Financial Numerical Recipes.

Bernt Arne Ødegaard

9 September 1999

Abstract

This is a discussion of algorithms and computer code for advanced financial calculations. It was written for use in a course teaching derivative securities. It contains the basic and some advanced algorithms for option pricing, and some algorithms dealing with term structure modeling and pricing of fixed income securities.

All computer code is in the C++ language, and implemented as self–contained subroutines that can be compiled on any standard C++ compiler.

Contents

1 Introduction. **4**

2 Cash flow algoritms. **5**

2.1 Present value. 5

2.2 Internal rate of return. 6

2.2.1 Check for unique irr. 7

3 Basic Option Pricing, analytical solutions. **8**

3.1 Setup. 8

3.2 European call and put options, The Black Scholes analysis. 8

3.2.1 Analytical option prices, Black Scholes case. 9

3.2.2 Partial derivatives. 10

3.2.3 Implied Volatility. 11

3.3 Adjusting for payouts of the underlying. 13

3.3.1 Continous Payouts from underlying. 14

3.3.2 Dividends. 14

3.4 American options. 15

3.4.1 Exact american call formula when stock is paying one dividend. 15

4 Binomial option pricing. **18**

4.1 Introduction 18

4.2 European Options. 19

4.3 American Options. 20

4.4 Estimating partials. 20

4.5 Binomial approximation, dividends. 22

4.5.1 Proportional dividends. 23

4.5.2 Discrete dividends. 24

5 Finite Differences **26**

5.1 European Options. 26

5.2 American Options. 27

6 Simulation **30**

6.1 Simulating the sample path. 30

6.2 Hedge parameters 31

7 Approximations **33**

7.1 A quadratic approximation to American prices due to Barone–Adesi and Whaley. 33

7.2 An approximation to the American Put due to Geske and Johnson . . . 35

8 Futures algoritms. 37

8.1 Pricing of futures contract. . . . 37
8.2 Options on futures . . . 37
8.2.1 Black's model . . . 37
8.2.2 Binomial approximation. . . . 38

9 Foreign Currency Algoritms 39

9.1 Foreign Currency Options . . . 39
9.1.1 European options. . . . 39
9.1.2 American options. . . . 40

10 Bond Algoritms. 41

10.1 Bond Price. . . . 41
10.2 Yield to maturity. . . . 43
10.3 Duration. . . . 43
10.3.1 Simple duration. . . . 43
10.3.2 Macaulay Duration . . . 44
10.3.3 Modified Duration . . . 45
10.4 Convexity . . . 45

11 Exotic options. 47

11.1 Lookback options. . . . 47

12 A general approach to option pricing by simulation. 48

12.1 Simulating prices of underlying. . . . 48
12.2 Defining payoffs. . . . 49
12.3 Pricing . . . 50
12.4 Improving the estimates, control variates. . . . 51

13 Alternatives to the Black Scholes type option formula. 52

13.1 Merton's Jump diffusion model. . . . 52
13.2 Stochastic volatility. . . . 53

14 Mean Variance Analysis. 54

14.1 Introduction. . . . 54
14.2 Mean variance portfolios. . . . 55
14.3 Short sales constraints . . . 55

15 Term Structure algorithms. 58

15.1 Term structure calculations. . . . 58
15.2 Using the currently observed term structure. . . . 58

15.2.1 Linear Interpolation. 59
15.3 Term structure approximations. 60
15.3.1 The Nelson and Siegel(1987) functional form. 60
15.3.2 Bliss (1989) 60
15.3.3 Cubic spline. 61
15.4 Term structure models. 61
15.4.1 The Vasicek model. 61
15.4.2 The original Cox Ingersoll Ross model. 62
15.4.3 The estimated CIR model used by Brown and Dybvig. 63

16 Fixed Income modelling, with emphasis on contingent claims. 65
16.1 Black Scholes bond pricing. 65
16.2 The Rendleman and Bartter model 65
16.3 Vasicek bond pricing. 66

A Normal Distribution approximations. 69
A.1 References 71

B A note on C++ and the source code 72
B.1 Source availability 72
B.2 Libraries. 72
B.3 On `C++` for non-`C++` programmers 72
B.4 Mathematical operators. 73
B.4.1 Exponentiation. 73
B.4.2 Increment and decrement. 73
B.5 Dynamically sized arrays. 73
B.6 The form of the `for` statement. 73
B.7 The `#include` construct. 74
B.8 The `class` concept. 74
B.9 Implementation in other programming languages. 74
B.10 References. 75

C Acknowledgements. 76

Chapter 1

Introduction.

Very often, when teaching finance, we only present the formula that gives the answer to how to calculate a given number. For teaching the material it is often useful to actually look at the implementation of *how* the calculation is done in practice.

In this document I am trying to show how some finance algorithms actually are implemented in addition to describing them. The implementation will be in the form of computer subroutines. The actual programming language is not important, but we do need that the algorithms are testable, so I have chosen to use `C++`as the programming language. `C++`has other advantages, but that is in a larger setting. I have tried to make the algorithms as "generic" as possible, without relying on particularities of any given language, but I have found it necessary to use some algoritms and templates from the new `C++`library standard. (The Standard Template Library)

Unfortunately, it turned out to be impossible to be completely "generic," in particular when using the term structure of interest rates. In many financial algorithms we need to take present values using the term structure. The problem is that there is a large numbers of ways to actually calculate/approximate/model a term structure. I have therefore hidden this complexity, by defining a `class` that is just an abstract representation of a term structure, which the various algorithms use. To give a full description of this class would take too long. Companion documents describing various classes for data structuring is available from my homepage.

I will first document the algorithm, and then present code that implements it. Note that I do not make any guarantees about the correctness of the algorithms, these are meant for teaching.

As you will see, this document is by no means complete, but it is being gradually added to as I get time.

This is not a textbook in the underlying theory, for that there are many good alternatives. For most of the material the best textbook to refer to is Hull [1993], since I have used that as a reference, and the notation follows that.

Chapter 2

Cash flow algoritms.

We look at some of the "basic" calculations in finance, used for project evaluation etc.

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 and r be the interest rate. Suppose we have N future cash flows that occur at times $t_1, t_2, \cdots, t_N$.

If compounding is continous, would calculate the present value as follows

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

This calculation is implemented as follows:

```
// file cflow_pv.cc
// author: Bernt Arne Oedegaard.

#include <cmath>
#include <vector>

double cash_flow_pv( vector<double>& cflow_times, vector<double>& cflow_amounts, double r){
    // calculate present value of cash flow, continous discounting
    double PV=0.0;
    for (unsigned int t=0; t<cflow_times.size();t++) {
        PV += cflow_amounts[t] * exp( -r * cflow_times[t]);
    };
    return PV;
};
```

If discounting was discrete, would calculate the present value as

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

which is implemented as

```
// file cflow_pv_discrete.cc
// author: Bernt Arne Oedegaard.
// calculate the present value of a stream of cash flows using discrete compounding

#include <cmath>
#include <vector>

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

2.2 Internal rate of return.

The internal rate of return of a cash flow is the interest rate that makes the present value of a cash flow equal to zero.

Finding an internal rate of return is thus to find a root of of the equation

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

As any textbook in basic finance, such as Brealey and Myers [1996], Ross et al. [1996] or Sharpe and Alexander [1990] will tell, there is a number of problems with 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.

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

#include <cmath>
#include <algorithm>
#include <vector>

#include "fin_algoritms.h"

const double ERROR=-1e30;

double cash_flow_irr(vector<double>& cflow_times, 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.
};
```

2.2.1 Check for unique irr.

If you worry about finding more than one IRR, the following 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]).

```
// file cflow_irr_test_unique.cc
// author Bernt A Oedegaard

#include <cmath>
#include <vector>

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

bool cash_flow_unique_irr(vector<double>& cflow_times, vector<double>& cflow_amounts) {
    // check whether the cash flow has a unique irr.
    int sign_changes=0; // first check Descartes rule
    for (unsigned 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 (unsigned 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;
}
```

Chapter 3

Basic Option Pricing, analytical solutions.

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. In this chapter we will discuss ways of calculating the price of an option in the setting discussed in these original papers. The discussion is not complete, it needs to be supplemented by one of the standard textbooks, like Hull [1993].

3.1 Setup.

Let us start by reviewing the setup. The basic assumption used is about the stochastic process governing the price of the underlying asset the option is written on. In the following discussion we will use the standard example of a stock option, but the theory is not only relevant for stock options.

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 following *partial differential equation*:

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

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.

We will start by discussing the original example solved by Black, Scholes, Merton: European call and put options.

3.2 European call and put options, The Black Scholes analysis.

A call (put) option gives the holder the right, but not the obligation, to buy (sell) some underlying asset at a given price X, called the exercise price, on or before some given date T.

If the option is European, it can only be used (exercised) at the maturity date. If the option is American, it can be used at any date up to and including the maturity date.

