A C++ Encoded Hull-White Interest Rate Tree-Builder

John H. Li[1]
Duke University
Durham, NC
April 15, 2002

[1] John Li graduated from Trinity College, Duke University Class of 2002 with and BS degree and High Distinction honors in Economics. He also holds a minor in Computer Science. He is employed by Deutsche Bank and will be working in the Corporate Derivatives group of the Global Markets Division. He will reside in London and New York City.

Acknowledgement

I would like to gratefully thank my independent study advisor, Dr. Bjorn Eraker, for recommending this research topic and providing me excellent guidance during the study. With his enthusiasm and great efforts to explain things clearly and simply, he helped to make the esoteric parts of finance understandable for me during my financial engineering independent study. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching and many ideas.

I wish to acknowledge Deutsche Bank's Corporate Derivatives desk for providing me with up-to-date term structures and market prices for specific interest rate derivatives at my request. I particularly wish to express appreciation to my future manager and head of the desk, Raj Bhattacharrya, for his enthusiastic supervision during this project.

I am indebted to my uncle and Johnson Graduate School of Management professor, Dr. Charles Lee, for his continued support and assistance in the preparation of this manuscript. Gratitude is also extended to Brian Bevan and John Mayer for their support.

Lastly and most importantly, I wish to thank my parents, Dr. Pei-Luen Li and Ling Li. I am forever indebted to them for their understanding, endless patience and encouragement when it was most required. To them I dedicate this thesis.

Abstract

The Hull-White model is a single-factor, no arbitrage approach to modeling the term structure of interest rates. It models the term structure by describing the evolution of the short rate, or the instantaneous rate of interest. Implementing this model results in a trinomial pricing tree that can be used to price complex interest rate derivatives such as options on swaps and bonds. The difficulty of this model lies in its relative complexity and multi-stage implementation. The model's advantage over similar models is its calculation speed.

This paper does not develop a new method but rather explains the author’s original implementation of the algorithm behind the Hull-White interest rate model using C++ programming code. The paper will first explain the generalized Hull-White model. It will then explain the construction of the Hull-White tree by correlating each step in the model's two-stage tree-building procedure with the C++ program architecture. The paper will then run the Hull-White model using current market data to price a one-year bond option on a ten-year zero-coupon bond and briefly explain the discrepancies in the instrument's market price and calculated price. The paper will also describe some of the limitations of the program and discuss possible future improvements.

Introduction

Interest rate derivatives are financial instruments whose payoffs are a function of the term structure of interest rates. Like foreign exchange and equity derivatives, interest rate derivatives are extremely important in today's economy for both risk management and speculative purposes. As a result, many new financial products have been created to meet the needs of end users such as corporations, banks, money managers and insurance companies. The challenge lies in finding an accurate procedure for pricing and hedging these products.

Pricing interest rate products is more complicated than pricing foreign exchange or equity derivative for several reasons. A primary reason is because for the valuation of many interest rate products, it is necessary to develop a model describing the behavior of today's entire term structure of interest rates[2]. As a result, a no-arbitrage term structure model must be created. The no-arbitrage property ensures that the value generated by the term structure model is exactly consistent with the market's bond prices implied by the zero-coupon yield curve. There are generally two different approaches to building no-arbitrage yield curve models: describing the evolution of the forward rate and describing the evolution of the short-term interest rate[3]. Each approach has its strengths and weaknesses.

The first approach describes the evolution of the instantaneous forward rate and was first developed by Heath, Jarrow, and Morton. The Heath-Jarrow-Morton (HJM) model considers the current term structure with a user specification of the volatility of forward rates to build a tree to model the behavior of the instantaneous forward rate. "The HJM tree of forward rates is the fundamental unit representing the evolution of interest rates in a given period of time[4]." Brace, Gatarek and Musiella (1997) extended this model into the LIBOR Market Model (LMM) that allows one to apply the model to observable non-instantaneous forward rates, such as the 3-month LIBOR. The approach behind both models results in an easily understandable tree implementation that permits as complex of a volatility structure as desired. "The HJM-LMM models provide approaches that give the user complete

[2] Hull, J., Options, Futures and Other Derivatives, Prentice Hall, 2000. pg 530.
[2] Hull, J. and A. White, "The General Hull-White Model and Super Calibration," Financial Analysts Journal, Vol. 57, No. 6 (Nov/Dec 2001), pg 37.

freedom in choosing the volatility term structure[5]." The weakness of the approach is that the trees created cannot be represented as recombining trees; calculations are based on a statistical process. HJM-LMM models are thus difficult to implement by any other means other than Monte-Carlo simulation[6].   This makes the accurate pricing of interest rate derivatives both time and computationally inefficient.

The second approach to modeling the yield curve is to take the initial term structure as given and describe how the short-term interest rate, the rate that applies over the next short interval of time, can evolve[7]. Models of the short rate are implemented in the form of a recombining tree similar to the stock price tree first developed by Cox, Ross and Rubinstein (1979) and do not need to be statistically calculated[8]. As a result, interest rate trees implemented using this approach are both robust and computationally fast; most models used for routine interest rate derivatives pricing are based on this approach. The method's weakness is its relative complexity and lack of flexibility in the user specification of the volatility environment. Example models of the short-term interest rate include the Ho and Lee model as well as the Hull and White model. This study examines the implementation of the single-factor Hull-White model.

The Hull-White Model

The single-factor, no-arbitrage Hull-White model is a model where the function of the instantaneous interest rate (short rate), *r*, follows the following stochastic differential equation:

$$dy = (?(t)-ay)dt + sdz \qquad (1)$$

where $y = f(r)$ is some function of the short rate, $?(t)$ is a function of time chosen so that the model provides an exact fit to today's zero-coupon yield curve, a is the mean reversion rate, dt is a small change in time, s is the annual standard deviation of the short rate and dz is a Wiener process[9]. A

---

[4] "Heath-Jarrow-Morton Model," http://www.mathworks.com/access/helpdesk/ help/toolbox/finderiv/using8.shtml, The MathWorks Inc., 2001.
[5] Hull Options, pg 618.
[6] Hull "The General Hull-White" pg 38.
[7] Hull, Options, pg. 596.
[8] Hull, "The General Hull-White," pg. 39.
[9] "CurveTrader Online Help," http://www.powerfinance.com/help. Leap of Faith Research, Inc., 1998.

trinomial tree is used to construct a discrete time and space Markov approximation of the state variable y.

The parameters a and s make up the volatility parameter (state factor) that is chosen by the user to calibrate the model to the market prices of a set of actively traded interest-rate derivatives[10]. The model assumes that the short rate is normally distributed and subject to mean reversion, the well documented phenomenon where interest rates appear to drift to a long-run average level over time. The model also assumes there are no market frictions, taxes nor transaction costs. It is assumed that assets are perfectly divisible and trading takes place at discrete time steps[11].

For this study the author identifies the short rate as the state variable:

$$y = r$$

A downside to setting the state variable equal to the short rate is the possibility of negative interest rates. However, in practice this probability is small. Furthermore, when $y = r$ and a ? 0, the model reduces to the analytically tractable model:

$$dr = (?(t)-ar)dt + sdz \tag{2}$$

as shown and proven in Hull and White (1990). The model is analytically tractable because it allows for the pricing at time t of a zero-coupon bond maturing at time T in terms of the initial term structure and r at time t[12]. Specifically, Hull and White proved:

$$P(t,T) = A(t,T)e^{-B(t,T)r(t)} \tag{3}$$

where:

$$B(t,T) = \frac{1-e^{-a(T-t)}}{a} \tag{4}$$

$$\ln A(t,T) = \ln\frac{P(0,T)}{P(0,t)} - B(t,T)\frac{\partial \ln P(0,t)}{\partial t} - \frac{1}{4a^3}s^2\left(e^{-aT}-e^{-at}\right)^2\left(e^{2at}-1\right) \tag{5}$$

[10] Hull, "The General Hull-White," pg 39.
[11] Leippold, M. and Z. Wiener, "The Term Structure of Interest Rates II: The Hull-White Trinomial Tree of Interest Rates, 1999, pg 1.
[12] Hull, J. and A. White, "Using Hull-White Interest Rate Trees," Journal of Derivatives, Vol. 3, No. 3, (Spring 1996), pp. 28, 30.

In practice, bond prices are usually computed in terms of *R(t)*, the discrete *?t*-period at time *t* rather than *r*. Hull and White convert equation (3) to:

$$P(t,T) = \hat{A}(\mathrm{t},\mathrm{T})e^{-\hat{B}(t,T)R(t)} \tag{6}$$

where:

$$\ln\hat{A}(t,T) = \ln\frac{P(0,T)}{P(0,t)} - \frac{B(t,T)}{B(t,t+\Delta t)}\ln\frac{P(0,t+\Delta t)}{P(0,t)} - \frac{s^2}{4a}\left(1-e^{-2at}\right)B(t,T)\left[B(t,T)-B(t,t+\Delta t)\right] \tag{7}$$

$$\hat{B}(t,T) = \frac{B(t,T)}{B(t,t+\Delta t)}\Delta t \tag{8}$$

An alternative would be to set:

$$y = ln(r)$$

This prevents the chance of negative interest rates, but has no analytic tractability[13].

