Assignment 3

David Fleischer
MACF 402 - Mathematical & Computational Finance II

November 16, 2015

1 Question 1: Programming a Black-Scholes Implied Volatility Calculator

1.1 Numerical Solutions to the Normal CDF

We first investigate our (naive) numerical approximation to the standard normal CDF

$$\Phi(x) \approx 1 - \phi(x)(b_1 t + b_2 t^2 + b_3 t^3 + b_4 t^4 + b_5 t^5)$$

where $b_0 = 0.2316419, b_1 = 0.319381530, b_2 = -0.356563782, b_3 = 1.781477937,\ b_4 = -1.821255978, b_5 = 1.330274429$, and $t = \frac{1}{1+b_0 x}$.

Comparing our approximation to the true value of the normal CDF (as computed by R's `pnorm`) we see that the approximation is reasonable for positive values of x but is a failure for x negative. In particular, Figure 1 shows that values of the normal CDF approximation for $x \in [-3, 3]$ and $x \in [-4, -3]$ are not usable for x negative, becoming an increasingly poor approximation as x decreases. We also notice a nasty singularity present itself in our approximation when $1 + b_0 x = 0 \iff x = \frac{-1}{0.2316419} \approx -4.317$. Figure 2 completes the point, showing us that the error between the true value of the normal CDF and our approximation is acceptable for only $x > 0$.

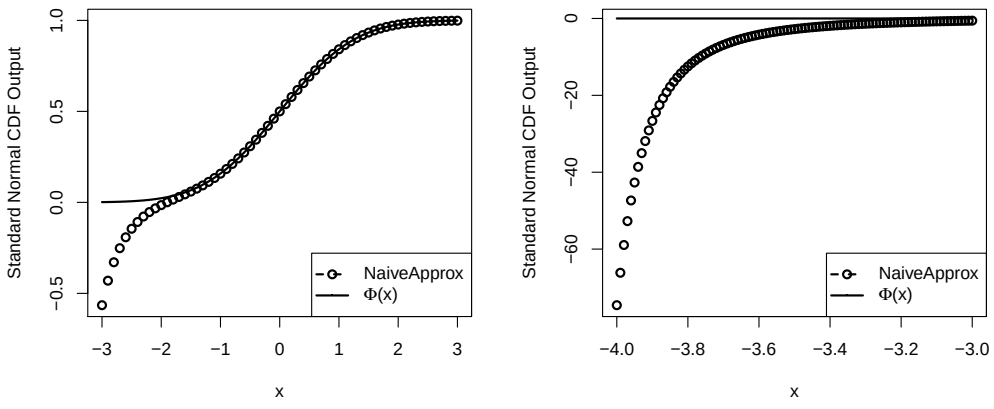


Figure 1: (Left) The approximation to the normal CDF described above is successful for positive x but for negative x is a poor approximation. (Right) Increasingly negative values of x show completely unusable approximations of the normal CDF.

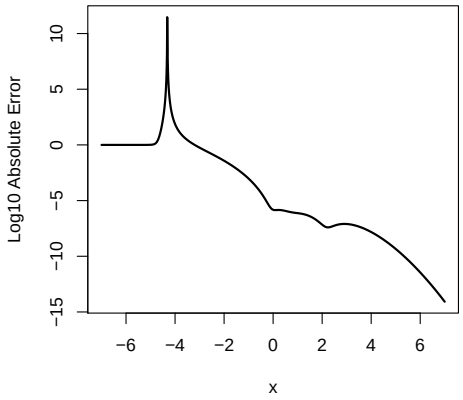


Figure 2: The error between the normal approximation and the true value of the normal CDF is sufficiently small for x large but for x small the error is too large to be usable. For x small the value of the error settles to $\approx 10^0 = 1$: Completely unusable as a CDF.

To solve this problem we note that our function is suitable for x positive and rely on the symmetry of the normal distribution. In particular, we use

$$\Phi(-x) = 1 - \Phi(x)$$

From this, we find our new approximation to the normal CDF

$$\Phi(x) \approx \begin{cases} 1 - \phi(x)(b_1 t(x) + b_2[t(x)]^2 + b_3[t(x)]^3 + b_4[t(x)]^4 + b_5[t(x)]^5) & x \geq 0 \\ \phi(-x)(b_1 t(-x) + b_2[t(-x)]^2 + b_3[t(-x)]^3 + b_4[t(-x)]^4 + b_5[t(-x)]^5) & x < 0 \end{cases}$$

Figure 3 shows us that this new implementation is a significant improvement to our earlier attempt. With the normal approximation complete we move on to other issues.

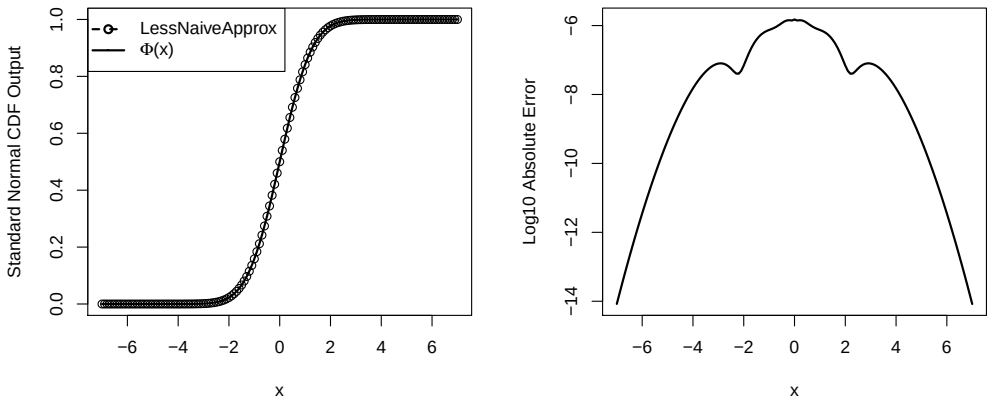


Figure 3: (Left) The new implementation of the normal approximation is appreciably more accurate than the implementation first introduced. (Right) The absolute error for all x is now sufficiently small (for our purposes).