We use the following notation:

- S: Price of the underlying, eg stock price.
- X: Exercise price.

- r: Risk free interest rate.
- σ: Standard deviation of the underlying asset, eg stock.
- t: Current date.
- T: Maturity date.
- $T-t$: Time to maturity.

At maturity, a call option is worth

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

and a put option is worth

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

This can be used in solving the pde above, since they define a *boundary condition* for the pde.

3.2.1 Analytical option prices, Black Scholes case.

The pde with the boundary condition

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

was shown by Black and Scholes to have an analytical solution of the following functional form

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

where

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

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

$$N(\cdot) = \text{ The cumulative normal distribution}$$

Similarly, the price for a put option is

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

Computer Algorithm, Black Scholes price.

```
// file: black_scholes_call
// author: Bernt A Oedegaard
// Calculation of the Black Scholes option price formula.

#include <cmath> // mathematical library
#include "normdist.h" // this defines the normal distribution

double option_price_call_black_scholes( double S, // spot price
                                        double X, // Strike (exercise) price,
                                        double r, // interest rate
                                        double sigma,
```

```
                                                    double time)
{
    double time_sqrt = sqrt(time);
    double d1 = (log(S/X)+r*time)/(sigma*time_sqrt) + 0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    double c = S * N(d1) - X * exp(-r*time) * N(d2);
    return c;
};
```

3.2.2 Partial derivatives.

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

Delta The first derivative of the option price with respect to the underlying 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.

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

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

Since delta is often used, here is a subroutine that calculates it

```
// file: black_scholes_delta_call.cc
// author: Bernt A Oedegaard
// The delta of the Black - Scholes formula

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

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

The remaining derivatives are more seldom used, but all of them are relevant.

Gamma The second derivative of the option price wrt the underlying stock. These are equal for puts and calls

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

Theta The partial with respect to time–to–maturity.

$$\Theta_c = \frac{\partial c}{\partial (T-t)} = -\frac{n(d_1)S\sigma}{2\sqrt{T-t}} - rXe^{-r(T-t)}N(d_2)$$

$$\Theta_p = \frac{\partial p}{\partial (T-t)} = -\frac{n(d_1)S\sigma}{2\sqrt{T-t}} + rXe^{-r(T-t)}N(-d_2)$$

Vega The partial with respect to volatility.

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

$$\text{Vega}_p = \frac{\partial p}{\partial \sigma} = S\sqrt{T-t}n(d_1)$$

Rho The partial with respect to the interest rate.

$$\text{Rho}_c = \frac{\partial c}{\partial r} = X(T-t)e^{-r(T-t)}N(d_2)$$

$$\text{Rho}_p = \frac{\partial p}{\partial r} = -X(T-t)e^{-r(T-t)}N(-d_2)$$

Computer algorithm, Black Scholes partials. Here is the algorithm that calculates all the above derivatives.

```
// black_scholes_partials_call.cc
// author: Bernt A Oedegaard
// The partial derivatives of the Black - Scholes formula

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

void option_price_partials_call_black_scholes( double S, // spot price
                                                double X, // Strike (exercise) price,
                                                double r, // interest rate
                                                double sigma, // volatility
                                                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/X)+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*X*exp( -r*time)*N(d2);
  Vega = S * time_sqrt * n(d1);
  Rho = X * time * exp(-r * time) * N(d2);
};
```

3.2.3 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

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

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.

Computer algorithm, implied volatility, bisections.

```
// file black_scholes_imp_vol_bisect.cc
// author: Bernt A Oedegaard
// calculate implied volatility of Black Scholes formula

#include "fin_algoritms.h"
#include <cmath>

double option_price_implied_volatility_call_black_scholes_bisections(
   double S, double X, double r, double time, double option_price)
{ // check for arbitrage violations:
    // if price at almost zero volatility greater than price, return 0

    double sigma_low=0.0001;
    double price = option_price_call_black_scholes(S,X,r,sigma_low,time);
    if (price>option_price) return 0.0;

    // 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;
};
```

Implied volatility, Newton-Raphson.

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. If

$$f(x) = 0$$

is the equation we want to solve, given a first guess x_0, we 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.

In our case

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

and, each new iteration will calculate

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

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.

Computer algoritm, implied volatility, Newton steps.

```
// file black_scholes_imp_vol_newt.cc
// author: Bernt A Oedegaard
// calculate implied volatility of Black Scholes formula using newton steps

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

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

   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
};
```

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

3.3 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.

3.3.1 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 (see [Hull, 1997, pg 263])

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

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

where

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

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

which are implemented below.

```
// file: black_scholes_price_payout_call.cc
// author: Bernt A Oedegaard
// Calculation of the Black Scholes option price formula,
// special case where the underlying is paying out a yield of q.

#include <cmath> // mathematical library
#include "normdist.h" // this defines the normal distribution

double option_price_european_call_payout( double S, // spot price
                                          double X, // Strike (exercise) price,
                                          double r, // interest rate
                                          double q, // yield on underlying
                                          double sigma, // volatility
                                          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;
};
```

3.3.2 Dividends.

A special case of payouts for the underlying is dividends. When the underlying 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.

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

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

double option_price_european_call_dividends( double S,
                                             double X,
                                             double r,
                                             double sigma,
                                             double time_to_maturity,
                                             vector<double>& dividend_times,
                                             vector<double>& dividend_amounts )
// reduce the current stock price by the amount of dividends.
{
  for (int i=0;i<dividend_times.size();i++) {
    if (dividend_times[i]<=time_to_maturity){
      S -= dividend_amounts[i] * exp(-r*dividend_times[i]);
    };
  };
  return option_price_call_black_scholes(S,X,r,sigma,time_to_maturity);
};
```

3.4 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. The known analytical American price is the case of a call on a stock that pays a known dividend, 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.

3.4.1 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. 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 ch 10.A2 of Hull [1993] for a textbook summary.

We use the following notation:

- S stock price.
- X exercise price.
- D_1: amount of dividend paid.
- t_1: Time of dividend payment.
- T: Maturity date of option.

- $\tau_1 = T - t_1$: Time to dividend payment.
- $\tau = T - t$: Time to maturity.
- C American call price.

A first check of early exercise is:

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

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

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

where $c(\cdot)$ is the regular Black Scholes formula.
Otherwise, the following is the formula for the American call value.

$$\begin{aligned} C &= (S - D_1 e^{-r\tau_1})[N(b_1) + N(a_1, -b_1, \rho)] \\ &\quad + Xe^{-r\tau)} N(a_2, -b_2, \rho) \\ &\quad - (X - D_1)e^{-r\tau_1} N(b_2) \end{aligned}$$

where

$$\rho = -\sqrt{\frac{\tau_1}{\tau}}$$

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

$$a_2 = a_1 - \sigma\sqrt{\tau}$$

$$b_1 = \frac{\log\left(\frac{S - D_1 e^{-t\tau_1}}{\bar{S}}\right) + (r + \frac{1}{2}\sigma^2)\tau_1}{\sigma\sqrt{\tau_1}}$$

$$b_2 = b_1 - \sigma\sqrt{\tau}$$

$$\tau_1 = t_1 - t$$

$$\tau = T - t$$

and $\bar{S}$ solves

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

Computer algorithm, Roll–Geske–Whaley call formula.

```
// file anal_price_am_call_div.cc
// author: Bernt A Oedegaard
// calculate dividend adjusted formula for american call option.

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

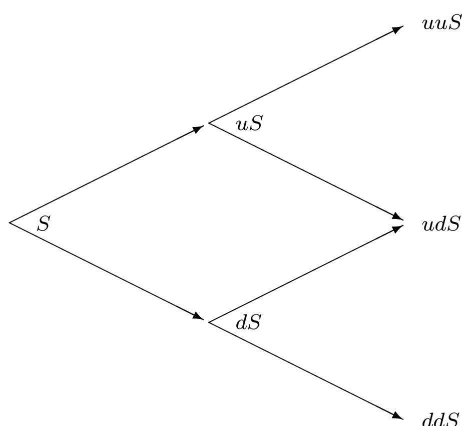
```
#include "fin_algoritms.h" // the regular black sholes formula

double option_price_american_call_dividend(double S,
                                           double X,
                                           double r,
                                           double sigma,
                                           double tau,
                                           double D1,
                                           double tau1)
{
    if (D1 <= X* (1.0-exp(-r*(tau-tau1)))) // check for no exercise
        return option_price_call_black_scholes(S-exp(-r*tau1)*D1,X,r,sigma,tau);

    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-X
    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,X,r,sigma,tau-tau1);
    double test = c-S_high-D1+X;
    while ( (test>0.0)
            && (S_high<=1e10) ) {
        S_high *= 2.0;
        c = option_price_call_black_scholes(S_high,X,r,sigma,tau-tau1);
        test = c-S_high-D1+X;
    };
    if (S_high>1e10) { // early exercise never optimal, find BS value
        return option_price_call_black_scholes(S-D1*exp(-r*tau1),X,r,sigma,tau);
    };

