Java Methods for Financial Engineering
Java Methods for Financial Engineering
Applications in Finance and Investment
To my wife Avril
Whose support, encouragement and patience
made this book possible
Contents

Preface ................................................................. xi

1. Introduction ..................................................... 1
   1.1. Numerical Accuracy & Errors ..................................... 1
   1.2. Core Math’s Classes ................................................... 2
      1.2.1. Root Finding - Interval Bisection .......................... 2
      1.2.2. Newton’s Method .................................................. 7
   1.3. Statistical Classes .................................................... 12
      1.3.1. Measures of Dispersion ....................................... 13
   1.4. Application Classes .................................................. 17
      1.4.1. Internal Rate of Return ........................................ 18
      1.4.2. Deriving Yield Approximations – Bisection Method ... 18
      1.4.3. Deriving Yield Approximations
             the Newton Raphson Method ................................... 21
      1.4.4. Portfolio Management ......................................... 21
      1.4.5. Portfolio Risk Measurement .................................... 28

2. Interest Rate Calculations ...................................... 33
   2.1. Compound Interest ................................................... 33
      2.1.1. Nominal and Effective Interest ............................... 33
   2.2. Present Value (PV) ..................................................... 37
      2.2.1. Compounding Cashflows ...................................... 37
      2.2.2. Perpetuity and Annuity ........................................ 39
   2.3. Internal Rate of Return .............................................. 40
   2.4. Term Structures ....................................................... 41
      2.4.1. Rate Interchanges ................................................ 41
      2.4.2. Spot Rates ......................................................... 43
      2.4.3. Deriving the Spot Curve ....................................... 49

3. Bonds ................................................................. 59
   3.1. Bonds – Fixed Interest ............................................. 59
   3.2. Bond Prices ............................................................. 68
      3.2.1. Interest Yields ..................................................... 69
      3.2.2. Yield to Maturity ................................................ 69
   3.3. Static Spread ............................................................ 70
   3.4. Credit Spreads .......................................................... 74
3.5. Bond Volatility Measures........................................................... 81
  3.5.1. Price Value of a Point.................................................... 85
3.6. Bond Pricing Characteristics.................................................. 88

4. Duration ....................................................................................... 99
  4.1. Macaulay Duration ............................................................. 99
  4.2. Effective Duration............................................................... 105

5. Futures.......................................................................................... 107
  5.1. Forward & Futures Pricing.................................................... 108
  5.2. Forward Price................................................................. 112
  5.3. Pricing On Different Markets............................................... 115
      5.3.1. Stock Index ................................................................. 115
      5.3.2. Currencies................................................................. 117
  5.4. Commodity Futures.............................................................. 118

6. Options.......................................................................................... 125
  6.1. Option Types.......................................................................... 125
  6.2. Option Specifications........................................................... 127
  6.3. Pricing Specification............................................................. 128
      6.3.1. Dividends and Stock Splits........................................ 130
      6.3.2. Option Quotes............................................................. 131
      6.3.3. Margin Accounts......................................................... 132
  6.4. Arbitrage in Option Prices..................................................... 133
      6.4.1. Main Components of Pricing.................................... 133
      6.4.2. Limits for Pricing......................................................... 136
  6.5. Early Exercise of American Options...................................... 141
  6.6. Option Convexity................................................................. 143
  6.7. Put Call Parity......................................................................... 145
  6.8. Strategies............................................................................... 149
      6.8.1. Hedge with a Protected Put ...................................... 150
      6.8.2. Reverse Protected Put Hedge.................................... 150
      6.8.3. Hedge with a Covered Call........................................ 150
      6.8.4. Reverse Covered Call Hedge..................................... 150
  6.9. Profit Diagrams...................................................................... 155

7. Modelling Stock Prices............................................................... 163
  7.1. The Stochastic Process.......................................................... 163
      7.1.1. Random Walks........................................................... 163
      7.1.2. Brownian Motion......................................................... 164
      7.1.3. Wiener Process............................................................ 164
      7.1.4. Ito Differential......................................................... 168
  7.2. Lognormal Modelling of Stock Prices.................................... 170
      7.2.1. Handling Empirical Data.......................................... 171
      7.2.2. Simulation with Monte Carlo.................................... 177
  7.3. The Lognormal Property....................................................... 182
8. The Binomial Model ......................................................... 191
  8.1. Stock Price .................................................................................. 191
    8.1.1. Cox Ross Rubinstein (CRR) Model .................................. 192
    8.1.2. Binomial Tree ................................................................ 193
  8.2. Trees for American & European Pricing ............................. 201

9. Analytical Option Pricing Methods ............................................ 205
  9.1. Black-Scholes-Merton ............................................................... 205
  9.2. Pricing with Black-Scholes ....................................................... 207
    9.2.1. Pricing without Dividends ............................................. 208
    9.2.2. Effects of Dividends ...................................................... 209
    9.2.3. Options Paying a Yield .................................................. 210
    9.2.4. Stock Index Options ..................................................... 211
    9.2.5. Options on Futures ....................................................... 212
    9.2.6. Currency Options ......................................................... 213
  9.3. Analytical Approximations for American Options ............... 216
    9.3.1. Roll Geske Whaley (RGW) Approximation ................. 216
    9.3.2. Bjerksund and Stensland (B&S) Approximation ............ 220
    9.3.3. Quadratic Approximation (Barone-Adesi Whaley Derivation) ........................................................ 223

10. Sensitivity Measures (The ‘Greeks’) ......................................... 231
  10.1. The Black-Scholes Pde ............................................................. 231
  10.2. Delta Sensitivity ........................................................................ 231
  10.3. Gamma Sensitivity .................................................................... 234
  10.4. Theta Sensitivity ....................................................................... 238
  10.5. Vega Sensitivity ....................................................................... 241
  10.6. Rho Sensitivity ....................................................................... 245
  10.7. Option Extensions .................................................................... 247
    10.7.1. Elasticity ......................................................................... 247
    10.7.2. Cost of Carry .................................................................... 249

