Assignment 1

David Fleischer
MACF 402 - Mathematical & Computational Finance II

October 1, 2015

Question 1. Show that

$$u = e^{R\Delta t}(1 + \sqrt{e^{\sigma^2 \Delta t} - 1}) \tag{1}$$
$$d = e^{R\Delta t}(1 - \sqrt{e^{\sigma^2 \Delta t} - 1}) \tag{2}$$

is equivalent to

$$u = 1 + \sigma\sqrt{\Delta t} + R\Delta t + \mathcal{O}(\Delta t^{3/2})$$
$$d = 1 + \sigma\sqrt{\Delta t} - R\Delta t + \mathcal{O}(\Delta t^{3/2})$$

for constant R, σ, as $\Delta t \longrightarrow 0$

Solution. For brevity, let

$$\begin{matrix} u \\ d \end{matrix} = \sigma\sqrt{\Delta t} \pm R\Delta t + \mathcal{O}(\Delta t^{3/2})$$

denote the system

$$\begin{cases} u = 1 + \sigma\sqrt{\Delta t} + R\Delta t + \mathcal{O}(\Delta t^{3/2}) \\ d = 1 + \sigma\sqrt{\Delta t} - R\Delta t + \mathcal{O}(\Delta t^{3/2}) \end{cases}$$

First, by Taylor's Theorem, if f has a continuous derivative up to order 2 then,

$$f(x) = f(0) + xf'(0) + \frac{1}{2}x^2 f''(\xi)$$

where $\xi \in [0, x]$. Assume $x \in [-1, 1]$ so that $f''(\xi)$ is bound by some constant then,

$$f(x) = f(0) + xf'(0) + \mathcal{O}(x^2)$$

So, for $|x|$ sufficiently small,

$$e^x = 1 + x + \mathcal{O}(x^2) \tag{3}$$
$$\sqrt{1 + x} = 1 + \frac{1}{2}x + \mathcal{O}(x^2) \tag{4}$$

Next, we introduce some useful properties of $\mathcal{O}(\cdot)$. Recall that, as $x \longrightarrow 0^+$, we say $f(x)$ is of the order $g(x)$ (denoted $f(x) = \mathcal{O}(g(x))$) if $\exists c, x_0 \in \mathbb{R}^+ : 0 \leq f(x) \leq cg(x)$ for all $x \leq x_0$. So, consider $x \longrightarrow 0^+$ and $c_1, c_2, k \in \mathbb{R}^+$,

1. If $f(x) = c_1\mathcal{O}(g(x))$ then,

$$c_1\mathcal{O}(g(x)) = c_1c_2g(x) = kg(x)$$

So we have $f(x) = c_1\mathcal{O}(g(x))$ bound by $kg(x)$. That is,

$$f(x) = c_1\mathcal{O}(g(x)) = \mathcal{O}(g(x))$$

2. If $f(x) = g_1(x)\mathcal{O}(g_2(x))$ then,

$$g_1(x)\mathcal{O}(g_2(x)) \leq g_1(x)kg_2(x) = kg_1(x)g_2(x)$$

So we have $g_1(x)\mathcal{O}(g_2(x))$ bound by $kg_1(x)g_2(x)$, so it is of the order $g_1(x)g_2(x)$. That is,

$$f(x) = g_1(x)\mathcal{O}(g_2(x)) = \mathcal{O}(g_1(x)g_2(x))$$

3. If $f(x) = \mathcal{O}(\mathcal{O}(x)^{n/m})$ then,

$$\begin{aligned}\mathcal{O}(\mathcal{O}(x)^{n/m}) &\leq c_1\mathcal{O}(x)^{n/m} \\ &\leq c_1(c_2x)^{n/m} = c_1c_2^{n/m}x^{n/m} = kx^{n/m}\end{aligned}$$

So we have $\mathcal{O}(\mathcal{O}(x)^{n/m})$ bound by $kx^{n/m}$, so it is of the order $x^{n/m}$. That is,

$$f(x) = \mathcal{O}(\mathcal{O}(x)^{n/m}) = \mathcal{O}(x^{n/m})$$

4. If $f_1(x) = \mathcal{O}(g_1(x))$ and $f_2(x) = \mathcal{O}(g_2(x))$ then,

$$\begin{aligned}\mathcal{O}(g_1(x)) + \mathcal{O}(g_2(x)) &\leq c_1g_1(x) + c_2g_2(x) \\ &\leq max(c_1, c_2)(g_1(x) + g_2(x)) = k(g_1(x) + g_2(x))\end{aligned}$$

So we have $\mathcal{O}(g_1(x)) + \mathcal{O}(g_2(x))$ bound by $k(g_1(x) + g_2(x))$, so it is of the order $g_1(x) + g_2(x)$. That is,

$$f(x) = \mathcal{O}(g_1(x)) + \mathcal{O}(g_2(x)) = \mathcal{O}(g_1(x) + g_2(x))$$

Now that we've set up our playing field we may evaluate the terms $e^{R\Delta t}$ and $e^{\sigma^2\Delta t}$ in (1) & (2) at $x = R\Delta t$ and $x = \sigma^2\Delta t$, respectively. This leaves us with,

$$\begin{aligned}{}^u_d &= (1 + R\Delta t + \mathcal{O}(R^2\Delta t^2))(1 \pm \sqrt{1 + \sigma^2\Delta t + \mathcal{O}(\sigma^4\Delta t^2) - 1}) \\ &= (1 + R\Delta t + \mathcal{O}(R^2\Delta t^2))(1 \pm \sqrt{\sigma^2\Delta t + \mathcal{O}(\sigma^4\Delta t^2)})\end{aligned}$$

Using the properties of $\mathcal{O}(\cdot)$ introduced above, for $\Delta t \longrightarrow 0^+$,

$$\begin{aligned}
\tfrac{u}{d} &= (1 + R\Delta t + \mathcal{O}(R^2\Delta t^2))(1 \pm \sqrt{\sigma^2\Delta t + \mathcal{O}(\sigma^4\Delta t^2)}) \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sqrt{\sigma^2\Delta t + \mathcal{O}(\Delta t^2)}) \quad \text{(by Property 1)} \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sqrt{\sigma^2\Delta t + \Delta t\mathcal{O}(\Delta t)}) \quad \text{(by Property 2)} \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sigma\sqrt{\Delta t}\sqrt{1 + \frac{1}{\sigma^2}\mathcal{O}(\Delta t)}) \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sigma\sqrt{\Delta t}\sqrt{1 + \mathcal{O}(\Delta t)}) \quad \text{(by Property 1)}
\end{aligned}$$