    S_bar = 0.5 * S_high; // now find S_bar that solves c=S_bar-D+X
    c = option_price_call_black_scholes(S_bar,X,r,sigma,tau-tau1);
    test = c-S_bar-D1+X;
    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,X,r,sigma,tau-tau1);
        test = c-S_bar-D1+X;
    };
    double a1 = (log((S-D1*exp(-r*tau1))/X) +( 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)
        - (X*exp(-r*tau))*N(a2,-b2,rho)
        - (X-D1)*exp(-r*tau1)*N(b2);
    return C;
};
```

Chapter 4

Binomial option pricing.

4.1 Introduction

When valuing options using the binomial approach, we assume the underlying option price follows a process where at given times the price can jump either up or down.



One way to think about what is happening is that we are approximating the terminal distribution of the price of the underlying at the maturity date of the option. At each node of this tree, the value of the option can be found using simple arbitrage arguments, it is possible to construct a portfolio by combining riskless borrowing and the risky asset to duplicate the payoff of an option, which means the price of the option is easy to find.

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

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

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

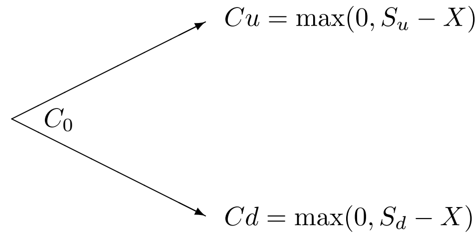
$$d = e^{-\sigma\sqrt{\Delta t}}$$

We also define the "risk neutral probabilities"

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

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

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

$$C_0 = e^{-r}(pC_u + (1-p)C_d)$$

The original source for binomial option pricing was the paper by Cox et al. [1979]. Good textbook discussions are in Cox and Rubinstein [1985] and Hull [1993].

4.2 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.

Computer algorithm.

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.

```
// file bin_eur_call.cc
// author: Bernt A Oedegaard
// calculate the binomial option pricing formula for an european call

#include <cmath> // standard mathematical library
#include <algorithm> // defining the max() operator
#include <vector> // STL vector templates

double option_price_call_european_binomial( double S, // spot price
                                            double X, // exercice price
                                            double r, // interest rate
                                            double sigma, // volatility
                                            double t, // time to maturity
                                            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];
};
```

4.3 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.

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

```
// file bin_am_call.cc
// author: Bernt A Oedegaard
// calculate the binomial option pricing formula for an American call

#include <cmath> // standard mathematical library
#include <algorithm> // defining the max() operator
#include <vector> // STL vector templates

double option_price_call_american_binomial( double S, // spot price
                                            double X, // exercice price
                                            double r, // interest rate
                                            double sigma, // volatility
                                            double t, // time to maturity
                                            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];
};
```

4.4 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 chapter 14 of Hull [1993]. The code below is for the non–dividend case.

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

```
// file bin_am_delta_call.cc
// author Bernt Arne Oedegaard

#include <cmath>
#include <algorithm>
#include <vector>

double option_price_delta_american_call_binomial(double S,
                                                  double X,
                                                  double r,
                                                  double sigma,
                                                  double t,
                                                  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;
};
```

Other hedge parameters.

```
// file bin_part.cc
// author Bernt Arne Oedegaard

#include <cmath>
#include <algorithm>
#include "fin_algoritms.h"

void option_price_partials_american_call_binomial(double S, // spot price
                                                   double X, // Exercise price,
                                                   double r, // interest rate
                                                   double sigma, // volatility
                                                   double time, // time to maturity
                                                   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;
};
```

4.5 Binomial approximation, dividends.

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.

4.5.1 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.

```
// file bin_am_prop_div_call.cc
// author: Bernt Arne Oedegaard
// binomial option pricing adjusting for dividends.

#include <cmath>
#include <algorithm>
#include <vector>
#include "fin_algoritms.h"

double option_price_call_american_proportional_dividends_binomial(
    double S,
    double X,
    double r,
    double sigma,
    double time,
    int no_steps,
    vector<double>& dividend_times,
    vector<double>& dividend_yields)
// given a dividend yield, the binomial tree recombines
{
    int no_dividends=dividend_times.size();
    if (no_dividends == 0) // just take the regular binomial
        return option_price_call_american_binomial(S,X,r,sigma,time,no_steps);
    double R = exp(r*(time/no_steps));
    double Rinv = 1.0/R;
    double u = exp(sigma*sqrt(time/no_steps));
    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];
};
```

4.5.2 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. In that case it will be a good deal faster to avoid the recursive calls. Look in [Hull, 1993, pg 347] for ways to achieve this by making some small assumptions.

Computer algorithm, binomial pricing with discrete dividends.

```
// file bin_am_div_call.cc
// author: Bernt Arne Oedegaard
// binomial option pricing adjusting for dividends.

#include <cmath>
#include <vector>
#include "fin_algoritms.h"

double option_price_call_american_discrete_dividends_binomial(double S,
                                                               double X,
                                                               double r,
                                                               double sigma,
                                                               double t,
                                                               int steps,
                                                               vector<double>& dividend_times,
                                                               vector<double>& dividend_amounts)
// given an amount of dividend, the binomial tree does not recombine, have to
// start a new tree at each ex-dividend date.
// do this recursively, at each ex dividend date, at each step, call the
// binomial formula starting at that point to calculate the value of the live
// option, and compare that to the value of exercising now.
{
    int no_dividends = dividend_times.size();
    if (no_dividends == 0) // just take the regular binomial
        return option_price_call_american_binomial(S,X,r,sigma,t,steps);
    int steps_before_dividend = (int)(dividend_times[0]/t*steps);

    double R = exp(r*(t/steps));
    double Rinv = 1.0/R;
    double u = exp(sigma*sqrt(t/steps));
    double uu= u*u;
    double d = 1.0/u;
    double pUp = (R-d)/(u-d);
    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] = uu*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,
                X, r, sigma,
                t-dividend_times[0], // time after first dividend
                steps-steps_before_dividend,
                tmp_dividend_times,
                tmp_dividend_amounts);
        // what is the value of keeping the option alive? Found recursively,
        // with one less dividend, the stock price is current value
        // less the dividend.
        call_values[i] = max(value_alive,(prices[i]-X)); // 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]-X); // check for exercise
        };
    };
    return call_values[0];
};
```

Chapter 5

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.

In the following we implement the two schemes described in chapter 14.7 of Hull [1993], the *implicit finite differences* and the the *explicit finite differences.*

5.1 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.

Computer algoritm, implicit finite differences. The following follows the discussion of finite differences on page 354 of Hull [1993]. Note the use of a linear algebra library to simplify the algoritm.

```
#include <cmath>
#include "newmat.h" // definitions for newmat matrix library
#include <vector> // standard STL vector template
#include <algorithm>

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);
};
```

Computer algoritm, explicit finite differences. An alternative to the implicit finite differences is the explicit finite difference method. The explicit version is faster, but a problem with the explicit version is

that it may not converge. The following follows the discussion of finite differences starting on page 356 of Hull [1993].

```
// findiff_exp_eur_put.cc
// author: Bernt A Oedegaard

#include <cmath>
#include <algorithm>
#include <vector>

double option_price_put_european_finite_diff_explicit(double S,
                                                       double X,
                                                       double r,
                                                       double sigma,
                                                       double time,
                                                       int no_S_steps,
                                                       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];
};
```

5.2 American Options.

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

Computer algoritm, implicit finite differences.

```
#include <cmath>
#include "newmat.h" // definitions for newmat matrix library
#include <vector>
#include <algorithm>
```

```
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;
    // 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;
    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);
};
```

Computer algoritm, explicit finite differences.

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

#include <cmath>
#include <algorithm>
#include <vector>

double option_price_put_american_finite_diff_explicit( 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 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];
}
```

Further Reading 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.

Chapter 6

Simulation

We now consider using simulations to estimate the price of an option.

For a call option, this is to calculate the current expected value of

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

or

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

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 underlying process, and find the estimated call price as the average of the simulated values. By a Law of Large Numbers, this average will converge to the actual call value, where the rate of convergence will depend on the way we simulate the sample path, and how many simulations we perform.

6.1 Simulating the sample path.

The assumed underlying price process is the *geometric Brownian Motion*

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

or

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

To simulate this, we need to *discretise* the process, by splitting the time between 0 and T into a number of periods of length Δt,

$$dt \approx \Delta t$$

$$dS \approx \Delta S = S_t - S_{t-1}$$

To simulate the term due to the Brownian motion (dB), remember that one of the defining characteristics of Brownian Motion is that the increment for any finite period of length s is distributed normally with mean zero and variance s. Since the variance is s, the standard deviation is $\sqrt{s}$. Thus, if ε_t is normally distributed with mean zero and unit variance, we can simulate the discrete process by

$$S_t - S_{t-1} = \mu \Delta t S_{t-1} + \sigma S_{t-1} \sqrt{\Delta t} \varepsilon_t$$

or

$$S_t = S_{t-1} + \mu \Delta t S_{t-1} + \sigma S_{t-1} \sqrt{\Delta t} \varepsilon_t$$

This expression can be used to generate a series of sample paths, which will approximate the sample path of a Brownian Motion process as the sampling interval Δt decreases.