11. Interest Rate Derivatives ............................................................. 255
  11.1. Market Price of Risk ............................................................... 255
  11.2. Martingales ............................................................................ 257
  11.3. Interest Rate Caps & Floors .................................................... 260
  11.4. Swap Options ........................................................................ 262
    11.4.1. Adjusting Rates for Convexity ...................................... 265
    11.4.2. Zero Coupon Bond as the Asset .................................... 266
    11.4.3. Valuation of Bond Options ........................................ 267
  11.5. Short Rate Modelling ............................................................... 269
    11.5.1. Rendleman and Bartter ................................................. 270
    11.5.2. The Vasicek Model ....................................................... 270
    11.5.3. Cox Ingersoll Ross
      (C.I.R) Model ...................................................................... 272
11.6. Arbitrage Free Models ................................................................. 272
  11.6.1. The Ho and Lee Model ....................................................... 273
  11.6.2. Hull and White Model ......................................................... 274

12. Conditional Options ................................................................. 279
  12.1. Executive Stock Options ........................................................ 279
    12.1.1. Forward Start Option ..................................................... 282
    12.1.2. Indexed Stock Options .................................................. 284
  12.2. Time Switch Option ............................................................... 288
  12.3. Chooser Option ..................................................................... 290
    12.3.1. Simple Chooser ............................................................. 292
    12.3.2. Complex Chooser Options ......................................... 296
  12.4. Options on Options ............................................................. 300
    12.4.1. Call on Call ................................................................. 301
    12.4.2. Put on Call ................................................................. 302
  12.5. Extendible Options .............................................................. 306
    12.5.1. Extendible Call ............................................................ 306
    12.5.2. Extendible Put ............................................................ 308
  12.6. Writer Extendible ................................................................. 313
  12.7. Rainbow Options .................................................................. 316
    12.7.1. Two Asset Correlated .................................................. 316
    12.7.2. Exchange Assets Option ............................................... 318
    12.7.3. American Exchange Option ......................................... 321
  12.8. Sequential Exchange Options ............................................... 323

13. Complex Conditional Options .................................................. 337
  13.1. Fixed Look Back Options ..................................................... 337
    13.1.1. Fixed Strike Lookback
      Call Option ........................................................................ 337
    13.1.2. Fixed Strike Lookback Put ........................................... 339
  13.2. Floating Strike Look Back Options ....................................... 343
    13.2.1. Floating Strike Lookback Put ...................................... 343
    13.2.2. Floating Strike Lookback Call ..................................... 344
  13.3. Partial Time Fixed Strike Lookback ...................................... 347
    13.3.1. Partial Time Fixed Strike Call .................................... 347
    13.3.2. Partial Time Fixed Strike Put .................................... 348
  13.4. Partial Time Floating Strike Lookback .................................. 353
    13.4.1. Partial Time Floating Strike Call .................................. 353
    13.4.2. Partial Time Floating Strike Put .................................. 356
  13.5. Min or Max of Two Risky Assets ......................................... 358
    13.5.1. Minimum of Two Risky Assets .................................... 358
    13.5.2. Maximum of Two Risky Assets .................................... 359
  13.6. Spread Option Approximation .............................................. 362
    13.6.1. Analytical Spread Approximation ............................... 362
13.7. Extreme Spreads ............................................................... 370
13.7.1. Extreme Spread ............................................................. 371
13.7.2. Reverse Extreme Spread .............................................. 371
13.8. Value or Nothing Options .................................................. 373
13.8.1. Cash-or-Nothing Option ................................................ 373
13.8.2. Asset-or-Nothing Option ............................................... 378

14. Barrier Type Options .......................................................... 383
14.1. In Barrier Valuation .......................................................... 383
14.1.1. Valuation with a Rebate ............................................... 385
14.1.2. Down and In Call Valuation ......................................... 386
14.1.3. Up and In Call Valuation ............................................. 391
14.1.4. Down and In Put Valuation ......................................... 391
14.1.5. Up and In Put Valuation ............................................. 393
14.2. Out Barrier Valuation ....................................................... 394
14.2.1. Down and Out Call Valuation ....................................... 395
14.2.2. Up and Out Call Valuation .......................................... 397
14.2.3. Down and Out Put Valuation ....................................... 397
14.2.4. Up and Out Put Valuation .......................................... 400

15. Double Barrier Options ..................................................... 407
15.1. Double Knock In/Out ........................................................ 407
15.1.1. Double Knock Out Call ............................................... 407
15.1.2. Double Knock Out Put ............................................... 408
15.1.3. Double Knock In Put ................................................... 411
15.2. Valuing With a Single Put/Call Model ............................... 414
15.2.1. Valuing Double Calls ................................................... 416
15.2.2. Valuing Double Put’s .................................................... 416

16. Digital Options ................................................................. 421
16.1. General (Rubinstein & Reiner)
    Method .................................................................................. 421
16.2. Valuation ........................................................................... 427
    16.2.1. In Valuation ............................................................... 427
    16.2.2. Out Options ............................................................... 432
16.3. Valuation as a Portfolio ..................................................... 435
    16.3.1. In Cash or Nothing Valuations .................................. 435
    16.3.2. In Asset or Nothing Valuations ................................. 438
    16.3.3. In Asset Dependent Valuations ................................. 439
    16.3.4. Out Asset Dependent Valuations .............................. 441
    16.3.5. Out Asset limited Valuations .................................... 445

17. Special Case Barrier Options ............................................ 455
17.1. Partial Time Options ........................................................ 455
17.2. Two Asset Options ........................................................... 465
17.3. Partial Time Two Asset Options ........................................ .... 472
17.4. Look Type Options ............................................................... 476

18. Other Exotics ................................................................. 485
18.1. Two Asset Cash or Nothing ................................................ 485
18.2. Gap Option .................................................................... 489
18.3. Soft Barrier Options .......................................................... 493
18.4. Sequential Barrier Type Options ........................................ 497
18.5. Supershares ..................................................................... 511
18.6. Asian Options ................................................................. 513
  18.6.1. Geometric Average Rate Option ................................. 513
  18.6.2. Arithmetic Approximations ...................................... 515
  18.6.3. Levy Method .............................................................. 518
18.7. Quantos ........................................................................ 521
  18.7.1. Fixed Exchange Valuation ....................................... 521
  18.7.2. Foreign Exchange Option ........................................ 524

Appendix 1 ........................................................................ 529

Appendix 2 ........................................................................ 549

Index .................................................................................. 555
Preface

The early chapters of this book are based on many years of teaching the material to my final year honours students in computer science and accountancy at Heriot-Watt University, Edinburgh. My approach has been guided by their response. The latter material has benefited greatly from my postgraduate student’s, many of whom contributed to addressing issues of practical, efficient implementation of derivative models. Although, in those days, the work was largely C++ based the principles (and problems) of object based deployment remain the same. In making use of the appropriate built-in Java data structures and the general development methodology I have relied heavily on my experience over the last decade in directing technical and operations teams deploying internet and intranet distributed financial tools for a wide range of financial organisations.