Evaluating (4) at $x = \mathcal{O}(\Delta t)$ we have,

$$\begin{aligned}
\tfrac{u}{d} &= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sigma\sqrt{\Delta t}(1 + \frac{1}{2}\mathcal{O}(\Delta t) + \mathcal{O}(\mathcal{O}(\Delta t)^2)) \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sigma\sqrt{\Delta t}(1 + \mathcal{O}(\Delta t) + \mathcal{O}(\mathcal{O}(\Delta t)^2)) \quad \text{(by Property 1)} \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sigma\sqrt{\Delta t}(1 + \mathcal{O}(\Delta t) + \mathcal{O}(\Delta t^2)) \quad \text{(by Property 3)} \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sigma\sqrt{\Delta t}(1 + \mathcal{O}(\Delta t + \Delta t^2)) \quad \text{(by Property 4)}
\end{aligned}$$

We see that $\mathcal{O}(\Delta t + \Delta t^2) = \mathcal{O}(\Delta t)$ (as $\Delta t \longrightarrow 0^+$) since we may find some positive constant k such that $\Delta t + \Delta t^2 \leq k\Delta t$. So,

$$\begin{aligned}
\tfrac{u}{d} &= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sigma\sqrt{\Delta t}(1 + \mathcal{O}(\Delta t)) \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2))(1 \pm \sigma\sqrt{\Delta t} \pm \sigma\sqrt{\Delta t}\mathcal{O}(\Delta t)) \\
&= (1 + R\Delta t + \mathcal{O}(\Delta t^2)(1 \pm \sigma\sqrt{\Delta t} + \mathcal{O}(\Delta t^{3/2})) \quad \text{(by Properties 1 \& 2)} \\
&= 1 + R\Delta t + \mathcal{O}(\Delta t^2) \pm \sigma\sqrt{\Delta t} \pm R\Delta t\sigma\sqrt{\Delta t} \pm \mathcal{O}(\Delta t^2)\sigma\sqrt{\Delta t} \\
&\qquad + \mathcal{O}(\Delta t^{3/2}) + R\Delta t\mathcal{O}(\Delta t^{3/2}) + \mathcal{O}(\Delta t^2)\mathcal{O}(\Delta t^{3/2}) \\
&= 1 + R\Delta t + \mathcal{O}(\Delta t^2) \pm \sigma\sqrt{\Delta t} + \mathcal{O}(\Delta t^{3/2}) + \mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta t^{3/2}) + \mathcal{O}(\Delta t^{3/2}) + \mathcal{O}(\Delta t^2)\mathcal{O}(\Delta t^{3/2}) \\
&\qquad \text{(by Properties 1 \& 2 and realizing that } \Delta t\sqrt{\Delta t} = \Delta t^{3/2} = \mathcal{O}(\Delta t^{3/2}))
\end{aligned}$$

Collecting like terms & applying Property 1 we get,

$$\tfrac{u}{d} = 1 + R\Delta t \pm \sigma\sqrt{\Delta t} + \mathcal{O}(\Delta t^{3/2}) + \mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta t^2)\mathcal{O}(\Delta t^{3/2})$$

Note,

$$\begin{aligned}
\mathcal{O}(\Delta t^2)\mathcal{O}(\Delta t^{3/2}) &= \mathcal{O}(\mathcal{O}(\Delta t^2)\Delta t^{3/2}) \quad \text{(by Property 2)} \\
&= \mathcal{O}(\mathcal{O}(\Delta t^2\Delta t^{3/2}) = \mathcal{O}(\mathcal{O}(\Delta t^{7/2})) \quad \text{(by Property 2)} \\
&= \mathcal{O}(\Delta t^{7/2}) \quad \text{(by Property 3)}
\end{aligned}$$

So, we are left with

$$\begin{aligned}
\tfrac{u}{d} &= 1 + R\Delta t \pm \sigma\sqrt{\Delta t} + \mathcal{O}(\Delta t^{3/2}) + \mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta t^{7/2}) \\
&= 1 + R\Delta t \pm \sigma\sqrt{\Delta t} + \mathcal{O}(\Delta t^{3/2} + \Delta t^2 + \Delta t^{7/2}) \quad \text{(by Property 4)}
\end{aligned}$$

Similar to above, we see that $\mathcal{O}(\Delta t^{3/2} + \Delta t^2 + \Delta t^{7/2}) = \mathcal{O}(\Delta t^{3/2})$, as $\Delta t \longrightarrow 0^+$, since we may find some positive constant k such that $\Delta t^{3/2} + \Delta t^2 + \Delta t^{7/2} \leq k\Delta t^{3/2}$. So,

$$u = 1 + R\Delta t + \sigma\sqrt{\Delta t} + \mathcal{O}(\Delta t^{3/2})$$
$$d = 1 + R\Delta t - \sigma\sqrt{\Delta t} + \mathcal{O}(\Delta t^{3/2})$$

as desired.

Question 2.

(a) Write a program implementing the N-step binomial model given some asset in order to price an option V_0^N.

(b) Write a program that applies the bisection algorithm to observed option prices in order to back out market volatility $\sigma^{implied}$.

(c) Apply the programs from (a)-(b) to given price data. Comment on convergence of implied volatility estimates for increasing values of N-steps.

(d) Plot the data from (c) as a function of *out-of-the-money* options for the largest value of N calculated. Is there a 'volatility smile'?.

Solution. Code output & code provided in a appendices A and B, respectively.

Implied volatility estimates given by the bisection algorithm appear to converge relatively rapidly. This is illustrated graphically in Figures 1 and 2. By $N = 10^2$ ($\Delta t = 14.16$ hours) the algorithm appears to give reasonably stable estimates and going beyond $N = 10^3$ yields little appreciable precision gains. For example, going from $N = 10^3$ to $N = 10^4$ yields additional accuracy of at most ± 0.0001. In fact, we see that all but three volatility estimates computed for $N = 10^5$ are identical up to six places to those computed for $N \geq 10^6$, with the three exceptions being identical up to five places (puts with strikes \$4.75, \$5.00, \$5.25).

