = sqrt(time);
    double nn = 2.0*b/sigma_sqr;
    double m = 2.0*r/sigma_sqr;
    double K = 1.0-exp(-r*time);
    double q2 = (-(nn-1)+sqrt(pow((nn-1),2.0)+(4*m/K)))*0.5;

    double q2_inf = 0.5 * ( -(nn-1) + sqrt(pow((nn-1),2.0)+4.0*m)); // seed value from paper
    double S_star_inf = X / (1.0 - 1.0/q2_inf);
    double h2 = -(b*time+2.0*sigma*time_sqrt)*(X/(S_star_inf-X));
    double S_seed = X + (S_star_inf-X)*(1.0-exp(h2));

    int no_iterations=0; // iterate on S to find S_star, using Newton steps
    double Si=S_seed;
    double g=1;
    double gprime=1.0;
    while ((fabs(g) > ACCURACY)
           && (fabs(gprime)>ACCURACY) // to avoid exploding Newton's
           && ( no_iterations++<500)
           && (Si>0.0)) {
        double c = option_price_european_call_payout(Si,X,r,b,sigma,time);
        double d1 = (log(Si/X)+(b+0.5*sigma_sqr)*time)/(sigma*time_sqrt);
        g=(1.0-1.0/q2)*Si-X-c+(1.0/q2)*Si*exp((b-r)*time)*N(d1);
        gprime=( 1.0-1.0/q2)*(1.0-exp((b-r)*time)*N(d1))
            +(1.0/q2)*exp((b-r)*time)*n(d1)*(1.0/(sigma*time_sqrt));
        Si=Si-(g/gprime);
    };
    double S_star = 0;
    if (fabs(g)>ACCURACY) { S_star = S_seed; } // did not converge
    else { S_star = Si; };
    double C=0;
    double c = option_price_european_call_payout(S,X,r,b,sigma,time);
    if (S>=S_star) {
        C=S-X;
    }
    else {
        double d1 = (log(S_star/X)+(b+0.5*sigma_sqr)*time)/(sigma*time_sqrt);
        double A2 = (1.0-exp((b-r)*time)*N(d1))* (S_star/q2);
        C=c+A2*pow((S/S_star),q2);
    };
    return max(C,c); // know value will never be less than BS value
};
```

Code 12.1: Barone Adesi quadratic approximation to the price of a call option

Consider the following set of parameters, used as an example in the Barone-Adesi and Whaley [1987] paper: $S = 100$, $X = 100$, $\sigma = 0.20$, $r = 0.08$, $b = -0.04$. Price a call option with time to maturity of of 3 months. The program

```
void test_baw_approximation_call(){
    double S = 100; double X = 100; double sigma = 0.20;
    double r = 0.08; double b = -0.04; double time = 0.25;
    cout << " Call price using Barone-Adesi Whaley approximation = "
         << option_price_american_call_approximated_baw(S,X,r,b,sigma,time) << endl;
};
```

provides the output

```
 Call price using Barone-Adesi Whaley approximation = 5.74339
```

Example 12.1: Example using the BAW approximation

Chapter 13

Average, lookback and other exotic options

Contents

We now look at a type of options that has received a lot of attention in later years. The distinguishing factor of these options is that they depend on the *whole price path* of the underlying security between today and the option maturity.

13.1 Bermudan options

A Bermudan option is, as the name implies,[1] a mix of an European and American option. It is a standard put or call option which can only be exercised at discrete dates throughout the life of the option. The simplest way to do the pricing of this is again the binomial approximation, but now, instead of checking at every node whether it is optimal to exercise early, only check at the nodes corresponding to the potential exercise times. Code 13.1 shows the calculation of the Bermudan price using binomial approximations. The times as which exercise can happen is passed as a vector argument to the routine, and in the binomial a list of which nodes exercise can happen is calculated and checked at every step.

[1] Since Bermuda is somewhere between America and Europe...

```
#include <cmath>            // standard C mathematical library
#include <algorithm>        // defines the max() operator
#include <vector>           // STL vector templates
using namespace std;

double option_price_put_bermudan_binomial( const double& S, // spot price
                                           const double& X, // exercice price
                                           const double& r, // interest rate
                                           const double& q, // continous payout
                                           const double& sigma, // volatility
                                           const double& time, // time to maturity
                                           const vector<double>& potential_exercise_times,
                                           const int& steps) { // no steps in binomial tree
    double delta_t=time/steps;
    double R = exp(r*delta_t);       // interest rate for each step
    double Rinv = 1.0/R;              // inverse of interest rate
    double u = exp(sigma*sqrt(delta_t)); // up movement
    double uu = u*u;
    double d = 1.0/u;
    double p_up = (exp((r-q)*delta_t)-d)/(u-d);
    double p_down = 1.0-p_up;
    vector<double> prices(steps+1); // price of underlying
    vector<double> put_values(steps+1); // value of corresponding put

    vector<int> potential_exercise_steps; // create list of steps at which exercise may happen
    for (int i=0;i<potential_exercise_times.size();++i){
        double t = potential_exercise_times[i];
        if ( (t>0.0)&&(t<time) ) {
            potential_exercise_steps.push_back(int(t/delta_t));
        };
    };

    prices[0] = S*pow(d, steps); // fill in the endnodes.
    for (int i=1; i<=steps; ++i) prices[i] = uu*prices[i-1];
    for (int i=0; i<=steps; ++i) put_values[i] = max(0.0, (X-prices[i])); // put payoffs at maturity
    for (int step=steps-1; step>=0; --step) {
        bool check_exercise_this_step=false;
        for (int j=0;j<potential_exercise_steps.size();++j){
            if (step==potential_exercise_steps[j]) { check_exercise_this_step=true; };
        };
        for (int i=0; i<=step; ++i) {
            put_values[i] = (p_up*put_values[i+1]+p_down*put_values[i])*Rinv;
            prices[i] = d*prices[i+1];
            if (check_exercise_this_step) put_values[i] = max(put_values[i],X-prices[i]);
        };
    };
    return put_values[0];
};
```

Code 13.1: Binomial approximation to Bermudan put option

The program

```
void test_bermudan_option(){
    double S=80;     double K=100;     double r = 0.20;
    double time = 1.0; double sigma = 0.25;
    int steps = 500;
    double q=0.0;
    vector<double> potential_exercise_times; potential_exercise_times.push_back(0.25);
    potential_exercise_times.push_back(0.5); potential_exercise_times.push_back(0.75);
    cout << " bermudan put price = "
         << option_price_put_bermudan_binomial(S,K,r,q,sigma,time,potential_exercise_times,steps)
         << endl;
};
```

provides the output

```
bermudan put price = 15.9079
```

13.2 Asian options

The payoff depends on the average of the underlying price. An *average price call* has payoff

$$C_T = \max(0, \bar{S} - X),$$

where $\bar{S}$ is the average of the underlying in the period between t and T.
Another Asian is the *average strike call*

$$C_T = \max(0, S_T - \bar{S})$$

There are different types of Asians depending on how the average $\bar{S}$ is calculated. For the case of S being lognormal and the average $\bar{S}$ being a geometric average, there is an analytic formula due to Kemna and Vorst [1990]. Hull [2003] also discusses this case. It turns out that one can calculate this option using the regular Black Scholes formula adjusting the volatility to $\sigma/\sqrt{3}$ and the dividend yield to

$$\frac{1}{2}\left(r + q + \frac{1}{6}\sigma^2\right)$$

in the case of continous sampling of the underlying price distribution.
Code 13.2 shows the calculation of the analytical price of an Asian geometric average price call.

```
#include <cmath>
using namespace std;
#include "normdist.h" // normal distribution definitions

double
option_price_asian_geometric_average_price_call(const double& S,
                                                const double& X,
                                                const double& r,
                                                const double& q,
                                                const double& sigma,
                                                const double& time){
    double sigma_sqr = pow(sigma,2);
    double adj_div_yield=0.5*(r+q+sigma_sqr);
    double adj_sigma=sigma/sqrt(3.0);
    double adj_sigma_sqr = pow(adj_sigma,2);
    double time_sqrt = sqrt(time);
    double d1 = (log(S/X) + (r-adj_div_yield + 0.5*adj_sigma_sqr)*time)/(adj_sigma*time_sqrt);
    double d2 = d1-(adj_sigma*time_sqrt);
    double call_price = S * exp(-adj_div_yield*time)* N(d1) - X * exp(-r*time) * N(d2);
    return call_price;
};
```

Code 13.2: Analytical price of an Asian geometric average price call

13.3 Lookback options

The payoff from lookback options depend on the maximum or minimum of the underlying achieved through the period. The payoff from the lookback call is the terminal price of the undelying less the minimum value

$$C_T = \max(0, S_T - \min_{\tau\in[t,T]} S_\tau)$$

For this particular option an analytical solution has been found, due to Goldman et al. [1979], which is shown in formula 13.1 and implemented in code 13.3

$$C = Se^{-q(T-t)}N(a_1) - Se^{-q(T-t)}\frac{\sigma^2}{2(r-q)}N(-a_1) - S_{min}e^{-r(T-t)}\left(N(a_2) - \frac{\sigma^2}{2(r-q)}e^{Y_1}N(-a3)\right)$$

$$a_1 = \frac{\ln\left(\frac{S}{S_{min}}\right) + (r - q + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

$$a_2 = a_1 - \sigma\sqrt{T-t}$$

$$a_3 = \frac{\ln\left(\frac{S}{S_{min}}\right) + \left(-r + q + \frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}$$

$$Y_1 = \frac{2\left(r - q - \frac{1}{2}\sigma^2\right)\ln\left(\frac{S}{S_{min}}\right)}{\sigma^2}$$

Formula 13.1: Analytical formula for a lookback call