The Hull-White model can be viewed as an extension of the Ho and Lee model with mean reversion rate of *a*; when $a = 0$, the model reduces to the Ho-Lee model. Furthermore, when written as:

$$dy = a\left(\frac{q(t)}{a} - y\right)dt + s\,dz$$

the Hull-White model can be characterized as an extension of the Vasicek model with a time-dependent reversion level of $\frac{q(t)}{a}$ at rate *a*[14].

A trinomial interest rate tree is a discrete representation of the stochastic process for the short rate[15]. The C++ implementation of the Hull-White model roughly follows the two-stage procedure for constructing trinomial trees. Each stage will be outlined.

[13] Hull, Options, pg. 587.
[14] Hull, Options, pg. 574.
[15] Hull, Options, pg. 578.

Theoretical Implementation- First Stage

The Hull-White interest rate tree is a discrete-time representation of the stochastic process for the short rate. Each step on the tree represents a point in time, t_i. The time step on the tree is $?t=t_{i+1}-t_i$. Each node on the interest rate tree at time t_i with a relative tree position j is denoted as node (i,j).

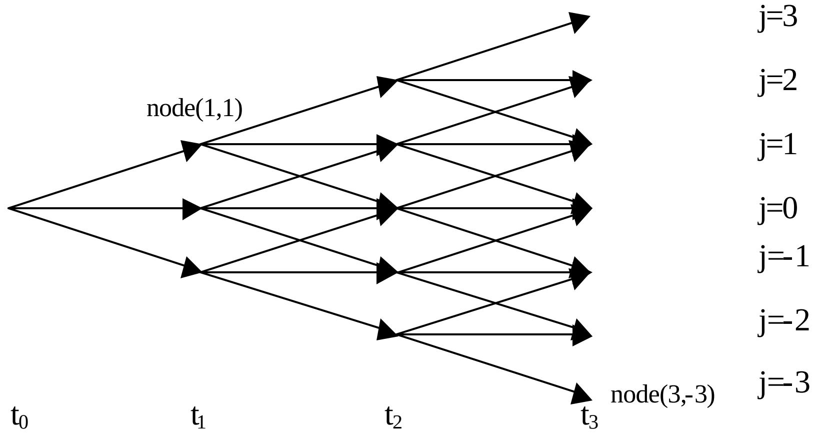

Figure 1

It is assumed that the $?t$-period interest rate at time t, $R(t)$, follows the same process as the short rate, r:

$$dR = (?(t)-aR)dt + sdz \qquad (9)$$

The goal of the first stage is to construct a tree such that the central node at each time step, node $(i,0)$, has a value of zero[16]. This is achievable by defining a new variable R^* obtained from R by setting both setting $?(t)$ and the initial value of R equal to zero[17]. R^* follows the process:

$$dR^* = -aR^*dt + sdz \qquad (10)$$

where the expected value of $R^*(t+\Delta t)-R^*(t)=-aR^*(t)dt$; its variance is equal to $s^2?t$[18].

Setting $?(t)$ to zero and the initial value of R^* is zero results in a process that is mean reverting to zero. If R^* starts at zero the unconditional expected value of R^* at all future times is zero (Hull 4). Define $?R$ as the interest rate spacing between the nodes on the tree, fit to represent the volatility of R, computed to the Hull-White recommended specifications for error minimization:

$$\Delta R = s\sqrt{3\Delta t}$$

[16] Hull, “Using Hull-White,” pg. 28.
[17] Hull, “Using Hull-White,” pg. 28.
[18] Hull, Options, pg. 581.

The upward, middle and downward branching probabilities for each node are then set to match the expected value and the standard deviation of the change in *R** for the process in equation (10)[19]. Solving this system of equations leads to the following probabilities.

$$?_{u} = \frac{1}{6} + \frac{a^2 j^2 \Delta t^2 - aj\Delta t}{2}$$

$$?_{m} = \frac{2}{3} - a^2 j^2 \Delta t^2$$

$$?_{d} = \frac{1}{6} + \frac{a^2 j^2 \Delta t^2 + aj\Delta t}{2}$$

The model incorporates mean reversion by setting a limit variables j_{max} equal to the smallest integer greater than $\frac{1.84}{a\Delta t}$ and $j_{min} = 0 - j_{max}$. If x at any node is greater than j_{max} or less than j_{min}, the branching will switch to a downward or upward branching pattern, respectively.

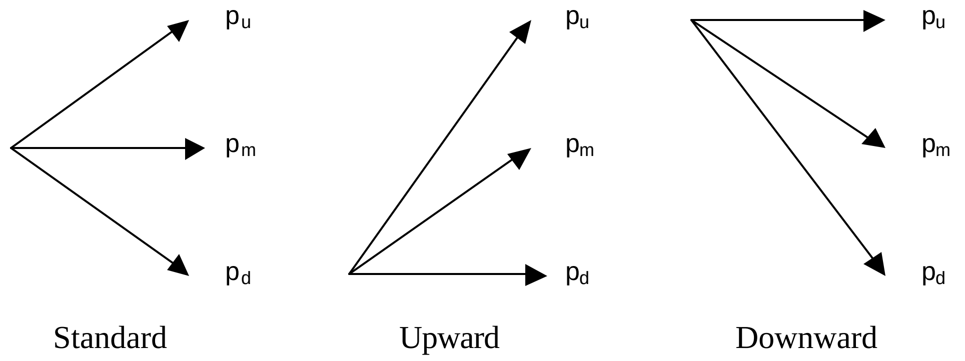

Figure 2

The probabilities for an upward-branching method are:

$$?_{u} = \frac{1}{6} + \frac{a^2 j^2 \Delta t^2 + aj\Delta t}{2}$$

$$?_{m} = -\frac{1}{3} - a^2 j^2 \Delta t^2 - 2aj\Delta t$$

[19] Hull, "Using Hull-White," pg. 27.

$$? _d = \frac{7}{6} + \frac{a^2 j^2 \Delta t^2 + 3aj\Delta t}{2}$$

The probabilities for a downward-branching method are:

$$? _u = \frac{7}{6} + \frac{a^2 j^2 \Delta t^2 - 3aj\Delta t}{2}$$

$$? _m = -\frac{1}{3} - a^2 j^2 \Delta t^2 + 2aj\Delta t$$

$$? _d = \frac{1}{6} + \frac{a^2 j^2 \Delta t^2 - aj\Delta t}{2}$$

Hull-White (1990) proved this equation prevents any of the probabilities of any node from being negative. The end product of the first stage is a recombining tree with a time step of $?t$ and vertical spacing of $?R$ that is symmetrical around 0.

Implementation- 1st Stage

This first stage is created by iterating through the tree twice- once to construct and connect nodes and once to add each node's respective probabilities. The end result is a tree represented as a vector of nodes.

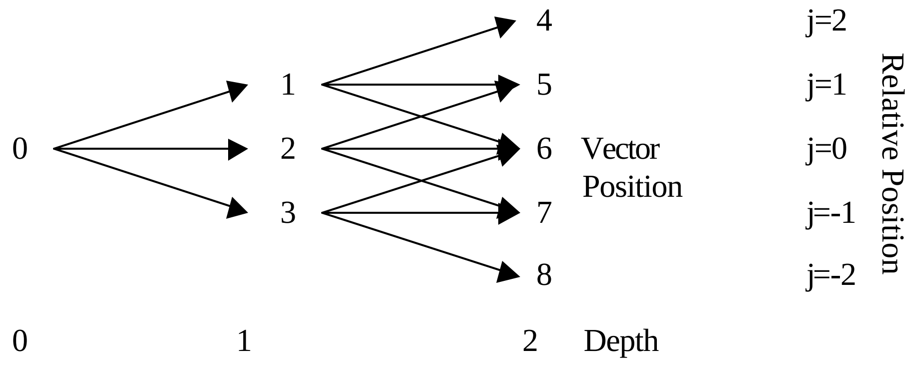

Figure 3

Vector Position	0	1	2	3	4	5	6	7	8
Depth	0	1	1	1	2	2	2	2	2
Relative Position	0	1	0	-1	2	1	0	-1	-2

Table 1

Each node is represented in C++ as a struct containing the following data:

```
struct myNode
{
  int   nodeNumber;        // node's numerical position on the vector
  int   depth;             // equals the depth of the node
  int   relativePosition;  // equals the relative position of the node
  float rate;              // equals the delta(t) rate for the node
  float presentValue;      // equals the node's Arrow-Debreu price
  float alpha;             // equals g(t)
  float pu;                // probability of branching up
  float pm;                // probability of branching middle
  float pd;                // probability of branching down

  myNode * up;             // the node connected via the highest branch
  myNode * middle;         // the node connected via the middle branch
  myNode * down;           // the node connected via the lowest branch
....
```

