C++ Implementation of localized Monte Carlo on trees

Sebastian Ferrando and Alex Korobchevsky

November 6, 2004

Abstract

This technical report describes the associated C++ library *localMC: A C++ Implementation of localized Monte Carlo on trees*. This library implements some of the algorithms described in the reference [1]. The goal of the project was mainly to develop the functionality needed to obtain the numerical output required in that paper.

Contents

1 Goals

The system goal is option pricing by using a variety of underlyings, derivatives and algorithms. The underlyings implemented so far are binomial and continium. The derivatives used are European versions of the average strike, fixed strike and lookback. The algorithms used are Basic Monte Carlo, Exact values on a Tree, Local Monte Carlo and Local Monte Carlo with fast shift. Let us describe briefly each of the concepts mentioned above.

From mathematical point of view, an underlying is a relation which receives an asset value as an argument and returns it's next value. A continious underlying has an infinite number of outcomes with a given argument where as a discrete underlying has a finite number of outcomes. Therefore a binomial underlying has only two outcomes. Obviously, a continious underlying is much more realistic. For instance, if a stock has a value of a hundred dollars today it can have any value tomorrow. A binomial model would assume a stock can either go up or down so the only values it can have is 90 or 110 dollars.

A derivative is a kind of option contract from financial point of view. We will use only European derivatives i.e. which can be exercised only at the end of the time period. The derivatives we use so far are average strike, fixed strike and lookback. Mathematically, we are looking how much each option will pay us of. Average Strike is an option which pays the difference between end value and the average along the path if positive, zero othervise. Look back is an option which pays the difference between end value and the minimum along the path and always excersiced. However, for the strike we do care about the values along the path because they make up the average.

The main focus of our research is algorithms. So far we use Basic Monte Carlo, Local Monte Carlo and Local MC with Fast Shift. Basic Monte Carlo is based on performing a large number of simulations, accumulating the payoffs and computing an average option value. This algorithm is very simple and slow. Most of the CPU time is spent on generating random numbers. We can generate less random numbers by applying the so called shift techniques. Generic shift reuses any given by the user amount of random numbers. Fast shift uses the previous value of the stock to compute the next one. In summary, shift algorithms are faster but less reliable then standard MC and should be used when a fast approximation is required.

2 Design

2.1 Class hierarchy

The system we were describing above was implemented in ANSI C++. It was tested on linux with g++2.96 but hopefully will run on most modern C++ compilers if configured and customized properly. The design is based heavily on standard OO concepts: classes, inheritance and virtual functions. The extractor which invokes the g2 library is wriiten in ANSI C and connected to C++

application by *system* calls.

The underlying class hierarchy is based on Abstract Base Class underlying which has prototypes for all functions and all variables required by every particular underlying.

Listing 1: ../Src/Und/Underlying.h [Line 32 to 66]

```
struct Underlying
{ double U;
  double D;
  double r;
  double T;
  int N;
  double P;
  double sigma;

  GSL_RandomGen *gen;



  /***************************** next ********************************
  Function: next
  Purpose: Calculates next stock value
  Param: double - Current stock value
  Returns: Next stock value
  ***************************************************************/
  virtual double next(double) = 0;



  /************************** generatePath ************************
  Function: generatePath
  Purpose: Dumps the whole path at once. Forces specified number of ups
  Param: int - length of the path
         int - number of ups forced in the path
  Returns: Array of ups and downs
  ***************************************************************/
  virtual double* generatePath(int N, int n) = 0;
};
```

Please note that this design is subject to change when we introduce more underlyings. In addition to standard underlying parameters a pointer to random generator is included (an OO concept of composition). Two most interesting methods are *next()* and *setBranchProb()*. Next returns the value of the asset one time increment later and uses the random generator heavily. *setBranchProb()* computes the probability of the stock to go up and applies to binomial

underlyings only. *Binom()* and *Continium()* classes implement all methods from Underlying class. As a remark, in *Continium()* branch probability doesn't depend on ups and downs so *setBranchProb()* method has an empty body.