This discretization is not recommended for actual use. Here we take into account that it is the log of the underlying price that follows a Brownian motion. A more correct discretization is to use

$$S_t = S_{t-1} exp\left(\left(\mu - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\varepsilon_t\sqrt{\Delta t}\right)$$

This is the discretization used when the payoff depend on the evolution of the underlying throughout the period. For examples of this, see the chapter on exotic options.

European Calls and Puts, however, do not depend on any intermediate observations, so it is therefore only necessary to simulate the terminal observation of the underlying.

Computer Algorithm, Monte Carlo European Option.

```
// file: simulated_call_euro.cc
// author: Bernt Arne Oedegaard

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

double option_price_call_european_simulated( double S,
                                             double X,
                                             double r,
                                             double sigma,
                                             double time,
                                             int no_sims)
{
    double sigma_sqr = sigma * sigma;
    double R = (r - 0.5 * sigma_sqr)*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);
    };
    double c = exp(-r*time) * (sum_payoffs/double(no_sims));
    return c;
};
```

6.2 Hedge parameters

```
// file simulated_delta_call.cc
// author: Bernt A Oedegaard
// estimation of the partials when doing monte carlo

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

double option_price_delta_call_european_simulated( double S,
                                                   double X,
                                                   double r,
                                                   double sigma,
                                                   double time,
                                                   int no_sims)
   // estimate the price using two different S values
{
    double sigma_sqr = sigma * sigma;
    double R = (r - 0.5 * sigma_sqr)*time;
    double SD = sigma * sqrt(time);
    double sum_payoffs = 0.0;
    double sum_payoffs_2 = 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_2 = (S+q)* exp(R + SD * Z);
        sum_payoffs_2 += max(0.0, S_T_2-X);
    };
    double c = exp(-r*time) * ( sum_payoffs/no_sims);
    double c2 = exp(-r*time) * ( sum_payoffs_2/no_sims);
    return (c2-c)/q;
};
```

Further Reading Boyle [1977]

Chapter 7

Approximations

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.

7.1 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 having a continuous payout, described in Barone-Adesi and Whaley [1987] (BAW). Their solution technique finds an approximate solution to the differential equation

$$\frac{1}{2}\sigma^2 S^2 V_S S + bSV_S - rV + V_t = 0 \tag{7.1}$$

The case we look at here is an option written on a commodity paying b as a continuous payout. Here V is the (unknown) formula that determines the price of the contingent claim. For an European option this can be explicitly solved, with the adjusted Black Scholes formula as the solution, but 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. This will also have to satisfy the partial differential equation. To come up with an approximation BAW transformed equation (7.1) 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

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

where

$$M = \frac{2r}{\sigma^2}$$

$$N = \frac{2b}{\sigma^2}$$

$$K(T) = 1 - e^{-r(T-t)}$$

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

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

and S^* solves

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

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

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

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

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

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

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

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

where $c(S)$ is the Black Scholes value for commodities.
Similarly, for a put option the approximation is

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

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

and we solve for S^{**} again by Newton's procedure, where now

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

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

```
// file approx_am_call.cc
// author: Bernt Arne Oedegaard
// implements the quadratic approximation of Barone-Adesi and Whaley
// described in Journal of Finance, june 87.

#include <cmath>
#include <algorithm>
#include "normdist.h" // normal distribution
#include "fin_algoritms.h" // define other option pricing formulas

const double ACCURACY=1.0e-6;
```

```
double option_price_american_call_approximated_baw( double S,
                                                    double X,
                                                    double r,
                                                    double b,
                                                    double sigma,
                                                    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;

    // seed value from paper
    double q2_inf = 0.5 * ( (-nn-1.0) + sqrt(pow((nn-1),2.0)+4.0*m));
    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
}
```

For further details of the Barone Adesi quadratic approximation: Main source is the original paper: Barone-Adesi and Whaley [1987]. The approximation is also discussed in section 14A of Hull [1993].

7.2 An approximation to the American Put due to Geske and Johnson

In Geske and Johnson [1984] a very nice approximation is developed for the American put problem. The solution technique is to view the American put as an sequence of Bermudan options, with the number of exercise dates increasing. The correct value is the limit of this sequence.

Define P_i to be the price of the put option with i dates of exercise left. P_1 is then the price of an european option, with the one exercise date the expiry date. P_2 is the price of an option that can be exercised twice, once halfway between now and expiry, the other at expiry. Geske-Johnson shows how these options may

be priced, and then develops a sequence of approximate prices that converges to the correct price. An approximation involving 3 option evaluations is

$$\hat{P} = P_3 + \frac{7}{2}(P_3 - P_2) - \frac{1}{2}(P_2 - P_1)$$

P_1 is the ordinary (european) Black Scholes value

$$P_2 = Xe^{-r\frac{(T-t)}{2}} \mathcal{N}\left(-d_2\left(\overline{S}_{\frac{T-t}{2}}\right)\right) ....$$

$$P_3 = ...$$

where

$$d_1(q, \tau) =$$

$$d_2(q, \tau) =$$

The evaluations of P_1 and P_2 are easy, the problem is the evalutation of P_3, since it involves the evaluation of a trivariate normal cumulative distribution. For this see the notes on the normal distribution, but it is a nontrivial problem, involving some numerical integration.

Chapter 8

Futures algoritms.

In this we discuss algoritms used in valuing futures contracts.

8.1 Pricing of futures contract.

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

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

#include <cmath>

double futures_price(double S, // value of underlying
                     double r, // risk free interest
                     double time_to_maturity) {
    return exp(r*time_to_maturity)*S;
};
```

8.2 Options on futures

One of the more interesting problems with futures contracts is to value an option written on a futures contract.

8.2.1 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]

See [Hull, 1993, Ch 11.4] for a textbook treatment.

- F: futures price.
- X: exercise price.
- r: interest rate.
- σ: volatility
- $T-t$: time to maturity

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

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

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

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

Computer algoritm.

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

#include <cmath> // mathematics library
#include "normdist.h" // normal distribution

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

8.2.2 Binomial approximation.

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

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

#include <cmath>
#include <algorithm>
#include <vector>

double futures_option_price_call_american_binomial(
     double F, double X, double r, double sigma, double time, int no_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];
};
```

Chapter 9

Foreign Currency Algoritms

9.1 Foreign Currency Options

9.1.1 European options.

To value european options, all that is necessary is a slight adjustment of the Black-Scholes equation to adjust for the interest-rate differential.

- S: exchange rate.
- X: exercise price.
- r: interest rate in domestic currency.
- r_f: interest rate in foreign currency.
- σ: volatility of exchange rate.
- $T-t$: time to maturity.

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

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

where

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

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

Computer Algoritm, European Currency Option

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

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

double currency_option_price_call_european( double S, // exchange_rate,
                                            double X, // exercise,
                                            double r, // r_domestic,
                                            double r_f, // r_foreign,
                                            double sigma, // volatility,
                                            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);
};
```

References. The original formulations of European foreign currency option prices are in Garman and Kohlhagen [1983] and Grabbe [1983].

See [Hull, 1993, Ch 11.3] for a textbook treatment.

9.1.2 American options.

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

Computer Algoritm, Binomial Currency Option

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

#include <cmath>
#include <algorithm>
#include <vector>

double currency_option_price_call_american_binomial(
    double S, double X, double r, double r_f, double sigma,
    double time, 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]-X));
    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]-X); // check for exercise
        };
    };
    return call_values[0];
};
```

Chapter 10

Bond Algoritms.

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.

10.1 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.

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

#include <cmath>
#include <vector>

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)
  // calculate bond price when term structure is flat,
  // given both coupon and principals
{
    double p = 0;
    for (unsigned i=0;i<coupon_times.size();i++) {
        p += exp(-r*coupon_times[i])*coupon_amounts[i];
    };
    for (unsigned i=0;i<principal_times.size();i++) {
        p += exp(-r*principal_times[i])*principal_amounts[i];
    };
    return p;
};
```

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

#include <cmath>
#include <vector>

double bonds_price(const vector<double>& cashflow_times,
                   const vector<double>& cashflows,
                   const double& r) {
    // calculate bond price when term structure is flat,
    double p=0;
    for (unsigned i=0;i<cashflow_times.size();i++) {
        p += exp(-r*cashflow_times[i])*cashflows[i];
    };
    return p;
};
```

There are two version of the routine listed, one which is called with both interest and principal vectors, another which is simply called with the cashflows. I put up 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 and what is principal. But in the simple cases considered here we stick to the case of cashflows, it makes the routines easier to follow.