```
#include <cmath>
using namespace std;
#include "normdist.h"

double option_price_european_lookback_call(const double& S,
                                           const double& Smin,
                                           const double& r,
                                           const double& q,
                                           const double& sigma,
                                           const double& time){
   if (r==q) return 0;
   double sigma_sqr=sigma*sigma;
   double time_sqrt = sqrt(time);
   double a1 = (log(S/Smin) + (r-q+sigma_sqr/2.0)*time)/(sigma*time_sqrt);
   double a2 = a1-sigma*time_sqrt;
   double a3 = (log(S/Smin) + (-r+q+sigma_sqr/2.0)*time)/(sigma*time_sqrt);
   double Y1 = 2.0 * (r-q-sigma_sqr/2.0)*log(S/Smin)/sigma_sqr;
   return S * exp(-q*time)*N(a1)- S*exp(-q*time)*(sigma_sqr/(2.0*(r-q)))*N(-a1)
      - Smin * exp(-r*time)*(N(a2)-(sigma_sqr/(2*(r-q)))*exp(Y1)*N(-a3));
};
```

Code 13.3: Price of lookback call option

13.4 Monte Carlo Pricing of options whose payoff depend on the whole price path

Monte Carlo simulation can be used to price a lot of different options. The limitation is that the options should be European. American options can not be priced by simulation methods. In chapter 11 we looked at a general simulation case where we wrote a generic routine which we passed a payoff function to, and the payoff function was all that was necessary to define an option value. The payoff function in that case was a function of the *terminal* price of the underlying security. The only difference to the previous case is that we now have to generate a price *sequence* and write the terminal payoff of the derivative in terms of that, instead of just generating the terminal value of the underlying security from the lognormal assumption.

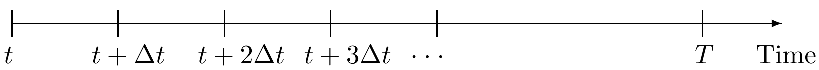
13.4.1 Generating a series of lognormally distributed variables

Recall that one will generate lognormally distributed variables as

$$S_T = S_t e^{\left(r-\frac{1}{2}\sigma^2\right)(T-t)+\sigma\sqrt{T-t}\tilde{x}}$$

where the current time is t and terminal date is T. To simulate a price sequence one splits this period into say N periods, each of length

$$\Delta t = \frac{T-t}{N}$$



Each step in the simulated price sequence is

$$S_{t+\Delta t} = S_t e^{\left(r-\frac{1}{2}\sigma^2\right)\Delta+\sigma\sqrt{\Delta t}\tilde{x}}$$

Code 13.4 shows how one would simulate a sequence of lognormally distributed variables.

```
#include <cmath>
#include <vector>
using namespace std;
#include "normdist.h"

vector<double>
simulate_lognormally_distributed_sequence(const double& S, // current value of underlying
                                          const double& r, // interest rate
                                          const double& sigma, // volatitily
                                          const double& time, // time to final date
                                          const int& no_steps){ // number of steps
    vector<double> prices(no_steps);
    double delta_t = time/no_steps;
    double R = (r-0.5*pow(sigma,2))*delta_t;
    double SD = sigma * sqrt(delta_t);
    double S_t = S;                       // initialize at current price
    for (int i=0; i<no_steps; ++i) {
        S_t = S_t * exp(R + SD * random_normal());
        prices[i]=S_t;
    };
    return prices;
};
```

Code 13.4: Simulating a sequence of lognormally distributed variables

```
#include <cmath>
using namespace std;
#include "fin_recipes.h"

double
derivative_price_simulate_european_option_generic(const double& S, // price of underlying
                                                 const double& X, // used by user provided payoff function
                                                 const double& r, // risk free interest rate
                                                 const double& sigma, // volatility
                                                 const double& time, // time to maturity
                                                 double payoff(const vector<double>& prices,
                                                               const double& X),
                                                 // user provided function
                                                 const int& no_steps, // number of steps in generated price sequence
                                                 const int& no_sims) { // number of simulations to run
    double sum_payoffs=0;
    for (int n=0; n<no_sims; n++) {
        vector<double>prices = simulate_lognormally_distributed_sequence(S,r,sigma,time,no_steps);
        sum_payoffs += payoff(prices,X);
    };
    return exp(-r*time) * (sum_payoffs/no_sims);
};
```

Code 13.5: Generic routine for pricing European options which

This code is then used in the generic routine to do calculations, as shown in code 13.5.

To price an option we are then only in need of a definition of a payoff function. We consider a couple of examples. One is the case of an Asian option, shown in code 13.6.

```
#include <cmath>
#include <numeric>
#include <vector>
using namespace std;

double payoff_arithmetric_average_call(const vector<double>& prices, const double& X) {
    double sum=accumulate(prices.begin(), prices.end(),0.0);
    double avg = sum/prices.size();
    return max(0.0,avg-X);
};

double payoff_geometric_average_call(const vector<double>& prices, const double& X) {
    double logsum=log(prices[0]);
    for (unsigned i=1;i<prices.size();++i){ logsum+=log(prices[i]); };
    double avg = exp(logsum/prices.size());
    return max(0.0,avg-X);
};
```

Code 13.6: Payoff function for Asian call option

Another is the payoff for a lookback, shown in code 13.7

```
#include <vector>
#include <algorithm>
using namespace std;

double payoff_lookback_call(const vector<double>& prices, const double& unused_variable) {
   double m = *min_element(prices.begin(),prices.end());
   return prices.back()−m; // always positive or zero
};

double payoff_lookback_put(const vector<double>& prices, const double& unused_variable) {
    double m = *max_element(prices.begin(),prices.end());
    return m−prices.back(); // max is always larger or equal.
};
```

Code 13.7: Payoff function for lookback option

13.5 Control variate

As discussed in chapter 11, a control variate is a price which we both have an analytical solution of and find the Monte Carlo price of. The differences between these two prices is a measure of the bias in the Monte Carlo estimate, and is used to adjust the Monte Carlo estimate of other derivatives priced using the same random sequence.

Code 13.8 shows the Black Scholes price used as a control variate. An alternative could have been the analytical lookback price, or the analytical solution for a geometric average price call shown earlier.

```
#include "fin_recipes.h"
#include <cmath>
using namespace std;

double
derivative_price_simulate_european_option_generic_with_control_variate(const double& S,
                                                                        const double& X,
                                                                        const double& r,
                                                                        const double& sigma,
                                                                        const double& time,
                                                                        double payoff(const vector<double>& prices,
                                                                                      const double& X),
                                                                        const int& no_steps,
                                                                        const int& no_sims) {
    double c_bs = option_price_call_black_scholes(S,S,r,sigma,time);// price an at the money Black Scholes call
    double sum_payoffs=0;
    double sum_payoffs_bs=0;
    for (int n=0; n<no_sims; n++) {
        vector<double> prices = simulate_lognormally_distributed_sequence(S,r,sigma,time, no_steps);
        double S1= prices.back();
        sum_payoffs += payoff(prices,X);
        sum_payoffs_bs += payoff_call(S1,S); // simulate at the money Black Scholes price
    };
    double c_sim = exp(-r*time) * (sum_payoffs/no_sims);
    double c_bs_sim = exp(-r*time) * (sum_payoffs_bs/no_sims);
    c_sim += (c_bs-c_bs_sim);
    return c_sim;
};
```

Code 13.8: Control Variate

References Exotic options are covered in Hull [2003]. Rubinstein [1993] has an extensive discussion of analytical solutions to various exotic options.

Chapter 14

Alternatives to the Black Scholes type option formula

Contents

A large number of alternative formulations to the Black Scholes analysis has been proposed. Very few of them have seen any widespread use, but we will look at some of these alternatives.

14.1 Merton's Jump diffusion model.

Merton has proposed a model where in addition to a Brownian Motion term, the price process of the underlying is allowed to have *jumps*. The risk of these jumps is assumed to not be priced.

In the following we look at an implementation of a special case of Merton's model, described in [Hull, 1993, pg 454], where the size of the jump has a normal distribution. λ and κ are parameters of the jump distribution. The price of an European call option is

$$c = \sum_{n=0}^{\infty} \frac{e^{\lambda'\tau}(\lambda'\tau)^n}{n!} C_{BS}(S, X, r_n, \sigma_n^2, T-t)$$

where

$$\tau = T - t$$

$$\lambda' = \lambda(1+\kappa)$$

$C_{BS}(\cdot)$ is the Black Scholes formula, and

$$\sigma_n^2 = \sigma^2 + \frac{n\delta^2}{\tau}$$

$$r_n = r - \lambda\kappa + \frac{n\ln(1+\kappa)}{\tau}$$

In implementing this formula, we need to terminate the infinite sum at some point. But since the factorial function is growing at a much higher rate than any other, that is no problem, terminating at about $n = 50$ should be on the conservative side. To avoid numerical difficulties, use the following method for calculation of

$$\frac{e^{\lambda'\tau}(\lambda'\tau)^n}{n!} = exp\left(\ln\left(\frac{e^{\lambda'\tau}(\lambda'\tau)^n}{n!}\right)\right) = exp\left(-\lambda'\tau + n\ln(\lambda'\tau) - \sum_{i=1}^{n}\ln i\right)$$

```
#include <cmath>
#include "fin_recipes.h"

double option_price_call_merton_jump_diffusion( const double& S,
                                                const double& X,
                                                const double& r,
                                                const double& sigma,
                                                const double& time_to_maturity,
                                                const double& lambda,
                                                const double& kappa,
                                                const double& delta) {
    const int MAXN=50;
    double tau=time_to_maturity;
    double sigma_sqr = sigma*sigma;
    double delta_sqr = delta*delta;
    double lambdaprime = lambda * (1+kappa);
    double gamma = log(1+kappa);
    double c = exp(-lambdaprime*tau)*option_price_call_black_scholes(S,X,r-lambda*kappa,sigma,tau);
    double log_n = 0;
    for (int n=1;n<=MAXN; ++n) {
        log_n += log(double(n));
        double sigma_n = sqrt( sigma_sqr+n*delta_sqr/tau );
        double r_n = r-lambda*kappa+n*gamma/tau;
        c += exp(-lambdaprime*tau+n*log(lambdaprime*tau)-log_n)*
            option_price_call_black_scholes(S,X,r_n,sigma_n,tau);
    };
    return c;
};
```

Code 14.1: Mertons jump diffusion formula

The program

```
#include <cmath>
#include "fin_recipes.h"
void test_merton_jump_diff_call(){
    double S=100;
    double K=100;
    double r=0.05;
    double sigma=0.3;
    double time_to_maturity=1;
    double lambda=0.5;
    double kappa=0.5;
    double delta=0.5;
    cout << " Merton Jump diffusion call = "
         << option_price_call_merton_jump_diffusion(S,K,r,sigma,time_to_maturity,lambda,kappa,delta)
         << endl;
};
```

provides the output

```
Merton Jump diffusion call = 23.2074
```

Example 14.1: Mertons Jump diffusion formula

Chapter 15

Using a library for matrix algebra

Contents

What really distinguishes C++ from standard `C` is the ability to *extend* the language by creating classes and collecting these classes into libraries. A library is a collection of classes and routines for one particular purpose. We have already seen this idea when creating the `date` and `term_structure` classes. However, one should not necessarily always go ahead and create such classes from scratch. It is just as well to use somebody else class, as long as it is correct and well documented and fulfills a particular purpose.

15.1 An example matrix class

Use `Newmat` as an example matrix class.

15.2 Finite Differences

We use the case of implicit finite difference calculations to illustrate matrix calculations in action.

The method of choice for any engineer given a differential equation to solve is to numerically approximate it using a finite difference scheme, which is to approximate the continous differential equation with a discrete *difference* equation, and solve this difference equation.

In the following we implement the *implicit finite differences*.

Explicit finite differences was discussed earlier, we postponed the implicit case to now because it is much simplified by a matrix library.

15.3 European Options

For European options we do not need to use the finite difference scheme, but we show how one would find the european price for comparison purposes.

15.4 American Options

We now compare the American versions of the same algoritms, the only difference being the check for exercise at each point.