As mentioned above, this project is based on using lots of random numbers. Random generator code was taken from GNU scientific library and encapsulated into a class.

Listing 2: ../Src/Rng/GSL_RandomGen.h [Line 33 to 126]

```
class GSL_RandomGen
{ public:

    /************************* GSL_RandomGen ***************************
     * Function: Constructor
     * Purpose: Creates new generator object
     * Param: generator name, seed to start, precision in number of digits after the decimal point
     * Returns: N/A
     ********************************************************************/
     GSL_RandomGen(string = "mt19937", long = 1, int = 4);



    /************************* ~GSL_RandomGen ***************************
     * Function: Destructor
     * Purpose: Dellocates memory by using routine defined in GSL
     * Param: N/A
     * Returns: N/A
     ********************************************************************/
    ~GSL_RandomGen() { gsl_rng_free(r_); }



    /************************** getName ********************************
     * Function: getName
     * Purpose: Accessor to generator name parameter
     * Param: N/A
     * Returns: generator name
     ********************************************************************/
    string getName() { return gsl_rng_name(r_); }



    /************************** setSeed ********************************
     * Function: setSeed
     * Purpose: Sets generator seed to specified value
     * Param: seed
     * Returns: N/A
```

Listing 2: ../Src/Rng/GSL_RandomGen.h [Line 33 to 126] (continued)

```
    ************************************************************/
   void setSeed(long);



   /***************************** next ****************************
    * Function: next
    * Purpose: Returns uniform [0..1] random value
    * Param: N/A
    * Returns: Uniform random value
    ************************************************************/
   double next() { return gsl_rng_uniform(r_); }



   /************************** nextInt ****************************
    * Function: nextInt
    * Purpose: Returns random integer in [0..range]
    * Param: upper range boundary
    * Returns: Random integer
    ************************************************************/
   int nextInt(int range) { return gsl_rng_uniform_int(r_, range); }



   /************************** gauss ******************************
    * Function: gauss
    * Purpose: Returns random normal with mean 0 and given standard deviation
    * Param: standard deviation
    * Returns: Random double
    ************************************************************/
   double gauss(double);



   /************************ print *****************************
    * Function: print
    * Purpose: Display contents of the Binom Random generator object
    * Param: fp - output file stream
    * Returns: N/A
    ************************************************************/
   void print(FILE* fp) { fprintf(fp, "Gen:  %s, seed:  %d\n", name_.c_str(), seed_);  }

 private:
```

Listing 2: ../Src/Rng/GSL_RandomGen.h [Line 33 to 126] (continued)

```
    string name_;
    long seed_;
    gsl_rng* r_;
};
```

The interface for random generator class includes functions *setSeed()*, *next()* and *getName()*. The user has three random number generator to chose from: *mnt*, *taus* and *gfsr*. These three selected generators are bith best performance ratings from over 60 generators available at GSL. The default random generator is *mnt*. The implementation if the generators is hidden from the user.

All derivatives, both path dependents and independents have *init()* and *update()* methods. *init()* is called before sampling the path, *update()* is called between the timesteps to change intermediate values (sums or averages). *Init()* and *update()* have empty bodies for path dependent derivatives. There are two versions of update. The update with final stock, front and initial asset value is used for fast shift only.