Let us also look at the case of discrete (annual) *compounding*:

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

#include <cmath>
#include <vector>

double bonds_price_discrete(const vector<double>& cashflow_times,
                            const vector<double>& cashflows,
                            const double& r) {
    // calculate bond price when term structure is flat,
    double p=0;
    for (unsigned i=0;i<cashflow_times.size();i++) {
        p +=cashflows[i]/(pow((1+r),cashflow_times[i]));
    };
    return p;
};
```

In the previous we assumed the term structure was *flat*. In most cases that is not a reasonable assumption, we need to use a general `term_structure` for discounting.

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

where $d(t)$ is a *function* that discounts a payment at time t to its value today. For example, a *flat* term structure has the following simple discount function:

$$d(t) = \frac{1}{(1+r)^t}.$$

For routines involving the term structure a `term_structure` class is assumed available.[1] For the purposes of the "algoritms paper," think of it as an abstract function that either return a `discount_factor` or a `yield`.

[1] The `term_structure` class is available in the same place this document is found.

```
// file bonds_price_termstru.cc
// author: Bernt Arne Odegaard

#include <vector>
#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;
};
```

10.2 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

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

The algoritm that follows 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."

```
// file bonds_yield.cc
// author: Bernt A Oedegaard
// calculate Yield to maturity for bond

#include <cmath>
#include "fin_algoritms.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 (fabs(diff)<ACCURACY) return r;
    if (diff>0.0) { bot=r;}
    else { top=r; };
    r = 0.5 * (top+bot);
  };
  return r;
};
```

10.3 Duration.

10.3.1 Simple duration.

$$D_1 = \sum \frac{tC(t)P(t)}{\sum C(t)P(t)}$$

where

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

First, case where term structure is flat

```
// file bonds_duration.cc
// author: Bernt Arne Odegaard

#include <cmath>
#include <vector>

double bonds_duration(const vector<double>& cashflow_times,
                      const vector<double>& cashflows,
                      const double& r)
// calculate the duration of a bond, simple case where the term
// structure is flat, interest rate r.
{
    double S=0;
    double D1=0;
    for (unsigned 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;
};
```

More general term structure:

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

#include <vector>
#include "term_structure_class.h"

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;
};
```

10.3.2 Macaulay Duration

$$D_2 = \sum \frac{tC(t)e^{-yt}}{\sum C(t)e^{-yt}}$$

where

- $C(t)$ is the cashflow in period t, and
- y is the (continous) yield to maturity.

```
// file bonds_duration_macaulay.cc
// author: Bernt A Oedegaard.
// calculate Macaulay duration.


#include "fin_algoritms.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
};
```

10.3.3 Modified Duration

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

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

#include <vector>
#include "fin_algoritms.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/y;
};
```

10.4 Convexity

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

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

```
// file bonds_convexity.cc
// author: Bernt A Oedegaard.
// calculate convexity of a bond.

#include <cmath>
#include <vector>

double bonds_convexity(const vector<double>& cashflow_times,
                       const vector<double>& cashflow_amounts,
                       const double& y )
{
    double C=0;
    for (unsigned i=0;i<cashflow_times.size();i++){
        double t = cashflow_times[i];
```

```
        C+= cashflow_amounts[i] * t * t * exp(-y*t);
    };
    return C;
};
```

Chapter 11

Exotic options.

With “exotic” options we think of any option with a more complicated payoff structure than the usual put and call payoff structures. There is virtually an unlimited number of possible exotic options available, only the imagination is the limit. In practice there is only a few that have seen much use, and we will look at some examples of these. For a large number of analytical solutions of exotic options, see Rubinstein [1993].

11.1 Lookback options.

The payoff from lookback options depend on the maximum or minimum of the underlying achieved through the period.

For this particular option an analytical solution has been found, due to Goldman et al. [1979], which is implemented below.

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

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

double exotics_lookback_european_call(double S,
                                      double Smin,
                                      double r,
                                      double q,
                                      double sigma,
                                      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));
};
```

An often used technique to estimate prices of various exotics is simulation, see the chapter on general simulations.

References: Chapter 16 of Hull [1993] has a short introduction to these options.

Chapter 12

A general approach to option pricing by simulation.

In an earlier chapter we looked at estimation of an European call option price by simulation.

Estimation by simulation is in fact completely general, and can be used to approximate the price on any European derivative. It is not feasible to use simulation on American derivatives, because the optimal exercise policy needs to be known. If the form of this was known, there would be a closed form solution.

But for European derivatives simulation is feasible. For more "exotic" options simulation is often the only solution.

I will show how we can make a general framework for simulation of derivatives prices.

12.1 Simulating prices of underlying.

Let us first look at how prices are simulated. We look at two cases. One where one only need the terminal price of the underlying at a given time. This is used for options that only depend on this, such as basic call and put options.

```
// file simulate_underlying_terminal.cc

#include <math.h>
#include "random.h"

double simulate_terminal_price(double S, // current value of underlying
                               double r, // interest rate
                               double sigma, // volatitily
                               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());
};
```

More generally we need to simulate price observations of the underlying at various times during the life of the derivative. This is what is implemented next.

```
// file: simulate_underlying_sequence.cc

#include <math.h>
#include <vector.h>
#include "random.h"

void simulate_price_sequence(double S, // current value of underlying
                             double r, // interest rate
                             double sigma, // volatitily
                             double time, // time to final date
                             int no_steps, // number of steps
                             vector<double>& prices ) {
   if (prices.size() < no_steps) {
      prices.clear();
      prices = vector<double>(no_steps);
   };
   double delta_t = time/no_steps;
   double R = (r-0.5*sigma*sigma)*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;
    };
};
```

12.2 Defining payoffs.

We next want to make general functions that defines the payoff of the derivative we analyze. Payoff is a function of either the terminal price or the full price history, and other parameters.

For example, the payoff for European call or put options are functions of the terminal price of the underlying and the exercise price of the option

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

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

The payoff is programmed as a function of two pieces of data, the terminal price and the exercise price.

```
// file payoff_black_scholes.cc

#include <algo.h>

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

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

As another example, let us look at options depending on the max or min of the price sequence. Here we have to pass the whole price history

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

```
// file payoff_max_min

#include <vector.h>
#include <algo.h>

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

double payoff_min(vector<double>& prices) {
    double m = *min_element(prices.begin(),prices.end());
    return prices.back()-m; // always positive or zero
};
```

One thing to note in this definition is the use of the function `max_element`. It returns the maximum of set of variables passed to it. It is part of the more recent additions to the `C++` language, the algorithms in the Standard Template Library.

Another example: Options on the average.

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

```
// file payoff_average.cc

#include <math.h>
#include <algo.h>
#include <vector.h>

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

double payoff_geometric_average(vector<double>& prices) {
    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-prices.back());
};
```

12.3 Pricing

We can now use the two previously defined routines to implement rather general simulation routines.

```
// file simulate_general.cc
// author: Bernt A Oedegaard
// estimation of an option price by simulation.
// This is a general routine, the payoff is defined by the user as a function
// payoff(vector<double>&) or
// payoff(vector<double>&,X),
// depending on whether there are other arguments beside the prices of the underlying needed.

#include <math.h>
#include <vector.h>
#include <algo.h>
#include "fin_algoritms.h"

double derivative_price_european_simulated ( double S, double X, double r, double sigma, double time,
                                             double payoff(double& price, double& X),
                                             int no_sims) {
    double sum_payoffs=0;
    for (int n=0; n<no_sims; n++) {
        double S_T = simulate_terminal_price(S,r,sigma,time);
        sum_payoffs += payoff(S_T,X);
    };
    return exp(-r*time) * (sum_payoffs/no_sims);
};

double derivative_price_european_simulated ( double S, double r, double sigma, double time,
                                             double payoff(vector<double>& prices),
                                             int no_steps, int no_sims) {
   double sum_payoffs=0;
   vector <double> prices(no_steps);
   for (int n=0; n<no_sims; n++) {
       simulate_price_sequence(S,r,sigma,time,no_steps,prices);
       sum_payoffs += payoff(prices);
   };
   prices.clear();
   return exp(-r*time) * (sum_payoffs/no_sims);
};

double derivative_price_european_simulated ( double S, double X, double r, double sigma, double time,
                                             double payoff(vector<double>& prices, double& X),
                                             int no_steps, int no_sims) {
   double sum_payoffs=0;
```

```
    vector <double> prices(no_steps);
    for (int n=0; n<no_sims; n++) {
        simulate_price_sequence(S,r,sigma,time,no_steps,prices);
        sum_payoffs += payoff(prices,X);
    };
    prices.clear();
    return exp(-r*time) * (sum_payoffs/no_sims);
};
```

12.4 Improving the estimates, control variates.

To improve on the simulation, we can use the *control variate* technique.