```cpp
#include <cmath>
#include "newmat.h" // definitions for newmat matrix library
using namespace NEWMAT;

#include <vector> // standard STL vector template
#include <algorithm>
using namespace std;

double option_price_put_european_finite_diff_implicit(double S,
                                                       double X,
                                                       double r,
                                                       double sigma,
                                                       double time,
                                                       int no_S_steps,
                                                       int no_t_steps) {
    double sigma_sqr = sigma*sigma;
    // need no_S_steps to be even:
    int M; if ((no_S_steps%2)==1) { M=no_S_steps+1; } else { M=no_S_steps; };
    double delta_S = 2.0*S/M;
    vector<double> S_values(M+1);
    for (int m=0;m<=M;m++) { S_values[m] = m*delta_S; };
    int N=no_t_steps;
    double delta_t = time/N;

    BandMatrix A(M+1,1,1); A=0.0;
    A.element(0,0) = 1.0;
    for (int j=1;j<M;++j) {
        A.element(j,j-1) = 0.5*j*delta_t*(r-sigma_sqr*j); // a[j]
        A.element(j,j)  = 1.0 + delta_t*(r+sigma_sqr*j*j); // b[j];
        A.element(j,j+1) = 0.5*j*delta_t*(-r-sigma_sqr*j); // c[j];
    };
    A.element(M,M)=1.0;
    ColumnVector B(M+1);
    for (int m=0;m<=M;++m){ B.element(m) = max(0.0,X-S_values[m]); };
    ColumnVector F=A.i()*B;
    for(int t=N-1;t>0;--t) {
        B = F;
        F = A.i()*B;
    };
    return F.element(M/2);
};
```

Code 15.1:

```
#include <cmath>
#include "newmat.h" // definitions for newmat matrix library
using namespace NEWMAT;

#include <vector>
#include <algorithm>
using namespace std;

double option_price_put_american_finite_diff_implicit( double S, double X,
                                                       double r, double sigma, double time,
                                                       int no_S_steps, int no_t_steps) {
    double sigma_sqr = sigma*sigma;
    int M; // need no_S_steps to be even:
    if ((no_S_steps%2)==1) { M=no_S_steps+1; } else { M=no_S_steps; };
    double delta_S = 2.0*S/M;
    double S_values[M+1];
    for (int m=0;m<=M;m++) { S_values[m] = m*delta_S; };
    int N=no_t_steps;
    double delta_t = time/N;

    BandMatrix A(M+1,1,1); A=0.0;
    A.element(0,0) = 1.0;
    for (int j=1;j<M;++j) {
        A.element(j,j-1) = 0.5*j*delta_t*(r-sigma_sqr*j); // a[j]
        A.element(j,j)  = 1.0 + delta_t*(r+sigma_sqr*j*j); // b[j];
        A.element(j,j+1) = 0.5*j*delta_t*(-r-sigma_sqr*j); // c[j];
    };
    A.element(M,M)=1.0;
    ColumnVector B(M+1);
    for (int m=0;m<=M;++m){ B.element(m) = max(0.0,X-S_values[m]); };
    ColumnVector F=A.i()*B;
    for(int t=N-1;t>0;--t) {
        B = F;
        F = A.i()*B;
        for (int m=1;m<M;++m) {  // now check for exercise
            F.element(m) = max(F.element(m), X-S_values[m]);
        };
    };
    return F.element(M/2);
};
```

Code 15.2:

Chapter 16

Mean Variance Analysis.

Contents

We now finally encounter a classical topic in finance, mean variance analysis. This has had to wait because we needed the tool of a linear algebra class before dealing with this.

16.1 Introduction.

The mean variance choice is one of the oldest finance areas, dating back to work of Markowitz. The basic assumption is that risk is measured by variance, and that the decision criterion should be to minimize variance given expected return, or to maximize expected return for a given variance.

Mean variance analysis is very simple when expressed in vector format.

Let

$$\mathbf{e} = E\begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

be the expected return for the n assets, and

$$\mathbf{V} = \begin{bmatrix} \sigma_{11} & \dots & \sigma_{1n} \\ & \vdots & \\ \sigma_{n1} & \dots & \sigma_{nn} \end{bmatrix}$$

be the covariance matrix.

$$\sigma_{ij} = \mathrm{cov}(r_i, r_j)$$

A portfolio of assets is expressed as

$$\omega = \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_n \end{bmatrix}$$

To find the expected return of a portfolio:

$$E[r_p] = \omega'\mathbf{e}$$

and the variance of a portfolio:

$$\sigma_p = \omega'\mathbf{V}\omega$$

```
#include "newmat.h"
#include <cmath>
using namespace std;
using namespace NEWMAT;

double mv_calculate_mean(const Matrix& e, const Matrix& w){
    Matrix tmp = e.t()*w;
    return tmp.element(0,0);
};

double mv_calculate_variance(const Matrix& V, const Matrix& w){
    Matrix tmp = w.t()*V*w;
    return tmp.element(0,0);
};

double mv_calculate_st_dev(const Matrix& V, const Matrix& w){
    double var = mv_calculate_variance(V,w);
    return sqrt(var);
};
```

Code 16.1:

16.2 Mean variance portfolios.

In the case where there are no short sales constraints, the minimum variance portfolio for any given expected return has an analytical solution and is therefore easy to generate.
The portfolio given the expected return $E[r_p]$ is found as

$$\omega_p = \mathbf{g} + \mathbf{h}E[r_p]$$

For the mathematics of generating the unconstrained MV frontier, see chapter 3 of Huang and Litzenberger [1988].

```
// file mv_calc_port_unconstrained.cc
// author: Bernt A Oedegaard

#include "newmat.h"
using namespace NEWMAT;

ReturnMatrix mv_calculate_portfolio_given_mean_unconstrained(const Matrix& e,
                                                              const Matrix& V,
                                                              const double& r){
    int no_assets=e.Nrows();
    Matrix ones = Matrix(no_assets,1); for (int i=0;i<no_assets;++i){ ones.element(i,0) = 1; };
    Matrix Vinv = V.i(); // inverse of V
    Matrix A = (ones.t()*Vinv*e); double a = A.element(0,0);
    Matrix B = e.t()*Vinv*e;   double b = B.element(0,0);
    Matrix C = ones.t()*Vinv*ones; double c = C.element(0,0);
    Matrix D = B*C - A*A;  double d = D.element(0,0);
    Matrix Vinv1=Vinv*ones;
    Matrix Vinve=Vinv*e;
    Matrix g = (Vinv1*b - Vinve*a)*(1.0/d);
    Matrix h = (Vinve*c - Vinv1*a)*(1.0/d);
    Matrix w = g + h*r;
    w.Release();
    return w;
};
```

Code 16.2:

16.3 Short sales constraints

In real applications, it is often not possible to sell assets short. In that case we need to add a constraint that all portfolio weights shall be zero or above.

When constraining the short sales, we need to solve a quadratic program.

$$\min \omega' \mathbf{V} \omega$$

subject to

$$\omega' \mathbf{1} = 1$$

$$\omega' \mathbf{e} = E[r_p]$$

$$\omega_i \in [0, 1] \quad \forall \; i$$

Chapter 17

Pricing of bond options, basic models

Contents

The area of fixed income securities is one where a lot of work is being done in creating advanced mathematical models for pricing of financial securities, in particular fixed income derivatives. The focus of the modelling in this area is on modelling the term structure of interest rates and its evolution over time, which is then used to price both bonds and fixed income derivatives. However, in some cases one does not need the machinery of term structure modelling which we'll look at in later chapters, and price derivatives by modelling the evolution of the bond price directly.

Specifically, suppose that the price of a Bond follows a Geometric Brownian Motion process, just like the case we have studied before. This is not a particularly realistic assumption for the long term behaviour of bond prices, since any bond price converges to the bond face value at the maturity of the bond. The Geometric Brownian motion may be OK for the case of short term options on long term bonds.

17.1 Black Scholes bond option pricing

Given the assumed Brownian Motion process, prices of European Bond Options can be found using the usual Black Scholes formula, as shown in code 17.1 for a zero coupon bond and code 17.2 for the case of an option on a coupon bond.

