Lecture 3
JAVA (46-935)
Somesh Jha
Overview of BDT

• Assume we are interested in building an interest rate tree upto time horizon $T$.

• Let $B_t$ be the bond maturing at time $t$ $(1 \leq t \leq T + 1)$.

• We use the binomial lattice. Each node is represented as $(t, U)$, where $t$ is the time and $U$ is the number of up-ticks on the path from the root to that node.
BDT (Contd)

- For each time $t$ ($1 \leq t \leq T$) we have two parameters $r_t$ and $k_t$.

- The short rate $r(t, U)$ at node $(t, U)$ is given by a function

  $$F(t, U, r_t, k_t)$$

- We assume that yields and the yield volatilities are given to us.
Algorithm

- Set the initial short rate $r(0, 0)$ to the yield of the bond $B_1$.

- Induction step.
  - **Assume:** We know the short rate for time less than $t$.
  - The yield and the yield volatilities of bond $B_{t+1}$ are a function of the parameters $r_t$ and $k_t$.
  - Solve for the parameters $r_t$ and $k_t$ by matching to the market data.
Common forms of Short rates

- **Lognormal**

  \[
  F(t, U, r_t, k_t) = r_t k_t \frac{2U-t}{2}
  \]

- **Normal**

  \[
  F(t, U, r_t, k_t) = r_t + k_t \frac{2U - t}{2}
  \]

- **Capped LogNormal**

  \[
  F(t, U, r_t, k_t) = \min\{r_t k_t \frac{2U-t}{2}, \lambda\}
  \]

- **Floored Normal**

  \[
  F(t, U, r_t, k_t) = \max\{r_t + k_t \frac{2U - t}{2}, \lambda\}
  \]
AbstractTermStructure class

/**
   Abstract class for building a BDT type interest-rate model.
   @author Somesh Jha
*/

package interestRate;

import mathUtil.*;

public abstract class AbstractTermStructure {

  private static final boolean DEBUG=false;

  //time horizon
  int T;

  //Used by the Newton-Raphson solver
  SlowYieldVolObject slowYieldVolObj;
  NewtonRaphson slowSolver;

  //parameters of the BDT model
  double r[];
  double k[];

  //bond yields and yield volatilities
  //at time 0
  double yield[];
  double volatilities[];

  //nodes[i] points to link list of
  //nodes with time i
  LinkList nodes[];
/**
Constructor takes following arguments.
@params T Time Horizon
@params yield Array of Bond yields
@params volatilites Array of Volatilities
*/
public AbstractTermStructure(int T, double yield[],
                             double volatilities[]){

    //set the time horizon
    this.T = T;

    //allocate space for the parameters
    r = new double[T+1];
    k = new double[T+1];

    //copy the bond yields and yield volatilities
    int arrayLength = yield.length;

    //assume volatility array has the same length
    //as yield array
    this.yield = new double[arrayLength];
    this.volatilities = new double[arrayLength];

    //copy the arrays
    System.arraycopy(yield,0,this.yield,0,arrayLength);
    System.arraycopy(volatilities,0,this.volatilities,0,arrayLength);

    //allocate the linked-list
    nodes = new LinkList[T+1];
    for(int i=0; i <= T; i++)
      nodes[i] = new LinkList();

    //Get the yield-vol object
    slowYieldVolObj = new SlowYieldVolObject(this);

    //Instantiate the solver
slowSolver = new NewtonRaphson(slowYieldVolObj);

} // end of AbstractTermStructure

/**
 * The form of the short rate at the node (time,up_ticks)
 */
public abstract double F(int time, int up_ticks,
double r, double k);

/**
 * Generate the entire pdag upto time horizon T.
 */
public void GenPdag() {

if (DEBUG) System.out.println("Entered GenPdag");
nodes[0].Insert(new Key(0,0));

for(int t=0; t < T; t++) {
    Node x=nodes[t].head;

    while (x != null) {
        if (DEBUG) {
            System.out.println("Considering node: ");
x.key.print();
        }

        int up_ticks = ((Key)x.key).up_ticks;

        //Generate successor nodes
        Key up_key = new Key(t+1,up_ticks+1);
        Key down_key = new Key(t+1,up_ticks);
x.succ[0] = nodes[t+1].Insert(up_key);
x.succ[1] = nodes[t+1].Insert(down_key);
    }
}

x = x.next;
    } //end of while
} //end of for

if (DEBUG) System.out.println("Leaving GenPdag");

