martingalePricing: A C++ Template Library to Price Financial Derivatives on Trees

Sebastian Ferrando and Brendan Bartlett

October 20, 2002

Abstract

A financial derivative system with several different data types, underlying types and derivatives. The system is implemented using generic programming techniques in C++.

Contents

1 Summary

martingalePricing is a financial derivative system which has two fundamental data types to represent the underlying and derivative. The system is implemented using generic programming and templates inspired by A. Alexandrescu[1].

1.1 Data Types

The two fundamental data types are:

1. Tree
   A fully balanced tree which can be either *binomial* or *trinomial*. Larger degree of trees are possible but they have not been tested.

2. Jagged matrix
   A jagged matrix is simply a matrix where the first column holds the root value and the next column height contains the children of the root value. Notice that in a jagged matrix, there is no concept of a *path* through the matrix, since each 'child' in the matrix can have multiple parents.

For both of these data types there are two different algorithms to price the underlying and derivative. The two algorithms are:

1. Complete
   The entire data structure is contained in memory. This can consume large amounts of memory, especially with a trinomial tree data structure.

2. Memory Saving
   Only a portion of the data structure is held in memory.

Notice that the data structure (either tree or jagged matrix) and algorithm (complete or memory saving) used for the underlying dictates the data structure and algorithm for the derivative[2]. For example, it is not possible to have the underlying as a complete tree with the derivative held in anything other then a complete tree.

1.2 Underlying Models

The different underlying types are different mathematical models to represent a particular type of stock. Three underlying types have been implemented:

1. Binom

2. Binom 2

3. Time Halving Trinom

[1] See chapters 1 and 2 of [1]
[2] Not true for a memory saving jagged matrix, discussed later.

1.3 Derivative Models

A derivative type is responsible for providing functions to calculate a particular type of derivative. Derivatives can be divided into two different categories:

1. Non Path Dependent
   These derivatives *do not* require any kind of path information to calculate the derivative value. Three different types have been implemented:
   (a) European Call
   (b) European Put
   (c) American Put
2. Path Dependent
   These *do* require path information to calculate the derivative value. Two of these have been implemented:
   (a) Average Strike European Call
   (b) Average Strike American Put

Notice that since there is not a concept of a 'path' in a jagged matrix, a jagged matrix *cannot* be used to calculate a path dependent derivative.

2 How to compile

martingalePricing has been implemented by following generic programming ideas. This means the actual implementation of the library uses C++ templates extensively. Due to the use of templates, it is necessary for martingalePricing to consist of header files which must be `#included` in any code which wants to use martingalePricing.

There is an optional feature to the library which allows *complete* data structures of either the underlying of derivative to be saved to disk as a `png` image file. In order to use this feature an extra library is required called the `gd` library which can be found at `http://www.boutell.com/gd/`. The version used is 1.8.4.

2.1 Generate

In the Test subdirectory, there is a program called `generate`. The purpose of this program is to ask the user several questions and generate a C++ program plus a Makefile which will call martingalePricing for what the user entered. For example, let's assume we want the following:

1. **Data Structure** - Tree
   (a) **Type** - Complete
2. **Underlying** - Binom
3. **Derivative** - European Call

With the input parameters of

Paramater	*Value*	*Meaning*
K	100.0	Derivative strike price
S0	100.0	Initial stock price
N	3	Number of time steps
dt[0]	1.0/3.0	Time difference between each step
U	1.1	Stock up jump value
D	0.9091	Stock down jump value
r	0.06	interest rate (yearly)

Please notice here that it is very important to include the decimal values for any paramaters which are double precision numbers (the only integer value is N). Failure to do so will yield incorrect results, as the compiler will incorrectly assume values are integers when they should be double precision numbers.

`generate` will produce two files: the actual cpp file and a Makefile for that source file. In this example, these files are called `sample.cpp` and `Makefile.sample`. The standard include files used by `sample.cpp` are shown below:

Listing 1: sample.cpp [Line 2 to 5]

```
#include <stdio.h>
#include <vector.h>
#include <stack.h>
#include <math.h>
```

Notice that the standard template library include files are used here. The martingalePricing included files are the following:

Listing 2: sample.cpp [Line 9 to 16]

```
#include "treenode.h"
#include "visitor.h"
#include "iterators.h"
#include "dstruct.h"
#include "multiplicative.h"
#include "underlying/binom.h"
#include "derivative/europeanCall.h"
#include "pricing.h"
```

If different combinations of data structure, underlying types, etc. were used then different include files would be here.

Below is the code which calls martingalePricing to initialize the underlying and derivative:

Listing 3: sample.cpp [Line 22 to 37]

```
int main(void)
{
        IP_Binom param;

        param.dt = vector<double>(1);

        param.S0 = 100.0;
        param.N = 3;
        param.dt[0] = 1.0/3.0;
        param.U = 1.1;
        param.D = 0.9091;
        param.r = 0.06;

        Pricing<UNDERLYING(Binom), Tree<Complete, Multiplicative>, EuropeanCall > *derivative =
          new Pricing<UNDERLYING(Binom), Tree<Complete, Multiplicative>, EuropeanCall >(param, 100.0);
```

After the code above runs, the underlying and derivative have been fully calculated. The syntax for the above declaration will be explained in the next section.

The code below calls martingalePricing to get the derivative root value:

Listing 4: sample.cpp [Line 41 to 56]

```
        /* The following methods are available in derivative:
         *
         * double getDerivativeRootValue()
         *
         * PostOrderIterator<double>& getStockPostOrderIterator()
         * BreadthFirstIterator<double>& getStockBreadthFirstIterator()
         *
         * PostOrderIterator<double>& getDerivativePostOrderIterator()
         * BreadthFirstIterator<double>& getDerivativeBreadthFirstIterator()
         *
         * void graphStock(char *fname, unsigned int width, unsigned int height)
         * void graphDerivative(char *fname, unsigned int width, unsigned int height)
         */

        printf("root derivative value is %f\n", derivative->getDerivativeRootValue());
}
```

Below is the relevant portion of `Makefile.sample` which shows the two variables which must be set by the user. The `GRAPHICS` option selects if the graphics library will be included with martingalePricing. The `INCLUDE` tells the compiler where to find the header files for martingalePricing.

```
# set GRAPHICS to either YES or NO
# set INCLUDE to either the relative or absolute location
# of the header files

GRAPHICS= YES
INCLUDE= ../Source

#don't touch anything below here

...
```

3 Structure of the library

As mentioned earlier, martingalePricing has been implemented by using generic programming ideas. This means that rather then following a traditional object oriented method, the code uses templates and *policies* extensively. Policies are `struct`ures of C code which implement a predefined set of functions. A number of different policies can be written which all implement the same set of functions. Code which uses these policies (called *host classes*) inherit from a template argument, where the template argument is the desired policy. This allows many different combinations of the code which calls the policies to the actual policies used.

Partial template specialization is used by code which uses policies to handle the different types of policies it supports.

martingalePricing uses policies extensively. The host class for the policies in martingalePricing is called `pricing`. It takes three policies, which are:

1. Underlying Type.

   As mentioned in the previous section, there are three choices here:

   - Binom
   - Binom2
   - TimeHalvingTrinom

2. Data Structure Type.

   There are two choices here:

   - Tree
   - Matrix

   Notice that these policies also use policies. This will be explained later.