The program's first iteration uses the user-inputted term structure. The program will build a tree whose depth, *D*, matches the depth of the term structure inputted and assumes that each interest rate given is for time *D*?t*. The time step, *?t*, is a user input. For simplicity, the root node is hard coded:

```
void HullTree::connectNodes(tvector<float> structure)
{
  // initializing the root node
  maxdepth = 0;
  myTree[0]->depth  = 0;
  myTree[0]->relativePosition = 0;
  myTree[0]->presentValue = 1.00000;

  if (structure.size() > 1)
  {
    myTree[0]->up     = myTree[1];
    myTree[0]->middle = myTree[2];
    myTree[0]->down   = myTree[3];
    myTree[1]->depth  = 1;
    myTree[2]->depth  = 1;
    myTree[3]->depth  = 1;
    myTree[1]->relativePosition = 1;
    myTree[2]->relativePosition = 0;
    myTree[3]->relativePosition = -1;
  }
...
```

For robustness, the calling one of two functions creates all other depths: *expand*, which expands the number of nodes vertically, and *maintain*, which maintains the number of nodes during the next time step. A portion of *maintain* is shown below:

```
int HullTree::maintain(int lastNodeNumber, int nodesInDepth, int tempDepth)
```

```
{
  int beginningNode = lastNodeNumber-nodesInDepth+1;
…
  // prevent the top node from expanding
  myTree[beginningNode]->up       = myTree[beginningNode + nodesInDepth];
  myTree[beginningNode]->middle   = myTree[beginningNode + nodesInDepth + 1];
  myTree[beginningNode]->down     = myTree[beginningNode + nodesInDepth + 2];

  // expand the middle nodes accordingly
  for (int i = 1; i <= nodesInDepth - 2; i++)
  {
    myTree[beginningNode + i]->up       = myTree[beginningNode + i +
nodesInDepth - 1];
    myTree[beginningNode + i]->middle   = myTree[beginningNode + i +
nodesInDepth];
    myTree[beginningNode + i]->down     = myTree[beginningNode + i +
nodesInDepth + 1];
  }

  myTree[lastNodeNumber]->up       = myTree[lastNodeNumber + nodesInDepth -
2];
  myTree[lastNodeNumber]->middle   = myTree[lastNodeNumber + nodesInDepth -
1];
  myTree[lastNodeNumber]->down     = myTree[lastNodeNumber + nodesInDepth];

  // adding relativePosition
  int divider = (nodesInDepth - 1) / 2;
  for (int a = 0; a < nodesInDepth; a++)
  {
    myTree[beginningNode + nodesInDepth + a]->relativePosition = divider - a;
  }
  return nodesInDepth;
}
```

The function, *connectNodes*, chooses to either expand or maintain the width of the tree by comparing the interest rate of node*(i,j)* to j_{min} and j_{max} limits.

```
// tempNode is the bottom-most node of the nodes in depth d
if (100 * tempNode->relativePosition * deltaR > jMax || 100 * tempNode-
>relativePosition * deltaR < jMin)
  {
    maintain(lastNodeNumber, nodesInDepth, count);
  }
  else expand(lastNodeNumber, nodesInDepth, count);
...
```

Adding branching probabilities requires a second iteration through the vector of nodes, this time adding the up, middle and down probabilities to each node based on the above formulas:

```
void HullTree::udm(myNode * node)
{
  if (node->relativePosition * deltaR > jMax)
  {
    node->pu = (7.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)-(3 * meanReversion * node->relativePosition*deltaT))/2);
    node->pm = (0.00000-(1.00000/3.00000)) - (meanReversion *
meanReversion * node->relativePosition * node->relativePosition * deltaT *
deltaT)+(2 * meanReversion * node->relativePosition*deltaT);
    node->pd = (1.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)-(meanReversion * node->relativePosition*deltaT))/2);
  }

  else if (node->relativePosition * deltaR < jMin)
  {
...
  }
  else
  {
    node->pu = (1.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)-(meanReversion * node->relativePosition*deltaT))/2);
    node->pm = (2.00000/3.00000) - (meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT);
    node->pd = (1.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)+(meanReversion * node->relativePosition*deltaT))/2);
  }
}
```

The end product is a tree represented as a vector of nodes, with each node containing pointers to other nodes within the vector and probabilities attached to the pointers.

Vector Position	0	1	2	3	4	5	6	7	8
R*	0.000%	1.732%	0.000%	-1.732%	3.464%	1.732%	0.000%	-1.732%	-3.464%
? $_u$	0.16667	0.121667	0.166667	0.221667	0.086667	0.121667	0.166667	0.221667	0.286667
? $_m$	0.66667	0.656667	0.666667	0.56667	0.626667	0.656667	0.666667	.0656667	0.62666
? $_d$	0.16667	0.221667	0.166667	0.121667	0.286667	0.221667	0.166667	0.121667	0.08666
Up (Vector Position)	1	4	5	6					
Middle	2	5	6	7					
Down	3	6	7	8					
Depth	0	1	1	1	2	2	2	2	2
Relative Position	0	1	0	-1	2	1	0	-1	-2

Table 2

Theoretical Implementation- 2nd Stage

The end product of the first stage tree is a tree that represents the process (10):

$$dR^* = -aR^*dt + sdz$$

The second stage in the tree construction is to convert the generic level tree into a tree whose interest rates are calibrated to the initial term structure. "Within the calibrated tree the prices of the zero bonds that mature at each tree time-period coincide with those implied by the yield curve currently observed in the market[20]." This is accomplished by defining *g:*

$$g(t) = R(t) - R^*(t) \qquad (11)$$

where:

$$dg = [q(t) - ag(t)]dt$$

Since *g(t)* is a function *?(t)* and the function *?(t)* is selected so that the model fits the term structure, the de facto process is to adjust the nodes in the tree so that it correctly prices discount bonds of all maturities[21].

In order for the interest rate tree to be exactly consistent to the initial term structure, *g* for each time-step must be calculated iteratively. Given a tree of node*(i,j)* where $(0 \leq i \leq n\, ; -m_i \leq i \leq m_i)$, define the following:

$R^*(i,j)$: value of R* at node*(i,j)*

$R(i,j)$: value of R at node*(i,j)*

$$g(t_i) = g_i = R(i,j) - R^*(i,j)$$

R(0,0) is equal to the term structure rate for 1**?t*. As a result, if *?t* = 1, *R(0,0)* is the one year rate on the yield curve and *R(i,j)* is the one year rate for year *i+1* on the interest rate tree.

Hull and White define *Q(i,j|h,k)* as an Arrow-Debreu (AD) price, the present value at node*(h,k)* of a security that pays off $1 at node*(i,j)* and zero at any other node[22]. *Q(i,j|h,k)* is equal to the probability of reaching *i,j* from *h,k*, discounted at *R(h,k)*. If *h = i-1*:

$$Q(i,j|i-1,k) = p(i, j \mid i-1, k)e^{-(x_{i-1,k}+g_{i-1})(t_i - t_{i-1})}$$

Q(i,j|0,0) can be denoted as Q_{ij} as denotes as the root AD price for node*(i,j)*:

[20] Leippold, "The Term Structure of," pg. 10.
[21] Hull, "The General Hull-White," pg. 42.
[22] Hull, "The General Hull-White," pg. 42.

$$
\begin{aligned}
Q_{ij} &= \sum_k Q(i, j \mid i-1, k) Q_{i-1,k} \\
&= \sum_k p(i, j \mid i-1, k) e^{-(x_{i-1,k}+g_{i-1})(t_i - t_{i-1})} Q_{i-1,k}
\end{aligned}
\tag{12}
$$

Q_{ij} is determined for every node j at step i[23]. Only after computing the Arrow-Debreu prices for node*(i,j)* one can begin to calibrate the interest rate tree to the term structure.

Calibration of the interest rate tree is accomplished by matching the price of a discount bond of length t_{i+1}, *P(i+1)*, to the summation of all of the Arrow-Debreu prices of nodes at time t_i, each Arrow-Debreu price discounted at each node's *R(i,j)* rate. For example, if *?t* =0.5, the summation of the values for *Q(1,1), Q(1,0)* and *Q(1,-1)* discounted at ($R^*_{1,1} + g_1$), ($R^*_{1,0} + g_1$) and ($R^*_{1,-1} + g_1$), respectively, would be calibrated to the price of a (2*.5) = 1 year discount bond.

Denote *P(i+1)* as the price at node*(0,0)* of a discount bond that pays \$1 at time t_{i+1}. *P(i+1)* is computed using the term structure:

$$P_{i+1} = e^{-(y_{i+1})(t_{i+1})}$$

where y_{i+1} is the interest rate on the current term structure for t_{i+1}. *P(i+1)* is matched to the summation of the present value of *Q(i,j)* prices. Denote *V(i,j)* as the present value of *Q(i,j)*, each value discounted to each node's *R = R*(t) + g(t)* rate:

$$V_{ij} = e^{-(x_{ij}+g_i)(t_{i+1}-t_i)}$$

Hull and White calculate the present value as

$$
\begin{aligned}
P_{i+1} &= \sum_j Q_{ij} V_{ij} \\
&= \sum_j Q_{ij} e^{-(x_{ij}+g_i)(t_{i+1}-t_i)}
\end{aligned}
\tag{13}
$$

The formula can be rearranged to solve for g_i:

$$g_i = \frac{\ln \sum_j Q_{ij} e^{-j\Delta R \Delta t} - \ln P_{(i+1)}}{\Delta t} \tag{14}$$

Denote the Arrow Debreu price at node*(0,0)* = 1. Based on (14), *R(0,0)* can be calculated. The next iteration uses *R(0,0)* to price *Q(1,1), Q(1,0)*, and *Q(1,-1)*. Only after *Q(1,1), Q(1,0)* and *Q(1,-1)* are

[23] Hull ,"The General Hull-White," pg. 42-43

determined can *R(1,1)*, *R(1,0)* and *R(1,-1)* be calculated through calibration techniques. This iterative technique is used until *i* = *n* total steps and the interest rate tree is complete.