Listing 3: ../Src/Der/Derivative.h [Line 22 to 63]

```
struct Derivative
{ /********************** init ***************************************
   * Function: init
   * Purpose: Initialises utility variables
   * Param: N/A
   * Returns: N/A
   * Note: Empty for path independent derivatives
   ***********************************************************************/
  virtual void init() = 0;




  /******************** payOff ***************************************
   * Function: payOff
   * Purpose: Returns the option payoff
   * Param: Final stock price
   * Returns: option payoff
   * Precondiion: Underlying and derivative parameter must be initialized
   ***********************************************************************/
  virtual double payOff(double) = 0;




  /********************** update ***************************************
   * Function: update
                                                                        ➥
```

Listing 3: ../Src/Der/Derivative.h [Line 22 to 63] (continued)

```
   * Purpose: Updates the intermidiate values between timesteps for
   *          path-dependent derivatives.
   * Param: Stock value at the given timestep
   * Returns: N/A
   * Note: Empty for path independent derivatives
   ****************************************************************/
  virtual void update(double) = 0;




  double min_St;
  double sum_St;
};
```

The algorithm implementation is quite different for every algorithm. But there is one method each algorithm is required to implement: *calcResults()* which computes option value and standard error. An algorithm has one parameter: starting asset value. As we already said the algorithms implemented so far are MC, Shift and Haar.

2.2 Physical file organization

All project files are put in *mcRAsum02* directory which has these subdirectories: *Doc*, *Src*, *Lib* and *Exe*. Two additional files are there: *readme* with a brief project description and *regs* that performs regression testing. regs compiles and runs underlying test program and the test application (subject to change). The Doc directory contains the pdweave file to allow extraction of code fragments with progdoc. A Makefile processes the pdweaved file with latex and create dvi and/or postcript versions of the documentation. In Exe subdirectory future application code will be placed. The Lib subdirectory contains the libraries used for the project. Each library includes header files, source files in case the user wishes to recompile the library and a static library file if no recompilation is needed. Each library has a benchmark to test it's basic functionality. The libraries we use so far are MyGSL (adapted from GNU scientific library and used to generate random numbers), G2 generates plots in PostScript format. When a maintaince programmer wishes to add a library he is adviced to follow this pattern.

We need more detailed coverage of Src subdirectory. It has four subdirectories: Rng contains the random generator code wrapped into a class. We must mention original GSL is implemented in C but we linked it in C++ and wrapped into a class for convenience. Und subdirectory contains all the underlying code. Note that all files are header files since all functions are brief and declared inline. A test directory includes underlying testing code. Dersubdirectory contains all the derivative code. All files are header files as in the underlyings but there is

no test directory since it makes little sense to test derivatives independently. However, algorithms do have implementation .cpp files since the functions are too complex to be placed inline. In the Alg directory we have a test subdirectory which contains current application code and a makefile to run it.

An Exe subdirectory contains the *gop, ngop* applications, the input, extractor files. In addition to that, all object files are put there.

Listing 4: ../Src/org [Line 2 to 41]

```
utilities.h*

Alg:
Algorithm.cpp*
Algorithm.h*
BasicMC.cpp*
BasicMC.h*
BasicShift.cpp*
BasicShift.h*
GenericShift.cpp*
GenericShift.h*
HaarCVMC.cpp*
HaarCVMC.h*
HaarNode.h*
Test/
testbinarytree.cpp*
Tree.h

Der:
Derivative.h*
Derivative.h~*
EuropeanCall.h*
EuropeanPut.h*
EuropeanStrike.h*
sourceLog*

Rng:
GSL_RandomGen.cpp*
GSL_RandomGen.h*
GSL_RandomGen.h~*
sourceLog*
Test/

Und:
Binom2.h
Binom.h*
Test/
Underlying.h
Underlying.h~
```

Listing 4: ../Src/org [Line 2 to 41] (continued)

3 Implementation

In this section we will go over some non trivial pieces of code in all directories.