Figure 3 informs us that the topic of increasing values of N is of some importance when considering run time. For $N \leq 10^3$ we compute volatility estimates with the help of an iterative algorithm for the binomial coefficient. The iteration (in N) for the binomial coefficient is compounded with the iteration (in N) required for option valuation in a binomial tree: A net run time proportional to N^2. For $N \geq 10^4$ we compute volatility estimates with the help of an approximation algorithm for the binomial coefficient instead of the iterative solution. In doing so we remove one iterative process leading to a run time proportional to N. Since a fairly convergent estimate of implied volatility can be obtained by $N = 10^4$, which can be computed in less than one second[1], going beyond these values of N may not be a good use of time.

[1] As run on a MacBook Pro (Late 2013) with 2.4GHz dual-core Intel Core i5 processor.

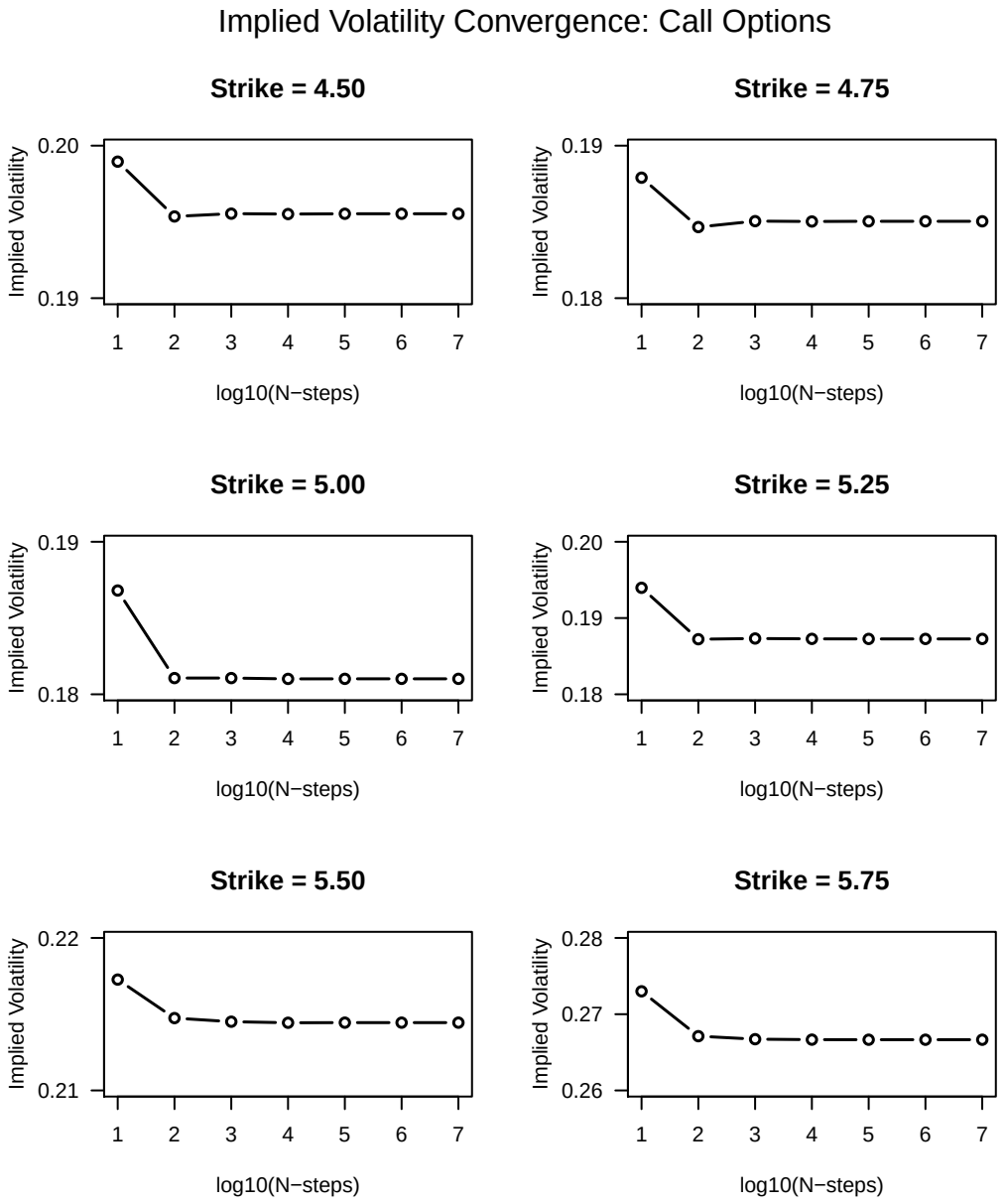

Figure 1: Implied volatility estimates given call option strike prices as a function of the N-steps of the binomial model. Qualitatively, convergence appears to occur relatively rapidly: By $N = 10^3$ little gains to volatility precision occurs.