Implementation- 2nd Stage

The completion of the first stage is an interest rate tree centered around zero. The second stage is implemented by iterating through the vector of nodes one depth at a time. The function that orchestrates the iterative process is function *addRates*. During the iteration of each depth within *addRates*, a temporary vector, *depthVector,* containing all nodes at time t is created.

```
void HullTree::addRates(tvector<float> structure)
{
  ...
  int tempDepth = 1;

  for (int a = 1; a < myTree.size(); a++)
  {
    // we're put all nodes of the same depth on a vector
    if (myTree[a]->depth == tempDepth)
    {
      depthVector.push_back(myTree[a]);
    }
    else
    {
      tempDepth++;
      ...
```

Each node within *depthVector* has its Arrow-Debreu prices calculated using each predecessor node's *R* rates as an input. Because C++ pointers are one-directional—nodes that node*(a,b)* points to as its highest, middle and lowest branch nodes do not have pointers back to node (a,b)—a search algorithm is created to find the predecessors of each node in depthVector and return these predecessor nodes through a temporary vector, *tempVector*.

```
tvector<myNode*> HullTree::findConnectors(myNode * node)
{
  tvector<myNode *> tempVector;
  for (int count = 0; count < myTree.size(); count++)
  {
    if (myTree[count]->up == node || myTree[count]->middle == node
||myTree[count]->down == node)
    {
      myNode * tempNode = myTree[count];
      tempVector.push_back(tempNode);
    }
  }
  return tempVector;
}
```

Given *tempVector*, the Arrow-Debreu price for the current node[a] in *depthVector* is calculated.

```
float HullTree::addPresentValue(myNode * node, tvector<myNode *> depthVector,
tvector<float> structure)
{
  float alpha= 0.00000;
  // going through those connecting nodes, finding Q for depthVector[a]
  for (int b = 0; b < tempVector.size(); b++)
  {
    if (tempVector[b]->up == depthVector[a])
    {
      depthVector[a]->presentValue = tempVector[b]->pu *
exp(0.00000 - (tempVector[b]->rate)*deltaT) * tempVector[b]->presentValue +
depthVector[a]->presentValue;
    }
    else if (tempVector[b]->middle == depthVector[a])
    {
      depthVector[a]->presentValue = tempVector[b]->pm *
exp(0.00000 - (tempVector[b]->rate)*deltaT) * tempVector[b]->presentValue +
depthVector[a]->presentValue;
    }
    else if (tempVector[b]->down == depthVector[a])
    {
      depthVector[a]->presentValue = tempVector[b]->pd *
exp(0.00000 - (tempVector[b]->rate)*deltaT) * tempVector[b]->presentValue +
depthVector[a]->presentValue;
    }
   ...
```

This process is run for each node in *depthVector*. Using *depthVector*'s Arrow-Debreu prices, g_i is determined.

```
  ...
  g = 0;
  for (int c = 0; c < depthVector.size(); c++)
  {
    g = g + depthVector[c]->presentValue * exp(0.00000 - (deltaR *
deltaT * depthVector[c]->relativePosition));
  }
  g = log(g);
  g = g - log(bondPrices[depthVector[0]->depth]);
  g = g / deltaT;
  return g;
}
```

This value is returned to the central node of time t_i, node*(i,0)* in *addRates*. The central node's *R*(i,0)* value is zero; its new calibrated interest rate, *R(i,0),* is set to g_i. The remaining nodes at t_i have their *R(i,j)* values calculated using function *addRemainingRates*.

```
void HullTree::addRemainingRates(myNode * tempNode, tvector<myNode *> 
depthVector)
{
  for (int a = 0; a < depthVector.size(); a++)
  {
    depthVector[a]->rate = depthVector[a]->relativePosition * deltaR + 
tempNode->rate;
  }
}
```

depthVector gets erased after each node in the vector has its *R(i,j)* values calculated. The vector gets recreated for the next time period, beginning the iteration for time *i+1*. This iterative process continues until the tree is calibrated. The C++ implementation of both stages is illustrated in Appendix A.

Pricing Analysis

The Hull-White tree generated allows for the pricing of numerous interest rate derivatives. Many of such instruments have their values derived from the U.S. Treasury term structure, shown below for April 10, 2002:

Term Structure	
Maturity	Yield (%)
3 month	1.7
6 month	1.97
2 year	3.44
5 year	4.6
10 year	5.25
30 year	5.71

Table 3

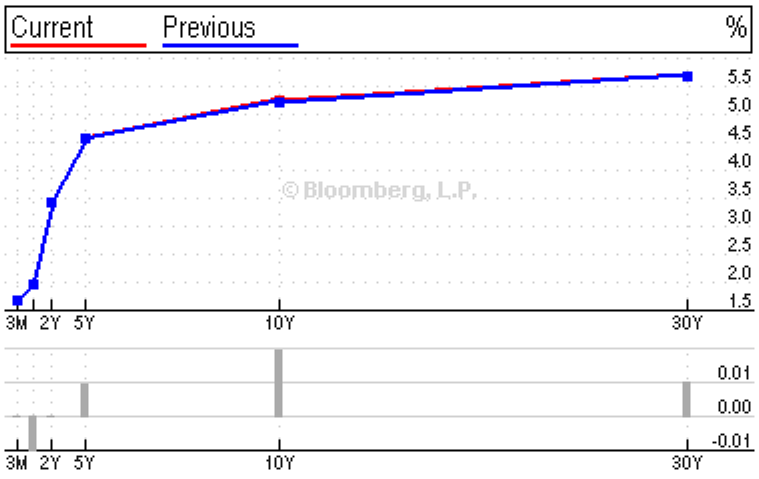

Figure 4

One such derivative is a European put option on a zero-coupon bond. A common instrument using a portfolio of such options is an interest rate cap, an option that provides a payoff whenever a specified interest rate exceeds a certain level[24]. The specific derivative priced is a one-year put option on a zero-coupon bond that will expire in ten years. The notional price is $1000. The strike price is at the money; the current price of the bond is $592.74. It is assumed that a = 1 and s = 0.01.

Consider the construction of an eight-step, one-year tree. The time step is $?t$ = 1 / 8 = 0.125 years and the rate step, $?r = s\sqrt{3\Delta t}$ = 0.0061237. j_{min} and j_{max} values are calculated as the smallest integer greater than 0.184 / $a\,?t$ = 15. Inputting a, s, $?t$ and the term rates for 1/8, 2/8, 3/8...8/8 year into the C++ tree generator results in a completed interest rate tree with $?t$-period rate, R, at each node. Rates for maturities between given interest rates in Table 3 are calculated using linear interpolation. Table 4 presents the completed tree and corresponding $g = R - R^*$ rates; appendix B displays the program's true output in full detail.

	Branching Probabilities			**?t rate, R**							
j	**? up**	**? middle**	**? down**	**i = 0**	**i = 1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**
7	0.1267	0.6590	0.2142								0.0761
6	0.1320	0.6610	0.2070							0.0675	0.0700
5	0.1374	0.6628	0.1999						0.0589	0.0614	0.0638
4	0.1429	0.6642	0.1929					0.0503	0.0528	0.0552	0.0577
3	0.1486	0.6653	0.1861				0.0416	0.0442	0.0467	0.0491	0.0516
2	0.1545	0.6660	0.1795			0.0331	0.0355	0.0381	0.0405	0.0430	0.0455

[24] Hull, Options, pg. 665.

1	0.1605	0.6665	0.1730		0.0247	0.0270	0.0294	0.0320	0.0344	0.0369	0.0393
0	0.1667	0.6667	0.1667	0.0162	0.0185	0.0209	0.0232	0.0258	0.0283	0.0308	0.0332
-1	0.1730	0.6665	0.1605		0.0124	0.0148	0.0171	0.0197	0.0222	0.0246	0.0271
-2	0.1795	0.6660	0.1545			0.0086	0.0110	0.0136	0.0160	0.0185	0.0210
-3	0.1861	0.6653	0.1486				0.0049	0.0075	0.0099	0.0124	0.0148
-4	0.1929	0.6642	0.1429					0.0013	0.0038	0.0063	0.0087
-5	0.1999	0.6628	0.1374						0.0000	0.0001	0.0026
-6	0.2070	0.6610	0.1320							0.0000	0.0000
-7	0.2142	0.6590	0.1267								0.0000
			g	0.0162	0.0185	0.0209	0.0232	0.0258	0.0283	0.0308	0.0332

Table 4

Figure 5 is a graphical representation of the tree without branching lines between nodes.