Many applications in finance and investment are readily solved using analytical methods. The use of analytical techniques such as the calculus cannot be used directly on a standard computer. The analytical methods require numerical approximation techniques to be applied, which allow a standard computer to be programmed. The resultant programs give an approximate solution (approximate to the solution which would be found by direct use of analytical techniques). The approximation methods provide solutions that are only ever partially correct (to a given degree of accuracy).

A number of applications in finance and investment require the use of methods which involve time-consuming and laborious iterative calculations. The direct application of analytical techniques would not be of any help, so there is little option but to use trial and error or ‘best guess’ techniques. In other situations many of the valuation methods used in financial engineering have no closed-form solutions and require analytical methods; these need to be approximated for solution on a standard computing platform.

The issues mentioned above are tackled within this book by providing a series of fundamental or core classes which will allow the implementation of analytical techniques. The core classes also provide methods for the solution of problems involving the tedious repetition or best guess route. There are fundamental methods available for the provision of ‘better’ approximations. However there is a point at which the continuous adjustment to an approximation exhibits diminishing returns. The decision taken here is to include the most widely used and robust methods which are used as the basis of a large number of financial engineering tools.
Statistical methods are widely used in investment and finance applications. Many statistical methods rely heavily on analytical techniques to solve problems, thus for computer implementation of these statistical methods one needs to make use of numerical methods to approximate the analytical components. The java classes developed in this book will provide a series of statistical methods which allow the direct application of the statistical techniques. The statistical classes, in some cases will make use of underlying core classes to provide the needed numerical methods. Other statistical classes will not inherit any of the core classes but will themselves be the fundamental class.

Application classes are the end product of the building process. An application class implements a solution to a given problem in financial engineering or finance and investment. The application classes embody the techniques that are used throughout financial engineering practice. Those techniques will invariably use the underlying statistical classes (which in turn may use the methods of core classes), the core classes or a combination of both. The categorisation of classes into core/CoreMath, statistical/BaseStats and application/FinApps allows the independent development of a library system which can be added to over time, without affecting the operation of applications already built.

The methodology employed here is to make the core classes static, where the function is unchanging in the application or as abstract as possible, where the function is largely affected by the application context. The core classes are used (extended or implemented) by calling classes, which become increasingly more concrete as application classes. The core, statistical and application classes are organised as packages. CoreMath is the package containing all of the classes dealing with numerical algorithms, BaseStats contains the statistical classes and FinApps contains the application classes.

The chapters follow a largely linear progression from an investigation of fundamental concepts of finance and investment tools through to implementation of the techniques which underpin a wide range of the option products being used in Financial Engineering in Chapter 7. There is a brief discussion of number representation and accuracy which sets the scene for much of the termination criteria and levels of acceptable accuracy used in algorithm development. The first two chapters cover the implementation of financial tools and portfolio management techniques and introduce some Java data structures. Chapters 3 and 4 develop the technical issues in Bond markets and provide Java implementations of Bond valuation methods. Chapters 5 and 6 provide an introduction to the basis of option markets and aspects of practical techniques. Chapters 7–9 give the theoretical basis for much of the work shown in later chapters. For those who are starting in financial engineering, the three chapters will provide the necessary background to understand the methods and limitations of the standard tools. For the experienced practitioner, these chapters will guide an understanding of the Java class implementations that follow. Chapters 10–18 provide the models and implementation of a wide range of Financial Engineering methods. These chapters are accessible directly by the practitioner who wishes to implement a specific type of model or specific methods within a model.
The Java classes and methods are designed to be used as modular ‘object-based’ tools that can be used as-is to implement the many techniques covered. However it is expected that the imaginative practitioner will want to combine the many methods to develop their own products; particular to their unique application context. The class structures developed here will encourage this approach.
1 Introduction

1.1. Numerical Accuracy & Errors

Since we are largely dealing with numeric approximations or iterative convergence to a desired solution the discussion of accuracy and error are important.

In the decimal system irrational (e.g. $\sqrt{2}$) and transcendental (e.g. $\pi$) numbers cannot have a precise representation, most rational numbers are also not represented precisely, in decimal notation. Representing $\frac{1}{3}$ as a decimal can be approximated by 0.3 or 0.333333333 or some other arbitrarily large representation. Providing a decimal value for $\frac{1}{3}$, means representing the division as a floating point number.

In Java floating point numbers can be stored up to 15 digits in length (as type double with 64 bits). The Java BigDecimal is capable of storing an arbitrarily large decimal number with no loss of accuracy. However the transition from rational to binary representation provides the opportunity for potential loss of accuracy. Floating point numbers are represented in the form $M \times R^e$, where the Mantissa is the integer part and Radix is the base of a particular computer’s numbering system, this is usually 2, but can be 10 (in calculator processors) or 16. The exponent e can be up to 38 in type float and 308 in type double. The Mantissa provides us with the available precision of a number and the exponent provides the range.

Since representing numbers in floating point arithmetic can have varying results dependent on the underlying machine architecture and the data type (single or double precision floating point) it is important to know the target machine limitations and also use the appropriate number representations in the executing code.

One of the more common errors encountered with floating point representation is rounding error. This results from having more digits in a number than can be accommodated by the system. As an example consider a computer with a particularly small representation of real numbers. In this machine we can store four integers in the Mantissa and have a single exponent.

This simple machine has a largest value of 0.9999E9; the next lowest value is 0.9998E9. The value in decimal of 0.9999E9 is 999900000; the value of 0.9998E9 is 999800000. If we do the subtraction 999900000 − 999800000 = 100000, we see that there is no way of representing 100,000 intermediate values. So, 999855000 is represented as 999900000, as is 999895000 and so on.
Rounding errors are machine number related and are an artefact of using fixed lengths (machine word lengths) of bits to represent an infinite variety of numbers. Because rounding errors are related to machine architecture, it is useful to have some knowledge of the target platform.

One of the benefits in programming with Java is that the code is portable in the sense that it will run on any platform with a JVM. Unfortunately code written on one machine architecture with a different number representation to the target machine is not guaranteed. Floating point calculations that have a large dependency on accuracy should be configurable at run time with knowledge of the runtime architecture. We will return to the issue of accuracy and error at points in following chapters as individual algorithms introduce their own particular representational characteristics.