```
// file simulate_general_control_variate.cc

#include "fin_algoritms.h"
#include <math.h>

double derivative_price_european_simulated_control_variate (
    double S,
    double r,
    double sigma,
    double time,
    double payoff(vector<double>& prices),
    int no_steps,
    int no_sims)
{
     double X = S;
     double c_bs = option_price_call_black_scholes(S,X,r,sigma,time);
     vector <double> prices(no_steps,0.0);
     double sum_payoffs=0;
     double sum_payoffs_bs=0;
     for (int n=0; n<no_sims; n++) {
         simulate_price_sequence(S,r,sigma,time,no_steps,prices);
         sum_payoffs += payoff(prices);
         sum_payoffs_bs += payoff_european_call(prices.back(),X);
     };
     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;
};
```

Chapter 13

Alternatives to the Black Scholes type option formula.

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.

13.1 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

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

```
// file merton_jump_diff_call.cc
// author: Bernt Arne Oedegaard

#include <cmath>
#include "fin_algoritms.h"

double option_price_call_merton_jump_diffusion( double S,
                                                double X,
                                                double r,
                                                double sigma,
                                                double time_to_maturity,
```

```
                                    double lambda,
                                    double kappa,
                                    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,sigma,r-lambda*kappa,tau);
    double log_n = 0;
    for (int n=1;n<=MAXN; ++n) {
        log_n += log(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,sigma_n,r_n,tau);
    };
    return c;
};
```

13.2 Stochastic volatity.

One of the areas of continuing research is how to adjust for changing volatility in the option price formulas.

Chapter 14

Mean Variance Analysis.

14.1 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.

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

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:

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

```
// file mv_calc.cc
// author: Bernt A Oedegaard
// basic calculations for mean variance analysis

#include "newmat.h"
#include <cmath>

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);
};
```

14.2 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"

ReturnMatrix mv_calculate_portfolio_given_mean_unconstrained(const Matrix& e,
                                                              const Matrix& V,
                                                              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;
};
```

14.3 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

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

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

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

To actually put the full code for a quadratic program here is beyond the purpose of these notes. In the following I therefore submit the program to a quadratic solver called `cfsqp`. At times is it best to trust in the OR department and their codes. The code merely indicates what the typical steps are in submitting this to a general optimization routine. All we have to do is to define an objective function and a set of constraints as `C` subroutines that can be interfaced with an optimizer.

```
// file mv_short.cc
// author: Bernt Arne Oedegaard
// calculate a mean variance portfolio under short sales constraints
#include "mv_calc.h"
#include "cfsqp_op.h" // optimize using CFSQP
#include "mymatrix.h" //

int MV_SHORT_no_assets; // global values because they need to be
double MV_SHORT_r; // available in criterion function
Matrix* MV_SHORT_e;
Matrix* MV_SHORT_V;

double MV_SHORT_objective(double x[]){
    Matrix X(MV_SHORT_no_assets,1);
    for (int i=0;i<MV_SHORT_no_assets;++i){
        X.element(i,0)=x[i];
    };
    return mv_calculate_variance(*MV_SHORT_V,X);
}

double MV_SHORT_constraints(double x[],int constraint_no){
    Matrix X(MV_SHORT_no_assets,1);
    for (int i=0;i<MV_SHORT_no_assets;++i){
        X.element(i,0)=x[i];
    };
    if (constraint_no==1) { // constraint w'1=1
        Matrix A = (X.t()*ones_matrix(MV_SHORT_no_assets,1));
        return A.element(0,0)-1.0;
    };
    if (constraint_no==2) { // constraint w'e=r
        Matrix A = (X.t()*(*MV_SHORT_e));
        return A.element(0,0)-MV_SHORT_r;
    };
    return 0;
};

ReturnMatrix mv_calculate_portfolio_given_mean_no_short_sales( Matrix& e,
                                                                Matrix& V,
                                                                double r){
    // solve the quadratic program to minimize variance given constraints
    // all weights are in [0,1]
    // the weights sum to one
    // expected return equals r
  MV_SHORT_no_assets = e.Nrows();
  MV_SHORT_V = &V;
  MV_SHORT_e = &e;
  MV_SHORT_r = r;
  Matrix w = null_matrix(MV_SHORT_no_assets, 1);
  // check whether it is feasible to find optimum. Only if r is between
  // min and max in e;
  double r_min = e.element(0,0);
  double r_max = e.element(0,0);
  int i_min = 0;
  int i_max = 0;
  int i;
```

```
  for (i=1;i<MV_SHORT_no_assets;++i) {
    if (e.element(i,0)<r_min) {
      r_min = e.element(i,0);
      i_min = i;
    };
    if (e.element(i,0)>r_max) {
      r_max = e.element(i,0);
      i_max = i;
    };
  };
  if ( (r>r_max) || (r<r_min) ) {
    cerr << " can not perform optimization " << endl;
    w.Release(); // returning zeros as weight
    return w;
  };
  double *x = new double[MV_SHORT_no_assets];
  double *x_min = new double[MV_SHORT_no_assets];
  double *x_max = new double[MV_SHORT_no_assets];
  for (i=0;i<MV_SHORT_no_assets;++i) {
    x[i] = 0;
    x_min[i] = 0;
    x_max[i] = 1;
  };
  // now generate feasible portfolio by combining r_max and r_min;
  double lambda = (r-r_max)/(r_min-r_max);
  x[i_min]=lambda;
  x[i_max]=(1-lambda);
  double min_var = CFSQP_optimize_with_constraints_and_bounds(
                                                  x,
                                                  MV_SHORT_no_assets,
                                                  MV_SHORT_objective,
                                                  0,0,0,2, // 2 linear equality constraints
                                                  MV_SHORT_constraints,
                                                  x_min,
                                                  x_max);
  for (i=0; i<MV_SHORT_no_assets; ++i) {
    w.element(i,0)=x[i];
  };
  delete [] x;
  delete [] x_min;
  delete [] x_max;
  w.Release();
  return w;
};
```

Chapter 15

Term Structure algorithms.

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.

15.1 Term structure calculations.

Some useful transformations. 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

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

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

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

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

The forward rate can also be calculated directly from yields as

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

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

#include "math.h"

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

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

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

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

15.2 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 yields.

15.2.1 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 both yields and forward rates.

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

```
// file lin_intp.cc
// author: Bernt A Oedegaard.
// given a set of yields, produce a term structure by linear interpolation.

#include "fin_algoritms.h"

double term_structure_yield_linearly_interpolated(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;
};
```

As an alternative to

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

```
// file lin_intp.cc
// author: Bernt A Oedegaard.
// given a set of forward yields, produce a term structure by linear interpolation.

#include "fin_algoritms.h"

double term_structure_forward_linearly_interpolated(double time,
                                                    const vector<double>& obs_times,
                                                    const vector<double>& obs_forwards) {
// assume observations 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_forwards[0]; // earlier than lowest obs.

    double t_max = obs_times[no_obs-1];
    if (time >= t_max) return obs_forwards[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_forwards[t-1] * lambda + obs_forwards[t] * (1.0-lambda);
    return r;
};
```

15.3 Term structure approximations.

Instead of explicitly modeling the term structure, one may want to approximate it by a *flexible functional form*. The following three are examples of this, proposed in respectively Nelson and Siegel [1987], Bliss [1989] and McCulloch [1971]. The first two model the *yield*, McCulloch model the discount factor (or bond price) directly.

15.3.1 The Nelson and Siegel(1987) functional form.

$$r(s) = \beta_0 + (\beta_1 + \beta_2)\left[\frac{1 - e^{-\frac{t}{\lambda}}}{\frac{t}{\lambda}}\right] + \beta_2\left[e^{-\frac{t}{\lambda}}\right]$$

```
// file nels_sie.cc
// author: Bernt A Oedegaard
// purpose: Calculate the term structure proposed by Nelson and Siegel
// Parsimonious Modeling of Yield Curves, Journal of Business, (1987)

#include <math.h>

double term_structure_yield_nelson_siegel(double t,
                                          double beta0, double beta1, double beta2,
                                          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;
};
```

15.3.2 Bliss (1989)

In Bliss [1989] a further development of Nelson and Siegel [1987] was proposed.

$$r(t) = \gamma_0 + \gamma_1\left[\frac{1 - e^{-\frac{t}{\lambda_1}}}{\frac{t}{\lambda_1}}\right] + \gamma_2\left[\frac{1 - e^{-\frac{t}{\lambda_2}}}{\frac{t}{\lambda_2}} - e^{-\frac{t}{\lambda_2}}\right]$$

This has 5 parameters to estimate: $\{\gamma_0, \gamma_1, \gamma_2, \lambda_1, \lambda_2\}$.

```
// file termstru_yield_bliss.cc
// author: Bernt A Oedgaard

#include <math.h>

double term_structure_yield_bliss(double t, double gamma0, double gamma1, double gamma2,
                                  double lambda1, double lambda2) {
    double r;
    double t1 = t/lambda1;
    double t2 = t/lambda2;
    r = gamma0
        + gamma1 * ( (1-exp(-t1)) / t1 )
```

```
        + gamma2 * ( ( 1-exp(-t2)) / t2 )
        + gamma2 * ( -exp(-t2));
    return r;
};
```

15.3.3 Cubic spline.

The cubic spline parameterization was first used by McCulloch [1971] to estimate the nominal term structure. He later added taxes in McCulloch [1975]. The cubic spline was also used by Litzenberger and Rolfo [1984].

$$d(t) = 1 + b_1 t + c_1 t^2 + d_1 t^3 + \sum_{j=1}^{K} F_j (t - t_j)^3 1_{\{t<t_j\}}$$

Here $1_{\{A\}}$ is the indicator function for an event A, and we have K *knots*.