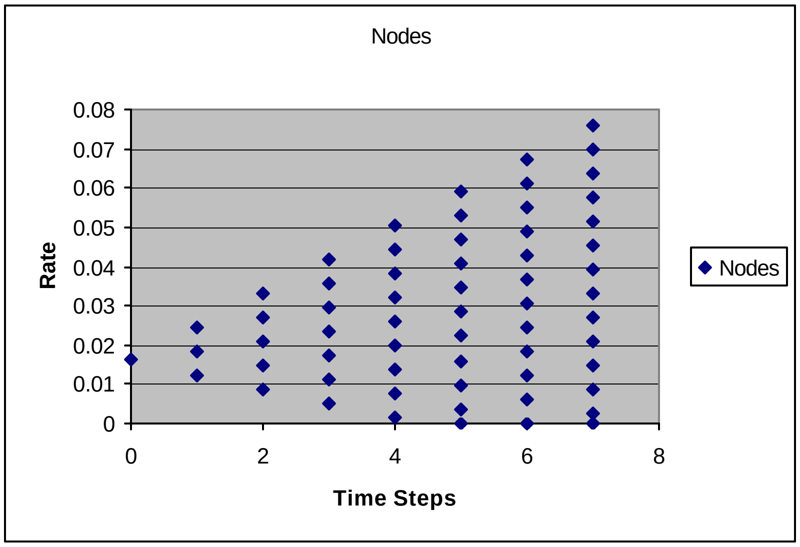

Figure 5

Next, the bond payoff, $P(t,T)$, at each terminal node ($t_i = 8$) must be calculated. This is done analytically through equations (3) through (8). Table 5 presents a spreadsheet which calculates the $P(t,T)$ values analytically.

User Inputs			Leaf Node Calculations	
			Leaf Node Rates	P(t,T)
t	*1*		*0.0761*	0.4764
T	*10*		*0.0700*	0.4941
deltaT	*0.125*		*0.0638*	0.5125
a	*0.1*		*0.0577*	0.5316
sigma	*0.01*		*0.0516*	0.5514

Term Structure Calculations			
P(0,t)	*0.9757*	*0.0455*	0.5720
P(0,T)	*0.5927*	*0.0393*	0.5933
P(0,t+deltaT)	*0.9714*	*0.0332*	0.6154
deltaR	*0.0061*	*0.0271*	0.6383
		0.0210	0.6621
		0.0148	0.6867
Calculations		*0.0087*	0.7123
B(t,T)	5.9343	*0.0026*	0.7388
Bhat(t,T)	5.9715	*0.0000*	0.7503
B(t,t+deltat)	0.1242	*0.0000*	0.7503
Ahat(t,T)	0.7503		

Table 5

User-inputted values are italicized. Appendix C shows the calculation of each formula using Microsoft Excel cell references.

Calculating the value of the put option whose strike price is the current bond price, \$592.74, is accomplished by calculating the option payoff, *1000*MAX(0.59274 - P(t,T), 0)*, where MAX takes the higher value of 0.59274-*P(t,T)* or zero. Table 6 provides these payoffs.

Leaf Node Calculations			
Strike Price (x1000)	Leaf Node Rates	P(t,T)	Put Option Price
0.592739659	0.0760759	0.476392195	116.3474638
	0.0699522	0.494135092	98.60456681
	0.0638285	0.512538811	80.20084794
	0.0577048	0.531627964	61.11169533
	0.0515811	0.551428079	41.31158047
	0.0454573	0.571965976	20.77368251
	0.0393336	0.593268453	0
	0.0332099	0.615364325	0
	0.0270862	0.638283142	0
	0.0209624	0.662055951	0
	0.0148387	0.686713766	0
	0.00871498	0.712290029	0
	0.00259126	0.738818865	0
	0	0.750339983	0
	0	0.750339983	0

Table 6

Each option price must then be discounted back through the tree. The process works in reverse: from end nodes*(i,j)*, the value at node*(i-1,k)* is computed as the discounted sum of all of its branching nodes' values. Specifically,

Standard Branching: $$v_{i-1,k} = \left(\Pi_{up} v_{i,k+1} + \Pi_{midde} v_{i,k} + \Pi_{down} v_{i,k-1}\right) e^{-(R_{i-1,k})(t_i - t_{i-1})}$$

Downward Branching: $$v_{i-1,k} = \left(\Pi_{up} v_{i,k} + \Pi_{midde} v_{i,k-1} + \Pi_{down} v_{i,k-2}\right) e^{-(R_{i-1,k})(t_i - t_{i-1})}$$

Upward Branching: $$v_{i-1,k} = \left(\Pi_{up} v_{i,k} + \Pi_{midde} v_{i,k+1} + \Pi_{down} v_{i,k+2}\right) e^{-(R_{i-1,k})(t_i - t_{i-1})}$$

Table 7 iterates from $i = 7$ backwards to $i = 0$ and discounts at each time step using the rates and transition probabilities in Table 4.

	Put Option Price							
j	**i = 0**	**i = 1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**
7								116.3475
6							96.3210	98.6046
5						76.4811	78.3106	80.2008
4					56.7367	58.1482	59.6071	61.1117
3				37.4190	38.1860	39.1058	40.1843	41.3116
2			20.8372	20.5673	20.3003	20.0950	20.1095	20.7737
1		9.7706	8.9816	8.0483	6.9027	5.4161	3.3187	0.0000
0	**4.0198**	3.3902	2.7176	2.0045	1.2660	0.5512	0.0000	0.0000
-1		0.8362	0.5363	0.2818	0.0951	0.0000	0.0000	0.0000
-2			0.0619	0.0170	0.0000	0.0000	0.0000	0.0000
-3				0.0000	0.0000	0.0000	0.0000	0.0000
-4					0.0000	0.0000	0.0000	0.0000
-5						0.0000	0.0000	0.0000
-6							0.0000	0.0000
-7								0.0000

Table 7

The price of the put option is calculated to be \$4.0198, or 0.402% of the bond's notional value.

Comparison

The market's ask price for the bond option is \$4.39, or 0.439% of the bond's notional. This is a \$0.37 increase over the tree-calculated option price. There are several explanations for this difference, listed in decreasing hypothesized significance:

- Illiquidity premium- While puts and calls on bonds are quite common, many are bundled into portfolios to create easily hedged interest rate caps and floors. One-year options for zero coupon bonds are seldom traded alone in the over-the-counter market and as a result, there may be an extra illiquidity premium added to the price of the option. When asked how often options off the Treasury curve are traded, the analyst covering the desk replied that she prices bond options of duration of this length once a year.
- Pricing inaccuracy- The interest rate derivatives analyst covering short-term options did not have a model immediately available that was capable of pricing this particular option. As a result, she entered the yield curve into a European swaption model to generate the price of the bond option. It is unclear what possible differences in assumptions or calculations the analyst used to generate the market price.
- Differences in volatility assumptions- The Hull-White model contains a single factor, volatility, that can be modified to fit the market price of the derivative. Determining the volatility parameters a and s is known as model calibration and is out of the scope of the article. In this example, a and s are assumed to be .1 and .01, respectively. These numbers were chosen because they match the volatility parameters of examples priced by Hull, Leippold and Wiener. It is unknown if the volatility parameters chosen result in a minimal or near minimal goodness-of-fit measure[25]. Likewise, the market model priced was without calibration; it is unknown what volatility assumptions should be used.
- Time step inaccuracy- The interest rate tree generated in the exercise has only eight steps and is considered a rough approximation of the option price because each time step is equivalent to over 45 days. When calculated using 20 time steps (?t = 18.25 days) the price of the bond option increases to $4.16 (appendix D). It is shown that continuing to increase the resolution of the tree will continue narrow the difference between the market price and calculated price[26].
- Linear interpolation of interest rates- For this exercise linear interpolation of the term structure is used. This violates the no-arbitrage assumption described in the beginning of the article as the term structure used in the tree is no longer exactly consistent with the market yield curve.

[25] Hull, Options, pg. 593.
[26] Hull, Options, pg. 589.

- Limitations of the model- The single-factor Hull-White model is limited in its flexibility in managing volatility. "The model can be made to provide a fit to volatilities observed in the market, but the user has no control over the volatilities at subsequent times[27]." A solution is to generate a model whose volatility is dependent on time. Hull and White (1996) create such a model:

  $$d(f(r)) = (?(t)+u-af(r))dt + s_1 dz_1$$

  where u has an initial value of zero and follows the process

  $$du=-budt+s_2 d_2$$

  The C++ tree-constructor is flexible enough to be easily modified to accommodate this feature and create three-dimensional trees. However, it was felt that a model with time-dependent volatility was not suitable for this article because its pricing would require intense calibration. Furthermore, making volatility time-dependent may lead many to believe that volatility as a function of time, implying that there can be assumptions on what the factor is in the future. This is an incorrect assumption.

It can only be concluded that the calculated price serves as an approximation of the true price of the bond option.