1.2. Core Math’s Classes

All of the Core classes are contained within the package CoreMath. This package covers functions, interpolation & extrapolation, roots of functions, series, linear algebra, Wiener, Brownian and Ito processes. The Java code for each of these classes is given in Appendix 1.

Core classes are designed as static or abstract classes, which in many cases require extending in other implementing classes (usually application classes). Some of the core classes are designed as standard Java classes, where it can be reasonably expected that the interface will be modified in the application context. The examples used throughout the text are working and tested ‘off the shelf’ Java code but are not developed as user ready applications. The intention is to show and explain Java methods that will run and perform a given function without adding the overhead of error trapping and exception handling.

We will often make use of core classes that provide roots (or zeros) of functions. The general methodology adopted for the book is best explained with the aid of an example that deals with providing the roots of a function. Our first example will be the development of a class that makes use of bracketing and bisection techniques to converge on a root with a given precision. The class is called IntervalBisection and is in the package CoreMath.

1.2.1. Root Finding - Interval Bisection

Figure 1.1 shows the Interval Bisection technique being applied to the function $y = 2 - e^x$. Interval bisection solves for a root of the equation by starting with two outlying values (the end points $X_0$ and $X_1$) that bracket the root. This is shown by arrow (1). The assumption is that the initial range of these end points contains the root. By evaluating the function at these points $f(x_0), f(x_1)$ and checking that the function changes sign we know the root is within the range. The assumption is also made that the function is continuous at the root and thus
1.2. Core Math's Classes

Figure 1.1. Bisection on $y = 2 - e^x$.

has at least one zero. The method takes the first approximation $x_2$ to the root by halving the initial range. So

$$x_2 = \frac{1}{2}(X_0 + X_1) \quad (1.2.1)$$

The function $f(x_2)$ is evaluated; there are three possible outcomes. First, in the interval, $[x_2, x_0]$. $f(x_0)f(x_2) < 0$. Means there is at least one root between these endpoints. Second, $f(x_0)f(x_2) = 0$ (we assume that $f(x_0) \neq 0$). This indicates that we have found the root $f(x_2)$. Third, $f(x_0)f(x) > 0$. This means the root is in the other interval half, $[x_2, x_1]$. Given that the second outcome is not initially achieved we continue with the process of halving the uncertainty until a root is found within the desired precision. From Figure 1.1 we see that the initial range, shown by arrow (1) is halved at $x = 0.5$. The function evaluates to 0.351, the function evaluates to $-0.7183$ at $x = 1.0$. The root therefore lies within the range shown by arrow (2). The halved value is at $x = 0.75$. The function evaluates to $-0.117$. The root therefore lies within the range now in the direction shown by arrow (3). The halved value is evaluated to be 0.131. The range is now in the direction shown by arrow (4). This process is continued until the desired precision is reached. The data in Table 1.1 shows the convergence for the function of Figure 1.1.

Listing 1.1 shows the method `evaluateRoot` in the abstract class `IntervalBisection`. The abstract method `ComputeFunction` is implemented in the extending class. The method `evaluateRoot` provides functionality for the bisection algorithm. This is outlined below in Listing 1.1.
public double evaluateRoot(double lower, double higher)
    //lower and higher are the initial estimates//
{
    double fa; //fa and fb are the initial ‘guess’ values.//
    double fb;
    double fc; //fc is the function evaluation , f(x)//
    double midvalue=0;
    double precvalue=0;
    fa=computeFunction(lower); //ComputeFunction is implemented
        //by the caller//
    fb=computeFunction(higher);

    //Check to see if we have the root within the range bounds//
    if (fa*fb>0)
    {
        //If fa^fb>0 then both are either positive//
        //or negative and don’t bracket zero.//
        midvalue=0; //Terminate program/
    }
    else
    do
    {
        precvalue=midvalue; //preceding value for testing
            //relative precision//
        midvalue=lower+0.5*(higher-lower);
        fc=computeFunction(midvalue) //Computes the f(x)//
            //for the mid value//
1.2. Core Math’s Classes

```java
if(fa*fc<0)
{
    higher=midvalue;
}
else
if(fa*fc>0)
{
    lower=midvalue;
}
while((abs(fc)>precisionvalue|i<iterations));
//loops until desired number of iterations or precision is reached/
return midvalue;
}
```

Listing 1.1. Method evaluateRoot from class IntervalBisection in package CoreMath

The return value, midvalue in this case is output when the converging solution is < 0.001. Note we might have used different precision criteria that would rely on the relative change in precision from one evaluation to the other. For example using the loop: while((abs(midvalue-precvalue)>precisionvalue|i<iterations)); would terminate when successive values of the intermediate evaluations are < 0.001.

Table 1.1 shows the output from IntervalBisection when evaluating the equation \( y = 2 - e^x \).

Column one shows the approximation output from the computation. Column two shows previous higher estimate minus the previous lower estimate. Columns three and four show the high and low estimates. For our example the initial ‘guesses’ were higher = 1.0 and lower = 0.5.

The approximation after 19 iterations reaches the desired precision to 1E-06, which is accurate to the ‘real’ solution by around -1E-07.

Listing 1.2 shows the complete class for IntervalBisection. ComputeFunction is an abstract method which has to be implemented in the calling class (which provides the actual function, in our example this is \( y = 2 - e^x \)). The constructor defaults to 20 iterations of the algorithm and the precision is set to 1e-3. The using class can pass other values through the alternate constructor (int iterations, double precisionvalue). Access to the internal values is via the get methods.

```java
public abstract class IntervalBisection
{
    //computeFunction is implemented to evaluate successive root estimates/
    public abstract double computeFunction(double rootvalue);
    protected double precisionvalue;
    protected int iterations;
    protected double lowerBound;
    protected double upperBound;
    //default constructor/
    protected IntervalBisection()
    {
        iterations=20;
    }