3.1 Algorithms

Thats how we implement the exact value continium for the look back option:

Listing 5: ../Src/Alg/ContExactValues.cpp [Line 39 to 42]

```
double value = S * cumulative(dtwo) -
   exp(-r * t) * S * cumulative(dtwo - sd * sqrt(t)) +
    exp(-r * t) * (pow(sd, 2) / (2 * r)) * S * (cumulative(-dtwo + 2 * r * sqrt(t) / sd) -
         exp(r * t) * cumulative(-dtwo));
```

Thats how we implement the tree exact value (only critical section is shown):

Listing 6: ../Src/Alg/ExactValues.cpp [Line 33 to 58]

```
for (int k = 0; k <= binom->N; k++)
{ int* path = new int[binom->N];
  for (int i = 0; i < binom->N - k; i++)
    path[i] = 0;
  for (int i = binom->N - k; i < binom->N; i++)
    path[i] = 1;

  int comb = combinations(binom->N, k);
  for (int j = 1; j <= comb; j++)
  { double ST = S;
    ec->init();
    ec->update(ST);

    for (int i = 0; i < binom->N; i++)
    { if (path[i] == 1)
         ST *= binom->U;
      else if (path[i] == 0)
         ST *= binom->D;
      ec->update(ST);
    }

    double CT = ec->payOff(ST);
    sum_CT += CT * logProbability(1 - binom->P, binom->N, k) / comb;
    permutePath(path, binom->N);
  }
}
```

➥

Listing 6: ../Src/Alg/ExactValues.cpp [Line 33 to 58] (continued)

As we mentioned above, Local Monte Carlo has been combined together with fast shift. Here is the section which includes the generic fast shift. The minimum is updated for lookback, while the average is updated for the strikes.

Listing 7: ../Src/Alg/LocalMC.cpp [Line 75 to 90]

```
  for (int shifttimes = 0; shifttimes < shift - 1; shifttimes++)
  { ec->sum_St = (ec->sum_St - S) / path[shifttimes] + ST;
    if (ST > S)
      ec->min_St = MIN(ec->min_St * path[binom->N - shifttimes - 1], S);
    else
      ec->min_St = MIN(ec->min_St / path[shifttimes], ST);

    double CT = ec->payOff(ST);
    sum_i[k] += CT;

    if (counter[k] > 1)
      error[k] = ((counter[k] - 2) * error[k] +
          (counter[k] - 1.0) / counter[k] * pow(ST - sum[k] / (counter[k] - 1), 2)) / (counter[k] - 1);
counter[k]++;
sum[k] += ST;
  }
```

Implementation of an error with Local Monte Carlo

Listing 8: ../Src/Alg/LocalMC.cpp [Line 102 to 106]

```
    // Error computation
    double varTotal = 0;
for (int k = 0; k <= binom->N; k++)
      if (counter[k] > 0)
        varTotal += pow(logProbability(1 - binom->P, binom->N, k), 2) * error[k] / counter[k];
```

For implementation details refer to Requirements Document.

3.2 Utilities

Here is a function used by Exact values on Continium. It uses numerical approximation to normal Cumulative function. Number of steps can be changed if we would like to have more presicion.

Listing 9: ../Src/utilities.cpp [Line 25 to 40]

```
double cumulative(double x)
{  double width = x / 10000;
   if (width < 0)
```

Listing 9: ../Src/utilities.cpp [Line 25 to 40] (continued)

```
      width *= (-1);
   double sum = 0;

   for (int i = 0; i < 10000; i++)
   {  double value = 1 / sqrt(2 * M_PI) * exp((-1) * (width*i) * (width*i) / 2);
      sum += value * width;
   }
   if (x > 0)
      return 0.5 + sum;
   return 0.5 - sum;
}
```

Another important function is logarithmic probability function which important for determining class probability in local Monte Carlo.

Listing 10: ../Src/utilities.cpp [Line 164 to 169]

```
double logProbability(double pU, int N, int n)
{ double S = 0;
  for (int i = 0; i < n; i++)
    S += log ((double)(N - i) / (i + 1));
  return exp(N * log (1 - pU) + n * log (pU / (1 - pU)) + S);
}
```

3.3 Main program

The main generic option plotter application is straightforward. It gets data from the user by using prompts and the input file. We will only demonstrate a single line that performs the computations. It uses the underlying and derivative objects created by getting all user input and invoke *calcResults* on an algorithm object created already. The ouput results are stored in vectors and saved to output files.