3. Derivative Type.

   There are five different choices here:

   - European Call
   - European Put
   - American Put
   - Average Strike European Call
   - Average Strike American Put

The different policies for `pricing` will be explained in the following sections.

3.1 Underlying Type

The underlying policy is responsible to accept input parameters and calculate several matrices which are used by `pricing` to calculate the stock price. The `Binom` underlying type will be explained here. The other types are similar [3].

Each underlying type accepts only one set of input parameters. The input parameters are implemented as a simple structure of values whose structure name *must* be the name of the underlying type prepended with `IP_`. For example, for the underlying type `Binom`, the input parameters are called `IP_Binom`. The data structure is defined to be:

Listing 5: ../Source/underlying/binom.h [Line 4 to 14]

```
struct IP_Binom
{
  double S0;
  unsigned int N;
  vector<double> dt;
  double U;
  double D;
  double r;
};

typedef struct IP_Binom IP_Binom;
```

Paramater	*Meaning*
S0	Initial Stock Value
N	Number of time steps
dt	vector of time gaps
U	Up step multiplier
D	Down step multiplier
r	Annual continuously compounded interest rate

The size of `dt` can be either one or N. If it is one, all of the time steps are equal, otherwise the vector specifies the gap between each time step. For `binom` the size must be one.

Note that these input parameters are specific for `binom` and they will be different for the other underlying types.

The signature, or format of the policy is shown below:

Listing 6: ../Source/underlying/binom.h [Line 18 to 34]

```
/* Binom ********************/
class Binom
{
 protected:
```

[3] See [7] for explanation of the formulas used

Listing 6: ../Source/underlying/binom.h [Line 18 to 34] (continued)

```
static const unsigned int Q = 2;
static const int I = 1;
static const int J = 1;
static const int K = 0;
static const bool recomb = true;

vector< vector<double> > M;
vector< vector<double> > P;
vector< double > disc;

double S0;
unsigned int N;
vector<double> dt;
```

Paramater	*Meaning*	*Value*
Q	Number of children per node	2
I	Size of 1st dimension of M	1
J	Size of 1st dimension of J	1
K	Size of 1st dimension of disc	0
recomb	True if can be stored in matrix	true
M	Matrix of multipliers to price stock	M[0][0] = D, M[0][1] = U
P	Up and down probabilities	calculated from input parameters
disc	Annual continuously compounded interest rate	calculated from input parameters

Where S0, N and dt are simply copied from the passed input parameters.

Unlike the input parameters, the table above describes the format of the policy. Every new policy *must* implement the above variables.

3.1.1 Implementing another underlying

Let's say we want to implement another underlying called `newUnderlying`. Everything in the left margin in text typeset is the actual code for the new underlying. I will present some code and explain it. If you take this code, you will have a (uninteresting) new underlying. You will place the code in the library in the `Source/underlying` subdirectory to make it available for finance++.

The first step is developing the input parameters this underlying needs from the user to operate. You would create a regular C structure and name it `IP_newUnderlying`. You *must* name the input parameters `IP_newUnderlying` for reasons which will be discussed later.

Let's pretend that the input parameters are similar to `Binom`:

```
struct IP_newUnderlying
{
double S0;
unsigned int N;
```

```
vector<double> dt;
double r;
};
```

You also need to `typedef` the structure, otherwise `pricing` will have problems:

```
typedef struct IP_newUnderlying IP_newUnderlying;
```

Now, as discussed in the previous section, you *must* implement several variables as `protected` class variables, otherwise it will not compile.

Let's pretend that those necessary variables (Q, I, J, K, recomb, M, P, disc, S0, N and dt) are similar to `Binom` as well:

```
class newUnderlying
{
protected:
static const unsigned int Q = 2;
static const int I = 1;
static const int J = 1;
static const int K = 0;
bool recomb;

vector< vector<double> > M;
vector< vector<double> > P;
vector< double > disc;

double S0;
unsigned int N;
vector<double> dt;
```

This declaration is required in all underlyings, including our new one `newUnderlying`. Notice that the variables may have different values (Q for a trinomial underlying would be 3).

Now, we will continue on with defining our underlying class. Here we define the constructor which is the only function in the class which is defined for an underlying. In the constructor, you perform any input validation (eg. N must be less than 16, vector dt cannot be larger than 1, etc.) with a simple `if` statement and print a message to `stderr` and then returning to the calling code. Additionaly, you perform any calculations and set the vectors of `M, P` and `disc` accordingly.

Let's pretend that our input validation consists of ensuring that the vector dt in the input paramaters is no larger then 1:

```
public:
newUnderlying(IP_newUnderlying inIP)
```

```
{
// dt can only have just one element

if( inIP.dt.size() > 1 )
{
  fprintf(stderr, "newUnderlying, invalid dt size %d, must be 1.\n", inIP.dt.size());
  return;
}
```

Now, we are finished our input validation. The underlying needs to set the hardcoded `recomb` variable to true or false. Let's pretend that `newUnderlying` is always recombining (ie. it may be stored in a matrix):

```
recomb = true;
```

At this point, we simply copy over the `S0, N` and `dt` values from the input parameters:

```
    S0 = inIP.S0;
    N = inIP.N;

    dt = vector<double>(inIP.dt.size());
    dt[0] = inIP.dt[0];
```

Now, we simply fill in the `M, P` and `disc` vectors according to the model for this underlying (for which this fictional one has hardcoded values, in reality it will have some kind of mathematical formula here):

```
vector<double> vd = vector<double>(2);

// fill in the M vector

vd[0] = 0.9;      // Di
vd[1] = 1.1;   // Ui

M.push_back(vd);

// fill in the P vector

vd = vector<double>(2);

vd[0] = 0;
vd[1] = 1;
```

```
P.push_back(vd);

disc.push_back(pow(M_E, (-inIP.r) * dt[i]));
}
    }
```

And that's it. You should now be able to use this underlying (assuming, of course, it is in a file named `newUnderlying.h` in the `Source/underlying` subdirectory.

3.2 Data Structure Type

There are two data structure types:

- Tree
- Matrix

This policy takes two template arguments which specify policies it uses. They are:

1. Type
   - Complete
     This holds the entire data structure for an underlying in memory.
   - Empty
     This holds the entire data structure for a derivative in memory. Is it identical to a Complete tree with the exception that after initialization, all node values are 0.0 in the tree.
   - MemorySaving
     This is used for a partial tree for both the underlying and derivative.
2. UnderlyingType
   How to actually calculate the underlying. The only choice is `multiplicative`.

Note that the data structure types are implemented in the file `dstruct.h`. The following will explain the code *in that file only.*

The MemorySaving version of the two data types are implemented in `pricing.h` and will be explained in that section.

Also, please note that any calculation code in `dstruct.h` is only responsible to calculate the *underlying.* The calculation code for the derivative is contained in `pricing.h` and will be explained in that section.

3.2.1 Tree

A tree consists of tree nodes. A tree node is defined as:

Listing 7: ../Source/treenode.h [Line 2 to 10]

```
template <class T>
class TreeNode
{
 public:
  T value;
  unsigned int depth;                  // 0 ..  N, the current depth of this node
  unsigned int childNum;               // which child # of the parent this node is
  vector<TreeNode<T> *> children;
};
```

The type T used in martingalePricing is `double`.