```
6  

    precisionvalue= 1e-3;
}
//Constructor with user defined repetitions and precision//
protected IntervalBisection(int iterations, double precisionvalue)
{
    this.iterations=iterations;
    this.precisionvalue=precisionvalue;
}
public int getiterations()
{
    return iterations;
}
public double getprecisionvalue()
{
    return precisionvalue;
}
public double evaluateRoot(double lower, double higher)
{
    double fa;
    double fb;
    double fc;
    double midvalue=0;
    double precvalue=0;
    fa=computeFunction(lower);
    fb=computeFunction(higher);
    //Check to see if we have the root within the range bounds//
    if (fa∗fb>0)
    {
        midvalue=0; //Terminate program//
    }
    else
    do
    {
        precvalue=midvalue; //preceding value for testing//
        //relative precision//
        midvalue=lower+0.5∗(higher-lower);
        fc=computeFunction(midvalue);
        if(fa∗fc<0)
        {
            higher=midvalue;
        }
        else
        if(fa∗fc>0)
        {
            lower=midvalue;
        }
    }
    while ((abs(fc)>precisionvalue<iterations));
    //loops until desired number of iterations or precision is reached//
    return midvalue;
}

Listing 1.2. IntervalBisection in package CoreMath
A class such as \textbf{IntervalBisection} has its core functionality controlled by the using class. To compute the function \( y = 2 - e^x \) we had to use an application class which extended the abstract method \textit{ComputeFunction}. The using class provided the controlling logic to provide the equation into \textbf{IntervalBisection}. The abstract class is there to provide a core technique (interval bisection) and not to perform other functionality. The strategy of keeping core functionality within static or abstract classes allows us to re-use the class in a variety of applications without the need to re-design or add to the core.

In this example we have used the class \textbf{IntervalBisection} to implement interval bisection on the function \( y = 2 - e^x \). Later we will use this same class to implement interval bisection on a yield equation. It will perform exactly the same functionality on a completely different equation; the controlling class (an application class) will implement the abstract method \textit{computeFunction} with the various input equations.

We will see later that more than one class is often required before we can implement an application. The interval bisection algorithm although generally robust is slower to converge than other root finding algorithms. The Newton Raphson algorithm (abbreviated to Newton’s method) is a method for a root finding algorithm which converges to a root much more quickly than interval bisection. Although the Newton method is quicker to converge, it requires the derivative of the function to be used in the solution. This is a good example of a series of classes being used to implement an application.

### 1.2.2. Newton’s Method

To use Newton’s method we will need to use the class \textbf{Derivative} from the CoreMath package. This abstract class provides the method derivation to provide functionality for providing the derivative of a single function. The class has its abstract method \textit{deriveFunction} extended by the using class which provides the controlling logic to provide the single functions for evaluation. Listing 1.3 provides the complete abstract class for \textbf{Derivative}.

The method derivation uses the technique of difference quotients to arrive at an approximation of a function. The method being implemented is based on the general definition of the derivative.

\[
f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \tag{1.2.2}
\]

The implementation used is based on the approximation which gives best accuracy with lower computational cost:

\[
f'(x) = \frac{f(x + h) - f(x - h)}{2h} \tag{1.2.3}
\]

The method which implements algorithm 1.1.3 is given below in Listing 1.3.
public double derivation (double InputFunc) {
    double value;
    double X2=deriveFunction(InputFunc-h);
    double X1=deriveFunction(InputFunc+h);
    value=((X1-X2)/(2*h));
    return value;
}

Listing 1.3. Method derivation in class Derivative package CoreMath

X1 and X2 take the value from the abstract method deriveFunction and implement the arithmetic from equation 1.1.3. The value of h is chosen to provide optimum accuracy. The smaller we can make h, the greater the accuracy we achieve (from theory). The analytic answer to the derivative of $e^x$ for $x = 1$, is $e$ itself. Column three in Table 1.2 shows the error in the derived approximation from the actual value of $e$. Column four shows the ratio of previous to present

<table>
<thead>
<tr>
<th>$\frac{1}{h}$</th>
<th>$f^\prime$</th>
<th>$\epsilon = e - f^\prime(x)$</th>
<th>Ratio($\epsilon_{n-1}/\epsilon_n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>2.8329678</td>
<td>-0.114685971</td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td>2.746685882</td>
<td>-0.028404053</td>
<td>4.03766</td>
</tr>
<tr>
<td>0.125</td>
<td>2.72536622</td>
<td>-0.007084391</td>
<td>4.00939</td>
</tr>
<tr>
<td>0.0625</td>
<td>2.720051889</td>
<td>-0.00177006</td>
<td>4.00234</td>
</tr>
<tr>
<td>0.03125</td>
<td>2.718724279</td>
<td>-4.42E-04</td>
<td>4.00059</td>
</tr>
<tr>
<td>0.015625</td>
<td>2.718392437</td>
<td>-1.11E-04</td>
<td>4.00015</td>
</tr>
<tr>
<td>0.0078125</td>
<td>2.71830948</td>
<td>-2.77E-05</td>
<td>4.00004</td>
</tr>
<tr>
<td>0.00390625</td>
<td>2.71828741</td>
<td>-6.91E-06</td>
<td>4.00001</td>
</tr>
<tr>
<td>0.001953125</td>
<td>2.718283557</td>
<td>-1.73E-06</td>
<td>4</td>
</tr>
<tr>
<td>0.77E-04</td>
<td>2.71828261</td>
<td>-4.32E-07</td>
<td>4</td>
</tr>
<tr>
<td>4.88E-04</td>
<td>2.71828136</td>
<td>-1.08E-07</td>
<td>4</td>
</tr>
<tr>
<td>2.44E-04</td>
<td>2.71828185</td>
<td>-2.70E-08</td>
<td>3.99996</td>
</tr>
<tr>
<td>1.22E-04</td>
<td>2.71828135</td>
<td>-6.75E-09</td>
<td>3.99984</td>
</tr>
<tr>
<td>6.10E-05</td>
<td>2.7182813</td>
<td>-1.69E-09</td>
<td>3.99719</td>
</tr>
<tr>
<td>3.05E-05</td>
<td>2.71828129</td>
<td>-4.19E-10</td>
<td>4</td>
</tr>
<tr>
<td>1.53E-05</td>
<td>2.71828129</td>
<td>-9.19E-11</td>
<td>4</td>
</tr>
<tr>
<td>7.63E-06</td>
<td>2.71828128</td>
<td>-1.92E-11</td>
<td>4</td>
</tr>
<tr>
<td>3.81E-06</td>
<td>2.71828129</td>
<td>-4.83E-11</td>
<td>0</td>
</tr>
<tr>
<td>1.91E-06</td>
<td>2.71828128</td>
<td>9.94E-12</td>
<td>-4.85718</td>
</tr>
<tr>
<td>9.54E-07</td>
<td>2.71828128</td>
<td>1.26E-10</td>
<td>0</td>
</tr>
<tr>
<td>4.77E-07</td>
<td>2.71828128</td>
<td>1.26E-10</td>
<td>1</td>
</tr>
<tr>
<td>2.38E-07</td>
<td>2.71828129</td>
<td>-3.39E-10</td>
<td>-0.37238</td>
</tr>
<tr>
<td>1.19E-07</td>
<td>2.71828128</td>
<td>5.92E-10</td>
<td>-0.57314</td>
</tr>
<tr>
<td>5.96E-08</td>
<td>2.71828128</td>
<td>5.92E-10</td>
<td>1</td>
</tr>
<tr>
<td>2.98E-08</td>
<td>2.71828135</td>
<td>-6.86E-09</td>
<td>-0.08632</td>
</tr>
<tr>
<td>1.49E-08</td>
<td>2.71828135</td>
<td>-6.86E-09</td>
<td>1</td>
</tr>
<tr>
<td>7.45E-09</td>
<td>2.71828135</td>
<td>-6.86E-09</td>
<td>1</td>
</tr>
<tr>
<td>3.73E-09</td>
<td>2.71828106</td>
<td>2.29E-08</td>
<td>-0.29893</td>
</tr>
<tr>
<td>1.86E-09</td>
<td>2.71828165</td>
<td>-3.67E-08</td>
<td>-0.62584</td>
</tr>
<tr>
<td>9.31E-10</td>
<td>2.718281984</td>
<td>1.56E-07</td>
<td>0.2352</td>
</tr>
</tbody>
</table>
error. The ratio of improvement is about 4. For each halving of h. This is true until 1/h is at 4.88E-04, thereafter the improvement oscillates widely.