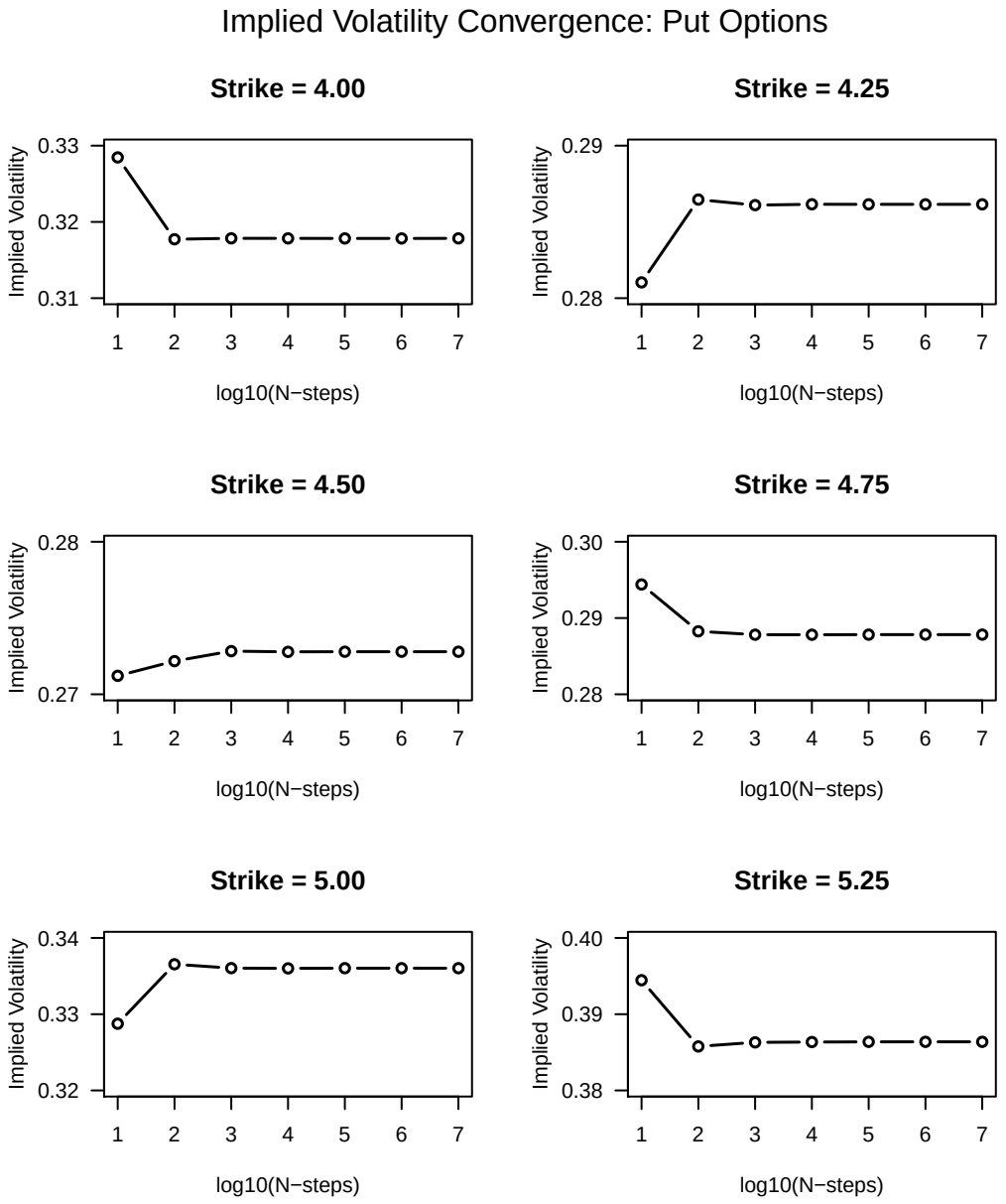

Figure 2: Implied volatility estimates given put option strike prices as a function of the N-steps of the binomial model. Similar to the volatility estimates of call options, convergence for puts appears by approximately $N = 10^3$.

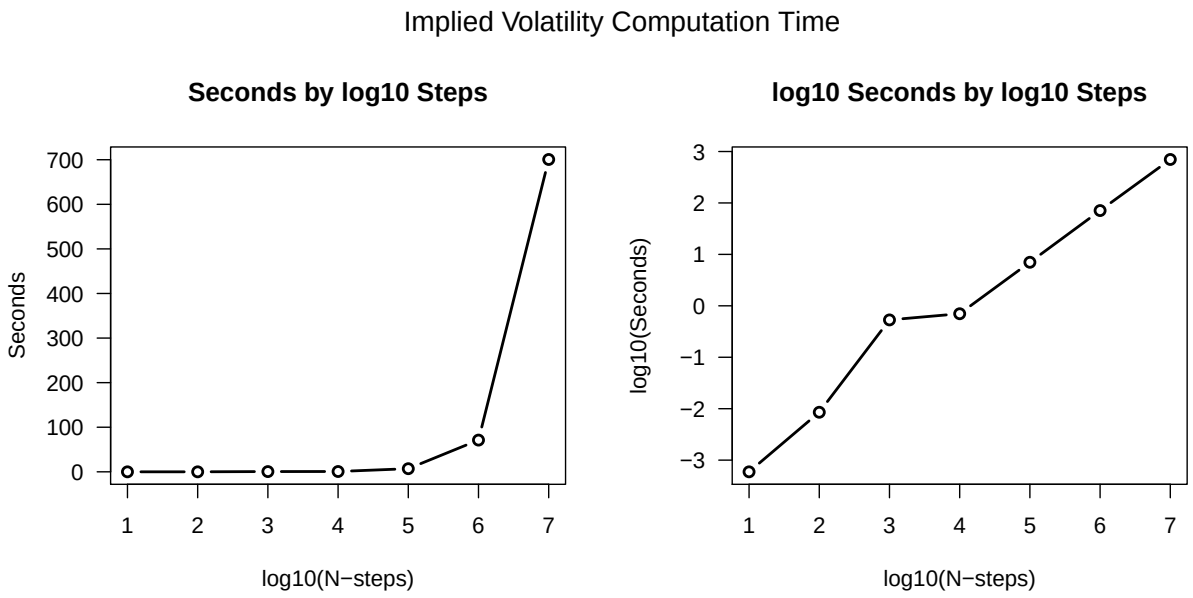

Figure 3: Run time as a function of the number of steps of the N-step binomial model. Given this particular algorithm we see nonlinear growth in time for $N \leq 10^3$ and linear growth in time for $N \geq 10^4$ when computing estimates of implied volatility.

Figure 4 gives us a visualization of the volatility estimates as a function of strike price. From this we immediately see the limitations of the binomial & Black-Scholes models. These models assume that volatility is constant across strike prices and that volatility is independent of call and put contracts for a constant strike. However we see that neither of these assumptions hold for our computed implied volatilities. Not only is there the clear volatility smile, but we also see that put volatility estimates are across-the-board higher than the corresponding call estimates. A smile can be seen when considering only call or put contracts, but if constructing the smile from only out-of-the-money calls/puts then we see a discontinuity in the smile when the strike is contract is at-the-money.