To estimate this we need to find the $3+K$ parameters:

$$\{b_1, c_1, d_1, F_1, \cdots, F_K\}$$

If the spline *knots* are known, this is a simple linear regression.

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

#include <math.h>
#include <vector.h>

double term_structure_discount_factor_cubic_spline(double t,
                                                   double b1,
                                                   double c1,
                                                   double d1,
                                                   const vector<double>& f,
                                                   const vector<double>& knots)
{
    // calculate the discount factor for the spline functional form.
    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;
};
```

15.4 Term structure models.

The previous section discussed methods that can be viewed as ways of nonparametrically estimating the term structure *function*. We next look at *economic* models of the term structure.

15.4.1 The Vasicek model.

Proposed by Vasicek [1977].

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

#include "math.h"

double term_structure_discount_factor_vasicek(double time,
                                               double r,
                                    double a, double b, 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;
}
```

15.4.2 The original Cox Ingersoll Ross model.

The term structure model described in Cox et al. [1985] is one of the most commonly used in academic work, because it is a general equilibrium model that still is "simple enough" to let us find closed form expressions for derivative securities.

Calculating discount factors. The short interest rate.

$$dr(t) = \kappa(\theta - r(t))dt + \sigma\sqrt{r(t)}dW$$

The discount factor for a payment at time T.

$$d(t,T) = A(t,T)e^{-B(t,T)r(t)}$$

where

$$\gamma = \sqrt{(\kappa+\lambda)^2 + 2\sigma^2}$$

$$A(t,T) = \left[\frac{2\gamma e^{\frac{1}{2}(\kappa+\lambda+\gamma)(T-t)}}{(\gamma+\kappa+\lambda)(e^{\lambda(T-t)}-1)+2\gamma}\right]^{\frac{2\kappa\theta}{\sigma^2}}$$

and

$$B(t,T) = \frac{2e^{\gamma(T-t)}-1}{(\gamma+\kappa+\lambda)(e^{\lambda(T-t)}-1)+2\gamma}$$

Five parameters

1. r – The short term interest rate.
2. κ – The mean reversion parameter.
3. λ – "market" risk parameter.
4. θ – the long–run mean of the process.
5. σ – the variance rate of the process.

Computer algorithm.

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

#include <math.h> // mathematics library

double term_structure_discount_factor_cir(double t,
                                          double r,
                                          double kappa,
                                          double lambda,
                                          double theta,
                                          double sigma)
    // this is the original CIR formulation of their term structure.
{
    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;
};
```

15.4.3 The estimated CIR model used by Brown and Dybvig.

If the Cox et al. [1985] model is estimated from a cross-section of bond prices, all the parameters are not identifiable from the data. As shown in Brown and Dybvig [1986], it is only possible to separately identify four factors $\{r, \phi_1, \phi_2, \phi_3\}$.

$$d_t(s) = A(s)e^{-B(s)r(t)}$$

$$\begin{aligned}
A_t(s) &= \left[\frac{\phi_1 e^{\phi_2 s}}{\phi_2(e^{\phi_1 s}-1)+\phi_1}\right]^{\phi_3} \\
B_t(s) &= \frac{e^{\phi_1 s}-1}{\phi_2(e^{\phi_1 s}-1)+\phi_1} \\
\phi_1 &= \sqrt{(\kappa+\lambda)^2+2\sigma^2} \\
\phi_2 &= (\kappa+\lambda+\phi_1)/2 \\
\phi_3 &= 2\kappa\theta/\sigma^2
\end{aligned}$$

Discount factor.

```
// file esti_cir.cc
// author Bernt Arne Oedegaard

#include <math.h>

double term_structure_discount_factor_estimated_cir( double t, // time to maturity.
                                                     double r, // short interest rate.
                                                     double phi1,
                                                     double phi2,
                                                     double phi3)
```

```
{
  double tmp = (phi2*(exp(phi1*t)-1.0)+phi1);
  double A = (phi1*exp(phi2*t))/tmp;
  A = pow(A,phi3);
  double B = (exp(phi1*t)-1.0)/tmp;
  double dfact = A*exp(-B*r);
  return dfact;
}
```

Chapter 16

Fixed Income modelling, with emphasis on contingent claims.

In this part we look at more advanced modelling related to derivatives written on fixed–income securities.

16.1 Black Scholes bond pricing.

The Black Scholes model can be used under restrictive assumptions, but the constant volatility assumption of the bond price is unrealistic.

Computer algoritm, Bond option price, Black Scholes

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

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

double bond_option_price_call_zero_black_scholes(
     double B, double X, double r, double sigma, 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 c = B * N(d1) - X * exp(-r*time) * N(d2);
     return c;
};
```

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

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

double bond_option_price_call_coupon_bond_black_scholes(
    double B, double X, double r, double sigma, double time,
     vector<double> coupon_times, vector<double> coupon_amounts){
     for (unsigned int i=0;i<coupon_times.size();i++) { // subtract present value of coupons
        if (coupon_times[i]<=time) { // coupon paid befor option expiry
           B -= coupon_amounts[i] * exp(-r*coupon_times[i]);
     };
     };
     return bond_option_price_call_zero_black_scholes(B,X,r,sigma,time);
};
```

16.2 The Rendleman and Bartter model

The Rendleman and Bartter approach to valuation of interest rate contingent claims 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, which has implications for how we do discounting.

I will follow the notation used in Hull [1993] (page 385).

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

#include <cmath>
#include <algorithm>
#include <vector>

double bond_option_price_call_zero_american_rendleman_bartter(double X,
                                                              double option_maturity,
                                                              double S,
                                                              double M, // term structure paramters
                                                              double interest, // current short interest rate
                                                              double bond_maturity, // time to maturity for
underlying bond

                                                              double maturity_payment,
                                                              int no_steps)
// call on a zero coupon bond.
{
    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;
    int i;
    for (i=1;i<=no_steps;++i){ r[i]=r[i-1]*uu;};
    vector<double> P(no_steps+1);
    for (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 (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 (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 (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];
};
```

Sources for further readings. Section 15 of Hull [1993] has a short discussion.

The original references are Rendleman and Bartter [1979] and Rendleman and Bartter [1980].

16.3 Vasicek bond pricing.

If the term structure model is Vasicek's model there is a solution for the price of a zero coupon model, 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 $a=0$,

$$v(t,T) = \sigma\sqrt{T-t}$$

$$\sigma_P = \sigma(s-T)\sqrt{T-t}$$

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

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

double bond_option_price_call_zero_vasicek(double X, // exercise price
                                           double r, // current interest rate
                                           double option_time_to_maturity,
                                           double bond_time_to_maturity,
                                           double a, // parameters
                                           double b,
                                           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;
};
```

Appendix A

Normal Distribution approximations.

Numerical approximations to the cumulative normal distribution, univariate and bivariate.

Distribution function.

$$n(z) = e^{\frac{1}{2}z^2}$$

If x is normally distributed with mean zero and variance 1, then

$$P(x < z) = N(z) = \int_{-\infty}^{z} n(x)dx$$

Let x, y be bivariate normally distributed, each with mean 0 and variance 1. The correlation between x and y is ρ, $\rho \in [-1, 1]$. Then

$$\begin{aligned} P(x < a, y < b) &= N(a, b, \rho) \\ &= \int_{-\infty}^{a} \int_{-\infty}^{b} \frac{1}{2\pi\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2}\frac{x^2 - 2\rho xy + y^2}{1-\rho^2}\right) dxdy \end{aligned}$$

```
// normdist.h
// author: Bernt A Oedegaard

#ifndef _NORMAL_DIST_H_
#define _NORMAL_DIST_H_

double n(double z); // normal distribution function
double n(double r,double mu, double sigmasqr); // normal distribution function
double N(double z); // cumulative probability of normal
double N(double a, double b, double rho); // cum prob of bivariate normal

#endif
```

```
// normdist.cc
// numerical approximations to univariate N(z) and bivariate N(a,b,rho)
// normal distributions

#include <cmath> // math functions.