This illustrates a phenomenon mentioned earlier in the introduction, namely machine (rather than theoretical) error. The errors being introduced are largely the result of rounding. The effects of repeated divisions of f(x+h) and f(x-h), together with machine representation (we are using type double for all floating calculations) are introducing practical implementation errors. If we used type float (32 bit) in the calculation things would be worse and we could expect significant error to be shown at around an h of -6.9E-06. We can achieve accuracy of -1.92E-11 before things deteriorate. For most applications this is good enough, but for some it could pose problems. You can use Table 1.1 to assess the size of h that might be suitable for your particular application.

Table 1.2 below shows the output from derivation with input = $e^x$. The values of h are decreasing from 0.5 down to 9.31E-10. The computed function f', is gradually converging on the ‘correct’ (high precision) answer.

Listing 1.4 gives the complete abstract class for Derivative.

```java
package CoreMath;
public abstract class Derivative {
    public abstract double deriveFunction(double fx);
    //returns a double...... the function/
    public double h; // degree of accuracy in the calculation/
    public double derivation(double InputFunc) {
        double value;
        double X2 = deriveFunction(InputFunc - h);
        double X1 = deriveFunction(InputFunc + h);
        value = ((X1 - X2) / (2 * h));
        return value;
    }
}

Listing 1.4. Derivative
```

Now we know something about the characteristics of our core class Derivative let’s examine the use of it in Newton’s method.

Newton’s method is based on linear approximations to the function. The approximation is based on the tangent line to the function curve.

$$\tan \theta = f'(x_0) = \frac{f(x_0)}{(x_0 - x_1)}$$

Thus, $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$ and $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$

In general, $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ for $n = 1, 2, 3 \ldots (0.2.4)$

To use Newton’s method we only require a single approximation for the root and the derivative of the function f(x). There can be problems with Newton’s method; one is where the derivative is zero near the root. In this case; $f(x_n)/f'(x_n) \rightarrow \infty$. 
Small values of \( f'(x_n) \) can cause large differences between iterations and slow convergence; also the calculation of \( f'(x_n) \) itself can be complicated.

Package CoreMath contains the class \texttt{NewtonRaphson}. This is an abstract class which implements the Newton Raphson algorithm and extends the abstract method \texttt{deriveFunction}.

\begin{verbatim}
public void newtraph(double lowerbound)
{
    double fx=newtonroot(lowerbound); // y = 2 - e^x in our example
    double x=derivation(lowerbound);
    double x=(lowerbound-(fx/Fx)); //\( x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \)
    while((abs(x-lowerbound)>precisionvalue&counter<iterate))
    {
        lowerbound=x;
        fx=newtonroot(lowerbound);
        Fx=derivation(lowerbound);
        x=(lowerbound-(fx/Fx));
        counter++;
    }
}
\end{verbatim}

Listing 1.5. Shows the method \texttt{newtraph}. This implements the algorithm of 1.1.4

The method \texttt{newtraph} takes the approximation as lowerbound. The abstract method \texttt{newtonroot} is extended in the calling class which provides the function for evaluation. The method \texttt{derivation} is used to calculate the derivative. The method iterates through calls until the desired precision or predefined number of iterations is reached. This is controlled through the while loop which implements: 
\[ |x_{n+1} - x_n| > \varepsilon & < \text{Iterations} \]

The precision value \( \varepsilon \) is defined in the method \texttt{accuracy} as is the value of the desired maximum number of iterations.

Table 1.3 shows the output from \texttt{NewtonRaphson} for \( y = 2 - e^x \).

Column one shows the approximation (guesses) input. The initial approximation was 1.0. The second column shows the actual (analytical) solution to the function minus the approximation. The third column shows log base 10 of the differences. From this column it can be intuitively appreciated that the error in the successive approximations is halving each time.