1.2 Computing Implied Volatilities

We now use our closed form expression (approximation) for the normal CDF to compute the Black-Scholes price of a European. That is, we have

$$C(S,K,t,T,r,\sigma) = \Phi(d_1)S - \Phi(d_2)Ke^{-r(T-t)}$$
$$P(S,K,t,T,r,\sigma) = \Phi(-d_2)Ke^{-r(T-t)} - \Phi(-d_1)S$$

with

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right]$$
$$d_2 = d_1 - \sigma\sqrt{T-t}$$

and option Delta & Vega[1]

$$\Delta_C = \Phi(d_1)$$
$$\Delta_P = -\Phi(-d_1)$$
$$\nu_C = \nu_P = S\phi(d_1)\sigma\sqrt{T-t}$$

With our ingredients in place we are now in a position to numerically solve for the Black-Scholes implied volatility for options written on an underlying asset with $S_0 = 4.75, r = 0.0492$ and $\tau = 59/365$ years to maturity.

[1] Derivations of option Delta and Vega are presented in an appendix.

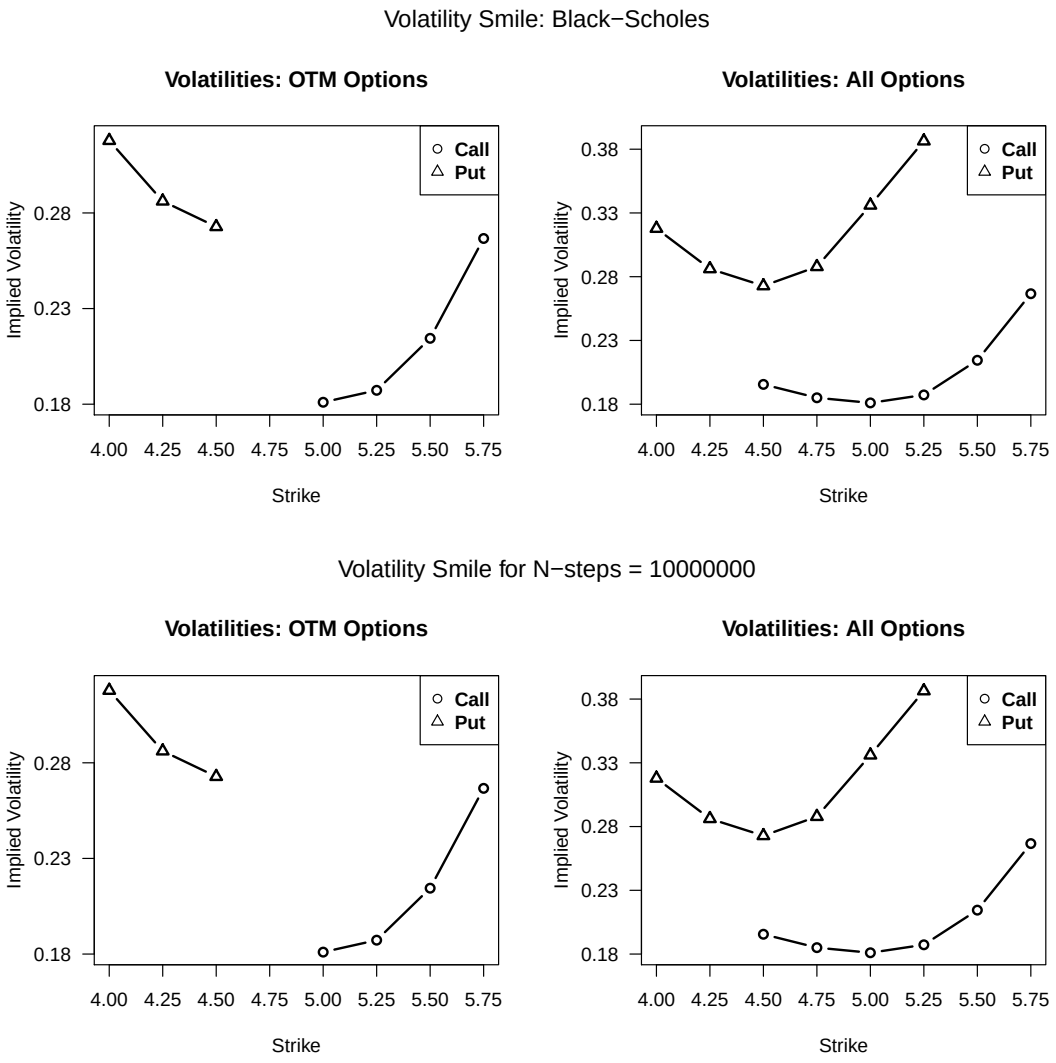


Figure 4: The volatility smile as computed by the Black-Scholes model (top) and the Binomail model (bottom) for the given asset using only OTM observed option prices (left) and all option prices (right).

Figure 4 shows us a notable smile, with a discontinuity at $K = S$, when computing the implied volatility with only OTM options. Notably, the Black-Scholes implementation produces implied volatility estimates that are (at least impressionistically) nearly identical to those calculated using the Binomial model. Yielding the same output, the Black-Scholes model is preferable since it is sufficiently closed-form and thus extremely fast. This is a point of comparison with the iterative solution for the Binomial model, which is comparatively slow.

2 **Question 2:** Fitting an Implied Volatility Surface

For Apple (AAPL) stock on February 17, 2012 with spot price \$502.12 we have computed the implied volatility curve for options expiring on March 17, 2012; April 17, 2012; and May 17, 2012, seen in Figure 5. Note that we see a decided volatility smile, as computed by OTM options written on AAPL, in all three maturities.