Listing 11: ../Exe/gop.cpp [Line 165 to 170]

```
vector<double> cv;
vector<double> se;
cout << "Generator:  " << ptr_und->gen->getName() << endl;
long start = clock();
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Call Value " << cv.back() << ", standard error " << se.back() <<  ", time = " << (clock() - start)/10000
```

4 Test plan

4.1 Random generator test

The random number generator needs to be chosen from over 60 ones found in GNU Scientific Library. We are looking for both speed and rondomness. We need to make sure a number chosen will be uniform in range 0..1 and be generated as fast as possible. For this reason a benchmark is written which accepts 3 command line arguments: generator name, fraction and number of hits. For instance, we generate one million uniform numbers and check how many are below 0.3. The ideal generator will give 300000 numbers. That's how we check the accuracy: how close we get to the theoretical number. To test the speed standard UNIX time utility can be used to check how long does it take to generate a million random numbers. Therefore, a trade off needs to be found between speed and presicion. The generator which performs best is mnt19937. We confirm that finding by gsl rating of it's generators.

Listing 12: ../Src/Rng/Test/testgen.cpp [Line 43 to 64]

```
int main(int argc, char** argv)
{ if (argc != 4)
  { cout << "Usage:  ./testgen generator fraction hits " << endl;
    exit(1);
  }

  GSL_RandomGen gen(argv[1]);
  double frac = atof(argv[2]);
  long sim = atol(argv[3]);

  int count = 0;
  for (int i = 0; i < sim; i++)
  { double eps = gen.next();
    if (eps < frac)
      count++;
  }
  cout << "Simulations:  " << sim << endl;
  cout << "Generator:  " << argv[1] << endl;
  cout << "Fraction:  " << frac << endl;
  cout << "Hits:  " << count << endl;
}
```

4.2 Underlying test

To test the binom underlying we hardcode underlying and derivative parameters. Number of timesteps assumed to be 3 and up/down probabilities assumed identical. So we have 8 paths: UUU, UUD, UDU, UDD, DUU, DUD, DDU, DDD. Each path needs to have identical number of occurences since it has the

same probability. If we get that the underlying is working.

Listing 13: ../Src/Und/Test/test.cpp [Line 35 to 102]

```
int main(int argc, char **argv)
{ char gen_desc[10];

  if (argc == 2)
    strcpy(gen_desc, argv[1]);
  else
    strcpy(gen_desc, "mt");
  Binom binom(gen_desc);

  binom.U = 1.2;
  binom.D = 0.9;
  binom.r = 0;
  binom.T = 1.0;
  binom.N = 3;
  binom.P = 0.5;

  int S = 100;
  int M = 100000;

  /* State counters; array not used for clarity */
  int UUU = 0; int UUD = 0; int UDU = 0; int UDD = 0;
  int DUU = 0; int DUD = 0; int DDU = 0; int DDD = 0;

  cout << "Generator:  " << binom.gen->getName() << endl;

  for (int j = 1; j <= M; j++)
  { double ST = S;
    string status = "";

    for (int i = 1; i <= binom.N; i++)
    { double STnew =  binom.next(ST);
      if (ST < STnew)
        status += "U";
      else
        status += "D";
      ST = STnew;
    }

    if (status == "UUU")
      UUU++;
    else if (status == "UUD")
      UUD++;
    else if (status == "UDU")
      UDU++;
    else if (status == "UDD")
      UDD++;
```