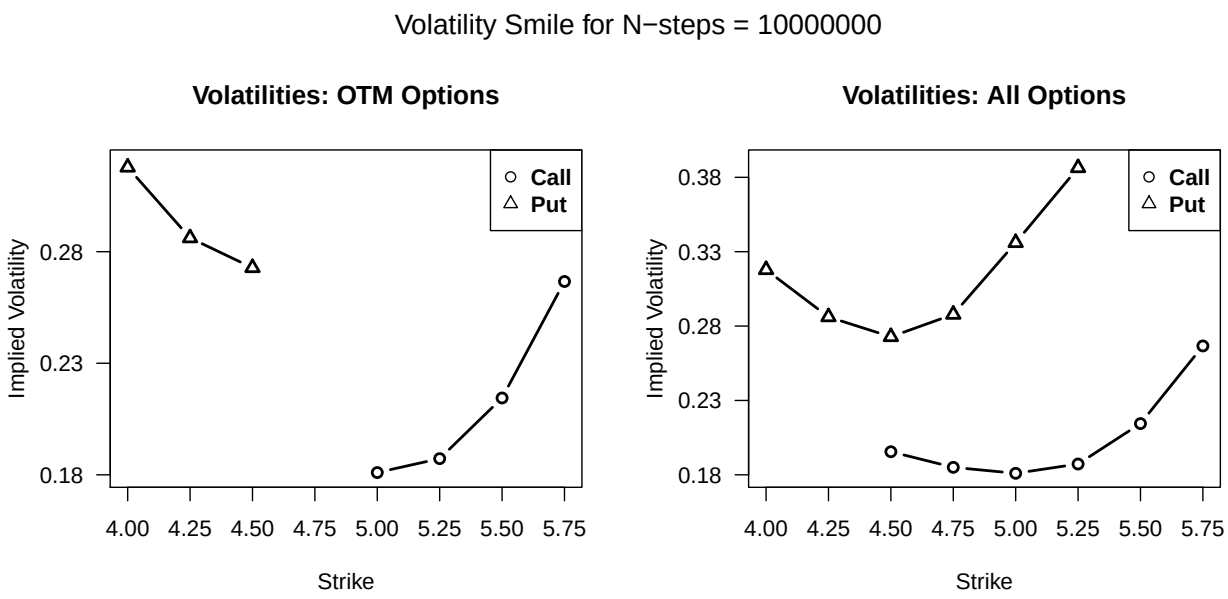

Figure 4: Volatility estimates from the bisection algorithm of the binomial model plotted as a function of strike price. Left: Volatilities for out-of-the-money options. Right: Volatilities for all options. At the strike price corresponding to at-the-money options ($4.75) note the discontinuity in implied volatility when considering only the out-of-the-money calls & puts.

Appendix A Code Output: Data Table

N	strike	moneyness	call	put
10	4	0.842105	nan	0.328461
10	4.25	0.894737	nan	0.281036
10	4.5	0.947368	0.198952	0.271212
10	4.75	1	0.187901	0.294403
10	5	1.05263	0.186806	0.328753
10	5.25	1.10526	0.19397	0.394455
10	5.5	1.15789	0.21727	nan
10	5.75	1.21053	0.273003	nan
100	4	0.842105	nan	0.317726
100	4.25	0.894737	nan	0.286465
100	4.5	0.947368	0.195358	0.272175
100	4.75	1	0.184662	0.288261
100	5	1.05263	0.181065	0.336548
100	5.25	1.10526	0.187225	0.385773
100	5.5	1.15789	0.214745	nan
100	5.75	1.21053	0.26712	nan
1000	4	0.842105	nan	0.317848
1000	4.25	0.894737	nan	0.286102
1000	4.5	0.947368	0.195541	0.272831
1000	4.75	1	0.185051	0.287819
1000	5	1.05263	0.181065	0.336042
1000	5.25	1.10526	0.187317	0.386307
1000	5.5	1.15789	0.214508	nan
1000	5.75	1.21053	0.266724	nan
10000	4	0.842105	nan	0.317841
10000	4.25	0.894737	nan	0.286156
10000	4.5	0.947368	0.195515	0.272785
10000	4.75	1	0.185028	0.287815
10000	5	1.05263	0.181011	0.336012
10000	5.25	1.10526	0.187271	0.386349
10000	5.5	1.15789	0.214439	nan
10000	5.75	1.21053	0.266663	nan
100000	4	0.842105	nan	0.317833
100000	4.25	0.894737	nan	0.286152
100000	4.5	0.947368	0.195534	0.272793
100000	4.75	1	0.18504	0.287827
100000	5	1.05263	0.181015	0.336031
100000	5.25	1.10526	0.187263	0.386375
100000	5.5	1.15789	0.214447	nan
100000	5.75	1.21053	0.266655	nan
1000000	4	0.842105	nan	0.317833
1000000	4.25	0.894737	nan	0.286152
1000000	4.5	0.947368	0.195534	0.272793
1000000	4.75	1	0.18504	0.287828

N	strike	moneyness	call	put
1000000	5	1.05263	0.181015	0.336033
1000000	5.25	1.10526	0.187263	0.386377
1000000	5.5	1.15789	0.214447	nan
1000000	5.75	1.21053	0.266655	nan
10000000	4	0.842105	nan	0.317841
10000000	4.25	0.894737	nan	0.286152
10000000	4.5	0.947368	0.195534	0.272793
10000000	4.75	1	0.18504	0.287828
10000000	5	1.05263	0.181015	0.336033
10000000	5.25	1.10526	0.187263	0.386377
10000000	5.5	1.15789	0.214447	nan
10000000	5.75	1.21053	0.266655	nan

Appendix B Code

B.1 Main.cpp

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <ctime> // calculate time between dates
#include <time.h> // time code
#include "Asset.h"
#include "BinomialModel.h"
#include "CallOption.h"
#include "PutOption.h"
#include "MathHelp.h"

int main() {
//    double ask = 0.33;
//    double S0 = 4.75;
//    double strike = 4.5;
//    double R = 0.0492;
//    double tau = 0.161;
//    int N = 100;
//
//    Asset A(S0, R);
//    CallOption C(A, strike, tau);
//    CallOption* CO = &C;
//
//    std::cout << BinomialModel::impliedVol(ask, CO, N) << std::endl;
//    //std::cout << BinomialModel::optionValuation(CO, N) << std::endl;
//    //std::cout << BinomialModel::assetPrice(A, tau, 5, N) << std::endl;

   double S0 = 4.75; // market spot price
   double R = 0.0492; // con't comp. interest rate
   // steps for the binomial tree calculations
   int N[] = {pow(10,1),pow(10,2),pow(10,3),pow(10,4),pow(10,5),pow(10,6),pow(10,7)};

   // specify the option chain strikes & asks
   double strikes[8] = {4,4.25,4.5,4.75,5,5.25,5.5,5.75};
   double call_asks[8] = {std::nan("1"),std::nan("1"),0.33,0.16,0.06,0.02,0.01,0.01};
   double put_asks[8] = {0.02,0.04,0.09,0.20,0.38,0.59,std::nan("1"),std::nan("1")};

   // compute tau from date endpoints
   struct std::tm date_a = {0,0,0,29,6,102}; // July 29th 2002
   struct std::tm date_b = {0,0,0,26,8,102}; // Sept 26th 2002
   std::time_t start_date = std::mktime(&date_a);
   std::time_t end_date= std::mktime(&date_b);
   double tau = std::difftime(end_date, start_date)/(60 * 60 * 24 * 365);

   Asset A(S0, R); // create our underlying asset

   // prepare to write data to csv
   std::ofstream outfile_vols, outfile_times;
   outfile_vols.open("../data/implied_vols.csv"); // initialize csv
```