```
#include <cmath>
#include "normdist.h"

double bond_option_price_put_zero_black_scholes(const double& B,
                                                const double& X,
                                                const double& r,
                                                const double& sigma,
                                                const double& time){
    double time_sqrt = sqrt(time);
    double d1 = (log(B/X)+r*time)/(sigma*time_sqrt) + 0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    double p = X * exp(-r*time) * N(-d2) - B * N(-d1);
    return p;
};
```

Code 17.1: Black scholes price for European call option on zero coupon bond

```
#include <cmath>
#include <vector>
using namespace std;
#include "normdist.h"
#include "fin_recipes.h"

double bond_option_price_put_coupon_bond_black_scholes( const double& B,
                                                        const double& X,
                                                        const double& r,
                                                        const double& sigma,
                                                        const double& time,
                                                        const vector<double> coupon_times,
                                                        const vector<double> coupon_amounts){
    double adjusted_B=B;
    for (unsigned int i=0;i<coupon_times.size();i++) {
        if (coupon_times[i]<=time) {
            adjusted_B -= coupon_amounts[i] * exp(-r*coupon_times[i]);
        };
    };
    return bond_option_price_put_zero_black_scholes(adjusted_B,X,r,sigma,time);
};
```

Code 17.2: Black scholes price for European call option on coupon bond

17.2 Binomial bond option pricing

Since we are in the case of geometric Brownian motion, the usual binomial approximation can be used to price American options, where the bond is the underlying security. Code 17.3 shows the calculation of a put price

```
#include <cmath>        // standard mathematical library
#include <algorithm>        // defining the max() operator
#include <vector>       // STL vector templates
using namespace std;

double bond_option_price_put_american_binomial( const double& B, // Bond price
                                                const double& K, // exercise price
                                                const double& r, // interest rate
                                                const double& sigma, // volatility
                                                const double& t, // time to maturity
                                                const int& steps){ // no steps in binomial tree
   double R = exp(r*(t/steps));        // interest rate for each step
   double Rinv = 1.0/R;                // inverse of interest rate
   double u = exp(sigma*sqrt(t/steps)); // up movement
   double uu = u*u;
   double d = 1.0/u;
   double p_up = (R-d)/(u-d);
   double p_down = 1.0-p_up;
   vector<double> prices(steps+1); // price of underlying
   vector<double> put_values(steps+1); // value of corresponding put

   prices[0] = B*pow(d, steps); // fill in the endnodes.
   for (int i=1; i<=steps; ++i) prices[i] = uu*prices[i-1];
   for (int i=0; i<=steps; ++i) put_values[i] = max(0.0, (K-prices[i])); // put payoffs at maturity
   for (int step=steps-1; step>=0; --step) {
      for (int i=0; i<=step; ++i) {
         put_values[i] = (p_up*put_values[i+1]+p_down*put_values[i])*Rinv;
         prices[i] = d*prices[i+1];
         put_values[i] = max(put_values[i],(K-prices[i]));   // check for exercise
      };
   };
   return put_values[0];
};
```

Code 17.3: Binomial approximation to american bond option price

The program

```
void test_bond_option_gbm_pricing(){
    double B=100;
    double K=100;
    double r=0.05;
    double sigma=0.1;
    double time=1;
    cout << " zero coupon put option price = "
         << bond_option_price_put_zero_black_scholes(B,K,r,sigma,time) << endl;

    vector<double> coupon_times; coupon_times.push_back(0.5);
    vector<double> coupons; coupons.push_back(1);
    cout << " coupon bond put option price = "
         << bond_option_price_put_coupon_bond_black_scholes(B,K,r,sigma,time,coupon_times,coupons);
    cout << endl;

    int steps=100;
    cout << " zero coupon american put option price, binomial = "
         << bond_option_price_put_american_binomial(B,K,r,sigma,time,steps) << endl;
};
```

provides the output

```
 zero coupon put option price = 1.92791
 coupon bond put option price = 2.22852
 zero coupon american put option price, binomial = 2.43282
```

Chapter 18

Credit risk

18.1 The Merton Model

Chapter 19

Term Structure Models

Contents

We now expand on the analysis of the term structure in chapter 3. As shown there, the term structure is best viewed as an abstract function of term to maturity, equally well described by the prices of zero coupon bonds (discount factors), yield on zero coupon bonds (spot rates) or forward rates. In the earlier case we considered two particular implementations of the term structure: A flat term structure or a term structure estimated by linear interpolations of the spot rates.

A number of alternative ways of estimating the term structure has been considered. Some are purely used as interpolation functions, while others are fully specified, dynamic term structure models. We show two examples of the first type, the approximating function proposed in Nelson and Siegel [1987] and a cubic spline used by e.g. McCulloch [1971]. We also consider the term structure models of Cox et al. [1985] and Vasicek [1977].

19.1 The Nelson Siegel term structure approximation

Proposed by Nelson and Siegel [1987].

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

double term_structure_yield_nelson_siegel(const double& t,
                                          const double& beta0,
                                          const double& beta1,
                                          const double& beta2,
                                          const double& lambda) {
   if (t==0.0) return beta0;
   double tl = t/lambda;
   double r = beta0 + (beta1+beta2) * ((1-exp(-tl))/tl) + beta2 * exp(-tl);
   return r;
};
```

Code 19.1:

19.2 Cubic spline.

Cubic splines are well known for their good interpolation behaviour.

```
#ifndef _TERM_STRUCTURE_CLASS_NELSON_SIEGEL_
#define _TERM_STRUCTURE_CLASS_NELSON_SIEGEL_

#include "term_structure_class.h"

class term_structure_class_nelson_siegel : public term_structure_class {
private:
    double beta0_, beta1_, beta2_, lambda_;
public:
    term_structure_class_nelson_siegel(const double& beta0,
                                       const double& beta1,
                                       const double& beta2,
                                       const double& lambda);
    virtual double yield(const double& T) const;
};

#endif
```

Code 19.2:

```
#include "term_structure_class_nelson_siegel.h"
#include "fin_recipes.h"

term_structure_class_nelson_siegel::term_structure_class_nelson_siegel( const double& b0,
                                                                        const double& b1,
                                                                        const double& b2,
                                                                        const double& l) {
   beta0_ = b0; beta1_ = b1; beta2_ = b2; lambda_ = l;
};

double term_structure_class_nelson_siegel::yield(const double& t) const {
   if (t<=0.0) return beta0_;
   return term_structure_yield_nelson_siegel(t,beta0_,beta1_,beta2_,lambda_);
};
```

Code 19.3:

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

double term_structure_discount_factor_cubic_spline(const double& t,
                                                   const double& b1,
                                                   const double& c1,
                                                   const double& d1,
                                                   const vector<double>& f,
                                                   const vector<double>& knots){
    double d = 1.0 + b1*t + c1*(pow(t,2)) + d1*(pow(t,3));
    for (int i=0;i<knots.size();i++) {
        if (t >= knots[i]) { d += f[i] * (pow((t-knots[i]),3)); }
        else { break; };
    };
    return d;
};
```

Code 19.4:

```
#ifndef _TERM_STRUCTURE_CLASS_CUBIC_SPLINE_
#define _TERM_STRUCTURE_CLASS_CUBIC_SPLINE_

#include "term_structure_class.h"
#include <vector>
using namespace std;

class term_structure_class_cubic_spline : public term_structure_class {
private:
    double b_;
    double c_;
    double d_;
    vector<double> f_;
    vector<double> knots_;
public:
    term_structure_class_cubic_spline(const double& b,
                                      const double& c,
                                      const double& d,
                                      const vector<double>& f,
                                      const vector<double> & knots);
    virtual ~term_structure_class_cubic_spline();
    virtual double discount_factor(const double& T) const;
};

#endif
```

Code 19.5:

```
#include "term_structure_class_cubic_spline.h"
#include "fin_recipes.h"

term_structure_class_cubic_spline::term_structure_class_cubic_spline ( const double& b,
                                                                        const double& c,
                                                                        const double& d,
                                                                        const vector<double>& f,
                                                                        const vector<double>& knots) {
    b_ = b;  c_ = c;  d_ = d;  f_.clear();   knots_.clear();
    if (f.size()!=knots.size()){ return; };
    for (int i=0;i<f.size();++i) {
        f_.push_back(f[i]);
        knots_.push_back(knots[i]);
    };
};

term_structure_class_cubic_spline::~term_structure_class_cubic_spline(){
    f_.clear();
    knots_.clear();
};

double term_structure_class_cubic_spline::discount_factor(const double& T) const {
   return term_structure_discount_factor_cubic_spline(T,b_,c_,d_,f_,knots_);
};
```

Code 19.6:

19.3 Cox Ingersoll Ross.

The Cox et al. [1985] model is the most well–known example of a continous time, general equilibrium model of the term structure.

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

double term_structure_discount_factor_cir(const double& t,
                                          const double& r,
                                          const double& kappa,
                                          const double& lambda,
                                          const double& theta,
                                          const double& sigma){
    double sigma_sqr=pow(sigma,2);
    double gamma = sqrt(pow((kappa+lambda),2)+2.0*sigma_sqr);
    double denum = (gamma+kappa+lambda)*(exp(gamma*t)-1)+2*gamma;
    double p=2*kappa*theta/sigma_sqr;
    double enum1= 2*gamma*exp(0.5*(kappa+lambda+gamma)*t);
    double A = pow((enum1/denum),p);
    double B = (2*(exp(gamma*t)-1))/denum;
    double dfact=A*exp(-B*r);
    return dfact;
};
```

Code 19.7:

```
#ifndef _TERM_STRUCTURE_CLASS_CIR_
#define _TERM_STRUCTURE_CLASS_CIR_

#include "term_structure_class.h"

class term_structure_class_cir : public term_structure_class {
private:
    double r_;           // interest rate
    double kappa_;       // mean reversion parameter
    double lambda_;      // risk aversion
    double theta_;       // long run mean
    double sigma_;       // volatility
public:
    term_structure_class_cir(const double& r,
                             const double& k,
                             const double& l,
                             const double& th,
                             const double& sigma);

    virtual double discount_factor(const double& T) const;
};

#endif
```

Code 19.8:

```
#include "term_structure_class_cir.h"
#include "fin_recipes.h"
#include "term_structure_models.h"

term_structure_class_cir::term_structure_class_cir(const double& r,
                                                   const double& k,
                                                   const double& l,
                                                   const double& th,
                                                   const double& sigma) {
    r_      = r;
    kappa_ = k;
    lambda_ = l;
    theta_ = th;
    sigma_ = sigma;
};

double term_structure_class_cir::discount_factor(const double& T) const{
  return term_structure_discount_factor_cir(T,r_,kappa_,lambda_,theta_,sigma_);
};
```

Code 19.9:

19.4 Vasicek

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

double term_structure_discount_factor_vasicek(const double& time,
                                              const double& r,
                                              const double& a,
                                              const double& b,
                                              const double& sigma){
    double A,B;
    double sigma_sqr = sigma*sigma;
    double aa = a*a;
    if (a==0.0){
        B = time;
        A = exp(sigma_sqr*pow(time,3))/6.0;
    }
    else {
        B = (1.0 - exp(-a*time))/a;
        A = exp( ((B-time)*(aa*b-0.5*sigma_sqr))/aa -((sigma_sqr*B*B)/(4*a)));
    };
    double d = A*exp(-B*r);
    return d;
}
```

Code 19.10:

```
#ifndef _TERM_STRUCTURE_CLASS_VASICEK_
#define _TERM_STRUCTURE_CLASS_VASICEK_

#include "term_structure_class.h"

class term_structure_class_vasicek : public term_structure_class {
private:
    double r_;
    double a_;
    double b_;
    double sigma_;
public:
    term_structure_class_vasicek(const double& r,
                                 const double& a,
                                 const double& b,
                                 const double& sigma);

    virtual double discount_factor(const double& T) const;
};

#endif
```

Code 19.11:

```
#include "term_structure_class_vasicek.h"
#include "fin_recipes.h"

term_structure_class_vasicek::term_structure_class_vasicek(const double& r,
                                                           const double& a,
                                                           const double& b,
                                                           const double& sigma) {
    r_    = r;    a_    = a;    b_    = b;    sigma_ = sigma;
};

double term_structure_class_vasicek::discount_factor(const double& T) const{
    return term_structure_discount_factor_vasicek(T,r_,a_,b_,sigma_);
};
```

Code 19.12:

Chapter 20

Binomial Term Structure models

Contents

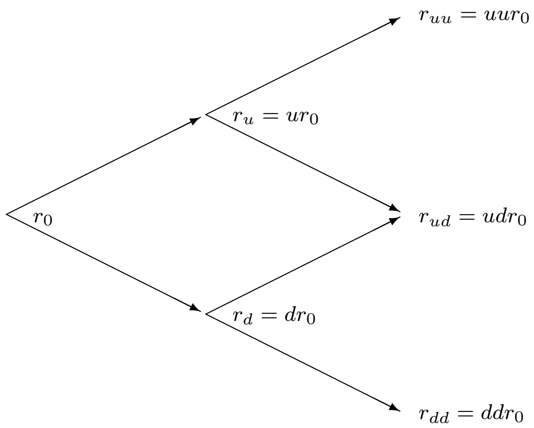
Pricing bond options with the Black Scholes model, or its binomial approximation, as done in chapter 17, does not always get it right. For example, it ignores the fact that at the maturity of the bond, the bond volatility is zero. The bond volatility decreases as one gets closer to the bond maturity. This behaviour is not captured by the assumptions underlying the Black Scholes assumption. We therefore look at more complicated term structure models, the unifying theme of which is that they are built by building *trees* of the interest rate.

20.1 The Rendleman and Bartter model

The Rendleman and Bartter approach to valuation of interest rate contingent claims (see Rendleman and Bartter [1979] and Rendleman and Bartter [1980]) is a particular simple one. Essentially, it is to apply the same binomial approach that is used to approximate options in the Black Scholes world, but the random variable is now the interest rate. This has implications for multiperiod discounting: Taking the present value is now a matter of choosing the correct sequence of spot rates, and it may be necessary to keep track of the whole "tree" of interest rates.

The general idea is to construct a tree as shown in figure 20.1.

Figure 20.1: Interest rate tree



The figure illustrates the building of an interest rate tree of one period spot rates by assuming that for any given period t the next period interest rate can only take on two values, $r_{t+1} = ur_t$ or $r_{t+1} = dr_t$, where u and d are constants. r_0 is the initial spot rate.

Code 20.1 shows how one can construct such an interest rate tree.

Code 20.2 implements the original algorithm for a call option on a (long maturity) zero coupon bond.

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

vector< vector<double> >
build_interest_rate_tree_rendleman_bartter(const double& r0,
                                           const double& u,
                                           const double& d,
                                           const int& n){
    vector< vector<double> > tree;
    for (int i=1;i<=n;++i){
        vector<double> r(i);
        for (int j=0;j<i;++j){
            r[j] = r0*pow(u,j)*pow(d,i-j-1);
        };
        tree.push_back(r);
    };
    return tree;
};
```

Code 20.1: Building an interest rate tree

20.2 Readings

General references include Sundaresan [2001].

Rendleman and Bartter [1979] and Rendleman and Bartter [1980] are the original references for building standard binomial interest rate trees.

```
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;

double bond_option_price_call_zero_american_rendleman_bartter(const double& X,
                                                              const double& option_maturity,
                                                              const double& S,
                                                              const double& M, // term structure paramters
                                                              const double& interest, // current short interest rate
                                                              const double& bond_maturity, // time to maturity for underlying b
                                                              const double& maturity_payment,
                                                              const int& no_steps) {
    double delta_t = bond_maturity/no_steps;

    double u=exp(S*sqrt(delta_t));
    double d=1/u;
    double p_up = (exp(M*delta_t)-d)/(u-d);
    double p_down = 1.0-p_up;

    vector<double> r(no_steps+1);
    r[0]=interest*pow(d,no_steps);
    double uu=u*u;
    for (int i=1;i<=no_steps;++i){ r[i]=r[i-1]*uu;};
    vector<double> P(no_steps+1);
    for (int i=0;i<=no_steps;++i){ P[i] = maturity_payment; };
    int no_call_steps=int(no_steps*option_maturity/bond_maturity);
    for (int curr_step=no_steps;curr_step>no_call_steps;--curr_step) {
        for (int i=0;i<curr_step;i++) {
            r[i] = r[i]*u;
            P[i] = exp(-r[i]*delta_t)*(p_down*P[i]+p_up*P[i+1]);
        };
    };
    vector<double> C(no_call_steps+1);
    for (int i=0;i<=no_call_steps;++i){ C[i]=max(0.0,P[i]-X); };
    for (int curr_step=no_call_steps;curr_step>=0;--curr_step) {
        for (int i=0;i<curr_step;i++) {
            r[i]   = r[i]*u;
            P[i] = exp(-r[i]*delta_t)*(p_down*P[i]+p_up*P[i+1]);
            C[i]=max(P[i]-X, exp(-r[i]*delta_t)*(p_up*C[i+1]+p_down*C[i]));
        };
    };
    return C[0];
};
```

Code 20.2: RB binomial model for European call on zero coupon bond

Chapter 21

Term Structure Derivatives

Contents

21.1 Vasicek bond option pricing

If the term structure model is Vasicek's model there is a solution for the price of an option on a zero coupon bond, due to Jamshidan [1989].

Under Vacisek's model the process for the short rate is assumed to follow.

$$dr = a(b - r)dt + \sigma dZ$$

where a, b and σ are constants. We have seen earlier how to calculate the discount factor in this case. We now want to consider an European Call option in this setting.

Let $P(t, s)$ be the time t price of a zero coupon bond with a payment of \$1 at time s (the discount factor). The price at time t of a European call option maturing at time T on on a discount bond maturing at time s is(See Jamshidan [1989] and Hull [1993])

$$P(t, s)N(h) - XP(t, T)N(h - \sigma_P)$$

where

$$h = \frac{1}{\sigma_P} \ln \frac{P(t, s)}{P(t, T)X} + \frac{1}{2}\sigma_P$$

$$\sigma_P = v(t, T)B(T, s)$$

$$B(t, T) = \frac{1 - e^{-a(T-t)}}{a}$$

$$v(t, T)^2 = \frac{\sigma^2(1 - e^{-a(T-t)})}{2a}$$

In the case of $a = 0$,

$$v(t, T) = \sigma\sqrt{T - t}$$

$$\sigma_P = \sigma(s - T)\sqrt{T - t}$$

```
#include "normdist.h"
#include "fin_recipes.h"
#include <cmath>

using namespace std;

double bond_option_price_call_zero_vasicek(const double& X, // exercise price
                                           const double& r, // current interest rate
                                           const double& option_time_to_maturity,
                                           const double& bond_time_to_maturity,
                                           const double& a, // parameters
                                           const double& b,
                                           const double& sigma){
    double T_t = option_time_to_maturity;
    double s_t = bond_time_to_maturity;
    double T_s = s_t-T_t;
    double v_t_T;
    double sigma_P;
    if (a==0.0) {
        v_t_T = sigma * sqrt ( T_t ) ;
        sigma_P = sigma*T_s*sqrt(T_t);
    }
    else {
        v_t_T = sqrt (sigma*sigma*(1-exp(-2*a*T_t))/(2*a));
        double B_T_s = (1-exp(-a*T_s))/a;
        sigma_P = v_t_T*B_T_s;
    };
    double h = (1.0/sigma_P) * log (
        term_structure_discount_factor_vasicek(s_t,r,a,b,sigma)/
        (term_structure_discount_factor_vasicek(T_t,r,a,b,sigma)*X) )
        + sigma_P/2.0;
    double c =
        term_structure_discount_factor_vasicek(s_t,r,a,b,sigma)*N(h)
        -X*term_structure_discount_factor_vasicek(T_t,r,a,b,sigma)*N(h-sigma_P);
    return c;
};
```

Code 21.1:

Appendix A

Normal Distribution approximations.

Contents

We will in general not go into detail about more standard numerical problems not connected to finance, there are a number of well known sources for such, but we show the example of calculations involving the normal distribution.

A.1 The normal distribution function

The nurmal distribution function

$$n(x) = e^{-\frac{x^2}{2}}$$

is calculated as

```
#include <cmath> // c library of math functions
using namespace std; // which is part of the standard namespace

// most C compilers define PI, but just in case it doesn't
#ifndef PI
#define PI 3.141592653589793238462643
#endif

double n(double z) { // normal distribution function
    return (1.0/sqrt(2.0*PI))*exp(−0.5*z*z);
};
```

Code A.1: The normal distribution function

A.2 The cumulative normal distribution

The solution of a large number of option pricing formulas are written in terms of the cumulative normal distribution. For a random variable x the cumulative probability is the probability that the outcome is lower than a given value z. To calculate the probability that a normally distubuted random variable with mean 0 and unit variance is less than z, $N(z)$, one have to evaluate the integral

$$\text{Prob}(x \leq z) = N(z) = \int_{-\infty}^{z} n(x)dx = \int_{-\infty}^{z} e^{-\frac{x^2}{2}} dx$$

There is no explicit closed form solution for calculation of this integral, but a large number of well known approximations exists. Abramowiz and Stegun [1964] is a good source for these approximations. The following is probably the most used such approximation, it being pretty accurate and relatively fast. The arguments to the function are assumed normalized to a (0,1) distribution.

```
#include <cmath> // math functions.
using namespace std;

double N(double z) {
    if (z > 6.0) { return 1.0; }; // this guards against overflow
    if (z < -6.0) { return 0.0; };

    double b1 = 0.31938153;
    double b2 = -0.356563782;
    double b3 = 1.781477937;
    double b4 = -1.821255978;
    double b5 = 1.330274429;
    double p = 0.2316419;
    double c2 = 0.3989423;

    double a=fabs(z);
    double t = 1.0/(1.0+a*p);
    double b = c2*exp((-z)*(z/2.0));
    double n = ((((b5*t+b4)*t+b3)*t+b2)*t+b1)*t;
    n = 1.0-b*n;
    if ( z < 0.0 ) n = 1.0 - n;
    return n;
};
```

Code A.2: The cumulative normal

A.3 Multivariate normal

The normal distribution is also defined for several random variables. We then characterise the *vector* of random variables

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

A probability statement about this vector is a joint statement about all elements of the vector.