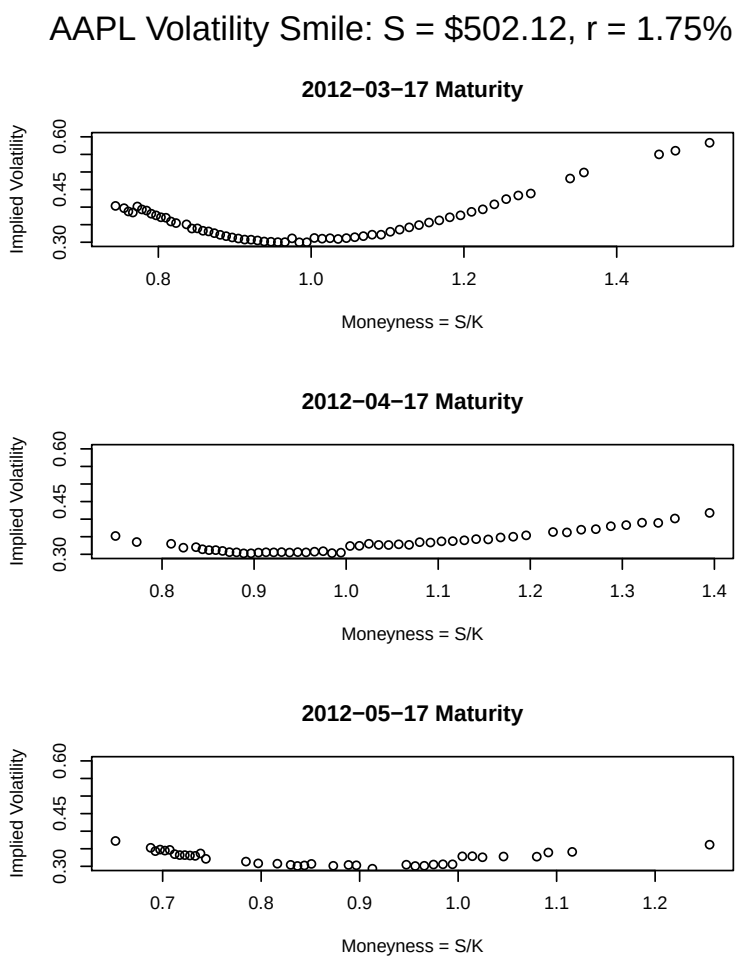


Figure 5: The implied volatility curve for AAPL as computed by OTM options. We note the clear volatility smile.

2.1 One Dimensional Fitted Estimates

In order to determine an effective smoothing parameter h for the one dimensional Nadaraya-Watson estimator we perform a bit of exploration in a region of plausible parameters. Plotted in Figures 6, 7, and 8 are the fitted smiles for March, April, and May maturities, using $h \in \{0.025, 0.05, 0.075, 0.1\}$. Pathological cases for h extremely small/large are considered in the two dimensional estimator.

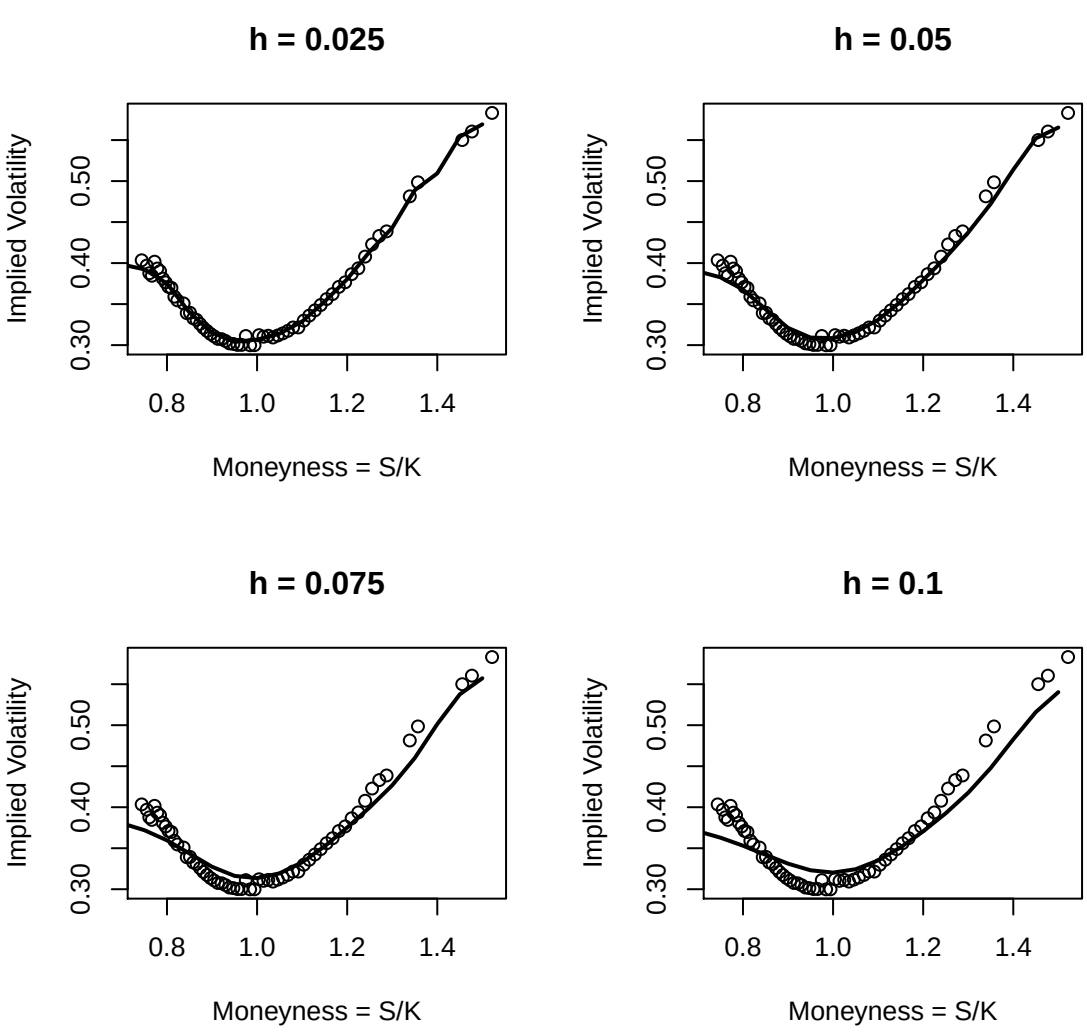


Figure 6: Fitted smiles for options written on AAPL expiring March 17, 2012.