Listing 13: ../Src/Und/Test/test.cpp [Line 35 to 102] (continued)

```
    else if (status == "DUU")
      DUU++;
    else if (status == "DUD")
      DUD++;
    else if (status == "DDU")
      DDU++;
    else if (status == "DDD")
      DDD++;
  }

  cout << "Total simulations:  " << M << endl;
  cout << "Status:  UUU, hits:  " << UUU << ", prob.  = " << (1 - binom.P) * (1 -  binom.P) * (1 -  binom.P) << er
  cout << "Status:  UUD, hits:  " << UUD << ", prob.  = " << (1 - binom.P) * (1 -  binom.P) * binom.P << endl;
  cout << "Status:  UDU, hits:  " << UDU << ", prob.  = " << (1 - binom.P) *  binom.P * (1 -  binom.P) << endl;
  cout << "Status:  UDD, hits:  " << UDD << ", prob.  = " << (1 - binom.P) *  binom.P * binom.P << endl;
  cout << "Status:  DUU, hits:  " << DUU << ", prob.  = " << binom.P * (1 -  binom.P) * (1 -  binom.P) << endl;
  cout << "Status:  DUD, hits:  " << DUD << ", prob.  = " << binom.P * (1 -  binom.P) * binom.P << endl;
  cout << "Status:  UUU, hits:  " << DDU << ", prob.  = " << binom.P * binom.P * (1 -  binom.P) << endl;
  cout << "Status:  UUU, hits:  " << DDD << ", prob.  = " << binom.P * binom.P * binom.P << endl;
  return 0;
}
```

4.3 Testing in large

System testing is algorithm testing. For more information on testing ideas refer to Requirements Documents. In test appication we hardcode all underlying parameters. The binom2 underlying is equivalent so it's never tested in that appliction. For each algorithm we try to run all derivatives available. Basic Monte Carlo can be compared with Brendan's true values. Generic Shift (and other variations) must converge to Basic Monte Carlo. Haar gives precise option values for relatively small number of simulations due to it's control variate. The user is encouraged to hardcode other underlying parameters and check the results. For a large number of simulation the processing time will be large. For large number of timesteps results might not be exact due to round off errors especially for Fast Shift and Haar algorithms.

Listing 14: ../Src/Alg/Test/test.cpp [Line 59 to 211]

```
BasicMC mc;
Algorithm *ptr_a = &mc;
mc.S = 100;
ptr_a->M = 5000;

cout << "Generator:  " << binom.gen->getName() << endl;
cout << "Algorithm used:  Basic Monte Carlo" << endl;
```

Listing 14: ../Src/Alg/Test/test.cpp [Line 59 to 211] (continued)

```
// European Call
EuropeanCall ec(100);
Derivative* ptr_der = &ec;
long start = clock();
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Call Value " << cv.back() << ", standard error " << se.back() <<  ", time = " << (clock() - start)/10000
//copy(cv.begin(), cv.end(), ostream_iterator<double>(cout, " "));
cv.clear();


// European Strike
AverageStrikeEuropeanCall es(binom.N);
ptr_der = &es;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Strike Value " << cv.back() << ", standard error " << se.back() << endl;
//copy(cv.begin(), cv.end(), ostream_iterator<double>(cout, " "));
cv.clear();

// Look back
LookBack lb;
ptr_der = &lb;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Look Back Value " << cv.back() << ", standard error " << se.back() << endl;
//copy(cv.begin(), cv.end(), ostream_iterator<double>(cout, " "));
cv.clear();

// Down Out
DownOut dout(100, 99);
ptr_der = &dout;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Down Out Value " << cv.back() << ", standard error " << se.back() << endl;
//copy(cv.begin(), cv.end(), ostream_iterator<double>(cout, " "));
cv.clear();


ExactValues ex;
ptr_a = &ex;
ex.S = 100;
cout << "Algorithm used:  Exact Values" << endl;

// European Call
ptr_der = &ec;
start = clock();
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Call Value " << cv.front() << ", standard error " << se.back() <<  ", time = " << (clock() - start)/1000
cv.clear();
```