The protected variables in the tree are shown below:

Listing 8: ../Source/dstruct.h [Line 138 to 141]

```
protected:
  TreeNode<double> *sRoot_;
  BreadthFirstIterator<double> *sBreadthFirstIter_;
  PostOrderIterator<double> *sPostOrderIter_;
```

There are access methods for the iterators:

Listing 9: ../Source/dstruct.h [Line 171 to 179]

```
BreadthFirstIterator<double> *GetBreadthFirstIterator(void)
{
  return sBreadthFirstIter_;
}

PostOrderIterator<double> *GetPostOrderIterator(void)
{
  return sPostOrderIter_;
}
```

For a complete tree, the constructor code is shown below:

Listing 10: ../Source/dstruct.h [Line 146 to 155]

```
Tree(int inQ, double S0, int N, vector<vector<double> >M, TFunctor *inVisitListener)
{
  Constr(inQ);
  sRoot_->value = S0;

  UpdateCalc(N, 0, sRoot_, M);

  sBreadthFirstIter_ = new BreadthFirstIterator<double>(sRoot_, degree_);
  sPostOrderIter_ = new PostOrderIterator<double>(sRoot_, degree_, inVisitListener);
}
```

➥

Listing 10: ../Source/dstruct.h [Line 146 to 155] (continued)

The first four arguments are from the underlying and the final argument is a pointer to a member function (a functor) which is called every time the Post Order Iterator visits a new node. This is needed to price path dependent derivatives which will be discussed in the `pricing.h` section. Also see appendices B and C for more information.

Constr is defined to be:

Listing 11: ../Source/dstruct.h [Line 40 to 51]

```
void Constr(int inQ)
{
  degree_ = inQ;

  sRoot_ = new TreeNode<double>;
  sRoot_->value = 0.0;
  sRoot_->childNum = 0;
  sRoot_->depth = 0;

  sBreadthFirstIter_ = NULL;
  sPostOrderIter_ = NULL;
}
```

The code which calculates the underlying tree values is shown below:

Listing 12: ../Source/dstruct.h [Line 55 to 72]

```
 void UpdateCalc(int N, int depth, TreeNode<double> *node, vector< vector<double> > M)
 {
   if(depth == N)
     return;

   node->children = vector<TreeNode<double> *>(degree_);

   for(int c = 0; c < degree_; c++)
     {
node->children[c] = new TreeNode<double>();

node->children[c]->value = underlyingChildValue(node->value, c, depth, M);
node->children[c]->childNum = c;
node->children[c]->depth = depth+1;

UpdateCalc(N, depth+1, node->children[c], M);
     }
 }
```

This code is simply recursive code which allocates memory for each new depth of the tree and calls `underlyingChildValue` for each newly created tree node.

`underlyingChildValue` is defined in `multiplicative.h` as:

Listing 13: ../Source/multiplicative.h [Line 3 to 22]

```
struct Multiplicative
{
  /* returns the stock value for a child given:
   *
   * parent is the parent's stock value
   * c is the requested child # stock value
   * T is the current time step
   * M is the vector of factors (of size [N][Q] if time dependent,
   *   otherwise size [1][Q])
   */
  double underlyingChildValue(double parent, int c, int T, vector<vector<double> > M)
  {
    assert((unsigned int)c < M[0].size());

    if(M.size() > 1)                  // is M time dependent?
      return parent * M[T][c];
    else
      return parent * M[0][c];     // no, use 0th vector for multiplicative factor
  }
};
```

The complete tree also contains a function called `dumpNodes` which uses the `gd` library to render the tree as a png image file.

An empty tree is identical to a Complete tree with the exception that `underlyingChildValue` is not called when the tree is created. This means that all of the node values are set to 0.0 after initialization. This type of tree is used by `pricing.h` to hold a complete tree derivative in memory.

It's constructor is defined to be:

Listing 14: ../Source/dstruct.h [Line 159 to 167]

```
Tree(int inQ, int inN)
{
  Constr(inQ);

  UpdateEmpty(inN, 0, sRoot_);

  sBreadthFirstIter_ = new BreadthFirstIterator<double>(sRoot_, degree_);
  sPostOrderIter_ = new PostOrderIterator<double>(sRoot_, degree_, NULL);
}
```

Where UpdateEmpty is:

Listing 15: ../Source/dstruct.h [Line 76 to 95] (continued)

Listing 15: ../Source/dstruct.h [Line 76 to 95]

```
 void UpdateEmpty(int N, int depth, TreeNode<double> *node)
 {
   if(depth == N)
     return;

   node->value = 0.0;

   node->children = vector<TreeNode<double> *>(degree_);

   for(int c = 0; c < degree_; c++)
     {
node->children[c] = new TreeNode<double>();
node->children[c]->value = 0.0;
node->children[c]->childNum = c;
node->children[c]->depth = depth+1;

UpdateEmpty(N, depth+1, node->children[c]);
     }
 }
```

3.2.2 Matrix

A complete matrix is simply a vector of other vectors. The height of the matrix is initially one, to hold the root value. Then, it grows by either one or two depending on if it is binomial or trinomial, respectively.

Here is how the matrix is implemented:

Listing 16: ../Source/dstruct.h [Line 250 to 251]

```
protected:
  vector< vector< MatrixNode <double> *> > jag_;
```

Each node in the matrix is held in a simple data structure:

Listing 17: ../Source/dstruct.h [Line 227 to 232]

```
template <class T>
class MatrixNode
{
public:
  T value;
};
```

Listing 17: ../Source/dstruct.h [Line 227 to 232] (continued)

Each matrix has access methods:

Listing 18: ../Source/dstruct.h [Line 419 to 432]

```
void Set(int col, int row, double inValue)
{
  jag_[col][row]->value = inValue;
}

double Get(int col, int row)
{
  return jag_[col][row]->value;
}

unsigned int Height(int col)
{
  return jag_[col].size();
}
```

As with the tree, there are two different variations of the matrix, a complete one which holds the underlying and an empty one which holds the derivative. The empty matrix is identical to a complete matrix with the exception that at initialization time, all nodes have 0.0 in them.

The complete matrix constructor is defined as:

Listing 19: ../Source/dstruct.h [Line 298 to 334]

```
 Matrix(int inQ, double S0, unsigned int inN, vector<vector<double> >M)
 {
   MatrixNode<double> *node = new MatrixNode<double>;
   vector<MatrixNode<double> *> column = vector<MatrixNode<double> *>(1);

   Constr(inQ);

   // set the root node value.
   jag_[0][0]->value = S0;

   unsigned int columnHeight;
   // now add the remaining nodes
   for(unsigned int columnNum = 1; columnNum < inN+1; columnNum++)
     {
columnHeight = 1 + columnNum * (inQ-1);
column = vector<MatrixNode<double> *>(columnHeight);

for(int rowNum = columnHeight - 1; rowNum >= 0; rowNum--)
  {
    node = new MatrixNode<double>;
```

Listing 19: ../Source/dstruct.h [Line 298 to 334] (continued)

```
    unsigned int sRowNum = rowNum;

    if(sRowNum == 0)
      node->value = underlyingChildValue(jag_[columnNum-1][0]->value, 0, columnNum, M);
    else if(sRowNum >= jag_[columnNum-1].size())
      node->value = underlyingChildValue(jag_[columnNum-1][jag_[columnNum-1].size() - 1]->value,
                                         sRowNum - jag_[columnNum-1].size() + 1, columnNum, M);
    else
      node->value = underlyingChildValue(jag_[columnNum-1][sRowNum - (sRowNum % inQ)]->value,
                                         sRowNum % inQ, columnNum, M);

    column[rowNum] = node;
  }

jag_.push_back(column);
     }
 }
```