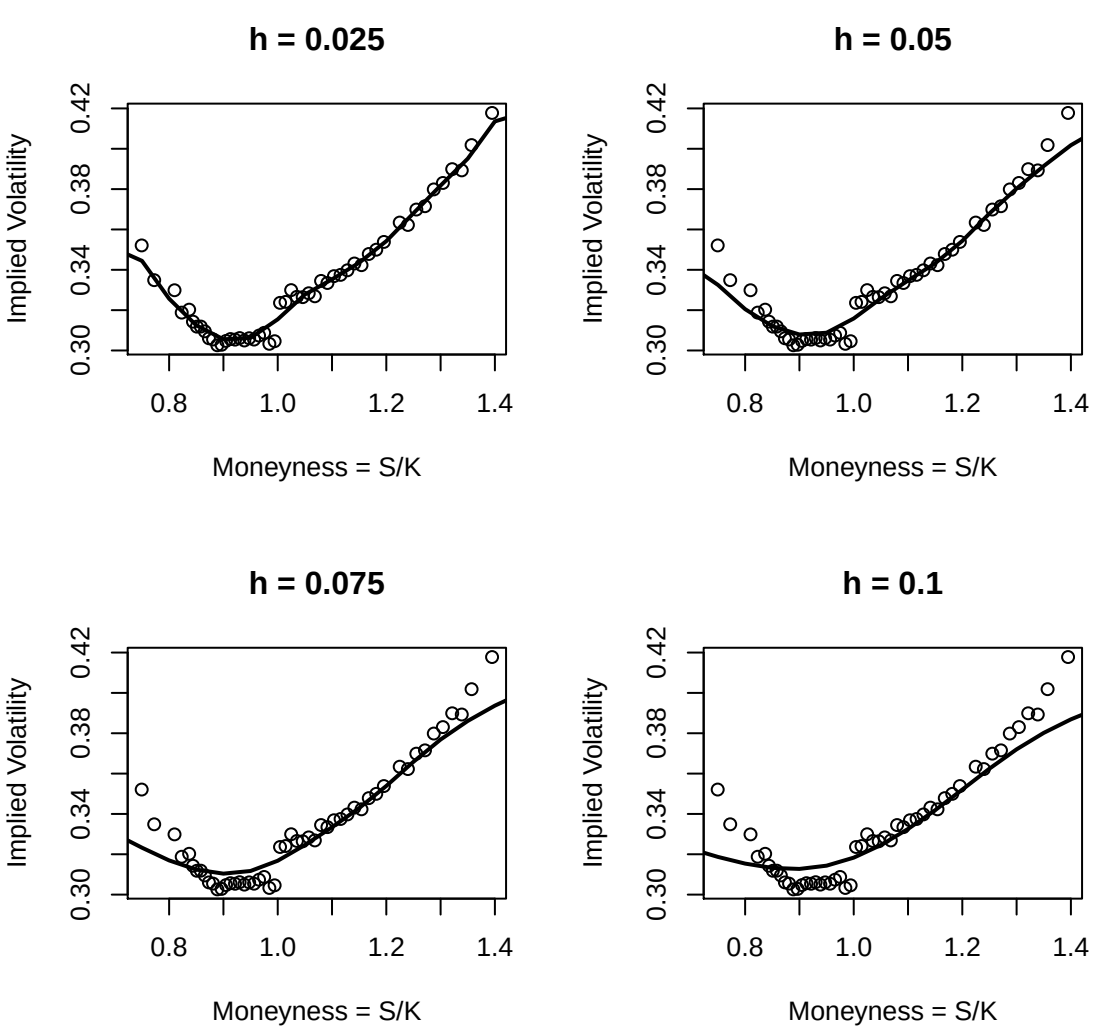


Figure 7: Fitted smiles for options written on AAPL expiring on April 17, 2012.

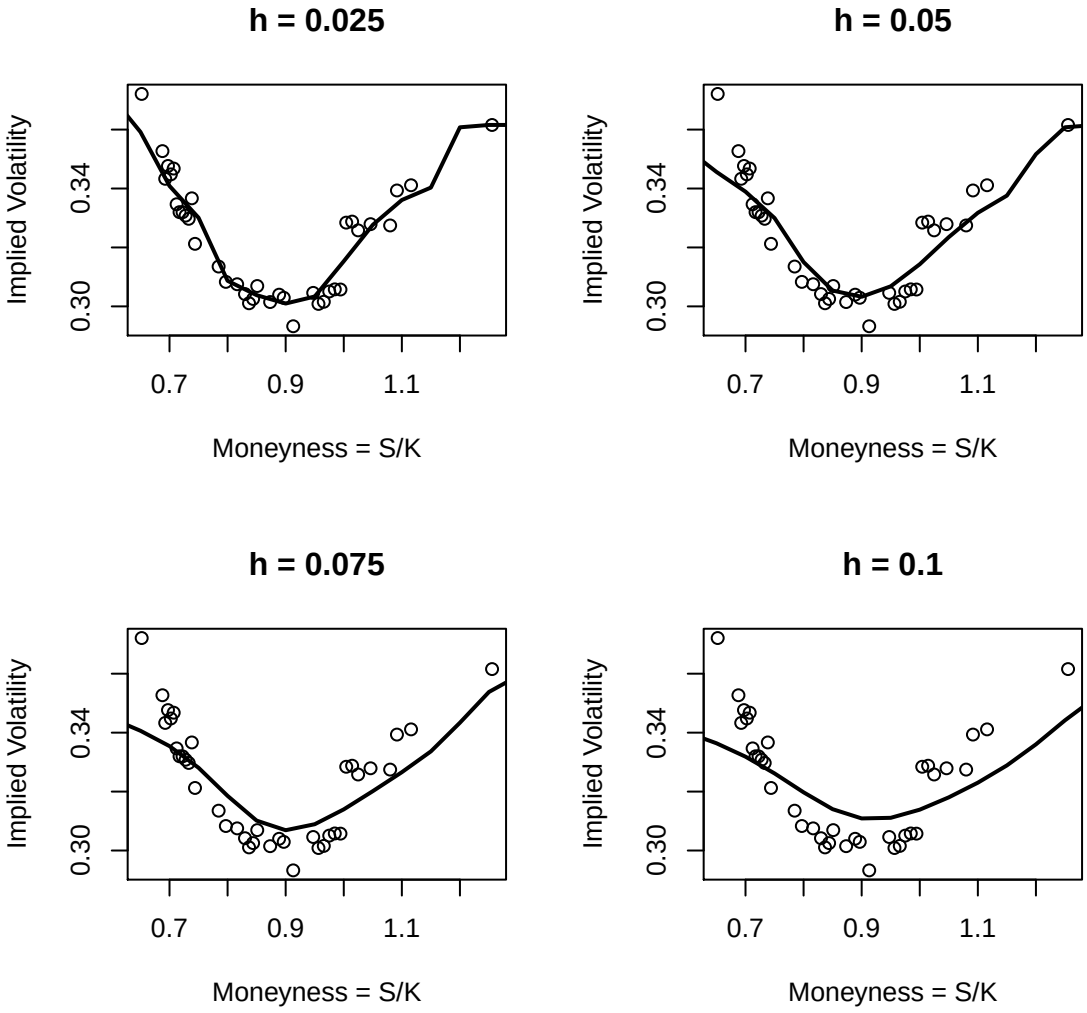


Figure 8: Fitted smiles for options written on AAPL expiring on May 17, 2012.