A.4 Calculating cumulative bivariate normal probabilities

The most used multivariate normal calculation is the bivariate case, where we let x and y be bivariate normally distributed, each with mean 0 and variance 1, and assume the two variables have correlation of ρ. By the definition of correlation $\rho \in [-1, 1]$. The cumulative probability distribution

$$\begin{aligned} P(x < a, y < b) &= N(a, b, \rho) \\ &= \int_{-\infty}^{a} \int_{-\infty}^{b} \frac{1}{2\pi\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2}\frac{x^2 - 2\rho xy + y^2}{1-\rho^2}\right) dxdy \end{aligned}$$

There are several approximations to this integral. We pick one such, discussed in [Hull, 1993, Ch 10], shown in code A.3

```
#include <cmath> // include the standard library mathematics functions
using namespace std; // which are in the standard namespace

double N(double); // define the univariate cumulative normal distribution as a separate function

#ifndef PI
const double PI=3.141592653589793238462643;
#endif

inline double f(double x, double y, double aprime, double bprime, double rho) {
    double r = aprime*(2*x-aprime) + bprime*(2*y-bprime) + 2*rho*(x-aprime)*(y-bprime);
    return exp(r);
};

inline double sgn( double x) { // sign function
    if (x>=0.0) return 1.0;
    return -1.0;
};

double N(double a, double b, double rho) {
    // Numerical approximation to the bivariate normal distribution,
    // as described e.g. in Hulls book
    if ( (a<=0.0) && (b<=0.0) && (rho<=0.0) ) {
        double aprime = a/sqrt(2.0*(1.0-rho*rho));
        double bprime = b/sqrt(2.0*(1.0-rho*rho));
        double A[4]={0.3253030, 0.4211071, 0.1334425, 0.006374323};
        double B[4]={0.1337764, 0.6243247, 1.3425378, 2.2626645 };
        double sum = 0;
        for (int i=0;i<4;i++) {
            for (int j=0; j<4; j++) {
                sum += A[i]*A[j]* f(B[i],B[j],aprime,bprime,rho);
            };
        };
        sum = sum * ( sqrt(1.0-rho*rho)/PI);
        return sum;
    }
    else if ( a * b * rho <= 0.0 ) {
        if ( ( a<=0.0 ) && ( b>=0.0 ) && (rho>=0.0) ) {
            return N(a) - N(a, -b, -rho);
        }
        else if ( (a>=0.0) && (b<=0.0) && (rho>=0.0) ) {
            return N(b) - N(-a, b, -rho);
        }
        else if ( (a>=0.0) && (b>=0.0) && (rho<=0.0) ) {
            return N(a) + N(b) - 1.0 + N(-a, -b, rho);
        };
    }
    else if ( a * b * rho >= 0.0 ) {
        double denum = sqrt(a*a - 2*rho*a*b + b*b);
        double rho1 = ((rho * a - b) * sgn(a))/denum;
        double rho2 = ((rho * b - a) * sgn(b))/denum;
        double delta=(1.0-sgn(a)*sgn(b))/4.0;
        return N(a,0.0,rho1) + N(b,0.0,rho2) - delta;
    };
    return -99.9; // should never get here
};
```

Code A.3: Approximation to the cumulative bivariate normal

If one has more than two correlated variables, the calculation of cumulative probabilites is a nontrivial problem. One common method involves Monte Carlo estimation of the definite integral. We will consider this, but then it is necessary to first consider simulation of random normal variables.

A.5 Simulating random normal numbers

Generation of random numbers is a large topic and is treated at length in such sources as Knuth [1997]. The generated numbers can never be truly random, only "pseudo"-random, they will be generated according to some reproducible algorithm and after a (large) number of random number generations the sequence will start repeating itself. The number of iterations before replication starts is a measure of the quality of a random number generator. For anybody requiring high-quality random number generators the `rand()` function provided by the standard C++ library should be avoided, but for not getting into involved discussion of random number generations we use this function as a basis for the generation of uniformly distributed numbers in the interval $[0, 1)$, as shown in code A.4.

```
#include <cstdlib>

double random_uniform_0_1(void){
    return double(rand())/double(RAND_MAX); // this uses the C library random number generator.
};
```

Code A.4: Pseudorandom numbers from an uniform $[0, 1)$ distribution

These uniformly distributed distributed random variates are used as a basis for the Polar method for normal densities discussed in Knuth [1997] and inplemented as shown in code A.5

```
#include <cmath>
#include <cstdlib>
//using namespace std;

double random_uniform_0_1(void);

double random_normal(void){
    double U1, U2, V1, V2;
    double S=2;
    while (S>=1) {
        U1 = random_uniform_0_1();
        U2 = random_uniform_0_1();
        V1 = 2.0*U1-1.0;
        V2 = 2.0*U2-1.0;
        S = pow(V1,2)+pow(V2,2);
    };
    double X1=V1*sqrt((-2.0*log(S))/S);
    return X1;
};
```

Code A.5: Pseudorandom numbers from a normal $(0, 1)$ distribution

A.6 Cumulative probabilities for general multivariate distributions

When moving beyond

A.7 References

Tong [1990] discusses the multivariate normal distribution, and is a good reference.

A.8 Exercises

Exercise 13.

Replace the `random_uniform` function here by an alternative of higher quality, by looking into what numerical libraries is available on your computing platform, or by downloading a high quality random number generator from such places as `mathlib` or `statlib`.

Appendix B

C++ concepts

This chapter contains a listing of various `C`/C++ concepts and some notes on each of them.

`class`

`const`

`double`

`exp(x)` (C function). Defined in <cmath>. Returns the natural exponent e to the given power x, e^x.

`Indexation` (in vectors and matrices). To access element number `i` in an array `A`, use `A[i-1]`. Well known trap for people coming to `C` from other languages. Present in `C` for historical efficiency reasons. Arrays in `C` were implemented using pointers. When referring to the first element in the array,