```
   outfile_times.open("../data/times.csv"); // initialize csv

   // write csv columns & write data
   outfile_vols << "N,strike,moneyness,call,put" << std::endl;

   // display computation time for each N while looping
   std::cout << "N\t\tseconds" << std::endl;
   outfile_times << "N,seconds" << std::endl;

   for (int j = 0; j < (sizeof(N)/sizeof(N[0])); j++) { // loop over values of N

      clock_t t1, t2;
      t1 = clock();

      // loop over strike & ask values & compute implied volatility estimates
      for (int i = 0; i < sizeof(strikes)/sizeof(strikes[0]); i++) {

         CallOption C(A, strikes[i], tau);
         PutOption P(A, strikes[i], tau);

         CallOption* CO = &C;
         PutOption* PO = &P;

         outfile_vols << N[j] << "," << strikes[i] << "," << strikes[i]/S0 << ","
         << BinomialModel::impliedVol(call_asks[i],CO,N[j]) << ","
         << BinomialModel::impliedVol(put_asks[i],PO,N[j]) << std::endl;
      }

      t2 = clock();
      float diff ((float)t2 - (float)t1);
      std::cout << "10^" << log10(N[j]) << "\t\t" << diff/CLOCKS_PER_SEC << std::endl;
      outfile_times << N[j] << "," << diff/CLOCKS_PER_SEC << std::endl;
   }

   outfile_vols.close();
   outfile_times.close();
}
```

B.2 Asset.h

```
/*
   Header file for an Asset
   An asset is an object with a spot, interest rate, and (optional) volatility
*/
#ifndef ASSET_H
#define ASSET_H
class Asset {
   private:
      double spot; // asset spot value
      double R; // cont. comp. interest rate
      double sigma; // volatility parameter
```

```
   public:
      Asset(); // default constructor
      Asset(double spot_in, double R_in);
      Asset(double spot_in, double R_in, double sigma_in);
      void setSpot(double spot_in);
      double getSpot();
      void setR(double R_in);
      double getR();
      void setSigma(double sigma_in);
      double getSigma();
};
#endif
```

B.3 Asset.cpp

```
#include "Asset.h"

Asset::Asset() {}
Asset::Asset(double spot_in, double R_in) {
   spot = spot_in;
   R = R_in;
}
Asset::Asset(double spot_in, double R_in, double sigma_in) {
   spot = spot_in;
   R = R_in;
   sigma = sigma_in;
}
void Asset::setSpot(double spot_in) {
   spot = spot_in;
}
double Asset::getSpot() {
   return spot;
}
void Asset::setR(double R_in) {
   R = R_in;
}
double Asset::getR() {
   return R;
}
void Asset::setSigma(double sigma_in) {
   sigma = sigma_in;
}
double Asset::getSigma() {
   return sigma;
}
```

B.4 BinomialModel.h

```
/*
   Header file for BinomialModel
   A binomial model is an operation (or set of operations) on an object and
   thus is a static class that cannot be instantiated
*/
#include "Asset.h"
#include "Option.h"
#include "MathHelp.h"
#include <cmath>

#ifndef BINOMIALMODEL_H
#define BINOMIALMODEL_H
class BinomialModel {
   public:
      static double p; // risk neutral prob. of up state
      /*
         Are factors really static attributes of BinomialModels?
         On the one hand it doesn't make sense to have BinomialModel be instantiated,
         on the other hand up and down factors will vary across assets.
         Furthermore, up and down factors clearly should not be attributes of an Asset
      */
      static double u; // up factor
      static double d; // down factor

      // price asset given n up steps given N total steps
      static double assetPrice(Asset A, double tau, int n, int N);
      // value option back to time = 0 given N step binomial model
      static double optionValuation(Option* O, int N);
      // back out implied vol. estimates via bisection algorithm
      static double impliedVol(double V_obs, Option* O, int N, double eps = pow(10,-6));

};
#endif
```

B.5 BinomialModel.cpp

```
#include "BinomialModel.h"

double BinomialModel::p = 0.5;
double BinomialModel::u = 1;
double BinomialModel::d = 1;

double BinomialModel::assetPrice(Asset A, double tau, int n, int N) {
   // compute future asset price at time tau given
   // n up steps in a N steps in a binomial tree in tau time
   if (N <= n) return A.getSpot();

   double sigma = A.getSigma();
   double R = A.getR();
   double dt = tau/N; // time step interval
```

```
   // compute up & down factors
   u = exp(sigma * sqrt(dt) + (R - 0.5 * pow(sigma, 2)) * dt);
   d = exp(-sigma * sqrt(dt) + (R - 0.5 * pow(sigma, 2)) * dt);

   // Sn = S0 * u^n * d^(N-n)
   double S_N = A.getSpot() * pow(u, n) * pow(d, N - n);
   return S_N;
}
double BinomialModel::optionValuation(Option* O, int N) {
   // compute option value at time 0 given N steps in a binomial tree

   // we rely heavily on pointers since Option O will either contain s
   // CallOption or a PutOption and we wish to use their function implementations
   // so that the code may remain general as possible
   // (also, I'm also not very good at C++)

   Asset A = O->getAsset();
   double R = A.getR();
   double tau = O->getTau();
   double strike = O->getStrike();

   double S_N, V_N;
   double V_0 = 0;

   if (N <= 1000) { // if N < 1000 we can use an exact algorithm for the binomial coef.
      for (int k = 0; k <= N; k++) {
         // compute asset value at time N given k up steps
         S_N = assetPrice(A, tau, k, N);
         V_N = O->payoff(S_N);

         V_0 += MathHelp::bin_coef(N, k) * pow(p, k) * pow(1 - p, N - k) * V_N;
      }
   }
   else { // if N too large it then we get into data type representation issues
         // so we've got to do some trickery
      for (int k = 0; k <= N; k++) {
         // compute asset value at time N given k up steps
         S_N = assetPrice(A, tau, k, N);
         V_N = O->payoff(S_N); // option payoff

         // compute partial sum of the value V_0 of our option
         // exploit the fact that x*y*z = exp(log(x) + log(y)) * z
         // to avoid representation issues
         V_0 += exp(MathHelp::bin_coef(N, k) + k * log(p) + (N - k) * log (1 - p)) * V_N;
      }
   }
   return exp(-R * tau) * V_0; // multiply by discount factor
}
double BinomialModel::impliedVol(double V_obs, Option* O, int N, double eps) {
   // bisection algorithm to back out implied volatility estimate
   // given observed market prices V_obs, Option O, N steps in a binomial model
   // and some tolerance parameter eps

   if (V_obs != V_obs) { // exploit the fact that nan is not equal to itself
```

```
    // this exists just for simplicity when handling options without asks
    return std::nan("1");
  }

  Asset A = O->getAsset();
  double spot = A.getSpot();
  double R = A.getR();

  double a = 0; // vol can never be < 0
  double b = 1; // what is a reasonable upper limit?
  double sigma_guess = (a + b)/2; // bisect!