We see that using $h = 0.025$ leads to overfit curves for all maturities, causing several apparent discontinuities. Using $h = 0.05$ provides a reasonably smooth curve for March and April maturities (with the possible exception of extremely high moneyness options for these dates) but leads to overfitting for options expiring in May. Using $h = 0.075$ leads to smooth curves for all maturities, and so we consider $h = 0.075$ as our relatively estimate for h after these trials.

2.2 Two Dimensional Fitted Estimates

2.2.1 Exploration for an Optimal h_1, h_2

As the min/max of the domain for the two dimensional Nadayara-Watson estimator we use $(m, t) \in [m_{min}, m_{max}] \times [T_{min}, T_{max}] = [0.5, 1.5] \times [7/365, 105/365]$. We show in Figures 9 and 10 some examples of the volatility surface for different values of h_1, h_2. Some pathological cases of overfit/underfit surfaces are provided in an Appendix below. We in Figure 9 that using $h_1 = h_2 = 0.075$ leads to a smooth fitted surface that captures a reasonable amount of detail. Using $h_1 = h_2 = 0.05$ leads to a possibly overfit surface, due to its "bumpy" surface features, and so we instead prefer the surface with $h_1 = 0.075, h_2 = 0.075$.

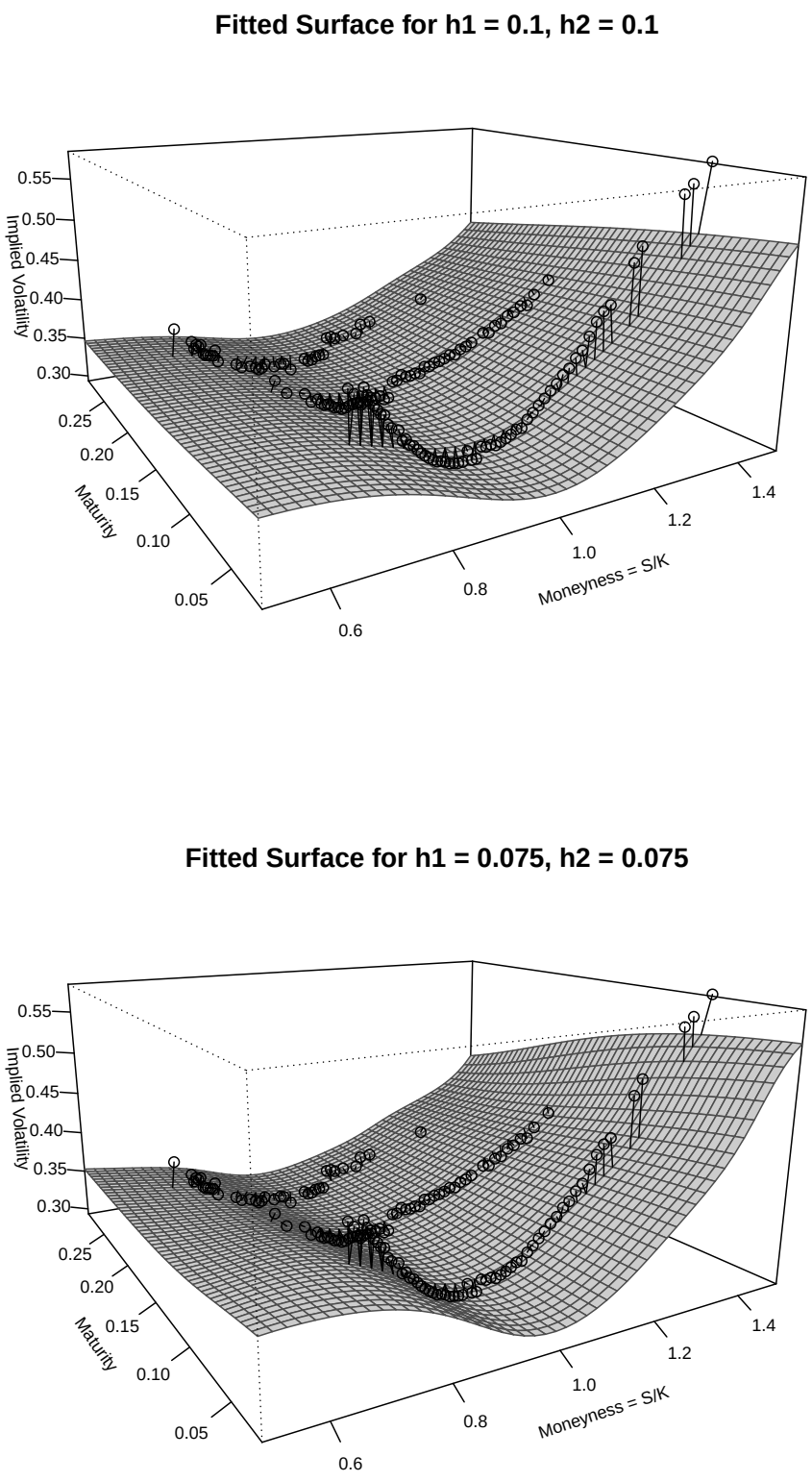


Figure 9: Fitted volatility surface for (top) $h_1 = 0.1, h_2 = 0.1$ and (bottom) $h_1 = 0.075, h_2 = 0.075$.