The C++ tree-builder itself can be improved. Firstly, the code should be able to handle and automatically interpolate inputted two, three, five, 10, and 30-year treasury rates. This function is currently done by hand. Later developments include having a built in bond-pricer and a function that automatically calculates user-inputted payoff functions. These functions are currently either hand-calculated or run on a fairly inflexible spreadsheet. These developments could be done by converting the source file into a Microsoft Excel-linkable dynamic link library or add-on. These functions would make this program comparable to the highly marketed Pricingtools.com-created Hull-White tree constructor. Long-run developments include a calibrating function that connects to a listing of prices of similar derivatives on the Internet.

[27] Hull, Options, pg. 601.

Summary

In this article the single factor Hull-White term structure model is explained. No new theoretical aspects are added to the model; rather, an advanced C++-encoded algorithm implementing its tree-building procedure is described. The program's vital functions are shown to roughly match the two-stage construction process first developed by Hull. When executed, the program is shown to be a flexible tree-generator capable of modeling the short-term interest rate and pricing interest rate derivatives. An example bond option is then priced twice using the current term structure, once using eight time-steps and once at twenty time-steps. Finally, time is devoted to discussing the possible reasons for the discrepancies between the market price and the model-generated price of the option.

References

"CurveTrader Online Help," http://www.powerfinance.com/help. Leap of Faith Research, Inc., 1998.

"Heath-Jarrow-Morton Model," http://www.mathworks.com/access/helpdesk/help/toolbox/finderiv/using8.shtml, The MathWorks Inc., 2001.

Hull, J., Options, Futures and Other Derivatives, Prentice Hall, 2000.

Hull, J. and A. White, "The General Hull-White Model and Super Calibration," Financial Analysts Journal, Vol. 57, No. 6 (Nov/Dec 2001), pp. 34-43.

Hull, J. and A. White, "Using Hull-White Interest Rate Trees," Journal of Derivatives, Vol. 3, No. 3, (Spring 1996), pp. 26-36.

Leippold, M. and Z. Wiener, "The Term Structure of Interest Rates II: The Hull-White Trinomial Tree of Interest Rates, 1999, 1-17.

Appendix A- Hullwhite.Cpp

```
// -----------------------------------------------
// Written by John Li
// 3/18/2002
// -----------------------------------------------

#include <iostream>
#include <string>
#include "tvector.h"
#include "math.h"
#include "fstream.h"
#include "hulltree.h"

float meanReversion;
float deltaT;
float deltaR;
float jMax;
float jMin;
tvector<float> termStructure;

void getInputs()
{
   float sigma;
   float tempFloat;
   string tempfile;

   ifstream inputFile;
   cout << "Enter File: ";
   cin >>  tempfile;                 // what's the file
   inputFile.open(tempfile.c_str());  // open it

   inputFile >> meanReversion;
   inputFile >> sigma;
   inputFile >> deltaT;

   // defining term structure
   while (inputFile >> tempFloat)
   {
     termStructure.push_back(tempFloat);
   }
   // figuring out deltaR
   deltaR = sigma * sqrt(3 * deltaT);
   // figuring out jMin and jMax
   jMax = ceil( 0.184 / (meanReversion * deltaT) );
   jMin = 0 - jMax;
}

int main()
{
    HullTree myHullTree;
    getInputs();
    myHullTree.buildTree(termStructure, meanReversion, deltaT, deltaR,
jMin, jMax);
}
```

Appendix A- Hulltree.H

```
#ifndef _HULLTREE_H
#define _HULLTREE_H
// The hull tree
// John Li

#include "tvector.h"
#include "math.h"

struct myNode
{
      int   nodeNumber;
        int   depth;             // equals depth of the tree
        int   relativePosition;  // equals j (-2,-1,-0, 1, 2) for the node
        float rate;              // equals R for the node
        float presentValue;      // equals Q for the node
        float alpha;             // equals value of center node= term struct

      float pu;                // probability of going up
      float pm;                // probability of going in the middle
      float pd;            // probability of going down

        myNode * up;
        myNode * middle;
        myNode * down;

        myNode(int & i, int & a, int & b, float & c, float & d, float & e,
float & z, float & y, float & x, myNode * f = NULL, myNode * g = NULL, myNode
* h = NULL)
        : nodeNumber(i),
        depth(a),
          relativePosition(b),
          rate(c),
          presentValue(d),
          alpha(e),
        pu(z),
        pm(y),
        pd(x),
          up(f),
          middle(g),
          down(h)
        { }
};

class HullTree
{
      public:

      HullTree();       // constructor
      ~HullTree();      // destructor

      void              udm(myNode * node);
```

```
        void              addBondPrices(tvector<float> structure);
      tvector<myNode *> findConnectors(myNode * node);
      float             addPresentValue(myNode * node, tvector<myNode *>
depthVector, tvector<float> structure);
      void              addRemainingRates(myNode * tempNode, tvector<myNode *>
depthVector);
      void              addRates(tvector<float> structure);
        int               expand(int lastNodeNumber, int nodesInDepth, int
tempDepth);
        int               maintain(int lastNodeNumber, int nodesInDepth, int
tempDepth);
      void              connectNodes(tvector<float> structure);
      void              outputTree();
      void              buildTree(tvector<float> structure, float a, float dT,
float dR, float min, float max);


      private:

      tvector<myNode *> myTree;

        // same size as structure, saves alpha values per term
      tvector<float> alphaStructure;

      // same size as structure, saves bond prices per term
        tvector<float> bondPrices;

      // same size as structure, saves width per term
      tvector<int> width;

      myNode * rootNode;
      float    meanReversion;
      float    deltaT;
      float    deltaR;
      float    jMin;
      float    jMax;

};

#endif
```