`log(x)` (C function). Defined in <cmath>

`namespace`

`standard namespace`

`vector` (C++ container class). Defined in <vector>

Appendix C

Summarizing routine names

In many of the algorithms use is made of *other* routines. To simplify the matter all routines are summarised in one header file, `fin_recipes.h`. This appendix shows this file.

```
// file: fin_recipes.h
// author: Bernt Arne Oedegaard
//   defines all routines in the financial numerical recipes "book"

#include <vector>
using namespace std;

///////// present value ////////////////////////////////////
// discrete coumpounding

double bonds_price_discrete(const vector<double>& cashflow_times,
                            const vector<double>& cashflows,
                            const double& r);

double cash_flow_pv_discrete (const vector<double>& cflow_times,
                              const vector<double>& cflow_amounts,
                              const double& r);
// continous compounding
double cash_flow_pv (const vector<double>& cflow_times,
                     const vector<double>& cflow_amounts,
                     const double& r);

double cash_flow_irr(const vector<double>& cflow_times,
                     const vector<double>& cflow_amounts);

bool cash_flow_unique_irr(const vector<double>& cflow_times,
                          const vector<double>& cflow_amounts);

double bonds_price(const vector<double>& coupon_times,
                   const vector<double>& coupon_amounts,
                   const vector<double>& principal_times,
                   const vector<double>& principal_amounts,
                   const double& r);

double bonds_price(const vector<double>& cashflow_times,
                   const vector<double>& cashflows,
                   const double& r);


double bonds_duration(const vector<double>& cashflow_times,
                      const vector<double>& cashflows,
                      const double& r);

double bonds_yield_to_maturity( const vector<double>& cashflow_times,
                                const vector<double>& cashflow_amounts,
                                const double& bondprice);

double bonds_duration_macaulay(const vector<double>& cashflow_times,
                               const vector<double>& cashflows,
                               const double& bond_price);

double bonds_duration_modified (const vector<double>& cashflow_times,
                                const vector<double>& cashflow_amounts,
                                const double& bond_price,
                                const double& r);
```

```
double bonds_convexity(const vector<double>& cashflow_times,
                       const vector<double>& cashflow_amounts,
                       const double& y);

/// term structure basics

double term_structure_yield_from_discount_factor(const double& dfact, const double& t);
double term_structure_discount_factor_from_yield(const double& r, const double& t);
double term_structure_forward_rate_from_disc_facts(const double& d_t, const double& d_T, const double& T);
double term_structure_forward_rate_from_yields(const double& r_t1, const double& r_T,
                                               const double& t1, const double& T);
double term_structure_yield_linearly_interpolated(const double& time,
                                                  const vector<double>& obs_times,
                                                  const vector<double>& obs_yields);
//// Futures pricing
double futures_price(const double& S, const double& r, const double& time_to_maturity);

/// Binomial option pricing

double option_price_call_european_binomial( const double& S, const double& K, const double& r,
                                            const double& u, const double& d); // one periode binomial

double option_price_call_european_binomial( const double& S, const double& K, const double& r,
                                            const double& u, const double& d, const int& no_periods);
    // multiple periode binomial

vector< vector<double> > binomial_tree(const double& S0,
                                       const double& u,
                                       const double& d,
                                       const int& no_steps);

/// Black Scholes formula //////////////////////////////////////////

double option_price_call_black_scholes(const double& S, const double& K, const double& r,
                                       const double& sigma, const double& time) ;
double option_price_put_black_scholes (const double& S, const double& K, const double& r,
                                       const double& sigma, const double& time) ;

double
option_price_implied_volatility_call_black_scholes_newton( const double& S, const double& K,
                                                           const double& r, const double& time,
                                                           const double& option_price);

double option_price_implied_volatility_call_black_scholes_bisections( const double& S, const double& K,
                                                                      const double& r, const double& time,
                                                                      const double& option_price);
double option_price_delta_call_black_scholes(const double& S, const double& K, const double& r,
                                             const double& sigma, const double& time);
double option_price_delta_put_black_scholes (const double& S, const double& K, const double& r,
                                             const double& sigma, const double& time);
void option_price_partials_call_black_scholes(const double& S, const double& K, const double& r,
                                              const double& sigma, const double& time,
                                              double& Delta, double& Gamma, double& Theta,
                                              double& Vega, double& Rho);
void option_price_partials_put_black_scholes(const double& S, const double& K, const double& r,
                                             const double& sigma, const double& time,
                                             double& Delta, double& Gamma, double& Theta,
                                             double& Vega, double& Rho);

/// warrant price
double warrant_price_adjusted_black_scholes(const double& S, const double& K,
                                            const double& r, const double& sigma,
                                            const double& time,
                                            const double& no_warrants_outstanding,
                                            const double& no_shares_outstanding);
```

```
double warrant_price_adjusted_black_scholes(const double& S, const double& K,
                                            const double& r, const double& q,
                                            const double& sigma, const double& time,
                                            const double& no_warrants_outstanding,
                                            const double& no_shares_outstanding);

/// Extensions of the Black Scholes model //////////////

double option_price_european_call_payout(const double& S, const double& K, const double& r,
                                         const double& b, const double& sigma, const double& time);
double option_price_european_put_payout (const double& S, const double& K, const double& r,
                                         const double& b, const double& sigma, const double& time);
double option_price_european_call_dividends(const double& S, const double& K, const double& r,
                                            const double& sigma, const double& time,
                                            const vector<double>& dividend_times,
                                            const vector<double>& dividend_amounts );
double option_price_european_put_dividends( const double& S, const double& K, const double& r,
                                            const double& sigma,const double& time,
                                            const vector<double>& dividend_times,
                                            const vector<double>& dividend_amounts);
double option_price_american_call_one_dividend(const double& S, const double& K, const double& r,
                                               const double& sigma,
                                               const double& tau, const double& D1, const double& tau1);
double futures_option_price_call_european_black(const double& F, const double& K, const double& r,
                                                const double& sigma, const double& time);
double futures_option_price_put_european_black(const double& F, const double& K, const double& r,
                                               const double& sigma, const double& time);
double currency_option_price_call_european(const double& S, const double& K, const double& r,
                                           const double& r_f, const double& sigma, const double& time);
double currency_option_price_put_european(const double& S, const double& K, const double& r,
                                          const double& r_f, const double& sigma, const double& time);
double option_price_american_perpetual_call(const double& S, const double& K, const double& r,
                                            const double& q, const double& sigma);
double option_price_american_perpetual_put(const double& S, const double& K, const double& r,
                                           const double& q, const double& sigma);

// binomial option approximation ////////////////

double option_price_call_european_binomial(const double& S, const double& K, const double& r,
                                           const double& sigma, const double& t, const int& steps);
double option_price_put_european_binomial (const double& S, const double& K, const double& r,
                                           const double& sigma, const double& t, const int& steps);
double option_price_call_american_binomial(const double& S, const double& K, const double& r,
                                           const double& sigma, const double& t, const int& steps);
double option_price_put_american_binomial (const double& S, const double& K, const double& r,
                                           const double& sigma, const double& t, const int& steps);
double option_price_call_american_binomial(const double& S, const double& K,
                                           const double& r, const double& y,
                                           const double& sigma, const double& t, const int& steps);
double option_price_put_american_binomial (const double& S, const double& K, const double& r,
                                           const double& y, const double& sigma,
                                           const double& t, const int& steps);

double option_price_call_american_discrete_dividends_binomial( const double& S, const double& K,
                                                               const double& r,
                                                               const double& sigma, const double& t,
                                                               const int& steps,
                                                               const vector<double>& dividend_times,
                                                               const vector<double>& dividend_amounts);

double option_price_put_american_discrete_dividends_binomial(const double& S, const double& K,
                                                             const double& r,
                                                             const double& sigma, const double& t,
                                                             const int& steps,
                                                             const vector<double>& dividend_times,
                                                             const vector<double>& dividend_amounts);
```

```
double option_price_call_american_proportional_dividends_binomial(const double& S, const double& K,
                                                                   const double& r, const double& sigma,
                                                                   const double& time, const int& no_steps,
                                                                   const vector<double>& dividend_times,
                                                                   const vector<double>& dividend_yields);

double option_price_put_american_proportional_dividends_binomial( const double& S, const double& K, const double& r,
                                                                  const double& sigma, const double& time, const int& no_steps,
                                                                  const vector<double>& dividend_times,
                                                                  const vector<double>& dividend_yields);

double option_price_delta_american_call_binomial(const double& S, const double& K, const double& r,
                                                 const double& sigma, const double& t, const int& no_steps);
double option_price_delta_american_put_binomial(const double& S, const double& K, const double& r,
                                                const double& sigma, const double& t, const int& no_steps);
void option_price_partials_american_call_binomial(const double& S, const double& K, const double& r,
                                                  const double& sigma, const double& time, const int& no_steps,
                                                  double& delta, double& gamma, double& theta,
                                                  double& vega, double& rho);

void option_price_partials_american_put_binomial(const double& S, const double& K, const double& r,
                                                 const double& sigma, const double& time, const int& no_steps,
                                                 double& delta, double& gamma, double& theta,
                                                 double& vega, double& rho);

double futures_option_price_call_american_binomial(const double& F, const double& K, const double& r, const double& sigma,
                                                   const double& time, const int& no_steps);

double futures_option_price_put_american_binomial( const double& F, const double& K, const double& r, const double& sigma,
                                                   const double& time, const int& no_steps);

double currency_option_price_call_american_binomial( const double& S, const double& K, const double& r, const double& r_f,
                                                     const double& sigma, const double& t, const int& n);

double currency_option_price_put_american_binomial( const double& S, const double& K, const double& r, const double& r_f,
                                                    const double& sigma, const double& t, const int& n);

//////////////////// finite differences //////////////////

double option_price_call_american_finite_diff_explicit( const double& S, const double& K, const double& r,
                                                        const double& sigma, const double& time,
                                                        const int& no_S_steps, const int& no_t_steps);

double option_price_put_american_finite_diff_explicit( const double& S, const double& K, const double& r,
                                                       const double& sigma, const double& time,
                                                       const int& no_S_steps, const int& no_t_steps);

double option_price_call_european_finite_diff_explicit( const double& S, const double& K, const double& r,
                                                        const double& sigma, const double& time,
                                                        const int& no_S_steps, const int& no_t_steps);

double option_price_put_european_finite_diff_explicit( const double& S, const double& K, const double& r,
                                                       const double& sigma, const double& time,
                                                       const int& no_S_steps, const int& no_t_steps);

double option_price_call_american_finite_diff_implicit( const double& S, const double& K, const double& r,
                                                        const double& sigma, const double& time,
                                                        const int& no_S_steps, const int& no_t_steps);

double option_price_put_american_finite_diff_implicit( const double& S, const double& K, const double& r,
                                                       const double& sigma, const double& time,
                                                       const int& no_S_steps, const int& no_t_steps);

double option_price_call_european_finite_diff_implicit( const double& S, const double& K, const double& r,
                                                        const double& sigma, const double& time,
                                                        const int& no_S_steps, const int& no_t_steps);
```

```
double option_price_put_european_finite_diff_implicit( const double& S, const double& K, const double& r,
                                                      const double& sigma, const double& time,
                                                      const int& no_S_steps, const int& no_t_steps);

///////////////////////// simulated option prices ////////////////////////////////////
// Payoff only function of terminal price
double option_price_call_european_simulated(const double& S, const double& K,
                                            const double& r, const double& sigma,
                                            const double& time_to_maturity, const int& no_sims);
double option_price_put_european_simulated(const double& S, const double& K,
                                           const double& r, const double& sigma,
                                           const double& time_to_maturity, const int& no_sims);
double option_price_delta_call_european_simulated(const double& S, const double& K,
                                                  const double& r, const double& sigma,
                                                  const double& time_to_maturity, const int& no_sims);
double option_price_delta_put_european_simulated(const double& S, const double& K,
                                                 const double& r, const double& sigma,
                                                 const double& time_to_maturity, const int& no_sims);
double simulate_lognormal_random_variable(const double& S, const double& r, const double& sigma,
                                          const double& time);

double
derivative_price_simulate_european_option_generic( const double& S, const double& K,
                                                   const double& r, const double& sigma,
                                                   const double& time,
                                                   double payoff(const double& price, const double& K),
                                                   const int& no_sims);
double
derivative_price_simulate_european_option_generic_with_control_variate(const double& S, const double& K,
                                                                       const double& r,const double& sigma,
                                                                       const double& time,
                                                                       double payoff(const double& price,
                                                                                     const double& K),
                                                                       const int& no_sims);

double
derivative_price_simulate_european_option_generic_with_antithetic_variate(const double& S, const double& K,
                                                                          const double& r,
                                                                          const double& sigma,
                                                                          const double& time,
                                                                          double payoff(const double&price,
                                                                                        const double& K),
                                                                          const int& no_sims);
/////////////////////////////
// payoffs of various options, to be used as function arguments in above simulations
double payoff_call(const double& price, const double& K);
double payoff_put (const double& price, const double& K);
double payoff_cash_or_nothing_call(const double& price, const double& K);
double payoff_asset_or_nothing_call(const double& price, const double& K);

/////////// approximated option prices ////////////////////////

double option_price_american_call_approximated_baw(const double& S, const double& K,
                                                   const double& r, const double& b,
                                                   const double& sigma, const double& time);
double option_price_american_put_approximated_baw(const double& S, const double& K,
                                                  const double& r, const double& b,
                                                  const double& sigma, const double& time);


////////////// path dependent and other exotic options ////////////////////////////////

double option_price_call_bermudan_binomial(const double& S, const double& X, const double& r,
                                           const double& q, const double& sigma, const double& time,
                                           const vector<double>& potential_exercise_times,
                                           const int& steps);
```

```
double option_price_put_bermudan_binomial( const double& S, const double& X, const double& r,
                                           const double& q, const double& sigma, const double& time,
                                           const vector<double>& potential_exercise_times,
                                           const int& steps);

double option_price_european_lookback_call(const double& S, const double& Smin, const double& r,
                                           const double& q, const double& sigma, const double& time);

double option_price_european_lookback_put(const double& S, const double& Smin, const double& r,
                                          const double& q, const double& sigma, const double& time);


double
option_price_asian_geometric_average_price_call(const double& S, const double& K, const double& r,
                                                const double& q, const double& sigma, const double& time);

vector<double> simulate_lognormally_distributed_sequence(const double& S, const double& r,
                                                         const double& sigma, const double& time, const int& no_steps);
double
derivative_price_simulate_european_option_generic( const double& S, const double& K, const double& r,
                                                   const double& sigma, const double& time,
                                                   double payoff(const vector<double>& price,
                                                                 const double& K),
                                                   const int& no_steps, const int& no_sims);


double
derivative_price_simulate_european_option_generic_with_control_variate(const double& S, const double& K,
                                                                       const double& r, const double& sigma,
                                                                       const double& time,
                                                                       double payoff(const vector<double>& price,
                                                                                     const double& K),
                                                                       const int& nosteps, const int& nosims);


/////////////////////////////
// payoffs of various options, to be used as function arguments in above simulations

double payoff_arithmetric_average_call(const vector<double>& prices, const double& K);
double payoff_geometric_average_call(const vector<double>& prices, const double& K);
double payoff_lookback_call(const vector<double>& prices, const double& unused_variable);
double payoff_lookback_put(const vector<double>& prices, const double& unused_variable);

////////////////// alternative stochastic processes ////////////////

double option_price_call_merton_jump_diffusion( const double& S, const double& K, const double& r,
                                                const double& sigma, const double& time_to_maturity,
                                                const double& lambda, const double& kappa, const double& delta);

// fixed income derivatives, GBM assumption on bond price

double bond_option_price_call_zero_black_scholes(const double& B, const double& K, const double& r,
                                                 const double& sigma, const double& time);
double bond_option_price_put_zero_black_scholes(const double& B, const double& K, const double& r,
                                                const double& sigma, const double& time);
double bond_option_price_call_coupon_bond_black_scholes(const double& B, const double& K, const double& r,
                                                        const double& sigma, const double& time,
                                                        const vector<double> coupon_times,
                                                        const vector<double> coupon_amounts);
double bond_option_price_put_coupon_bond_black_scholes(const double& B, const double& K, const double& r,
                                                       const double& sigma, const double& time,
                                                       const vector<double> coupon_times,
                                                       const vector<double> coupon_amounts);
double bond_option_price_call_american_binomial( const double& B, const double& K, const double& r,
                                                 const double& sigma, const double& t, const int& steps);
double bond_option_price_put_american_binomial( const double& B, const double& K, const double& r,
                                                const double& sigma, const double& t, const int& steps);
```

```
////////////////////////////////////////////////////////////////////////////////
// term structure models

double term_structure_yield_nelson_siegel(const double& t,
                                          const double& beta0, const double& beta1, const double& beta2,
                                          const double& lambda );


double term_structure_discount_factor_cubic_spline(const double& t,
                                                   const double& b1,
                                                   const double& c1,
                                                   const double& d1,
                                                   const vector<double>& f,
                                                   const vector<double>& knots);

double term_structure_discount_factor_cir(const double& t, const double& r,
                                          const double& kappa,
                                          const double& lambda,
                                          const double& theta,
                                          const double& sigma);

double term_structure_discount_factor_vasicek(const double& time,
                                              const double& r,
                                              const double& a,const double& b, const double& sigma);



/////////////////
/// binomial term structure models
/// bond option, rendlemann bartter (binomial)

double
bond_option_price_call_zero_american_rendleman_bartter(const double& K, const double& option_maturity,
                                                       const double& S, const double& M,
                                                       const double& interest,
                                                       const double& bond_maturity,
                                                       const double& maturity_payment,
                                                       const int& no_steps);
```