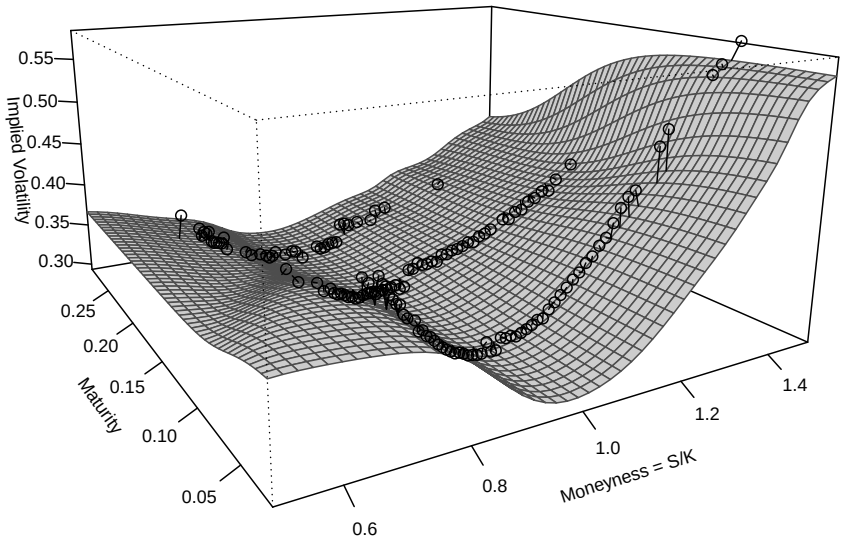


Figure 10: Fitted volatility surface for $h_1 = 0.05, h_2 = 0.05$.

Contracting the domain of the Nadaraya-Watson estimator to the min/max of the observed data, we see an appreciable improvement to the fitted volatility surface. Figure 11 shows us this improved fit by reducing the domain to $(m, t) \in [m_{min}, m_{max}] \times [T_{min}, T_{max}] = [502.12/770, 502.12/330] \times [29/365, 90/365]$. The observed data appears closer to the volatility surface while still avoiding undue "bumpiness" in the surface. In addition, we are able to get away with smaller smoothing parameters $h_1 = h_2 = 0.05$.

Reflecting on these results we may conclude that is likely not a very good idea to extrapolate far beyond the observed data. This makes intuitive sense since the Nadayara-Waton Estimator uses observations within the entire range to weight the estimate at a particular point. If we extrapolate beyond the min/max then we are forced to weight the estimate using only data prior to these points, since have no observations beyond these extrema to complete the weighting.

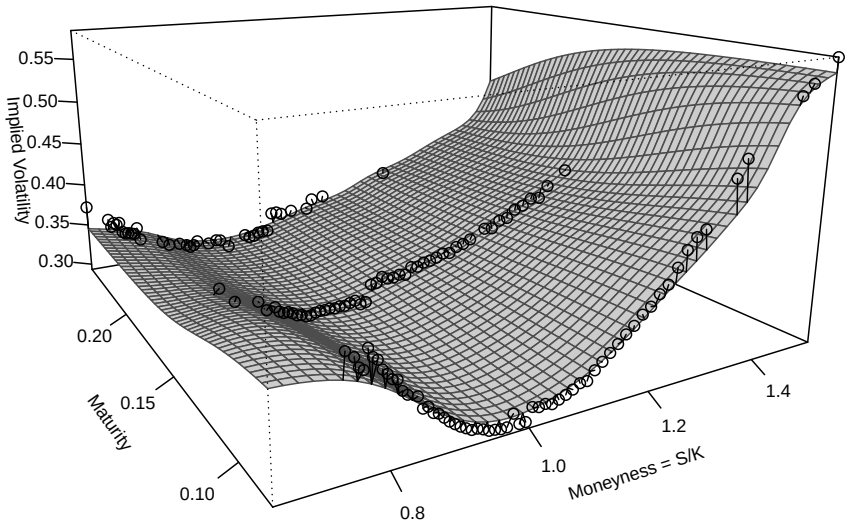


Figure 11: Fitted volatility surface under the new domain $[m_{min}, m_{max}] \times [T_{min}, T_{max}] = [502.12/770, 502.12/330] \times [29/365, 90/365]$, with $h_1 = 0.05, h_2 = 0.05$.

3 **Question 3:** No Arbitrage Restriction on the Volatility Surfaces

For brevity we consider $t = 0$, thus letting T be the time to maturity of our option. We first compute

$$\begin{aligned}
\frac{\partial C}{\partial K} &= \frac{\partial}{\partial K}\Big(\Phi(d_1)S - \Phi(d_2)Ke^{-rT}\Big) \\
&= \phi(d_1)\frac{\partial d_1}{\partial K}S - \phi(d_2)\frac{\partial d_2}{\partial K}Ke^{-rT} - \Phi(d_2)e^{-rT} \\
&= \frac{-1}{K\sigma\sqrt{T}}\Big(\phi(d_1)S - \phi(d_2)Ke^{-rT}\Big) - \Phi(d_2)e^{-rT}
\end{aligned}$$

but from the derivation of option Δ (in an appendix below) we have

$$\begin{aligned}
\phi(d_1)S - \phi(d_2)Ke^{-rT} &= 0 \\
\implies \frac{\partial C}{\partial K} &= \frac{-1}{K\sigma\sqrt{T}} \cdot 0 - \Phi(d_2)e^{-rT} \\
&= -\Phi(d_2)e^{-rT}
\end{aligned}$$

and from Put-Call parity we have

$$\begin{aligned}
C &= P + S - Ke^{-rT} \\
\implies -\Phi(d_2)e^{-rT} &= \frac{\partial P}{\partial K} - e^{-rT} \\
\implies \frac{\partial P}{\partial K} &= \big(1 - \Phi(d_2)\big)e^{-rT}
\end{aligned}$$

For the second derivatives $\frac{\partial^2 C}{\partial K^2}, \frac{\partial^2 P}{\partial K^2}$ we have

$$\begin{aligned}
\frac{\partial^2 C}{\partial K^2} &= \frac{\partial}{\partial K}\big(- \Phi(d_2)e^{-rT}\big) \\
&= -\phi(d_2)\frac{\partial d_2}{\partial K}e^{-rT} \\
&= -\phi(d_2)\frac{-1}{K\sigma\sqrt{T}}e^{-rT} = \frac{\phi(d_2)e^{-rT}}{K\sigma\sqrt{T}} \\
\frac{\partial^2 P}{\partial K^2} &= \frac{\partial}{\partial K}\big(1 - \Phi(d_2)\big)e^{-rT} \\
&= \frac{\partial}{\partial K}\big(- \Phi(d_2)e^{-rT}\big) \\
&= \frac{\partial^2 C}{\partial K^2} = \frac{\phi(d_2)e^{-rT}}{K\sigma\sqrt{T}}
\end{aligned}$$