} //end of GenPdag

/**
   * print the entire dag.
   */
public void print() {

    try {
        for(int t=0; t <= T; t++) {
            System.out.print("Nodes at time ");
            System.out.println(t);
            System.out.println("-------------BEGIN-------------------");
            nodes[t].print();
            System.out.println("-------------END-------------------");
        }
    } //
    catch (LinkListException e) {
        System.err.println("Shouldn’t happen! "+e.getMessage());
        System.exit(1);
    }
} //end of print

//price of the bond of a given maturity
//(t,up_ticks). Assume that the parameters
//r, k are known upto time maturity-1
private double slowPrice(int t, int up_ticks,
                        int maturity) {

    //Handle the base case
    if (t == maturity) return(1);
else {
    // Recursive call price on successor node
    double price_up = slowPrice(t+1, up_ticks+1, maturity);
    double price_down = slowPrice(t+1, up_ticks, maturity);

    double returnPrice =
        (0.5/(1+F(t, up_ticks, r[t], k[t]))) * (price_up + price_down);

    if (DEBUG) {
        System.out.print("slowPrice:AbstractTermStructure returnPrice ");
        System.out.println(returnPrice);
    }
    return(returnPrice);
} // end of else

} // end of slowPrice

/**
 * yield of the bond of a given maturity
 * at the node (t, up_ticks).
 */
public double slowYield(int t, int up_ticks, int maturity) {

    double bondPrice = slowPrice(t, up_ticks, maturity);

    double bondYield = Math.pow(1.00/bondPrice, 1.00/(maturity-t))-1;
    if (DEBUG) {
        System.out.print("slowYield:AbstractTermStructure bondYield ");
        System.out.println(bondYield);
    }
    return(bondYield);

} // end of slowYield
/**
   volatility of the yield of the bond
   at node (t, up_ticks)
*/
double slowLogVol(int t, int up_ticks, int maturity) {
    double up_yield = Math.log(slowYield(t+1, up_ticks+1, maturity));
    double down_yield = Math.log(slowYield(t+1, up_ticks, maturity));

    double expectedSquareVal = 0.5*(up_yield*up_yield +
    down_yield*down_yield);

    double expectedVal = 0.5*(up_yield+down_yield);
    return(Math.sqrt(expectedSquareVal-expVal*expVal));
}

//end of slowLogVol

//solve for the parameters at time t
private void slowSolve(int t) {

    //Handle the base case
    if (t==0) {
        r[0] = yield[0];
        k[0] = 1;
    }
    else {
        //update the maturity in the slowYieldVolObj
        slowYieldVolObj.maturity = t+1;
        double initialVal[] = new double[2];

        //Set the initial value of the parameters
        //time t to parameter values at time t-1
        initialVal[0] = r[t-1];
        initialVal[1] = k[t-1];
    }
}
double result[] = slowSolver.solve(initialVal);
r[t] = result[0];
k[t] = result[1];

} // end of else

} // end of slowSolve

/**
 * Solve for the entire interest rate tree.
 */
public void slowSolve() {
    // Call slowSolve iteratively
    for (int t=0; t <= T; t++) {
        slowSolve(t);
        // fill the short-rate at the nodes
        Node x=nodes[t].head;
        while (x != null) {
            Key key = (Key)x.key;
            key.short_rate = F(t,key.up_ticks,r[t],k[t]);
            x=x.next;
        }
    } // end of slowSolve

} // end of AbstractTermStructure
package interestRate;

import mathUtil.*;

public class SlowYieldVolObject extends AbstractFunctionObject {

    AbstractTermStructure termStructureObj;

    //Yield and volatility are computed for the
    //bond of that maturity
    public int maturity;

    private static final boolean DEBUG=false;

    public SlowYieldVolObject(AbstractTermStructure aTerm) {

        //call the constructor for the super class
        super(2);
        termStructureObj = aTerm;
    }//end of constructor

    //If i==0 calculate the yield and otherwise
    //calculate the vol. Use values as value
    //of r[t] and k[t]
    public double evaluate(int i, double val[]) {
        termStructureObj.r[maturity-1]=val[0];
        termStructureObj.k[maturity-1]=val[1];

        if (i==0) {
            double tempYield =termStructureObj.slowYield(0,0,maturity);
            if (DEBUG) {
                System.out.println("SlowYieldVolObject:evaluate: maturity tempYield val");
                System.out.println(maturity);
                System.out.println(tempYield);
            }
        }
    }
}
System.out.println(val[0]);
System.out.println(val[1]);
}

    return(tempYield-termStructureObj.yield[maturity-1]);
}
else {
    double tempVol = termStructureObj.slowLogVol(0,0,maturity);
    if (DEBUG) {
        System.out.println("SlowYieldVolObject:evaluate: maturity tempYield val");
        System.out.println(maturity);
        System.out.println(tempVol);
        System.out.println(val[0]);
        System.out.println(val[1]);
    }
    return(tempVol-termStructureObj.voltalities[maturity-1]);
}
}//end of evaluate