  O->setAsset(Asset(spot, R, sigma_guess));
  double V_guess = optionValuation(O, N);

  while (std::abs(V_guess - V_obs) >= eps) {
    if (V_guess > V_obs) { // guessed price too high, guessed vol too high
      b = sigma_guess; // shrink right endpoint
      sigma_guess = (a + b)/2; // bisect!
    }
    else { // guessed price too low, guessed vol too low
      a = sigma_guess; // shrink left endpoint
      sigma_guess = (a + b)/2; // bisect!
    }

    O->setAsset(Asset(spot, R, sigma_guess)); // change value of sigma in underlying
        asset
    V_guess = optionValuation(O, N); // recompute our guess
  }
  return sigma_guess;
}
```

B.6 Option.h

```
/*
  Header file for an Option
  An option is an object that contains an asset, a strike, and a time to maturity
  Maybe it makes sense to implement this as an abstract class?
  It doesn't make sense to be able to instantiate a general 'Option'
*/
#include "Asset.h"

#ifndef OPTION_H
#define OPTION_H
class Option {
  protected:
    Asset A; // underlying asset
    double strike; // strike price
    double tau; // time to expiry

  public:
```

```
    Option(); // default constructor
    Option(Asset A_in, double strike_in, double tau_in);
    void setAsset(Asset A_in);
    Asset getAsset();
    void setStrike(double strike_in);
    double getStrike();
    void setTau(double tau_in);
    double getTau();

    virtual double payoff() = 0;
    virtual double payoff(double some_strike) = 0;
};
#endif
```

B.7 Option.cpp

```
#include "Option.h"

Option::Option() {}
Option::Option(Asset A_in, double strike_in, double tau_in) {
   A = A_in;
   strike = strike_in;
   tau = tau_in;
}
void Option::setAsset(Asset A_in) {
   A = A_in;
}
Asset Option::getAsset() {
   return A;
}
void Option::setStrike(double strike_in) {
   strike = strike_in;
}
double Option::getStrike() {
   return strike;
}
void Option::setTau(double tau_in) {
   tau = tau_in;
}
double Option::getTau() {
   return tau;
}
```

B.8 CallOption.h

```
/*
   Header file for a CallOption
   A call option is a child of an option that contains a specific implementation
   of a payoff function
```

```
*/
#include "Option.h"

#ifndef CALLOPTION_H
#define CALLOPTION_H
class CallOption: public Option {
   public:
      CallOption(); // default constructor
      CallOption(Asset A_in, double strike_in, double tau_in);

      double payoff();
      double payoff(double some_spot);
};
#endif
```

B.9 CallOption.cpp

```
#include "CallOption.h"
#include <cmath>

CallOption::CallOption() {}
CallOption::CallOption(Asset A_in, double strike_in, double tau_in)
   :Option(A_in, strike_in, tau_in){} // call the parent constructor
double CallOption::payoff() {
   return fmax(A.getSpot() - strike, 0);
}
double CallOption::payoff(double some_spot) {
   return fmax(some_spot - strike, 0);
}
```

B.10 PutOption.h

```
/*
   Header file for a PutOption
   A put option is a child of an option that contains a specific implementation
   of a payoff function
*/
#include "Option.h"

#ifndef PUTOPTION_H
#define PUTOPTION_H
class PutOption: public Option {
   public:
      PutOption(); // default constructor
      PutOption(Asset A_in, double strike_in, double tau_in);

      double payoff();
      double payoff(double some_spot);
};
```

```
#endif
```

B.11 PutOption.cpp

```
#include "PutOption.h"
#include <cmath>

PutOption::PutOption() {}
PutOption::PutOption(Asset A_in, double strike_in, double tau_in)
   :Option(A_in, strike_in, tau_in){} // call the parent constructor
double PutOption::payoff() {
   return fmax(strike - A.getSpot(), 0);
}
double PutOption::payoff(double some_spot) {
   return fmax(strike - some_spot, 0);
}
```

B.12 MathHelp.h

```
/*
   Header file for MathHelp
   Contains some useful functions for our purposes
*/
#ifndef MATHHELP_H
#define MATHHELP_H
class MathHelp {
   public:
      static double log_stir(int n);
      static double bin_coef(int n, int k);
};
#endif
```

B.13 MathHelp.cpp

```
#include "MathHelp.h"
#include <cmath>

double MathHelp::log_stir(int n) { // use log(stirling's approx) to get a value of n!
    for large n
   if (n == 0) return 0; // we want Choose(N,0) = Choose(N,0) = 1
      // to do this requires 0! = 1, but we pass to log_fact the param. 0 in these border
          cases.
      // we know that log(0) is undefined, but we want exp(log(x)) = 1
      // so lets pretend log(0) = 0 since we later exp. it as exp(0) = 1
   return log(sqrt(2 * M_PI * n)) + n * (log(n) - 1);
}
double MathHelp::bin_coef(int n, int k) {
```

```
   if (k > n) return 0; // this should never be called in our case

   if (n <= 1000) { // use this implementation for modest n in the bin. coef. Choose(n,k)
      double r = 1; // result

      // since bin. coef. is symmetric, consider the half with small k for fewer
          iterations
      if (k > n - k) k = n - k;

      for (int i = 1; i <= k; i++) {
         r *= (n - i + 1)/((double)i);
      }
      return r;
   }
   else { // use this implementation for large n in the bin. coef. Choose(n,k);
        // apply log properties to log(n!/(k!(n-k)!)
      return (log_stir(n) - log_stir(k) - log_stir(n - k));
   }
}
```

B.14 makeplots.R

```
##
## Script for plotting the implied volatilities output by the corresponding C++ program
## Most of this trash is written for aesthetic purposes
##

setwd("~/drive/concordia/2015-2016/1_fall2015/macf402/assignment1/")
implied_vols <- read.csv("./data/implied_vols.csv", stringsAsFactors = F)
times <- read.csv("./data/times.csv", stringsAsFactors = F)