Thus, for the first two no-arbitrage requirements

$$\begin{aligned}
&\frac{\partial C}{\partial K} \leq 0, \quad \frac{\partial P}{\partial K} \geq 0 \\
&\frac{\partial^2 C}{\partial K^2} \geq 0, \quad \frac{\partial^2 P}{\partial K^2} \geq 0
\end{aligned}$$

we have

$$\begin{aligned}
&-\Phi(d_2)e^{-rT} \leq 0, \quad \big(1 - \Phi(d_2)\big)e^{-rT} \geq 0 \\
&\frac{\phi(d_2)e^{-rT}}{K\sigma\sqrt{T}} \geq 0
\end{aligned}$$

If we implement these sufficiently closed form expressions with our volatility surface we see the following figures:

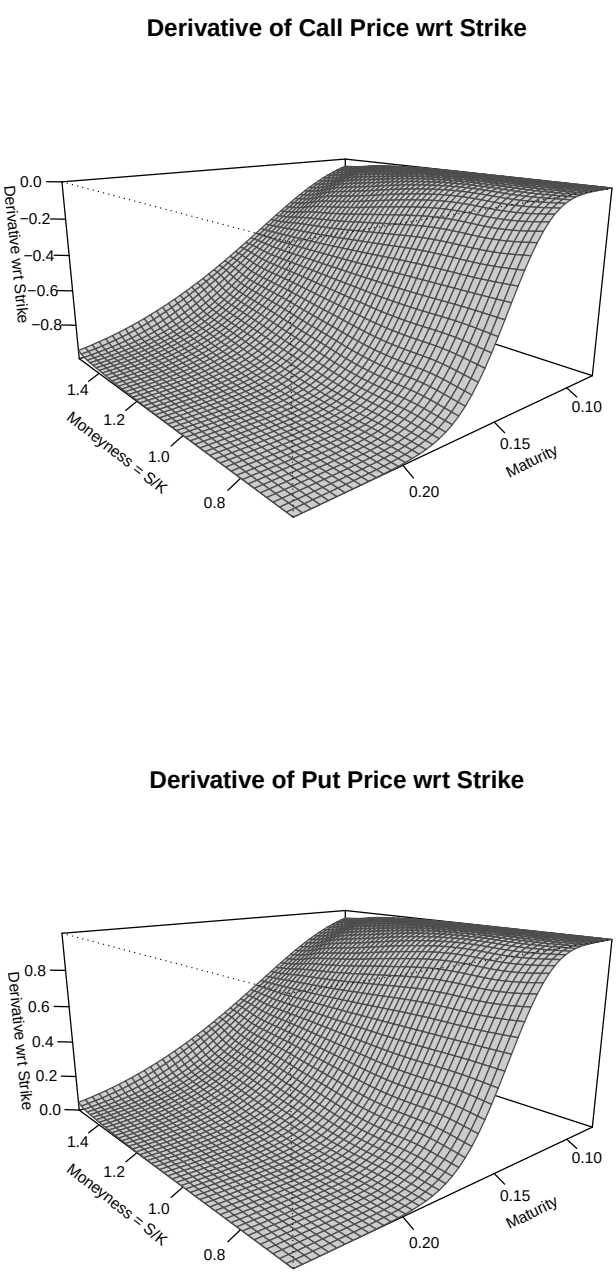


Figure 12: Computed values for (top) $\frac{\partial C}{\partial K}$ and (bottom) $\frac{\partial P}{\partial K}$ given the fitted volatility surface for $h_1 = 0.05, h_2 = 0.05$, with the domain of the fitted surface being the min/max of the observed data.

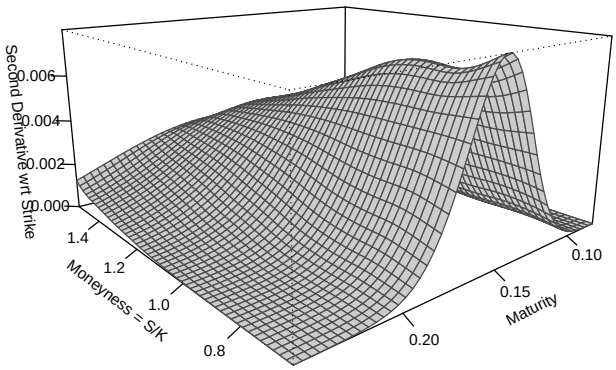


Figure 13: Computed values for $\frac{\partial C}{\partial K} = \frac{\partial P}{\partial K}$ given the fitted volatility surface for $h_1 = 0.05, h_2 = 0.05$, with the domain of the fitted surface being the min/max of the observed data.

Using this fitted surface ($h_1 = 0.05, h_2 = 0.05$, with the min/max of each dimension corresponding to the min/max of the observed data) we see that our volatility estimates do not predict call/put spread arbitrage or butterfly arbitrage opportunities, satisfying the requirements presented above. We are pleased and move on to discussing calendar spread arbitrage. We find that

$$\begin{aligned}
\frac{\partial C}{\partial T} &= \frac{\partial}{\partial T}\left(\Phi(d_1)S - \Phi(d_2)Ke^{-rT}\right) \\
&= \phi(d_1)\frac{\partial d_1}{\partial T}S - \phi(d_2)\frac{\partial d_2}{\partial T}Ke^{-rT} + \Phi(d_2)Kre^{-rT} \quad \text{but} \\
\frac{\partial d_2}{\partial T} &= \frac{\partial d_1}{\partial T} - \frac{\sigma}{2\sqrt{T}} \quad \text{so} \\
\frac{\partial C}{\partial T} &= \phi(d_1)\frac{\partial d_1}{\partial T}S - \phi(d_2)\frac{\partial d_1}{\partial T}Ke^{-rT} + \frac{\phi(d_2)Ke^{-rT}\sigma}{2\sqrt{T}} + \Phi(d_2)Kre^{-rT} \\
&= \frac{\partial d_1}{\partial T}\left(\phi(d_1)S - \phi(d_2)Ke^{-rT}\right) + \frac{\phi(d_2)Ke^{-rT}\sigma}{2\sqrt{T}} + \Phi(d_2)Kre^{-rT}
\end{aligned}$$