The empty matrix constructor is:

Listing 20: ../Source/dstruct.h [Line 271 to 294]

```
 Matrix(int inQ, unsigned int inN)
 {
   Constr(inQ);

   MatrixNode<double> *node;
   vector<MatrixNode<double> *> column;

   unsigned int columnHeight;
   for(unsigned int columnNum = 1; columnNum < inN+1; columnNum++)
     {
columnHeight = 1 + columnNum * (inQ-1);
column = vector<MatrixNode<double> *>(columnHeight);

for(unsigned int rowNum = 0; rowNum < columnHeight; rowNum++)
  {
    node = new MatrixNode<double>;

    node->value = 0.0;
    column[rowNum] = node;
  }

jag_.push_back(column);
     }
 }
```

Where Constr is:

Listing 21: ../Source/dstruct.h [Line 255 to 267]

```
public:
  void Constr(int inQ)
  {
    degree_ = inQ;

    MatrixNode<double> *node = new MatrixNode<double>;
    vector<MatrixNode<double> *> column = vector<MatrixNode<double> *>(1);

    // add the root node first.
    node->value = 0.0;
    column[0] = node;
    jag_.push_back(column);
  }
```

3.3 Derivative Type

There are two different types of derivatives:

- Non Path Dependent

  These derivatives do not care about the path taken to calculate the derivative at a particular node. They can be held in both trees and matrices.

- Path Dependent

  These derivatives need path information to calculate the value of the derivative at a node. They can only be held in trees, since matrices do not have a concept of an individual path to a node.

Derivatives are implemented as policies. The basic structure of every derivative is:

```
struct DerivativeName
{
  bool pathDependent;

  void derivativeInit(unsigned int N)
  {

  }

  void visitStock(unsigned int depth, unsigned int c, double inStockValue)
  {

  }
```

```
  double payOff(double K, double inStockValue)
  {

  }

  double derivativeValue(stack<double> *childStack, unsigned int T,
vector< vector<double> > P, vector<double> disc,
        double K, double stockValue)
  {

  }
};
```

`bool pathDepedent` is set to either true or false, depending on the derivative.

`void derivativeInit(unsigned int N)` is called by `pricing` before any other functions. Since the policies are implemented in structures, this is where the `pathDependent` variable can be set to true or false (as a hardcoded value). Path Dependent derivatives may also need to know N before they start, so it is provided here.

`void visitStock(unsigned int depth, unsigned int c, double inStockValue)` is for path dependent derivatives. It is called by `pricing` when a tree is used and nodes are visited. Notice that `pricing` uses a post order traversal to travel through the tree and `visitStock` is only called when nodes are advanced to, *so visitStock is not called for each individual path.* That would be unnecessary since the derivative needs to be aware that `pricing` uses a post order traveral and it can figure out the path to each node by understanding how `visitStock` is called.

Take for example, the tree in the included file `ecall-derivative.png`

Here is the order that `visitStock` will be called:

depth	*c*	*inStockValue*
0	0	100.00
1	0	90.91
2	0	82.65
3	0	75.13
3	1	90.91
2	1	100.00
3	0	90.91
3	1	110.00
1	1	110.00
2	0	100.00
3	0	90.91
3	1	110.00
2	1	121.00
3	0	110.00
3	1	133.10

`double payOff(double K, double inStockValue)` is called when `pricing` is at an end node. `payOff` returns the value of the derivative at this node, where `K` is the derivative parameter and `inStockValue` is the value of the stock at the corresponding end node.

`double derivativeValue(stack<double> *childStack, unsigned int T, vector< vector<double> > P, vector<double> disc, double K, double stockValue)` is called when `pricing` is not on an end node. This function returns the value of the derivative at this node. The paramaters are:

Name	*Meaning*
childStack	The child nodes of this derivative node
T	Current time step
P	vector of derivative probabilities
disc	Discounted interest rate
K	Derivative paramater
stockValue	Value of the stock at the corresponding node

Notice that it is this functions responsibility to pop the children nodes off of `childStack` when it is done with them.

Below is the entire European Call derivative. It is not path dependent.

Listing 22: ../Source/derivative/europeanCall.h [Line 3 to 53]

```
struct EuropeanCall
{
  bool pathDependent;

  void derivativeInit(unsigned int N)
  {
```

Listing 22: ../Source/derivative/europeanCall.h [Line 3 to 53] (continued)

```
    pathDependent = false;

    return;
  }

  void visitStock(unsigned int depth, unsigned int c, double inStockValue)
  {
    return;
  }

  double payOff(double K, double inStockValue)
  {
    return MAX(0.0, inStockValue - K);
  }

  double derivativeValue(stack<double> *childStack, unsigned int T, vector< vector<double> > P,
                         vector<double> disc, double unused1, double unused2)
  {
    int pindex;                     // index into first vector of P
                                    // needed because P might be time dependent or time independent

    if(P.size() > 1)     // is P time dependent?
      pindex = T;
    else
      pindex = 0;

    double derivativeValue = 0.0;

    for(int childNum = P[pindex].size() - 1; childNum >= 0; childNum--)
      {
        derivativeValue += childStack->top() * P[pindex][childNum];

        childStack->pop();
      }

    double r;

    if(disc.size() > 1)   // is R time dependent?
      r = disc[T];
    else
      r = disc[0];

    return r * derivativeValue;
  }
};
```

Notice that for `visitStock` this derivative simply returns. Since it is not path dependant, it has no need to keep track of any path information.

Also notice that the European Call has no early exercise condition, so it ignores the last two arguments given to it in `derivativeValue`.

For a derivative with an early exercise (American Put), it simply uses the last two variables, K and stockValue where the return value changes from the European Call:

```
return r * derivativeValue;
```

to computing a maximum in the return value:

```
return MAX(r * derivativeValue, K - stockValue);
```

Compare this derivative to a path dependent one, the Average Strike European Call:

Listing 23: ../Source/derivative/averageStrikeEuropeanCall.h [Line 2 to 64]

```
struct AverageStrikeEuropeanCall
{
  bool pathDependent;

  double stockSum;
  unsigned int averageStrikeN;

  void derivativeInit(unsigned int N)
  {
    stockSum = 0.0;
    averageStrikeN = N + 1;

    pathDependent = true;
  }

  double payOff(double K, double inStockValue)
  {
    double returnValue;

    returnValue = MAX(0.0, inStockValue - (stockSum / (double)averageStrikeN));

    stockSum -= inStockValue;

    return returnValue;
  }

  double derivativeValue(stack<double> *childStack, unsigned int T, vector< vector<double> > P,
                         vector<double> disc, double K, double stockValue)
  {
    int pindex;                    // index into first vector of P
                                   // needed because P might be time dependent or time independent
```

➥

Listing 23: ../Source/derivative/averageStrikeEuropeanCall.h [Line 2 to 64] (continued)

```
    if(P.size() > 1)     // is P time dependent?
      pindex = T;
    else
      pindex = 0;

    double derivativeValue = 0.0;

    for(int childNum = P[pindex].size() - 1; childNum >= 0; childNum--)
      {
        derivativeValue += childStack->top() * P[pindex][childNum];

        childStack->pop();
      }

    double r;

    if(disc.size() > 1)   // is R time dependent?
      r = disc[T];
    else
      r = disc[0];

    stockSum -= stockValue;

    return r * derivativeValue;
  }

  void visitStock(unsigned int depth, unsigned int c, double inStockValue)
  {
    stockSum += inStockValue;
  }
};
```

The Average Strike European Call needs to have the sum of all the nodes along the path to an end node (actually the average, but the sum is required to calculate that). When `visitStock` is called, this derivative simply adds the value of the stock to it's own internal variable `stockSum`. After either of `payOff` or `derivativeValue` is called, the derivative subtracts that stock value from it's sum, since that node has been visited.