Appendix D

Installation

The routines discussed in the book are available for download.

D.1 Source availability

The algorithms are available from my home page as a ZIP file containing the source code. These have been tested with the latest version of the GNU C++ compiler. As the algorithms in places uses code from the Standard Template Library, other compilers may not be able to compile all the files directly. If your compiler complains about missing header files you may want to check if the STL header files have different names on your system. The algorithm files will track the new ANSI standard for C++ libraries as it is being settled on. If the compiler is more than a couple of years old, it will not have STL. Alternatively, the GNU compiler `gcc` is available for free on the internet, for most current operating systems.

List of Codes

Appendix E

Acknowledgements.

After this paper was put up on the net, I've had quite a few emails about them. Some of them has pointed out bugs etc.

Among the ones I want to say thanks to for making improving suggestions and pointing out bugs are

EARAUJO@pactual.com.br
Jens Larsson <d96-jla@d.kth.se>
Michael L Locher
Lars Gregori
Steve Bellantoni <scj@cs.toronto.edu>
Steven Leadbeater <steven.leadbeater@edfman.com>
Ariel Almedal
Lotti Luca, Milano, Italy
jean-paul.beveraggi@gadz.org

Index

Bibliography

Milton Abramowiz and Irene A Stegun. *Handbook of Mathematical Functions.* National Bureau of Standards, 1964.

Giovanni Barone-Adesi and Robert E Whaley. Efficient analytic approximation of American option values. *Journal of Finance*, 42(2):301–20, June 1987.

Per Berck and Knut Sydsæter. *Matematisk Formelsamling for økonomer.* Universitetsforlaget, 2 edition, 1995.

Fisher Black. The pricing of commodity contracts. *Journal of Financial Economics*, 3:167–79, 1976.

Fisher Black and Myron S Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 7: 637–54, 1973.

Peter L Bossaerts and Bernt Arne Ødegaard. *Lectures on Corporate Finance.* World Scientific Press, Singapore, 2001.

Phelim P Boyle. Options: A Monte Carlo approach. *Journal of Financial Economics*, 4:323–38, 1977.

Richard A Brealey and Stewart C Myers. *Principles of Corporate Finance.* McGraw–Hill, fifth edition, 1996.

Michael Brennan and Eduardo Schwartz. Finite difference methods and jump processes arising in the pricing of contingent claims: A synthesis. *Journal of Financial and Quantitative Analysis*, 13:461–74, 1978.

John Cox and Mark Rubinstein. *Options markets.* Prentice–Hall, 1985.

John C Cox, Jonathan E Ingersoll, and Stephen A Ross. A theory of the term structure of interest rates. *Econometrica*, 53:385–408, 1985.

John C Cox, Stephen A Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7:229–263, 1979.

M B Garman and S W Kohlhagen. Foreign currency option values. *Journal of International Money and Finance*, 2: 231–37, 1983.

Robert Geske. The valuation of compound options. *Journal of Financial Economics*, 7:63–81, March 1979.

M Barry Goldman, Howard B Sosin, and Mary Ann Gatto. Path–dependent options: Buy at the low, sell at the high. *Journal of Finance*, 34, December 1979.

J O Grabbe. The pricing of call and put options on foreign exchange. *Journal of International Money and Finance*, 2: 239–53, 1983.

Chi-fu Huang and Robert H. Litzenberger. *Foundations for financial economics.* North–Holland, 1988.

John Hull. *Options, Futures and other Derivative Securities.* Prentice–Hall, second edition, 1993.

John Hull. *Options, Futures and other Derivatives.* Prentice–Hall, Fifth edition, 2003.

F Jamshidan. An exact bond option pricing formula. *Journal of Finance*, 44:205–9, March 1989.

A Kemna and A Vorst. A pricing method for options based on average asset values. *Journal of Banking and Finance*, 14: 113–29, March 1990.

Donald E Knuth. *The Art of Computer Programming Volume 2, Seminumerical Algotithms.* Addison–Wesley, third edition, 1997.

Stanley B Lippman and Jos'ee Lajoie. *C++ primer.* Addison–Wesley, third edition, 1998.

Harry Markowitz. Portfolio selection. *Journal of Finance*, 7: 77–91, 1952.

J Houston McCulloch. Measuring the term structure of interest rates. *Journal of Business*, 44:19–31, 1971.

Robert McDonald and Daniel Siegel. The value of waiting to invest. *Quarterly Journal of Economics*, pages 707–727, November 1986.

Robert L McDonald. *Derivatives Markets.* Addison Wesley, 2002.

Robert C Merton. The theory of rational option pricing. *Bell Journal*, 4:141–183, 1973.

Franco Modigliani and Merton M Miller. The cost of capital, corporation finance and the theory of investment. *American Economic Review*, 48:261–97, 1958.

Charles R Nelson and Andrew F Siegel. Parsimonious modelling of yield curves. *Journal of Business*, 60(4):473–89, 1987.

Carl J Norstrom. A sufficient conditions for a unique nonnegative internal rate of return. *Journal of Financial and Quantitative Analysis*, 7(3):1835–39, 1972.

William Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992.

Richard J Rendleman and Brit J Bartter. Two–state option pricing. *Journal of Finance*, 34(5):1093–1110, December 1979.

Richard J Rendleman and Brit J Bartter. The pricing of options on debt securities. *Journal of Financial and Quantitative Analysis*, 15(1):11–24, March 1980.

Richard Roll. An analytical formula for unprotected American call options on stocks with known dividends. *Journal of Financial Economics*, 5:251–58, 1977.

Stephen A Ross, Randolph Westerfield, and Jeffrey F Jaffe. *Corporate Finance.* Irwin, fourth edition, 1996.

Mark Rubinstein. The valuation of uncertain income streams and the valuation of options. *Bell Journal*, 7:407–25, 1976.

Mark Rubinstein. Exotic options. University of California, Berkeley, working paper, 1993.

Robert J. Shiller. The term structure of interest rates. In B M Friedman and F H Hahn, editors, *Handbook of Monetary Economics*, volume 2, chapter 13, pages 627–722. Elsevier, 1990.

Bjarne Stroustrup. *The C++ Programming language*. Addison–Wesley, third edition, 1997.

Suresh Sundaresan. *Fixed Income Markets and their Derivatives*. South-Western, 2 edition, 2001.

Y L Tong. *The Multivariate Normal Distribution*. Springer, 1990.

O Vasicek. An equilibrium characterization of the term structure. *Journal of Financial Economics*, 5:177–88, 1977.

Robert E Whaley. On the valuation of American call options on stocks with known dividends. *Journal of Financial Economics*, 9:207–1, 1981.

Paul Wilmott, Jeff Dewynne, and Sam Howison. *Option Pricing, Mathematical models and computation*. Oxford Financial Press, 1994. ISBN 0 9522082 02.