}//end of SlowYieldVolObject
Explanation

- **T**
  Type: `int`
  Time horizon

- **slowYieldVolObject**
  Type: `SlowYieldVolObject`
  Compute the yield and the volatility of the bond with a certain maturity.

- **slowSolver**
  Type: `NewtonRaphson`
  Newton Raphson solver to match with market data. Instantiated with `slowYieldVolObject`. 
Explanation Continued

• \(r\) and \(k\)
  
  **Type:** Array of double
  
  Holds the parameters for our model. Elements \(r[t]\) and \(k[t]\) are the parameters corresponding to time \(t\).

• yield and volatilities
  
  **Type:** Array of double
  
  Holds the market yields and yield volatilities of bonds. Elements \(yield[t - 1]\) and \(volatilities[t - 1]\) hold the yield and yield volatilities of the bond \(B_t\).

• nodes
  
  **Type:** Array of LinkLists
  
  \(nodes[t]\) is the linked-list of nodes corresponding to time \(t\).
Constructor

- Takes the time horizon and market yields and yield volatilities and parameters.

- Allocates space for arrays $r$, $k$, $yields$, and $volatilities$.

- Copies the yield and yield volatilities into its local array.

- Allocates the linked-list.

- Instantiates the slowYieldVolObj and slowSolver.
Method F

- This is an abstract function and provides the \textit{form} of the short rate.

- A class extending this class will provide an implementation for F.
**Method GenPdag**

- Very similar to the abstract option class.

- Generates the lattice starting from the initial time $t = 0$ and going up to the time-horizon.
Method print

- Prints all the nodes in the lattice.

- Starts from the initial time and goes up to the time-horizon.

- Calls the \texttt{print} method in the \texttt{LinkList} class.

- If it catches an exception, then prints the exception and exits.
**Method** slowPrice

- Computes the *price* of bond with a given maturity at the node \((t, \text{up\_ticks})\).

- Notice the recursion.

- Notice that we use the abstract method \(F\).
Method slowYield

- Computes the price of bond with a given maturity at the node \((t, \text{up\_ticks})\).

- Let yield and price of bond \(B_\tau\) at node \((t, U)\) be denoted by \(y(t, U, \tau)\) and \(P(t, U, \tau)\). We have the following relationship between yield and price

\[
P(t, U, \tau) = \frac{1}{(1 + y(t, U, \tau))^{\tau-t}}
\]
Method slowLogVol

\begin{itemize}
  \item Computes the volatility of the log of the yield at node \((t, \text{up_ticks})\) for bond that matures at time maturity.
  
  \item Notice that we need the yields at the successors of the node.
\end{itemize}
Method slowSolve(int t)

- Assume that we have the interest rate tree for time upto $t - 1$.

- This routine solves for the parameters $r[t]$ and $k[t]$.

- Notice that we use bond that matures at time $t + 1$ to solve for the parameters $r[t]$ and $k[t]$.

- The maturity field in the slowYieldVolObj is changed to $t + 1$.

- The initial value to the Newton-Raphson solver is the value of the parameters at time $t - 1$. 
**Method** slowSolve

- Solve for the parameters for all the times.

- Fill in the short-rates.

- Calls `slowSolve(t)`. 
Class SlowYieldVolObject

• Is a subclass of AbstractFunctionObject.

• Has to provide implementation of the method evaluate.
Method evaluate

- For $i = 0$ calculates the difference between yield of the bond whose maturity is maturity at the initial node $(0, 0)$ and the market yield.

- For $i \neq 0$ calculates the difference between the yield volatility of the bond whose maturity is maturity at the initial node $(0, 0)$ and the market volatility.