3.3.1 Implementing another path dependent derivative

Let's say we want to implement a barrier derivative, where the derivative parameter (K) represents a barrier and for any node along a path if the stock in that path leading to that node goes below the barrier (K), the pay off is 0, otherwise the derivative is a standard european one.

You would need to add another variable to this structure (let's say it is a stack called `barrier` which accepts integers). Initally, you put nothing on the stack.

`derivativeInit` will need to be modified to accept two variables, N and K. It does not right now - all that is sent is N. This change involves going through pricing to send the parameter K along with N. The derivatives will have to be changed as well to accept this new parameter (easy since they ignore it). When derivativeInit sends K to this new barrier derivative, it copies the value to inside the structure where it can further work with it.

Alternatively, visitStock can be modifed to pass K along with the depth, c and inStockValue. This is more difficult, since it involves modifying the functor to pass it. The memory saving tree algorithm in pricing would have to be modified as well, but that is far easier then modifying the functor (for which the complete tree uses).

Then, when visitStock is called, the code simply compares inStockValue with K. If it has gone below (or above, depending on the derivative) it pushes a 0 onto the stack. Otherwise, it pushes a 1 on the stack.

When payOff or derivativeValue are called, the code will act like a european option, except before returning it will have to examine the entire stack to see if there is a 0 on it. If so, these functions return 0, otherwise (the stack contains all ones), return the european value. Before returning (but after examining the stack), pop the top value from the stack.

This approach should work for calculating a barrier derivative (although it has not been implemented or tested). A naive approach would use a single multiplier, changing it from 0 to 1 when the barrier is reached, but this approach runs into problems because it is not trivial to change the 0 back to a 1 when the path has gone away from the barrier.

3.4 Pricing

In martingalePricing, the code which calls the policies is called `pricing` and it takes three different template arguments, where each template argument is a policy. Here is how code outside the library will call `pricing`:

```
Pricing<UNDERLYING(Binom), Tree<Complete, Multiplicative>, EuropeanCall >
*derivative = new Pricing<UNDERLYING(Binom),
             Tree<Complete, Multiplicative>, EuropeanCall >(param, 100.0);
```

The first template argument specifies the desired underlying type (Binom, Binom2, TimeHalvingTrinom). Notice that the type of underlying is passed to an include macro called `UNDERLYING`, which is defined in pricing.h to be:

Listing 24: ../Source/pricing.h [Line 2 to 2]

```
#define UNDERLYING(type)        type, IP_type
```

The purpose of this macro is to take one argument and expand it into two arguments, where the first argument is left untouched and the new second argument is simply the name of the original argument with `IP_` prepended to

it. Without this macro, any code which calls `pricing` would need to specify both the desired underlying type and the input parameter structure for that underlying type. Since the input parameters for each underlying type must be named `IP_<name of underlying>`, the `UNDERLYING` macro eliminates the need for calling code to specify the input arguments structure type.

The next argument to `pricing` is the desired data structure. In the example above, it is `Tree<Complete, Multiplicative>`. The data structure is a policy which itself uses policies (`Complete` and `Multiplicative`). There are only two choices for the data structure here:[4]

1. Tree
2. Matrix

The final argument to `pricing` is the desired derivative type. As listed in the previous section, there are several choices:

- European Call
- European Put
- American Put
- Average Strike European Call
- Average Strike American Put

Notice that it is only possible to run a path dependent derivative (Average Strike European Call or Average Strike American Put) with a tree as the data structure. An attempt to use a matrix will yield a run time error.

All of the work for calculating the derivative is done in the constructor. Thus, after the piece of code in listing 24 runs, the variable `derivative` will contain a priced derivative. Each type of data structure for pricing will include the method `getDerivativeRootValue` which contains a `double` of the priced derivative root. Other data structures may add additional access methods, see the next sections.

Since there are four distinct algorithms that `pricing` works with, each one will be described in a seperate section.

3.4.1 Complete Tree

Here is the code which calculates the derivative for a complete tree:

Listing 25: ../Source/pricing.h [Line 29 to 68]

```
public:
 Pricing(inputParameters inParam, double K) : UnderlyingModel(inParam)
```

[4]Of course, those policies themselves take policies

Listing 25: ../Source/pricing.h [Line 29 to 68] (continued)

```
   {
     derivativeInit(N);

     specificFunctor_ = new TSpecificFunctor< Pricing<UnderlyingModel, inputParameters,
Tree<Complete, UnderlyingMethod>, DerivativeMethod> >(this, &DerivativeMethod::visitStock);

     sTree_ = new Tree<Complete, UnderlyingMethod>(Q, S0, N, M, specificFunctor_);
     dTree_ = new Tree<Complete, Empty>(Q, N);

     TreeNode<double> *sNode;
     TreeNode<double> *dNode;

     PostOrderIterator<double> *ptrDPostOrderIter = dTree_->GetPostOrderIterator();
     PostOrderIterator<double>& dPostOrderIter  = *ptrDPostOrderIter;

     PostOrderIterator<double> *ptrSPostOrderIter = sTree_->GetPostOrderIterator();
     PostOrderIterator<double>& sPostOrderIter = *ptrSPostOrderIter;

     stack<double> *childStack = new stack<double>;

     while(! sPostOrderIter.isDone() && ! dPostOrderIter.isDone())
{
  sNode = *sPostOrderIter;
  dNode = *dPostOrderIter;

  if(sNode->children.size() == 0)
    dNode->value = payOff(K, sNode->value);
  else
    dNode->value = derivativeValue(childStack, sNode->depth, P, disc, K, sNode->value);

  childStack->push(dNode->value);

  ++sPostOrderIter;
  ++dPostOrderIter;
}

     rootDval = dNode->value;
   }
```

Intially, the algorithm registers a functor with the post order iterator for the stock[5]. This is necessary so that the `visitStock` method of the derivative can be called. Notice that `visitStock` is *not* called by this code, as that is left for the post order iterator to call when it advances over the tree.

As discussed earlier, to calculate the derivative a post order iterator is used over the stock tree. Each time the iterator for the stock is advanced (`sPostOrderIter`), the corresponding iterator for the derivative tree (`dPostOrderIter`) is advanced so that they are both synchronized.

[5] See appendix B

When the algorithm starts off, the first node that the iterator gives is the extreme left child of the tree. The algorithm will then call the derivative method `payOff` for that node and push the returned result onto the stack `childStack` which will be passed back to the derivative when `derivativeValue` is called. When the post order iterator returns a parent node, the algorithm calls `derivativeValue` and pushes that result on the stack `childStack`.

Once the post order iterator runs out of nodes, the variable `dNode` holds the root derivative value which is copied to the variable `rootDval` where it is ready to be returned to calling code through `getDerivativeRootValue`.