## get volatility data only for the largest N
maxN_callvols <- implied_vols$call[implied_vols$N == max(implied_vols$N)]
maxN_putvols <- implied_vols$put[implied_vols$N == max(implied_vols$N)]

## begin plotting smile
pdf("./plots/smile.pdf", height = 4.75, width = 9.5)
par(mfrow = c(1,2), oma = c(0, 0, 2, 0)) #
## OTM OPTIONS: plot volatility smile for the largest N computed
otm_callvols <- maxN_callvols
otm_callvols[which(unique(implied_vols$moneyness) <= 1)] <- NaN
otm_putvols <- maxN_putvols
otm_putvols[which(unique(implied_vols$moneyness) >= 1)] <- NaN
# endpoints for y axis, consider the highest/lowest volatilities & round them up/down
lo <- floor(min(otm_callvols, otm_putvols, na.rm = T)*100)/100
hi <- ceiling(max(otm_callvols, otm_putvols, na.rm = T)*100)/100

plot(otm_callvols ~ unique(implied_vols$strike),
     lwd = 2, type = 'b', ylim = c(lo, hi),
     xlab = "Strike", ylab = "Implied Volatility", main = "Volatilities: OTM Options",
     xaxt = "n", yaxt = "n") # xaxt = 'n', yaxt = 'n', suppress drawing axis values
```

```
axis(2, at = seq(lo, hi, 0.05), las = 1) # draw y axis ticks from lo to hi every 0.05
axis(1, at = unique(implied_vols$strike), las = 1) # draw y axis ticks from lo to hi
    every 0.05
lines(otm_putvols ~ unique(implied_vols$strike), lwd = 2, pch = 24, type = 'b', yaxt =
    "n")
legend("topright", legend = c("Call","Put"), pch = c(21,24), text.font = 2)

## plot ALL strikes regardless of moneyness
# endpoints for y axis, consider the highest and lowest volatilities
lo <- floor(min(maxN_callvols, maxN_putvols, na.rm = T)*100)/100
hi <- ceiling(max(maxN_callvols, maxN_putvols, na.rm = T)*100)/100

## ALL OPTIONS: smile for largest N
plot(maxN_callvols ~ unique(implied_vols$strike),
     lwd = 2, type = 'b', ylim = c(lo, hi),
     xlab = "Strike", ylab = "Implied Volatility", main = "Volatilities: All Options",
     xaxt = "n", yaxt = "n") # xaxt = 'n', yaxt = 'n', suppress drawing axis values
axis(2, at = seq(lo, hi, 0.05), las = 1) # draw y axis ticks from lo to hi every 0.05
axis(1, at = unique(implied_vols$strike), las = 1) # draw y axis ticks from lo to hi
    every 0.05
lines(maxN_putvols ~ unique(implied_vols$strike), lwd = 2, pch = 24, type = 'b', yaxt =
    "n")
legend("topright", legend = c("Call","Put"), pch = c(21,24), text.font = 2)
mtext(paste0("Volatility Smile for N-steps = ", max(implied_vols$N)), outer = T, cex =
    1.25)
dev.off()

## calls
## plot convergence of implied volatility for increasing N
pdf("./plots/call_convergence.pdf", height = 6.5, width = 5.5)
# get unique strike values of observed call options
call_strikes <- unique(implied_vols$strike[!is.nan(implied_vols$call)])
par(mfrow = c(3,2), oma = c(0, 0, 2, 0)) # create a frame with 3x2 boxes

for (i in 1:length(call_strikes)) { # loop over all strike prices
  vol_subset <- implied_vols[implied_vols$strike == call_strikes[i],]
  lo <- floor(min(vol_subset$call)*100)/100 # round y lower axis endpoints
  hi <- ceiling(max(vol_subset$call)*100)/100 # round y upper axis endpoints

  plot(vol_subset$call ~ log10(unique(implied_vols$N)),
       type = 'b', ylim = c(lo, hi),
       xlab = 'log10(N-steps)', ylab = 'Implied Volatility',
       main = paste0("Strike = ",sprintf("%.2f",call_strikes[i])),
       yaxt = "n", lwd = 1.5) # yaxt = 'n' suppress drawing the y axis values
  axis(2, at = seq(lo, hi, by = 0.01), las = 1) # draw y axis ticks from lo to hi every
      0.01

}
mtext("Implied Volatility Convergence: Call Options", outer = T, cex = 1)
dev.off()

## puts
## plot convergence of implied volatility for increasing N
pdf("./plots/put_convergence.pdf", height = 6.5, width = 5.5)
```

```
# get unique strike values of observed put options
put_strikes <- unique(implied_vols$strike[!is.nan(implied_vols$put)])
par(mfrow = c(3,2), oma = c(0, 0, 2, 0)) # create a frame with 3x2 boxes

for (i in 1:length(put_strikes)) { # loop over all strike prices
  vol_subset <- implied_vols[implied_vols$strike == put_strikes[i],]
  lo <- floor(min(vol_subset$put)*100)/100 # round y axis lower endpoint
  hi <- ceiling(max(vol_subset$put)*100)/100 # round y axis upper endpoint

  plot(vol_subset$put ~ log10(unique(implied_vols$N)),
       type = 'b', ylim = c(lo, hi),
       xlab = 'log10(N-steps)', ylab = 'Implied Volatility',
       main = paste0("Strike = ",sprintf("%.2f",put_strikes[i])),
       yaxt = "n", lwd = 1.5) # yaxt = 'n' suppress drawing the y axis values
  axis(2, at = seq(lo, hi, by = 0.01), las = 1) # draw y axis ticks from lo to hi every
      0.01
}
mtext("Implied Volatility Convergence: Put Options", outer = T, cex = 1)
dev.off()

## plot computation time as a fxn of N
pdf("./plots/times.pdf", height = 4.75, width = 9.25)
par(mfrow = c(1,2), oma = c(0, 0, 2, 0)) # create a frame with 1x2 boxes

plot(times$seconds ~ log10(times$N), type = 'b',
     ylab = "Seconds", xlab = "log10(N-steps)",
     main = "Seconds by log10 Steps", las = 1, lwd = 2)
plot(log10(times$seconds) ~ log10(times$N), type = 'b',
     ylab = "log10(Seconds)", xlab = "log10(N-steps)",
     main = "log10 Seconds by log10 Steps", las = 1, lwd = 2)
mtext("Implied Volatility Computation Time", outer = T, cex = 1.25)
dev.off()
```