It is instructive to compare the number of iterations and the convergence characteristics shown in Table 1.1 for the bisection algorithm and Tables 1.3 and

\begin{verbatim}
\begin{tabular}{|c|c|c|c|}
\hline
N & \( x_n \) & \text{(actual-}\( x_n \)) & \text{log}_{10}(\text{actual-}\( x_n \)) \\
\hline
1  & 1.0000000000000000 & -0.3068528194400547 & -0.5130698819559578 \\
2  & 0.7357598961783395 & -0.04261271561839375 & -1.3704607882472601 \\
3  & 0.6940422746273281 & -8.9509190278477404E-4 & -3.048132371584211 \\
4  & 0.6931475844952419 & -4.0395366556109E-7 & -6.3936881956713165 \\
5  & 0.6931471805598971 & 4.8183679268731794E-14 & -13.317100040678492 \\
\hline
\end{tabular}
\end{verbatim}

Table 1.3. Newton Raphson method on \( y = 2 - e^x \)
1.2. Core Math’s Classes

<table>
<thead>
<tr>
<th>N</th>
<th>f(x)</th>
<th>f'(x)</th>
<th>(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.718281828459</td>
<td>-2.718292257953</td>
<td>0.735759896178</td>
</tr>
<tr>
<td>2</td>
<td>-0.087067344576</td>
<td>-2.087063855072</td>
<td>0.694042272463</td>
</tr>
<tr>
<td>3</td>
<td>-0.001790985234</td>
<td>-2.001798726781</td>
<td>0.693147584495</td>
</tr>
<tr>
<td>4</td>
<td>-0.000000807871</td>
<td>-2.000000165481</td>
<td>0.693147180560</td>
</tr>
<tr>
<td>5</td>
<td>0.000000000000</td>
<td>-2.000000165481</td>
<td>0.693147180560</td>
</tr>
</tbody>
</table>

Table 1.4 shows the output from the method *newtraph*. Column one is the function evaluation with the ‘guess’ value as input. Column two is the derivative of the function with the variable set to the ‘guess’ value. Column three shows the successive approximations for x.

Clearly Newton’s method converges within four iterations whereas the bisection method takes 19 iterations for the same degree of precision.

Listing 1.6 gives the complete class for *NewtonRaphson*. Since the Newton Raphson method requires the use of the derivative, this class extends the abstract class *Derivative*. It was mentioned earlier that we often use several classes to provide an application with the needed methods. In this case we have *NewtonRaphson* making use of *Derivative* (and extending the abstract method). However the class *NewtonRaphson* is itself only designed to provide the means for carrying out Newton’s algorithm. To do the computation on an actual function, *NewtonRaphson* needs to have its abstract method *newtonroot* extended by an application which provides the function to be evaluated. *NewtonRaphson* also needs to pass this function to the *Derivative* method which requires it.

```java
package CoreMath;
public abstract class NewtonRaphson extends Derivative {
    public abstract double newtonroot(double rootvalue);
    //the requesting function implements the calculation fx//
    public double precisionvalue;
    public int iterate;
    public void accuracy(double precision, int iterations) {
        super.h=precision;//sets the superclass derivative//
        //to the desired precision//
        this.precisionvalue=precision;
        this.iterate=iterations;
    }

    public double newtraph(double lowerbound) {
        int counter=0;
        double fx=newtonroot(lowerbound);
        double Fx=derivation(lowerbound);
        double x=(lowerbound-(fx/Fx));
        return x;
    }
}
```
1. Introduction

```java
while(abs(abs(x)-abs(lowerbound)) > precisionvalue||counter<iterate)
{
    lowerbound=x;
    newtraph(lowerbound);//recursive call//
    // to newtraph//
    counter++;
}
return x;
}
```

```java
public double deriveFunction(double inputa)
{
    double x1=newtonroot(inputa);
    return x1;
}
```

Listing 1.6. NewtonRaphson

1.3. Statistical Classes

The statistical classes implement methods for the manipulation and analysis of data. Statistical classes provide standard re-usable techniques as methods for use in application classes where the specific functionality of the methods are needed to create a sophisticated technique (from possibly many methods). The class structures are minimal in the sense that a particular technique will usually be applied through the use of a series of classes that implement a particular part of the technique.

For example, the data in Table 1.5 is to be used to provide the standard deviation for the sample. The standard deviation will not be directly computed, rather the mean, followed by the variance then standard deviation will be used to provide the desired result Table 1.5 shows data with associated probability. Table 1.6 contains data only.

| Data Item : 1 | Data 12.000000 Probability 0.100000 |
| Data Item : 2 | Data 7.000000 Probability 0.200000 |
| Data Item : 3 | Data 11.000000 Probability 0.100000 |
| Data Item : 4 | Data 23.000000 Probability 0.100000 |
| Data Item : 5 | Data 44.000000 Probability 0.075000 |
| Data Item : 6 | Data 58.000000 Probability 0.025000 |
| Data Item : 7 | Data 22.000000 Probability 0.200000 |
| Data Item : 8 | Data 33.000000 Probability 0.100000 |
| Data Item : 9 | Data 56.000000 Probability 0.050000 |
| Data Item : 10| Data 76.000000 Probability 0.050000 |

Table 1.5. Input dataset
When we make use of the statistical methods in BaseStats to provide us with the mean and variance of this data it’s reasonable to expect that we can use the same methods. This is achieved by making extensive use of method overloading. The majority of the statistical classes are implemented as static methods. Statistical classes are placed in the package BaseStats.

### 1.3.1. Measures of Dispersion

The class `DataDispersion` in the package BaseStats is a general purpose class with static methods for direct use. A range of methods that deal with aspects of data dispersion are supplied to enable an application class to make direct use of specific techniques, or to combine methods into a more global technique.

The data in Tables 1.5 and 1.6 are input from a controlling class which makes use of the class `DataDispersion` to evaluate the standard deviation of the data. The methods used for this operation are shown in Listing 1.7.

```java
// uses the algorithm \( \frac{1}{n} \sum_{i=1}^{n} X_i \) //
public static double mean(double[] x) //arithmetic mean for a single list//
{
    double total=0.0;
    for(int i=0;i<x.length;i++)
        total+=x[i];
    return total/x.length;
}

// uses the algorithm \( \sum_{i=1}^{n} (X_i P_i) \) //
public static double mean(double[][] x) //returns expected value//
    //for variable * probability//
{
    double total=0.0;
    double probability=0.0;
    for(int i=0;i<x.length;i++)//the number of rows//
Listing 1.7. Methods used to provide Standard Deviation

```java
{  
    total+=(x[i][0]*x[i][1]);
    probability+=x[i][1];
}
if(probability!=1.0)
    System.out.println("WARNING ! The probabilities do not sum to 1.0");
return total;
}

// uses the algorithm
\[
\frac{\sum_{i=1}^{n} x_i^2 - \left(\frac{\sum_{i=1}^{n} x_i}{n}\right)^2}{n-1}
\]
// uses the algorithm
public static double variance(double[] v1)
    //variance of a single variable with equal likelihood/
    {
        double sumd=0.0;
        double total=0.0;
        for(int i=0;i<v1.length;i++)
        {
            total+=v1[i];
            sumd+=pow(v1[i],2);//sum of x sqrd
        }
        return (sumd-total)/((v1.length)-1);
        //true value of convergence as length is large/
    }