Pricing provides several access methods for outside code to examine both the derivative and stock. The methods are:

- `double getDerivativeRootValue(void)` - returns the root value of the derivative. All algorithms provide this method.
- `PostOrderIterator<double>& getStockPostOrderIterator(void)` - returns a post order iterator over the stock tree.
- `PostOrderIterator<double>& getDerivativePostOrderIterator(void)` - returns a post order iterator over the derivative tree.
- `BreadthFirstIterator<double>& getStockBreadthFirstIterator(void)` - returns a breadth first iterator over the stock tree.
- `BreadthFirstIterator<double>& getDerivativeBreadthFirstIterator(void)` - returns a breadth first iterator over the derivative tree.

If graphics were enabled, then the following two additional methods are available:

- `graphStock(char *fname, unsigned int width, unsigned int height)` - plots the stock using the `gd` library to a `png` file.
- `graphDerivative(char *fname, unsigned int width, unsigned int height)` - plots the derivative using the `gd` library to a `png` file.

Graphic example

For example, say you have a stock that you wish to dump to a file named `stock.png` with width of 800 pixels and height of 600 pixels. After you run `pricing` and have a pricing variable to work with (eg. the variable is called `derivative`), the following code will work:

```
derivative->graphStock("stock.png", 800, 600);
```

3.4.2 Memory Saving Tree

The memory saving tree algorithm uses two stacks: one for the stock values and another for the derivative values.

Listing 26: ../Source/pricing.h [Line 140 to 162]

```
public:
  Pricing(inputParameters inParam, double inK) : UnderlyingModel(inParam),
    Tree<MemorySaving, UnderlyingMethod>()
    {
      K = inK;

      sRoot_ = new TreeNode<double>();
      sRoot_->value = S0;

      stack<double> stock;
      stack<double> derivative;

      derivativeInit(N);

      stock.push(S0);
      visitStock(0, 0, stock.top());

      createTree(&stock, &derivative, 0);

      rootDval = derivative.top();

      return;
    }
```

Where `createTree` is defined as a recursive function[6] which simply does a post order traversal over a non-existant tree all the while keeping the stack for the stock and derivative in sync and updated. Since this code does not use an interator like the complete tree does, it is responsible for calling the `visitStock` derivative function.

The memory saving tree algorithm only provides the following access method:

- `double getDerivativeRootValue(void)` - returns the root value of the derivative. All algorithms provide this method.

3.4.3 Complete Matrix

The complete matrix algorithm performs two checks before it calculates the derivative:

1. The underlying *must* be recombining
2. The derivative *must not* be path dependent

If either of the two conditions are not met, the algorithm prints out a message to `stderr` and returns back to the calling code without calculating anything.

[6] Although difficult, it is possible to do this iteratively, see appendix A

Listing 27: ../Source/pricing.h [Line 212 to 258]

```
public:
 Pricing(inputParameters inParam, double K) : UnderlyingModel(inParam)
 {
   derivativeInit(N);

   rootDval = 0.0;

   if(! recomb)
     {
fprintf(stderr, "Error:  cannot run a non-recombining underlying on a matrix\n");
return;
     }

   if(pathDependent)
     {
fprintf(stderr, "Error:  cannot run a path dependent derivative on a matrix\n");
return;
     }

   sJag_ = new Matrix<Complete, UnderlyingMethod>(Q, S0, N, M);
   dJag_ = new Matrix<Complete, Empty>(Q, N);

   stack<double> *Stk = new stack<double>();

   for(unsigned int colNum = N; ;colNum--)
     {
for(unsigned int rowNum = 0; rowNum < sJag_->Height(colNum); rowNum++)
  {
    if(colNum == N)
      dJag_->Set(colNum, rowNum, payOff(K, sJag_->Get(N, rowNum)));
    else
      {
        for(unsigned int childNum = 0; childNum < Q; childNum++)
          Stk->push(dJag_->Get(colNum + 1, rowNum + childNum));

        dJag_->Set(colNum, rowNum, derivativeValue(Stk, colNum, P, disc, K,
                                                   sJag_->Get(colNum, rowNum)));

      }
  }

if(colNum == 0)
  break;
     }

   rootDval = dJag_->Get(0, 0);
 }
```

Listing 27: ../Source/pricing.h [Line 212 to 258] (continued)

The algorithm then starts in the rightmost column and calls the derivative method `payOff` to calculate the end nodes of the derivative.

Then the algorithm moves one column to the left and calculates the value of the derivative at the current node by pushing it's two "children" nodes in the column to the right onto the stack `Stk` where it is passed to the derivative method `derivativeValue`.

The algorithm repeats the above until the reaches the leftmost column of the matrix. At that point the node in the lower left corner of the matrix represents the "root" node of the matrix and its value is copied to `rootDval` where it can be returned by the access method `getDerivativeRootValue`.

Technical note: The reason for the strange for loop exit condition is caused by the fact that the control variable `colNum` is held in an `unsigned int` where it is not permitted to go negative, so the check for the exit has to be at the bottom of the loop.

The following access methods are provided by the complete matrix pricing algorithm:

- `double getDerivativeRootValue(void)` - returns the root value of the derivative. All algorithms provide this method.
- `double getStockValue(unsigned int col, unsigned int row)` - returns the stock value in the given column and row of the matrix
- `double getDerivativeRootValue(void)` - returns the derivative value in the given column and row of the matrix

If graphics were enabled, then the following two additional methods are available:

- `graphStock(char *fname, unsigned int width, unsigned int height)` - plots the stock using the `gd` library to a `png` file.
- `graphDerivative(char *fname, unsigned int width, unsigned int height)` - plots the derivative using the `gd` library to a `png` file.

3.4.4 Memory Saving Matrix

The memory saving matrix performs the exact same two checks as the complete matrix does (see above).

This algorithm is the only one where the underlying and the derivative are not stored in the same data structures. In this algorithm, the stock is stored in a complete matrix because there is no efficent formula to calculate the value of the stock in the matrix on the fly. Thus, it is necessary to store the entire matrix for the stock. The derivative is temporarily held in a STL queue of size `Q`.

Listing 28: ../Source/pricing.h [Line 304 to 375]

```
public:
  Pricing(inputParameters inParam, double K) : UnderlyingModel(inParam)
  {
    derivativeInit(N);

    rootDval = 0.0;

    if(! recomb)
      {
        fprintf(stderr, "Error:  cannot run a non-recombining underlying on a matrix\n");
        return;
      }

    if(pathDependent)
      {
        fprintf(stderr, "Error:  cannot run a path dependent derivative on a matrix\n");
        return;
      }

    sJag_ = new Matrix<Complete, UnderlyingMethod>(Q, S0, N, M);

    queue<double> *Que[Q];

    for(unsigned int i = 0; i < Q; i++)
      Que[i] = new queue<double>();

    stack<double> *Stk = new stack<double>();

    double dVal;

    for(unsigned int colNum = N; ;colNum--)
      {
        for(unsigned int rowNum = 0; rowNum < sJag_->Height(colNum); rowNum++)
          {
            if(colNum == N)
              dVal = payOff(K, sJag_->Get(N, rowNum));
            else
              {
                for(unsigned int childNum = 0; childNum < Q; childNum++)
                  {
                    Stk->push(Que[childNum]->front());
                    Que[childNum]->pop();
                  }

                dVal = derivativeValue(Stk, colNum, P, disc, K, sJag_->Get(colNum, rowNum));
              }
```