Appendix A- Hulltree.Cpp

```
// ---------------------------------------------
// Written by John Li
// 3/19/2002
// ---------------------------------------------

#include "hulltree.h"
#include "tvector.h"
#include "math.h"
#include <string>
#include <fstream>

// default constructor
HullTree::HullTree()
{

}

// destructor
HullTree::~HullTree()
{

}

//this figures out the Pu, Pm, and Pd for each node
void HullTree::udm(myNode * node)
{
  if (node->relativePosition * deltaR * 100 > jMax)
  {
    node->pu = (7.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)-(3 * meanReversion * node->relativePosition*deltaT))/2);
    node->pm = (0.00000-(1.00000/3.00000)) - (meanReversion *
meanReversion * node->relativePosition * node->relativePosition * deltaT *
deltaT)+(2 * meanReversion * node->relativePosition*deltaT);
    node->pd = (1.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)-(meanReversion * node->relativePosition*deltaT))/2);
  }

  else if (node->relativePosition * deltaR * 100 < jMin)
  {
    node->pu = (1.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)+(meanReversion * node->relativePosition*deltaT))/2);
    node->pm = (0.00000-(1.00000/3.00000)) - (meanReversion *
meanReversion * node->relativePosition * node->relativePosition * deltaT *
deltaT) - (2 * meanReversion * node->relativePosition*deltaT);
    node->pd = (7.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)+(3 * meanReversion * node->relativePosition*deltaT))/2);
```

```
  }
  else
  {
    node->pu = (1.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)-(meanReversion * node->relativePosition*deltaT))/2);
    node->pm = (2.00000/3.00000) - (meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT);
    node->pd = (1.00000/6.00000) + (((meanReversion * meanReversion *
node->relativePosition * node->relativePosition * deltaT *
deltaT)+(meanReversion * node->relativePosition*deltaT))/2);
  }

}


// calculates bond prices based on the term structure
void HullTree::addBondPrices(tvector<float> structure)
{
  for (int a = 0; a < structure.size(); a++)
  {
    float temp = exp(0.00000 - (structure[a] * (a+1)*deltaT));

    bondPrices.push_back(temp);
  }

}

// finds and returns connecting nodes for the calculation of Q
tvector<myNode*> HullTree::findConnectors(myNode * node)
{
  tvector<myNode *> tempVector;
  for (int count = 0; count < myTree.size(); count++)
  {
    if (myTree[count]->up == node || myTree[count]->middle == node
||myTree[count]->down == node)
    {
      myNode * tempNode = myTree[count];
      tempVector.push_back(tempNode);
    }
  }
  return tempVector;
}


// adds Q
float HullTree::addPresentValue(myNode * node, tvector<myNode *> depthVector,
tvector<float> structure)
{

  float alpha= 0.00000;

  for (int a = 0; a < depthVector.size(); a++)
  {
    depthVector[a]->presentValue = 0;

    // find connecting nodes to each node of the same depth
    tvector<myNode *> tempVector = findConnectors(depthVector[a]);

    // going through those connecting nodes, finding Q for depthVector[a]
    for (int b = 0; b < tempVector.size(); b++)
    {
      if (tempVector[b]->up == depthVector[a])
      {
        depthVector[a]->presentValue = tempVector[b]->pu * exp(0.00000 -
((tempVector[b]->rate)*deltaT)) * tempVector[b]->presentValue +
depthVector[a]->presentValue;
      }
      else if (tempVector[b]->middle == depthVector[a])
      {
        depthVector[a]->presentValue = tempVector[b]->pm * exp(0.00000 -
((tempVector[b]->rate)*deltaT)) * tempVector[b]->presentValue +
depthVector[a]->presentValue;
      }
      else if (tempVector[b]->down == depthVector[a])
      {
        depthVector[a]->presentValue = tempVector[b]->pd * exp(0.00000 -
((tempVector[b]->rate)*deltaT)) * tempVector[b]->presentValue +
depthVector[a]->presentValue;
      }
    }
 }

  for (int c = 0; c < depthVector.size(); c++)
  {
    alpha = alpha + (depthVector[c]->presentValue * exp(0.00000 - (deltaR
* deltaT * depthVector[c]->relativePosition)));
  }

  alpha = log(alpha);

  alpha = alpha - log(bondPrices[depthVector[0]->depth]);

  alpha = alpha / deltaT;

  return alpha;

}

// adds the remaining rates
void HullTree::addRemainingRates(myNode * tempNode, tvector<myNode *>
depthVector)
{
  for (int a = 0; a < depthVector.size(); a++)
  {

    depthVector[a]->rate = depthVector[a]->relativePosition * deltaR +
tempNode->rate;

    }

}


// adds the term structure onto the center nodes
void HullTree::addRates(tvector<float> structure)
{
  myTree[0]->rate = structure[0];
  myTree[0]->presentValue = 1;

  myNode * tempNode = myTree[0];

  tvector<myNode *> depthVector;
  int tempDepth = 1;


  for (int a = 1; a < myTree.size(); a++)
  {
    // we're put all nodes of the same depth on a vector
    if (myTree[a]->depth == tempDepth)
    {
      depthVector.push_back(myTree[a]);
    }
    else
    {
      // getting the center node
      tempNode = tempNode->middle;

      // calling present value with the center node and vector of nodes with
the same depth
      tempNode->rate = addPresentValue(tempNode, depthVector, structure);

      // add remaining rates to the nodes in the same depth as tempNode
      addRemainingRates(tempNode, depthVector);

      tempDepth++;

      // clear and add the first node of the next depth
      depthVector.clear();
      a--;
    }
  }

  // getting the center node
  tempNode = tempNode->middle;

  // calling present value with the center node and vector of nodes with the
same depth
  tempNode->rate = addPresentValue(tempNode, depthVector, structure);

  addRemainingRates(tempNode, depthVector);
```

```
}

int HullTree::expand(int lastNodeNumber, int nodesInDepth, int tempDepth)
{
  int beginningNode = lastNodeNumber-nodesInDepth+1;
  int t = 0;

  // temp variables, then adding on new blank nodes
  int aa = 0;
  float bb = 0;

  for (int c = 0; c < nodesInDepth + 2; c++)
  {
    myNode * tempNode = new
myNode(aa,tempDepth,aa,bb,bb,bb,bb,bb,bb,NULL,NULL,NULL);
    myTree.push_back(tempNode);
  }

  while (t < nodesInDepth)
  {
    myTree[beginningNode+t]->up       = myTree[beginningNode + t +
nodesInDepth];
    myTree[beginningNode+t]->middle   = myTree[beginningNode + t +
nodesInDepth + 1];
    myTree[beginningNode+t]->down     = myTree[beginningNode + t +
nodesInDepth + 2];
    t++;
  }

  // adding relativePosition
  int divider = (nodesInDepth + 1) / 2;
  for (int a = 0; a < nodesInDepth+2; a++)
  {
    myTree[beginningNode + nodesInDepth + a]->relativePosition = divider - a;
  }

  return nodesInDepth + 2;
}

int HullTree::maintain(int lastNodeNumber, int nodesInDepth, int tempDepth)
{
  int beginningNode = lastNodeNumber-nodesInDepth+1;

  // temp variables, then adding on new blank nodes
  int aa = 0;
  float bb = 0;

  for (int c = 0; c < nodesInDepth; c++)
  {
    myNode * tempNode = new
myNode(aa,tempDepth,aa,bb,bb,bb,bb,bb,bb,NULL,NULL,NULL);
    myTree.push_back(tempNode);
  }
  // prevent the top node from expanding
  myTree[beginningNode]->up       = myTree[beginningNode + nodesInDepth];
  myTree[beginningNode]->middle   = myTree[beginningNode + nodesInDepth + 1];
  myTree[beginningNode]->down     = myTree[beginningNode + nodesInDepth + 2];

  // expand the middle nodes accordingly
  for (int i = 1; i <= nodesInDepth - 2; i++)
  {
    myTree[beginningNode + i]->up       = myTree[beginningNode + i +
nodesInDepth - 1];
    myTree[beginningNode + i]->middle   = myTree[beginningNode + i +
nodesInDepth];
    myTree[beginningNode + i]->down     = myTree[beginningNode + i +
nodesInDepth + 1];
  }

  myTree[lastNodeNumber]->up       = myTree[lastNodeNumber + nodesInDepth -
2];
  myTree[lastNodeNumber]->middle   = myTree[lastNodeNumber + nodesInDepth -
1];
  myTree[lastNodeNumber]->down     = myTree[lastNodeNumber + nodesInDepth];

  // adding relativePosition
  int divider = (nodesInDepth - 1) / 2;
  for (int a = 0; a < nodesInDepth; a++)
  {
    myTree[beginningNode + nodesInDepth + a]->relativePosition = divider - a;
  }

  return nodesInDepth;
}


// goes down the vector of myNodes and connects the nodes to each other
void HullTree::connectNodes(tvector<float> structure)
{
  // temporary variables
  myNode * tempNode;

  // temp variables, originally making a max of 9 nodes
  int aa = 0;
  float bb = 0;

  for (int c = 0; c < 9; c++)
  {
    myNode * tempNode = new myNode(aa,aa,aa,bb,bb,bb,bb,bb,bb,NULL,NULL,NULL);
    myTree.push_back(tempNode);
  }

  // initializing the root node
  myTree[0]->depth  = 0;
  myTree[0]->relativePosition = 0;
  myTree[0]->presentValue = 1.00000;
  width.push_back(1);

  if (structure.size() > 1)
  {
    myTree[0]->up     = myTree[1];
    myTree[0]->middle = myTree[2];
    myTree[0]->down   = myTree[3];
    myTree[1]->depth  = 1;
    myTree[2]->depth  = 1;
    myTree[3]->depth  = 1;
    myTree[1]->relativePosition = 1;
    myTree[2]->relativePosition = 0;
    myTree[3]->relativePosition = -1;
    width.push_back(3);
  }

  if (structure.size() > 2)
  {
    myTree[1]->up     = myTree[4];
    myTree[1]->middle = myTree[5];
    myTree[1]->down   = myTree[6];
    myTree[2]->up     = myTree[5];
    myTree[2]->middle = myTree[6];
    myTree[2]->down   = myTree[7];
    myTree[3]->up     = myTree[6];
    myTree[3]->middle = myTree[7];
    myTree[3]->down   = myTree[8];
    myTree[4]->depth  = 2;
    myTree[5]->depth  = 2;
    myTree[6]->depth  = 2;
    myTree[7]->depth  = 2;
    myTree[8]->depth  = 2;
    myTree[4]->relativePosition = 2;
    myTree[5]->relativePosition = 1;
    myTree[6]->relativePosition = 0;
    myTree[7]->relativePosition = -1;
    myTree[8]->relativePosition = -2;
    width.push_back(5);
  }

  if (structure.size() > 3)
  {
    for (int count = 3; count < structure.size(); count++)
    {
      tempNode=myTree[myTree.size()-1];

      // see if tempNode->relativePosition * deltaR is greater than jMax or
less than jMin
      if (100 * tempNode->relativePosition * deltaR > jMax || 100 * tempNode-
>relativePosition * deltaR < jMin)
      {
        width.push_back(maintain(myTree.size()-1, width[count-1], count));
      }
      // if not, then make it even bigger
      else width.push_back(expand(myTree.size()-1, width[count-1], count));
    }
  }
}


// used for debugging purposes
void HullTree::outputTree()
{
  string filename = "OUTPUT";
  ofstream output(filename.c_str());

  for (int count = 0; count < myTree.size(); count++)
  {
    myTree[count]->nodeNumber = count;
  }

  for (int count = 0; count < myTree.size(); count++)
  {
    output << count << " ";
    output << myTree[count]->depth << " ";
    output << myTree[count]->relativePosition << " ";
    output <<  myTree[count]->rate << " ";
    output <<  myTree[count]->pu << " ";
    output <<  myTree[count]->pm << " ";
    output <<  myTree[count]->pd << endl;
  }
}


// builds the tree
void HullTree::buildTree(tvector<float> structure, float a, float dT,
float dR, float min, float max)
{
  // variables which equal referenced values
  meanReversion = a;
  deltaT = dT;
  deltaR = dR;
  jMin = min;
  jMax = max;

  connectNodes(structure);  // connect the nodes

  for (int count = 0; count < myTree.size(); count++)
  {
    udm(myTree[count]);
  }

  addBondPrices(structure);
  addRates(structure);
  outputTree();
}
```