// uses \[ \sum_{i=1}^{n}(X_i^2P_i) - \sum_{i=1}^{n}(X_iP_i)^2 \] 
// uses \[ \sum_{i=1}^{n}(X_i^2P_i) - \sum_{i=1}^{n}(X_iP_i)^2 \] 
public static double variance(double[][] v1)
    //variance of a variable with different probability of outcome/
    {
        double sumd=0.0;
        double total=0.0;
        double totalpow=0.0;
        double probability=0.0;
        for(int i=0;i<v1.length;i++)
        {
            total+=(v1[i][0]*v1[i][1]);//mean or expected value/
            totalpow+=(pow(v1[i][0],2)*v1[i][1]);//E[X2]/
            probability+=v1[i][1];
        }
        if(probability!=1.0)
            System.out.println("WARNING !The probabilities do not approximate to sum to 1.0");
        total=pow(total,2);
        return (totalpow-total);
    }

public static double standardDeviation(double s1)
    // computes standard deviation for variance s1/
    {
        double sdev;
        return sdev=sqrt(s1);
    }
```

Listing 1.7. Methods used to provide Standard Deviation
In each case the controlling class passes the data to the method. The JVM determines which particular implementation of mean and variance to use. In the case of Table 1.5, mean and variance for the single list take the data. For Table 1.6 data, the methods mean and variance for the double list take the data. The method `standardDeviation` takes the variance and produces the measure. This simple example shows that a combination of overloading methods and constructing a global technique (or algorithm) from simpler ones, is a very efficient technique. The output from Tables 1.5 and 1.6 are:

Table 1.5: Mean: 25.050000 Variance: 342.297500 Standard Deviation: 18.501284
Table 1.6: Mean: 34.200000 Variance: 547.955556 Standard Deviation: 23.408451

Class `DataDispersion` is used in a later example when we make use of covariance and standard deviation. The remaining methods to complete the listing for `DataDispersion` is shown in Listing 1.8:

```java
package BaseStats;
import java.util.ArrayList;
import java.io.*;
import static java.lang.Math.*;
public class DataDispersion {
    // use algorithm \frac{1}{n} \sum_{i} X_i for both entries //
    public static double[] dumean(double[][] x) // arithmetic mean //
        // for a double list //
    {
        double x1=0.0;
        double y=0.0;
        double[] total=new double[2];
        for(int i=0;i<x.length;i++)
        {
            x1+=x[i][0];
            y+=x[i][1];
        }
        total[0]=x1/x.length;
        total[1]=y/x.length;
        return total;
    }
    // use algorithm \frac{1}{n-1} \sum_{i=1}^{n} X_i //
    public static double convmean(double[] x) // for large length //
    {
        double total=0.0;
        for(int i=0;i<x.length;i++)
            total+=x[i];
        return total/(x.length-1);
    }
}
```
public static double[] variances(double[][] v1)
    //variance of a single variable with equal likelihood/
    //for double inputs/
{
    double[] output=new double[2];
    double sumd=0.0;
    double sumd1=0.0;
    double total=0.0;
    double total1=0.0;
    for(int i=0;i<v1.length;i++)
    {
        total+=v1[i][0];
        total1+=v1[i][1];
        sumd+= pow(v1[i][0],2);//sum of x sqrd
        sumd1+= pow(v1[i][1],2);//sum of x sqrd
    }
    total=(pow(total,2)/v1.length);//sum of [x]sqrd/n
    total1=(pow(total1,2)/v1.length);//sum of [x]sqrd/n
    output[0]=((sumd-total)/((v1.length)-1));
    output[1]=((sumd1-total1)/((v1.length)-1));
    return output;
}
public static double covar(double[][] outcomes)
    //equally likely outcomes/
{
    double sa=0.0;
    double sb=0.0;
    double product=0.0;
    int size=outcomes.length;
    for(int i=0;i<size;i++)
    {
        sa+=outcomes[i][0];//x values or proportions/
        sb+=outcomes[i][1];//y values or proportions/
    }
    double samn=sa/size;//expected value of x/
    double sbmn=sb/size;//expected value of y/
    for(int i=0;i<size;i++)
    {
        product+=((outcomes[i][0]-samn)∗
            (outcomes[i][1]-sbmn));
        //sum of the products of deviations/
    }
    return product/size;//covariance/
}
public static double covar2(double[][] outcomexyp)
    //inputs of non equal joint outcomes/
{
1.4. Application Classes

The application classes comprise those classes which provide the functionality for solving application problems in Financial Engineering computation. Application class methods provide the controlling logic for calling other classes from the CoreMath and BaseStats packages. The majority of application classes are self contained within the package FinApps, a limited number of the classes involve combinations of methods from others within FinApps. Application classes comprise the focus of this book. An application class which makes use of the root finding algorithms discussed earlier is the Yield evaluator and another which makes use of a range of methods from the CoreMath and BaseStats packages is the Portfolio evaluator. See Elton & Gruber (1995) for background theory.

Yield evaluation can be accomplished with a range of numerical methods. For our examples we will use the bisection algorithm implemented in IntervalBisection and the Newton Raphson algorithm implemented in the NewtonRaphson class. This example gives us the opportunity to see how the concrete classes

```java
// data in the form A value, B value. Probability(P)
// of B and A the same
double productx=0.0;
double producty=0.0;
int size=outcomexyp.length;
double covariance=0.0;
for(int i=0;i<size;i++)
{
    // A[n][0],B[n][1],P[n][2]...........
    productx+=outcomexyp[i][0]*outcomexyp[i][2];
    probability*observed value
    producty+=outcomexyp[i][1]*outcomexyp[i][2];
}
for(int j=0;j<size;j++)
{
    double xdevs=outcomexyp[j][0]-productx;
    double ydevs=outcomexyp[j][1]-producty;
    double devproduct=xdevs*ydevs;
    double covprobs=devproduct*outcomexyp[j][2];
    covariance+=covprobs;
}
return covariance;

// \rho_{ij} = \frac{\sigma_i}{\sigma_j} \\
public static double correlation(double cov, double sd1, double sd2)
{
    double cor=cov/sd1*sd2;
    return cor;
}
```

Listing 1.8. Class DataDispersion