➥

Listing 28: ../Source/pricing.h [Line 304 to 375] (continued)

```
            if(rowNum == 0)
                Que[0]->push(dVal);
            else if(rowNum == sJag_->Height(colNum)-1)
                Que[Q-1]->push(dVal);
            else if((rowNum == 1) && Q == 3)
              {
                Que[0]->push(dVal);
                Que[1]->push(dVal);
              }
            else if( (rowNum == sJag_->Height(colNum)-2) && Q == 3)
              {
                Que[Q-1]->push(dVal);
                Que[1]->push(dVal);
              }
            else
              for(unsigned int childNum = 0; childNum < Q; childNum++)
                  Que[childNum]->push(dVal);
          }

        if(colNum == 0)
          break;
      }

    rootDval = dVal;
  }
```

This algorithm is bit complicated, since it uses a queue to keep track of the derivative values. This queue is used by the algorithm to keep track of the "children" of nodes in the derivative. This is complicated since children nodes in a matrix can have up to 3 parents (when Q is 3), but *not every* child node has this situation, so the algorithm has to detect when a child node has only one parent, two parents, etc. and set up the queue appropiately.

Otherwise, this algorithm proceeds in a similar structure as the complete matrix one does, starting with the rightmost column in the matrix and proceeding to the left. A major difference is that when calling `derivativeValue` where a stack of child node values is required, this algorithm cannot simply consult the derivative matrix, since it does not exist. Thus a queue is used which holds the child node values, and these queue values are pushed into a stack which is passed to `derivativeValue`.

The memory saving matrix algorithm only provides the following access method:

- `double getDerivativeRootValue(void)` - returns the root value of the derivative. All algorithms provide this method.

A Post Order Traversal Code

Included here is code which does both a recursive and iterative post order traversal of an imaginary tree (one where there is no data structure present).

Please note that the iterative memory saving code is not present in martingalePricing. Right now, it is simply recursive post order tree traversal.

By carefully seeing the similarities of the recursive code presented here with the current memory saving tree algorithm, it is not terribly difficult to convert the recursive code over to an iterative version since the algorithm for an iterative post order tree traversal is shown below.

Listing 29: postorder.cpp [Line 2 to 91]

```
// post order tree traversal

#include <stack.h>
#include <stdio.h>

void iterative(unsigned int Q, unsigned int N);
void recursive(unsigned int Q, unsigned int N, unsigned int depth, unsigned int c);

main()
{
  unsigned int Q = 2;
  unsigned int N = 4;

  printf("Recursive traversal\n");
  recursive(Q, N, 0, 0);

  printf("Iterative traversal\n");
  iterative(Q, N);
}

void recursive(unsigned int Q, unsigned int N, unsigned int depth, unsigned int c)
{
  if(depth == N)
    return;

  for(unsigned int i = 0; i < Q; i++)
    recursive(Q, N, depth+1, i);

  printf("%d, %d\n", depth, c);
}

void iterative(unsigned int Q, unsigned int N)
{
  stack<unsigned int> *c = new stack<unsigned int>();
  unsigned int tmpInt;
```

➥

Listing 29: postorder.cpp [Line 2 to 91] (continued)

```
printf("%d, %d\n", 0, 0);

for(unsigned int i = 0; i < N-1; i++)
  {
    if(i == N-3)
      {
        printf("%d, 0\n", c->size()+1);
        c->push(1);
      }
    else
      {
        c->push(0);

        if(i != N-2)
          printf("%d, %d\n", c->size(), c->top());
      }
  }

while(!c->empty())
  {
    printf("%d, %d\n", c->size(), c->top());

    tmpInt = c->top();
    c->pop();
    c->push(tmpInt + 1);


    if(c->top() > Q-1)
      {
        c->pop();

        while(!c->empty() && c->top()+1 > Q-1)
          {
            printf("%d, %d\n", c->size(), c->top());
            c->pop();
          }

        if(c->empty())
          {
            printf("0, 0\n");
            return;
          }

        printf("%d, %d\n", c->size(), c->top());

        tmpInt = c->top();
        c->pop();
        c->push(tmpInt + 1);
```

➥

Listing 29: postorder.cpp [Line 2 to 91] (continued)

```
          for(unsigned int i = c->size(); i < N-1; i++)
            c->push(0);
        }
    }
}
```

B Tree Iterators

B.1 Breadth First Iterator

The `BreadthFirstIterator` implements a tree breadth first iterator using a queue from the Standard Template Library:

Listing 30: ../Source/iterators.h [Line 8 to 22]

```
template <class NodeData>
class BreadthFirstIterator
{
private:
  TreeNode<NodeData> *root;
  unsigned int degree;
  queue<TreeNode<NodeData> *> Q;

public:
  BreadthFirstIterator(TreeNode<NodeData> *inRoot, unsigned int inDegree);
  void Reset();
  bool isDone();
  TreeNode<NodeData> *operator*();
  void operator++();
};
```

The empty arguments constuctor is not valid and throws an exception if called. A `BreadthFirstIterator` object is instantiated by the `Tree` class only. The Tree passes its root Node to the constructor of the `BreadthFirstIterator` which is defined as follows:

Listing 31: ../Source/iterators.h [Line 50 to 56]

```
template <class NodeData>
BreadthFirstIterator<NodeData>::BreadthFirstIterator(TreeNode<NodeData> *inRoot, unsigned int inDegree)
{
  degree = inDegree;
  root = inRoot;
  Q.push(root);
}
```

All this does is record the degree integer and root node pointer as well as initializing the queue Q with the root node.

The Reset method simply empties the queue and places the root at the front of the queue:

Listing 32: ../Source/iterators.h [Line 60 to 67]

```
template <class NodeData>
void BreadthFirstIterator<NodeData>::Reset()
```

Listing 32: ../Source/iterators.h [Line 60 to 67] (continued)

```
{
  while(!Q.empty())
    Q.pop();

  Q.push(root);
}
```

The isDone method checks to see if the queue is empty:

Listing 33: ../Source/iterators.h [Line 71 to 75]

```
template <class NodeData>
bool BreadthFirstIterator<NodeData>::isDone()
{
  return Q.empty();
}
```

The dereferencing operator of the iterator simply returns the head value of the queue:

Listing 34: ../Source/iterators.h [Line 79 to 83]

```
template <class NodeData>
TreeNode<NodeData> *BreadthFirstIterator<NodeData>::operator *()
{
  return Q.front();
}
```

The increment operator removes the top node from the queue and the children of that node are pushed into the queue. The `if` statement makes sure that we do not try to push the children of a leaf node into the queue.

Listing 35: ../Source/iterators.h [Line 87 to 101]

```
template <class NodeData>
void BreadthFirstIterator<NodeData>::operator++()
{
  TreeNode<NodeData> *head, *child;

  head = Q.front();
  Q.pop();

  if(head->children.size() == degree)
    for(unsigned int curDegree = 0; curDegree < degree; curDegree++)
      {
        child = head->children[curDegree];
        Q.push(child);
      }
```

Listing 35: ../Source/iterators.h [Line 87 to 101] (continued)

```
}
```

B.2 Post Order Tree Iterator

The `PostOrderIterator` implements a tree post order iterator using two stacks from the Standard Template Library. It also allows a member function of an object to register itself where it will be called whenever the iterator travels down a path. This is used by path dependent derivatives to capture path information.