Appendix B

Node #	i	j	R	?up	?middle	?down
0	0	0	0.016175	0.166667	0.666667	0.166667
1	1	1	0.02465	0.160495	0.66651	0.172995
2	1	0	0.018527	0.166667	0.666667	0.166667
3	1	-1	0.012403	0.172995	0.66651	0.160495
4	2	2	0.033125	0.154479	0.666042	0.179479
5	2	1	0.027001	0.160495	0.66651	0.172995
6	2	0	0.020878	0.166667	0.666667	0.166667
7	2	-1	0.014754	0.172995	0.66651	0.160495
8	2	-2	0.00863	0.179479	0.666042	0.154479
9	3	3	0.041602	0.14862	0.66526	0.18612
10	3	2	0.035479	0.154479	0.666042	0.179479
11	3	1	0.029355	0.160495	0.66651	0.172995
12	3	0	0.023231	0.166667	0.666667	0.166667
13	3	-1	0.017108	0.172995	0.66651	0.160495
14	3	-2	0.010984	0.179479	0.666042	0.154479
15	3	-3	0.00486	0.18612	0.66526	0.14862
16	4	4	0.050333	0.142917	0.664167	0.192917
17	4	3	0.044209	0.14862	0.66526	0.18612
18	4	2	0.038085	0.154479	0.666042	0.179479
19	4	1	0.031961	0.160495	0.66651	0.172995
20	4	0	0.025838	0.166667	0.666667	0.166667
21	4	-1	0.019714	0.172995	0.66651	0.160495
22	4	-2	0.01359	0.179479	0.666042	0.154479
23	4	-3	0.007467	0.18612	0.66526	0.14862
24	4	-4	0.001343	0.192917	0.664167	0.142917
25	5	5	0.058912	0.13737	0.66276	0.19987
26	5	4	0.052789	0.142917	0.664167	0.192917
27	5	3	0.046665	0.14862	0.66526	0.18612
28	5	2	0.040541	0.154479	0.666042	0.179479
29	5	1	0.034418	0.160495	0.66651	0.172995
30	5	0	0.028294	0.166667	0.666667	0.166667
31	5	-1	0.02217	0.172995	0.66651	0.160495
32	5	-2	0.016046	0.179479	0.666042	0.154479
33	5	-3	0.009923	0.18612	0.66526	0.14862
34	5	-4	0.003799	0.192917	0.664167	0.142917
35	5	-5	-0.00232	0.19987	0.66276	0.13737
36	6	6	0.067494	0.131979	0.661042	0.206979
37	6	5	0.06137	0.13737	0.66276	0.19987
38	6	4	0.055246	0.142917	0.664167	0.192917
39	6	3	0.049122	0.14862	0.66526	0.18612
40	6	2	0.042999	0.154479	0.666042	0.179479
41	6	1	0.036875	0.160495	0.66651	0.172995
42	6	0	0.030751	0.166667	0.666667	0.166667
43	6	-1	0.024628	0.172995	0.66651	0.160495
44	6	-2	0.018504	0.179479	0.666042	0.154479
45	6	-3	0.01238	0.18612	0.66526	0.14862
46	6	-4	0.006256	0.192917	0.664167	0.142917

47	6	-5	0.000133	0.19987	0.66276	0.13737
48	6	-6	-0.00599	0.206979	0.661042	0.131979
49	7	7	0.076076	0.126745	0.65901	0.214245
50	7	6	0.069952	0.131979	0.661042	0.206979
51	7	5	0.063829	0.13737	0.66276	0.19987
52	7	4	0.057705	0.142917	0.664167	0.192917
53	7	3	0.051581	0.14862	0.66526	0.18612
54	7	2	0.045457	0.154479	0.666042	0.179479
55	7	1	0.039334	0.160495	0.66651	0.172995
56	7	0	0.03321	0.166667	0.666667	0.166667
57	7	-1	0.027086	0.172995	0.66651	0.160495
58	7	-2	0.020962	0.179479	0.666042	0.154479
59	7	-3	0.014839	0.18612	0.66526	0.14862
60	7	-4	0.008715	0.192917	0.664167	0.142917
61	7	-5	0.002591	0.19987	0.66276	0.13737
62	7	-6	-0.00353	0.206979	0.661042	0.131979
63	7	-7	-0.00966	0.214245	0.65901	0.126745

Appendix C

B10	C	D	E	F	G
11	*User Inputs*			Leaf Node Calculations	
12	*t*	*1*		*Leaf Node Rates*	P(t,T)
13	*T*	*10*		*0.0761*	\$D\$27*EXP(0-\$D\$25*F13)
14	*deltaT*	*0.125*		*0.0700*	\$D\$27*EXP(0-\$D\$25*F14)
15	*a*	*0.1*		*0.0638*	\$D\$27*EXP(0-\$D\$25*F15)
16	*sigma*	*0.01*		*0.0577*	\$D\$27*EXP(0-\$D\$25*F16)
17				*0.0516*	\$D\$27*EXP(0-\$D\$25*F17)
18	Term Structure Calculations			*0.0455*	\$D\$27*EXP(0-\$D\$25*F18)
19	P(0,t)	Term Structure-DONE'!E5		*0.0393*	\$D\$27*EXP(0-\$D\$25*F19)
20	P(0,T)	Term Structure-DONE'!E13		*0.0332*	\$D\$27*EXP(0-\$D\$25*F20)
21	P(0,t+deltaT)	Term Structure-DONE'!E6		*0.0271*	\$D\$27*EXP(0-\$D\$25*F21)
22	deltaR	D16*SQRT(3*D14)		*0.0210*	\$D\$27*EXP(0-\$D\$25*F22)
23	Calculations			*0.0148*	\$D\$27*EXP(0-\$D\$25*F23)
24	B(t,T)	(1-EXP(0-(D15*(D13-D12))))/D15		*0.0087*	\$D\$27*EXP(0-\$D\$25*F24)
25	Bhat(t,T)	D24*D14/D26		*0.0026*	\$D\$27*EXP(0-\$D\$25*F25)
26	B(t,t+deltat)	((1-EXP(0-(D15*D14)))/D15)		*-0.0035*	\$D\$27*EXP(0-\$D\$25*F26)
27	Ahat(t,T)	EXP(LN(D20/D19)-((D24/D26)*LN(D21/D19))-(((D16*D16)/(4*D15))*(1-EXP(0-(2*D15*D12)))*D24*(D24-D26)))		*-0.0097*	\$D\$27*EXP(0-\$D\$25*F27)

Appendix D

User Inputs	
t	1
T	10
deltaT	0.05
a	0.1
sigma	0.01
Term Structure Calculations	
P(0,t)	0.9757
P(0,T)	0.5927
P(0,t+deltaT)	0.9740
deltaR	0.0039
Calculations	
B(t,T)	5.9343
Bhat(t,T)	5.9492
B(t,t+deltat)	0.0499
Ahat(t,T)	0.7474

Leaf Node Calculations					
Leaf Node Rates	P(t,T)	Leaf Node Rates	P(t,T)	Leaf Node Rates	P(t,T)
0.10754	0.39418	0.04945	0.55692	0.00000	0.74739
0.10367	0.40337	0.04557	0.56990	0.00000	0.74739
0.09980	0.41277	0.04170	0.58318	0.00000	0.74739
0.09592	0.42239	0.03783	0.59678	0.00000	0.74739
0.09205	0.43224	0.03396	0.61069	0.00000	0.74739
0.08818	0.44231	0.03008	0.62492	0.00000	0.74739
0.08430	0.45262	0.02621	0.63949	0.00000	0.74739
0.08043	0.46317	0.02234	0.65439	0.00000	0.74739
0.07656	0.47397	0.01846	0.66965	0.00000	0.74739
0.07268	0.48501	0.01459	0.68525		
0.06881	0.49632	0.01072	0.70123		
0.06494	0.50789	0.00684	0.71757		
0.06107	0.51972	0.00297	0.73430		
0.05719	0.53184	0.00000	0.74739		
0.05332	0.54423	0.00000	0.74739		

Leaf Node Calculations			
Strike Price (x1000)	Leaf Node Rates	P(t,T)	Put Option Price
0.592739659	0.107542	0.394178702	198.5609571
	0.103669	0.40336644	189.3732193
	0.0997957	0.412769067	179.9705919
	0.0959228	0.422389868	170.3497908
	0.0920498	0.432235168	160.5044913
	0.0881768	0.442309947	150.4297121
	0.0843038	0.452619555	140.1201043
	0.0804308	0.463169465	129.5701944
	0.0765578	0.473965278	118.7743813
	0.0726849	0.485012437	107.7272219
	0.0688119	0.496317378	96.42228055
	0.0649389	0.507885822	84.85383732
	0.0610659	0.519723909	73.01575033
	0.0571929	0.531837924	60.90173459
	0.0533199	0.5442343	48.50535859
	0.049447	0.556919287	35.82037224
	0.045574	0.569900273	22.83938581
	0.041701	0.583183828	9.555831284
	0.037828	0.596777003	0
	0.033955	0.610687015	0
	0.030082	0.62492125	0
	0.0262091	0.639486884	0
	0.0223361	0.654392402	0
	0.0184631	0.669645346	0

	0.0145901	0.685253815	0
	0.0107171	0.701226094	0
	0.00684414	0.717570494	0
	0.00297115	0.734295985	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0
	0	0.747390658	0