Listing 14: ../Src/Alg/Test/test.cpp [Line 59 to 211] (continued)

```
// European Strike
ptr_der = &es;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Strike Value " << cv.front() << ", standard error " << se.back() << endl;
cv.clear();

// Look back
ptr_der = &lb;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Look Back Value " << cv.front() << ", standard error " << se.back() << endl;
cv.clear();

// Down Out
ptr_der = &dout;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Down Out Value " << cv.front() << ", standard error " << se.back() << endl;
cv.clear();



LocalMC lmc;
ptr_a = &lmc;
lmc.S = 100;
lmc.shift = 2;
ptr_a->M = 10000;

cout << "Algorithm used:  Local Monte Carlo" << endl;

// European Call
ptr_der = &ec;
start = clock();
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Call Value " << cv.back() << ", standard error " << se.back() <<  ", time = " << (clock() - start)/10000
cv.clear();

// European Strike
ptr_der = &es;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Strike Value " << cv.back() << ", standard error " << se.back() << endl;
//copy(se.begin(), se.end(), ostream_iterator<double>(cout, " "));
cv.clear();

// Look back
ptr_der = &lb;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Look Back Value " << cv.back() << ", standard error " << se.back() << endl;
cv.clear();
```

Listing 14: ../Src/Alg/Test/test.cpp [Line 59 to 211] (continued)

```
// Down Out
ptr_der = &dout;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Down Out Value " << cv.back() << ", standard error " << se.back() << endl;
cv.clear();



Continium cont(arg);
ptr_und = &cont;
cont.sigma = 0.3;

ptr_und->r = 0.0;
ptr_und->T = 1.0;
ptr_und->N = 2;
ptr_und->P = (ptr_und->U - exp(ptr_und->r * ptr_und->T / ptr_und->N)) / (ptr_und->U - ptr_und->D);
//cout << "Down prob.  is " << binom.P << endl;

ptr_a = &mc;
mc.S = 100;
ptr_a->M = 1000;

cout << "Underlying:  continium " << endl;
cout << "Algorithm used:  Basic Monte Carlo" << endl;

// European Call
ptr_der = &ec;
start = clock();
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Call Value " << cv.back() << ", standard error " << se.back() <<  ", time = " << (clock() - start)/10000
cv.clear();

// European Strike
ptr_der = &es;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Strike Value " << cv.back() << ", standard error " << se.back() << endl;
cv.clear();

// Look back
ptr_der = &lb;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Look Back Value " << cv.back() << ", standard error " << se.back() << endl;
cv.clear();

// Down Out
ptr_der = &dout;
ptr_a->calcResults(ptr_und, ptr_der, cv, se);
cout << " Down Out Value " << cv.back() << ", standard error " << se.back() << endl;
```

Listing 14: ../Src/Alg/Test/test.cpp [Line 59 to 211] (continued)

```
cv.clear();
```

5 Installation manual

1. Copy the archive file *mcRAsum02.tgz* to your account
2. Untar the application *tar xvfz mcRAsum02.tgz*
3. Copy all unpacked files to some directory on your account.

6 User manual

1. The major application is called *gop*, which stands for *generic option plotter*. It generates data files with ouput from simulations. The user has flexibility to select random generator from the command line by just passing any of the following arguments to the command line: *mt, taus, gfsr* (the default is *mt* which does not need to be explicitly typed). The underlying, derivative and algorithm are selected by prompting the user, whereas other parameters are given from the input file.
2. The *extractor* application is called directly by *gop* after all computations are complete. Up to five algorithm can be plotted simultaneously. Two windows will pop up: one withe error display, the other with option values.
3. Please be sure to give meaningful names to files as they will appear as labels to the plots. The error files will have the same name as the data files with the prefix of *err*.
4. The user might want to use extractor directly after *gop* has been called. It receives the following command line arguments (where *outputfile* will be a postcript file): Points is the number of points extractor will read from the inputfiles (starting from beginning). Recall that the file *input* (used by gop to create the output data) uses *M0*, *gap* and *steps*.So *points* is actually equals to steps, where

   ```
   M = M0 + steps * gap
   ```

   All these information is in the file *E*, create dby *gop*. Here is an example for this file:

Listing 15: E [Line 2 to 2]

```
./extractor plots.ps 100 3000 13000 bmc
```

One small issue is that we may run gop in several windows so E will contain info only in one of these files. If all windows are run with compatible information, of course the output can be plotted via extractor. Output file names from *gop* are used for plotting and also to create error files.