Listing 36: ../Source/iterators.h [Line 26 to 44]

```
template <class NodeData>
class PostOrderIterator
{
private:
  TreeNode<NodeData> *root;
  unsigned int degree;
  stack<TreeNode<NodeData> *> S;
  stack<int> curDegrees;
  TFunctor *visitListener;

public:
  PostOrderIterator(TreeNode<NodeData> *inRoot, unsigned int inDegree, TFunctor *inVisitListener);
  void registerVisitListener(TFunctor *inVisitListener);
  void Reset();
  bool isDone();
  TreeNode<NodeData> *operator*();
  void operator++();
  void dump();
};
```

The functor argument to the constructor is allowed to be `null`, in which case the iterator will not call any function when it travels down a path.

As well, during the operation of the iterator, calling code may change or unregister the listener functor by calling `registerVisitListener`:

Listing 37: ../Source/iterators.h [Line 120 to 124]

```
template <class NodeData>
void PostOrderIterator<NodeData>::registerVisitListener(TFunctor *inVisitListener)
{
  visitListener = inVisitListener;
}
```

One stack contains the nodes to return `S` while the second stack, `curDegrees` contains integers indicating which children have been visited already, where 0 represents the left most child, 1 the child next to it, etc. `S` is populated by

first pushing the root onto the stack, then by pushing the roots left most child, followed by that childs leftmost child, etc., until the left most leaf in the tree is reached. `curDegrees` is initially populated with one 0 for each node visited, since only the left most node has been travelled to.

Thus if the top integer on the stack `curDegrees` contains a 1, this indicates that for the node on the top of the stack `S` we have already visited its second child (the one to the right of the leftmost child).

The Reset method does exactly what is described two paragraphs above, as well it calls the functor (if registered) when it travels down a path:

Listing 38: ../Source/iterators.h [Line 128 to 154]

```
template <class NodeData>
void PostOrderIterator<NodeData>::Reset()
{
  TreeNode<NodeData> *walker;

  while(!curDegrees.empty())
    curDegrees.pop();

  while(!S.empty())
    S.pop();

  for(walker = root; walker->children.size() == degree;
      walker = walker->children[0])
    {
      if(visitListener)
        (*visitListener)(walker);

      S.push(walker);
      curDegrees.push(0);
    }

  if(visitListener)
    (*visitListener)(walker);

  curDegrees.push(0);
  S.push(walker);
}
```

The isDone method simply checks to see if the stack `S` is empty. It could have also checked the stack `curDegrees` to see if it is empty, but since the two stacks are always the same size, the code does not bother.

Listing 39: ../Source/iterators.h [Line 158 to 162]

```
template <class NodeData>
bool PostOrderIterator<NodeData>::isDone()
{
```

Listing 39: ../Source/iterators.h [Line 158 to 162] (continued)

```
  return S.empty();
}
```

The dereferencing operator of the iterator simply returns the top node on the stack:

Listing 40: ../Source/iterators.h [Line 166 to 170]

```
template <class NodeData>
TreeNode<NodeData> *PostOrderIterator<NodeData>::operator *()
{
  return S.top();
}
```

The increment operator pops the top element from both stacks and if the stacks are empty, it stops. Otherwise, the new top of the stack `curDegree` integer is incremented by one to indicate we are now visting the children to the right of what was visited before. If this increment is greater then the degree of the tree, then we have visited all of the children of this node, so we do nothing leaving this node as the top of the stack.

If the increment is less then the degree of the tree, we still have children to visit, so, just like the reset code, the leftmost children are pushed onto stack `S`, and for each child child pushed a 0 is pushed onto the stack `curDegrees` indicating we have only visited the leftmost children of these nodes.

Whenever a node is visited, the visit listener functor is called if registered.

This technique yields a post order iteration of the tree.

Listing 41: ../Source/iterators.h [Line 174 to 222]

```
/* This code is quite complicated, as it generally is expressed
 * as a recursive function.
 *
 * We need two stacks: one for the objects themselves
 *                     another to keep track of which child we've visited
 */
template <class NodeData>
void PostOrderIterator<NodeData>::operator++()
{
  TreeNode<NodeData> *top, *walker;
  unsigned int curDegree;

  curDegrees.pop();
  S.pop();

  if(S.empty())
    return;
```

➥

Listing 41: ../Source/iterators.h [Line 174 to 222] (continued)

```
  top = S.top();

  curDegree = curDegrees.top();

  if(top->children.size() != degree)
    return;

  curDegree++;

  if(curDegree < degree)
    {
      curDegrees.pop();
      curDegrees.push(curDegree);

      for(walker = top->children[curDegree];
          walker->children.size() == degree; walker = walker->children[0])
        {
          if(visitListener)
            (*visitListener)(walker);

          S.push(walker);
          curDegrees.push(0);
        }

      if(visitListener)
        (*visitListener)(walker);

      S.push(walker);
      curDegrees.push(0);
    }
}
```

C Functor

A functor is simply a pointer to a member function of an object.

In martingalePricing, functors are used by the post order iterator to call a registered listener. As mentioned in the comments, the functor acts as a bridge between the iterator (which deals with TreeNodes) and the derivative (which deals with depth, child number and values).

Listing 42: ../Source/visitor.h [Line 2 to 41]

```
// see
// http://www.newty.de/CCPP/functor/functor.html#chapter4
//
// chapter 5 of andrei ignored, since it is far too complicated
// for what is needed here.

// generic functor
// implementation class inherits from this, but this is passed to functions
class TFunctor
{
public:
  virtual void operator()(TreeNode<double> *inNode) = 0;
};

// implementation class, the constructor takes a pointer to an object (pricing)
// and acts as a bridge between the iterator listener callback and
// the visitStock provided by one of the derivative classes (which is stored inside
// pricing)

template <class TClass> class TSpecificFunctor : public TFunctor
{
private:
  void (TClass::*fpt)(unsigned int depth, unsigned int c, double inStockValue);
  TClass *ptr2Object;

public:
  TSpecificFunctor(TClass *inPtr2Object, void (TClass::*_fpt)(unsigned int depth, unsigned int c,
                                                              double inStockValue))
  {
    ptr2Object = inPtr2Object;
    fpt = _fpt;
  }

  virtual void operator()(TreeNode<double> *inNode)
  {
    // translate the call from the callback from the iterator to the format
    // expected by one of the derivative classes
    (*ptr2Object.*fpt)(inNode->depth, inNode->childNum, inNode->value);
  }
```

Listing 42: ../Source/visitor.h [Line 2 to 41] (continued)

```
};
```

References

[1] A. Alexandrescu, *Modern C++ Design. Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[2] M.Avellaneda and P. Laurence, *Quantitative Modeling of Derivative Securities. From Theory to Practice.* Chapman and Hall, 2000.

[3] Les Clewlow, Chris Strickland, *Implementing Derivatives Models.* John Wiley and Sons.

[4] S. Lippman and J. Lajoie, *C++ Primer, 3rd. Edition.* Addison-Wesley, 1998.

[5] B. Stroustrup, *The C++ Programming Language, 3rd. Edition.* Addison-Wesley, 1997.

[6] P. Wilmott, J. Dewynne and S. Howison, *Option Pricing. Mathematical models and computation.* Oxford Financial Press, 1993.

[7] S. Ferrando, *Draft for martingalePricing V 0.6.1.* Private Document, 2002.