- Needs reference to the termStructure.
Object Diagram

Figure 1: TermStructure Object Hierarchy
LogNormal class

package interestRate;

public class LogNormal extends AbstractTermStructure {

    public LogNormal(int T, double yield[],
                     double volatilities[]) {
        super(T, yield, volatilities);
    }

    public double F(int time, int up_ticks, double r,
                     double k) {
        int sum = 2*up_ticks - time;
        return (r*Math.pow(k, sum));
    }

} // end of class
About LogNormal

- Is a subclass of AbstractTermStructure.

- Method F implements a lognormal short rate.
package testPrograms;

import interestRate.*;

public class testTermStructure {

    static public void main(String argv[]) {

        int T=4;
        double yield[] = new double[5];
        double volatilities[] = new double[5];

        yield[0]=0.10;
        volatilities[0] = 0.20;
        yield[1]=0.11;
        volatilities[1]=0.19;
        yield[2]=0.12;
        volatilities[2]=0.18;
        yield[3]=0.125;
        volatilities[3]=0.17;
        yield[4]=0.13;
        volatilities[4]=0.16;
        LogNormal termObj = new LogNormal(T,yield,volatilities);
        termObj.GenPdag();
        termObj.slowSolve();
        termObj.print();

    } //end of main

}//end of testNewtonRaphson
About the Test Program

- Builds the interest rate lattice for time horizon of 4.

- Clumsy! Would like to take the data from file.

- Next we will discuss file I/O.
I/O in JAVA

• Everything to do with I/O is in a package called java.io.

• It is kind of complicated. Why?

• Internationalization
  Supposed to handle many languages.

• Customization
  Users can plug-in there own I/O routines.
Testing I/O

package testPrograms;

import java.io.*;
import java.util.*;

public class testFileIO {

    static public double[] parseLine(String line) throws NumberFormatException {
        // instantiate the String tokenizer
        StringTokenizer tokenizer = new StringTokenizer(line);
        int size = tokenizer.countTokens();
        double result[] = new double[size];
        int counter = 0;
        while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            result[counter] = Double.valueOf(token).doubleValue();
            counter++;
        }
        return(result);
    }

    // end of parseLine

    static public void main(String argv[]) {
        String inputFileName=null;
        String outputFileName="blahblah";
        switch (argv.length) {
            case 1:
                inputFileName=argv[0];
                break;
            case 2:
                outputFileName=argv[1];
                break;
        }
    }
}
inputFileName=argv[0];
outputFileName=argv[1];
break;
default:
    System.out.println("Wrong number of arguments provide");
}

FileInputStream fiStream=null;
InputStreamReader isReader=null;
BufferedReader bReader=null;
FileOutputStream foStream=null;
PrintWriter pWriter=null;
try {
    fiStream = new FileInputStream(inputFileName);
isReader = new InputStreamReader(fiStream);
bReader = new BufferedReader(isReader);
foStream = new FileOutputStream(outputFileName);
pWriter = new PrintWriter(foStream);

String line;

    while ( (line = bReader.readLine()) != null) {
        System.out.println(line);
        try {
            double result[] = parseLine(line);
            for (int i=0; i < result.length; i++) {
                pWriter.print(result[i]);
pWriter.print(" ");
            }

        pWriter.println();
    }
    catch (NumberFormatException e) {
        pWriter.println("Error in that Line");
    }

}
pWriter.close();
foStream.close();
}
catch (FileNotFoundException e) {
    System.err.println("Input file was not found "+e.getMessage());
}
catch (IOException e) {
    System.err.println("IOException occured "+e.getMessage());
}

finally {
    try {
        if (fiStream != null)
            fiStream.close();
        }
        catch (IOException e) {
            System.err.println("Error while closing the file "+e.getMessage());
        }
    }
} //end of main
} //end of testFileIO
Explanation of the Program

- The program has to be invoked with a input filename.

- If the output filename is not supplied, the output filename is *blahblah*.

- If the output filename is supplied, it is used.

- Reads from the input file and parses the lines into *array of doubles* and prints them to output file.
Running the program

- Compile it.
  javac testFileIO.java

- No output file.
  java testPrograms.testFileIO testFile
  Writes output the file blahblah.

- Output file supplied.
  java testPrograms.testFileIO testFile outFile
  Writes output the file outFile.
Running the program (Contd)

• Input looks like:

  0.10  0.20
  0.11  0.19
  0.12  0.18
  0.125  0.17
  0.13  0.16
  xxx

• Output looks like:

  0.1  0.2
  0.11  0.19
  0.12  0.18
  0.125  0.17
  0.13  0.16
  Error in that Line
System class

- Is a final class (what does this mean?).

- Has system defined functionality.