5. *ngop* is made by make *ngop* this program will create data in terms of N. At the moment is comparing Basic Monte Carlo on the tree and continuous. It goes from *1* to N, and N is read from *input* file. For each run (for fixed N) *M0*, *gap* and *steps* are taken from *input*.

6. The user can change plot layout by editing any of the parameters below and recompiling the *extractor*. To do so, type

   ```
   make plot
   ```

   in the command prompt. Here is some option that can be edited,

Listing 16: ../Exe/extractor.c [Line 37 to 44]

```
#define AXIS_LABELS 10            /* Number of ticks on each axis */
#define SCALE 0.9                /* Relative scale of the plot */
#define TICK_LENGTH 3            /* Size of the ticks on both axis */
#define LAYOUTS 5                                        /* Number of line formats which represent output
#define X_SIZE 600                                     /* Horizontal size of the plotting area */
#define Y_SIZE 400                                     /* Vertical size of the plotting area */
#define X_ORIGIN 150                                     /* X of lower left corner of the plotting area */
#define Y_ORIGIN 50                                     /* Y of lower left corner of the plotting area */
```

Listing 17: ../Exe/extractor.c [Line 69 to 72]

```
/* Change those values for altering plot layout */
int colors[]  = {6, 19, 4, 10, 13};            /* Colors for each graph (0..26)*/
int width[]   = {1, 1.3, 1.6, 1.9, 2.3};          /* Line width for each graph */
int dashes[]  = {2, 2, 3, 2, 1, 2, 2, 3, 5, 2};   /* Dashes length for each graph */
```

7. To use exact continium formula for the lookback choose *Continium* Underlying, and the *Exact Value Continium* algorithm.

8. Here we include a shell session when running gop

Listing 18: exampleOfRunningGop [Line 2 to 72]

```
Select Underlying
1: Binomial Tree
2: Continious
1  //user input
Select Derivative:
```

Listing 18: exampleOfRunningGop [Line 2 to 72] (continued)

```
1: European Call
2: Average Strike
3: Look Back
4: Fixed Strike
2  //user input
Select Algorithm:
1: Basic Monte Carlo
2: Exact Values on Tree
3: Local Monte Carlo
4: Local Monte Carlo with Shift
5: Exact Values Continium
1   //user input
Generator: taus
  Call Value 5.7202482, standard error 0.18364636, time = 0.03sec
Enter file name
avStBTreeBmc
Compute more options? y/n
n   //user input
//END all
```

To run the test application just change to *mcRAsum02/Src/Alg/Test* and type *./test*. A command line argument can specify random generator to be chosen from *mnt* (default), *taus* or *gfsr*. Other parameters can be changed by only recompiling the code.

7 Developers guide to maintainance

1. To add an underlying extend the abstract base class Underlying. Put the header file into *Src/Und* directory.

2. To add a derivative extend the abstract base class Derivative. Put the header file into *Src/Der* directory.

3. To add an algorithm extend the abstract base class Algorithm. Put the header and source files into *Src/Alg* directory. Add a target to a main makefile.

4. We have included a shell script to run some benchmarks. Here is the file

Listing 19: ../regs [Line 14 to 28]

```
cd Src/Und/Test
make
./undtest
./testCont
./testint
```

Listing 19: ../regs [Line 14 to 28] (continued)

```
cd ../../Rng/Test
make
make gauss
make testint
./testgen mt 0.4 1000
./testGauss
./testint mt 10 10000
cd ../../Alg/Test
make
./test mt
```

References

[1] S.E. Ferrando and A. Bernal, Localized Monte Carlo algorithm to compute prices of path dependent options on trees. Submitted (June 2004) to *International Journal of Theoretical and Applied Finance*, 21 pages.