#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);
};

double n(double r, double mu, double sigma) { // normal distribution function
    double nv = 1.0/(sqrt(2.0*PI)*sigma);
    double z=(r-mu)/sigma;
    nv *= exp(-0.5*z*z);
    return nv;
};
```

```
// cumulative univariate normal distribution.
// This is a numerical approximation to the normal distribution.
// See Abramowitz and Stegun: Handbook of Mathemathical functions
// for description. The arguments to the functions are assumed
// normalized to a (0,1 ) distribution.

double N(double z) {
    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;

    if (z > 6.0) { return 1.0; }; // this guards against overflow
    if (z < -6.0) { return 0.0; };
    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;
}

// Numerical approximation to the bivariate normal distribution,
// as described e.g. in Hulls book

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) {
    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
};
```

A.1 References

Abramowiz and Stegun [1964] gives the univariate normal approximation.

This particular implementation of the bivariate normal is from [Hull, 1993, Ch 10]. See there for further references.

Appendix B

A note on C++ and the source code

B.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.

B.2 Libraries.

One of the major advances of `C++` relative to other programming languages is its ability to use libraries to extend the language. While I have tried to avoid use of libraries, in a few places I have used a library that performs linear algebra. This is mainly for clarity, the algorithms can be cleanly and simply expressed in vector notation, and implementing them directly would involve long, unnecessary loops, which has the added disadvantage of most likely introducing trivial mistakes. Using a well-designed linear algebra library will also in most cases be much more efficient than writing code for equation solving yourself.

The linear algebra library that is used in the algorithms is called `newmat`, which is a public domain library. The current version is found as a file `newmat09.zip` or `newmat09.tar.gz` on major ftp archives. Search using `archie`. The usage in the algorithms should be pretty clear from the context, and other linear algebra `C++` libraries will have similar notation, but here are some examples:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix}$$

Defining the variables:
```
Matrix A;
ColumnVector b;
```

Calculations:
Take inverse, A^{-1}: `Matrix Ainv = A.i();`
Matrix multiply, $\mathbf{C} = \mathbf{A} \times \mathbf{b}'$: `Matrix C = A * b.t();`
Solve system, $\mathbf{Ax} = \mathbf{b}$: `Matrix x = A.i()*b;`

For linear algebra, it is possible to take advantage of particular forms of matrices, such as banded matrices, diagonal matrices etc. `C++` is particularly useful here, a good linear algebra library will choose efficient algorithms depending on the form of the matrices, *without* the need for particular choice of subroutines by the programmer, the choice of algorithm is purely made from context. For an example of this, look at the algorithm for option pricing using implicit finite differences.

B.3 On `C++` for non-`C++` programmers

In some of the algorithms we use a few things that is not obvious if you don't know the `C/C++` language.

B.4 Mathematical operators.

B.4.1 Exponentiation.

The usual operators ($+, -, *, /$) are what you expect, but exponentiation is not a part of the language, these and other mathematical operators are implemented as functions in the math library

- `pow(x,n)` calculates x^n.
- `exp(r)` calculates e^r.

B.4.2 Increment and decrement.

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, 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.

B.5 Dynamically sized arrays.

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. In addition to the problem of ensuring that we did not exceed the length, this is also wasteful in terms of space.

In the latest version of the C++ standard, a library called the Standard Template Library is included. One of the most useful parts of this library is the `vector<>` template, which gets rid of the "bookkeeping" involved in the previous array manipulations. In most of the routines I use a vector of doubles instead of an array, because this reduces the numbers of parameters needed to pass.

B.6 The form of the `for` statement.

The `for` statement in `C` (and `C++`) is more general than similar constructs in `FORTRAN` and `Pascal`. To repeat an operation `n` times, use:

```
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, but 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 here I try to only use the typical for statement, avoiding any obfuscated ones.

B.7 The #include construct.

For the compiler to use a function, it must be defined before it is used. The solution in C/C++ is to use *header files* that defines functions and macros.

For example, in many of the formulas we use in the text, the cumulative normal distribution function (N(z)) enters. This function is declared in a file normdist.h that looks like the following.

```
// normdist.h
// author: Bernt A Oedegaard

#ifndef _NORMAL_DIST_H_
#define _NORMAL_DIST_H_

double n(double z); // normal distribution function
double n(double r,double mu, double sigmasqr); // normal distribution function
double N(double z); // cumulative probability of normal
double N(double a, double b, double rho); // cum prob of bivariate normal

#endif
```

This header file declares two versions of the normal distribution, the univariate and the bivariate.

When N() is called with one argument, it is the univariate distribution that is referenced.

N(0.0) should return 0.5

When N() is called with three arguments, it is the bivariate distribution which is referenced.

N(0.0, 0.0, 0.0) should return 0.25

To make these function available in a subroutine, use the include statement to define them. Assume the header file normdist.h holds the defintions. The following is a complete program.

```
#include "normdist.h"
int main(){
      cout << " Bivariate cum.  normal dist (0,0,0) = "
                        << N(0,0,0) << endl;
}
```

When compiled, linked and run, the program should print

```
Bivariate cum.  normal dist (0,0,0) = 0.25
```

B.8 The class concept.

B.9 Implementation in other programming languages.

The main problem in converting the algorithms presented here into FORTRAN77 and Pascal is the use of variable–length arrays of numbers in subroutines, and in function calls. You have to come up with a different

solution in those languages, which probably would involve fixed length arrays of some default size. For an alternative, look at how this is handled in Numerical Recipes in the respective languages.

Translating into both Mathematica and Matlab should be a trivial.

B.10 References.

For a good introductory book on C++, see Lippman [1992].

The main source on the language is Stroustrup [1991].

For the STL see Musser and Saini [1996]

Appendix C

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
Michael L Locher
Lars Gregori
Steve Bellantoni <scj@cs.toronto.edu>

Index

γ, 10
ρ, 11
θ, 10

Barone–Adesi and Whaley, 33
binomial option price, 18
Black
  futures option, 37
bond, 41
  price, 41
  yield, 43
bond option
  Black Scholes, 65
  Vasicek, 66
Brown and Dybvig, 63

cash flow, 5
CIR
  estimated, 63
class, 74
Cox Ingersoll Ross, see CIR, 63
currency, 39
  option, 39
currency option
  American, 40
  European, 39

delta, 10
  binomial, 21
duration, 43
  Macaulay, 44
  modified, 45
  simple, 43

early exercise premium, 33
exotic option, 47
exp() (`C++ statement`), 73
explicit finite differences, 26, 28

finite differences, 26
  American, 27
  explicit, 26, 28
  implicit, 26, 27
for (`C++ statement`), 73
futures
  option, 37

gamma, 10
geometric Brownian motion, 8
Geske and Johnson, 35

hedging parameters
  Black Scholes, 10

implicit finite differences, 26, 27
implied volatility
  calculation, 11
include (`C++ statement`), 74
internal rate of return, 6
introduction, 4
irr, 6
  unique, 7

Jump Diffusion, 52

libraries
  programming, 72
lookback option, 47

Merton
  Jump Diffusion, 52
modified duration, 45

np, 5

option
  currency, 39
  exotic, 47
  futures, 37
  lookback, 47
option price
  binomial, 18
  simulated, 30

partial derivatives
  binomial, 20
  Black Scholes, 11
partials
  Black Scholes, 10
pow() (`C++ statement`), 73
present value, 5

quadratic approximation, 33

rho, 11

simulation, 30
  general, 48

theta, 10

unique irr, 7

vega, 11
volatility
  implied, 11

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 M S Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 7:637–54, 1973.

Robert R Bliss. Fitting term structures to bond prices. Working paper, University of Chicago, January 1989.

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, fourth 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.

Stephen J Brown and Philip H Dybvig. The empirical implications of the Cox, Ingersoll, Ross theory of the term structure of interest rates. *Journal of Finance*, 41:617–32, 1986.

J Cox, S Ross, and M Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7:229–263, 1979.

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.

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.

Robert Geske and H E Johnson. The american put valued analytically. *Journal of Finance*, XXXIX(5), December 1984.

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, third edition, 1997.

F Jamshidan. An exact bond option pricing formula. *Journal of Finance*, 44:205–9, March 1989.

Stanley B Lippman. *C++ primer.* Addison–Wesley, 2 edition, 1992.

Robert H. Litzenberger and Jaques Rolfo. An international study of tax effects on government bonds. *Journal of Finance*, 39:1–22, 1984.

J Houston McCulloch. Measuring the term structure of interest rates. *Journal of Business*, 44:19–31, 1971.

J Houston McCulloch. The tax adjusted yield curve. *Journal of Finance*, 30:811–829, 1975.

Robert C Merton. The theory of rational option pricing. *Bell Journal*, 4:141–183, 1973.

David R Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ programming with the standard template library.* Addison–Wesley, 1996.

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. Exotic options. University of California, Berkeley, working paper, 1993.

William F Sharpe and Gordon J Alexander. *Investments*. Prentice–Hall, 4 edition, 1990.

Bjarne Stroustrup. *The C++ Programming language*. Addison–Wesley, 2 edition, 1991.

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.