- Example: out is constant (static final) of type PrintStream which is linked to the screen. Is defined inside class System.
Variables Explained


- `FileInputStream` is first created. I can only read binary data (or bytes) using this class (see page 409).

- `InputStreamReader` allows me to read characters but I want to read lines (see page 416).

- `BufferedReader` allows me to read lines and also does buffering for efficiency reasons (see page 400).

- Keep making the functionality more general.
StringTokenizer

• This class belongs to the package java.util.

• It allows us to break string into tokens.

• Consider the following code:

  StringTokenizer tokenizer = new StringTokenizer()
  System.out.println(tokenizer.nextToken());

• Will print abc.
Double **class**

- **Double** is not the same as **double**.

- **Double** extends a class **Number** and **double** is a primitive type. It is in the package **java.lang**.

- Lot of utilities inside the class **Double** (see page 453).

- The statement given below parses a string into an object of type **Double** and then calls method **doubleValue** to convert it into a **double**.
Homework Setup

- Implement \texttt{BlackScholesCallObject} and \texttt{BlackScholesPutObject} classes that extend \texttt{AbstractFunctionObject}.

- The classes given above implement the Black-Scholes formula for call and a put.

- The \texttt{evaluate} method takes the volatility as argument and calculates the difference between the Black-Scholes formula and the actual option price.
Implied Volatility Graphs

- Pick a stock that has option prices for various strike prices and expiration dates.

- Pick a stock which doesn’t pay dividends or has low dividend rate.

- Find the *implied volatilities* for this stock using the Black-Schole objects and the Newton Raphson Solver.

- Plot the following graphs:
  - Implied Volatilities against strike price for options with same maturity.
  - Implied Volatilities against different maturity dates with same strike price.
  - Plot the graphs for both puts and calls.
Extending the class

• Extend this class to implement Normal, Capped LogNormal, and Floored Normal models.

• Call these classes Normal, CappedLogNormal, and FlooredNormal.

• Notice that for the classes CappedLogNormal and FlooredNormal the constructor will have to take an extra argument.
• Use the idea of compound state-prices discussed earlier.

• For each node \((t, U)\) we have three compound state-prices
  
  \(- \lambda_0(t, U)\) (Compound state price of the node at the initial node \((0, 0)\)).
  
  \(- \lambda_u(t, U)\) (Compound state price of the node at the node \((1, 1)\)).
  
  \(- \lambda_d(t, U)\) (Compound state price of the node at the node \((1, 0)\)).
Consider the bond $B_{t+1}$.

The price of this bond at nodes $(0, 0)$, $(1, 1)$ and $(1, 0)$ is given by the following equations:

$$P(0, 0, t + 1) = \sum_{u=0}^{t} \lambda_0(t, u) \frac{1}{1 + r(t, u)}$$

$$P(1, 1, t + 1) = \sum_{u=0}^{t} \lambda_u(t, u) \frac{1}{1 + r(t, u)}$$

$$P(1, 0, t + 1) = \sum_{u=0}^{t} \lambda_d(t, u) \frac{1}{1 + r(t, u)}$$
Updating State Prices

- Each time you solve for the parameters $r[t]$ and $k[t]$ you have to update the state prices.

- $\lambda_0(t + 1, u)$ is computed using the forward equation given below:

\[
0.5 \left( \frac{\lambda_0(t, u)}{1 + r(t, u)} + \lambda(t, u - 1) \frac{1}{1 + r(t, u - 1)} \right)
\]

Similar equation holds for $\lambda_u$ and $\lambda_d$.

- Handle the boundary nodes separately. (nodes $(t + 1, 0)$ and $(t + 1, t + 1)$).
Overall Algorithm

• **Base case**
  The short rate at the initial node \((0, 0)\) is the yield of the bond \(B_1\).

• **Inductive Step**
  – Assume we have computed the short-rates and the compound state prices at the nodes corresponding to time less than \(t\).
  – Find the parameters \(r[t]\) and \(k[t]\) by matching the yield and the yield volatilities of the bond \(B_{t+1}\).
  – Use the state prices \(\lambda_0(t-1, u), \lambda_u(t-1, u), \) and \(\lambda_d(t-1, u)\) to compute the price of the bond \(B_{t+1}\) at the nodes \((0, 0)\), \((1, 1)\), and \((1, 0)\).
  – Compute the compound state prices \(\lambda_0(t, u), \lambda_u(t, u), \) and \(\lambda_d(t, u)\) using the forward equation.