but we have already shown that $\phi(d_1)S - \phi(d_2)Ke^{-rT} = 0$, so

$$\begin{aligned}
\frac{\partial C}{\partial T} &= \frac{\phi(d_2)Ke^{-rT}\sigma}{2\sqrt{T}} + \Phi(d_2)Kre^{-rT} \\
&= Ke^{-rT}\left(r\Phi(d_2) + \frac{\sigma\phi(d_2)}{2\sqrt{T}}\right)
\end{aligned}$$

and from Put-Call parity we have

$$\begin{aligned}
C &= P + S - Ke^{-rT} \\
\implies Ke^{-rT}\left(r\Phi(d_2) + \frac{\sigma\phi(d_2)}{2\sqrt{T}}\right) &= \frac{\partial P}{\partial T} + Kre^{-rT} \\
\implies \frac{\partial P}{\partial T} &= Ke^{-rT}\left(r\Phi(d_2) + \frac{\sigma\phi(d_2)}{2\sqrt{T}}\right) - Kre^{-rT} \\
&= Ke^{-rT}\left(r\Phi(d_2) + \frac{\sigma\phi(d_2)}{2\sqrt{T}} - r\right) \\
&= Ke^{-rT}\left(r\big(1 - \Phi(-d_2)\big) + \frac{\sigma\phi(d_2)}{2\sqrt{T}} - r\right) \\
&= Ke^{-rT}\left(-r\Phi(-d_2) + \frac{\sigma\phi(d_2)}{2\sqrt{T}}\right)
\end{aligned}$$

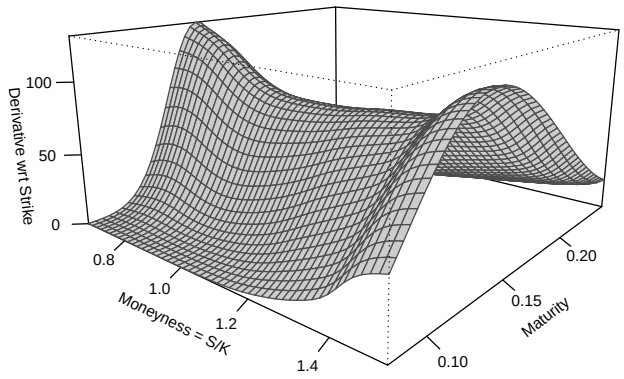
Thus, for the final no-arbitrage requirement

$$\frac{\partial C}{\partial T} \geq 0, \quad \frac{\partial P}{\partial T} \geq 0$$

we have

$$Ke^{-rT}\left(r\Phi(d_2) + \frac{\sigma\phi(d_2)}{2\sqrt{T}}\right) \geq 0, \quad Ke^{-rT}\left(-r\Phi(-d_2) + \frac{\sigma\phi(d_2)}{2\sqrt{T}}\right) \geq 0$$

If we implement these sufficiently closed form expressions for $\frac{\partial C}{\partial T}$, $\frac{\partial P}{\partial T}$ with our volatility surface we see the following figures:



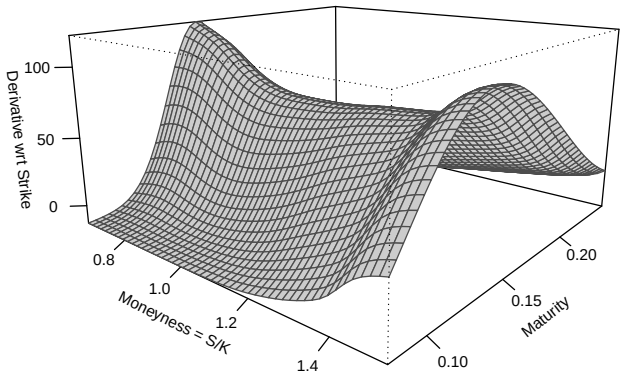


Figure 14: Computed values for (top) $\frac{\partial C}{\partial T}$ and (bottom) $\frac{\partial P}{\partial T}$ given the fitted volatility surface for $h_1 = 0.05, h_2 = 0.05$, with the domain of the fitted surface being the min/max of the observed data.

Notice that using the fitted volatility surface ($h_1 = 0.05, h_2 = 0.05$) we satisfy the no calendar spread arbitrage requirement for call options, but for put options we observe some negative values for $\frac{\partial P}{\partial T}$ at extremely low moneyness, short maturity options. Manipulating the smoothing parameters doesn't appear to remedy the situation (plots omitted), leaving these extreme cases predicting arbitrage opportunities. Thus, we suspect that the cause of these calendar spread arbitrage predictions is the presence of outliers in this range of our data set.

4 **Question 4:** Pricing a Vanilla Option Based on the Volatility Surface

With fixed maturity $T = 60/365$ we wish to price a European call option written on AAPL ($S = 502.12, r = 0.0175$) with strike $K = 572.50$ using our fitted surface. This particular surface used $N = 50$ grid points in each dimension. Finding the fitted volatility corresponding to the moneyness and maturity the nearest to the desired moneyness/maturity we get

```
closest_moneyness = 0.88278
closest_maturity = 0.164719
sigma_hat = 0.31168
```

We note that this is an error of 0.005714 in moneynes (strike error = \$3.7060). Similarly, we have an error in maturity of 0.0003354 years, approximately 2.94 hours. With this we price the European call option, and corresponding hedge ratio,

```
bs_price = 5.46787
bs_hedge_ratio = 0.17053
```

As an exercise, we up the number of grid point in our fitted surface to $N = 100$ in each dimension, we get

```
closest_moneyness = 0.88045
closest_maturity = 0.163858
sigma_hat = 0.312129
```

This which corresponds to a moneyness error of 0.00338 (strike error = \$2.2007) and a maturity error of 0.0005256 years, approximately 4.96 hours. This values give us

```
bs_price = 5.49106
bs_hedge_ratio = 0.170923
```

which corresponds to a difference in price of approximately 2.3 cents. For fun, we up the steps to $N = 1000$, giving us

```
closest_moneyness = 0.876652
closest_maturity = 0.164436
sigma_hat = 0.31262
bs_price = 5.51647
bs_hedge_ratio = 0.171353
```

with errors of 0.0004135 in moneyness (strike error = \$0.2700) and a maturity error of 0.00005244 years, approximately 27.56 minutes. These values give us a price difference of approximately 2.5 cents from the previous. Certainly these differences are appreciable to a financial institution, and we may recommend the use of some local averaging of the volatility surface to get a price that is as accurate as possible.

Appendix A Supplemental Figures

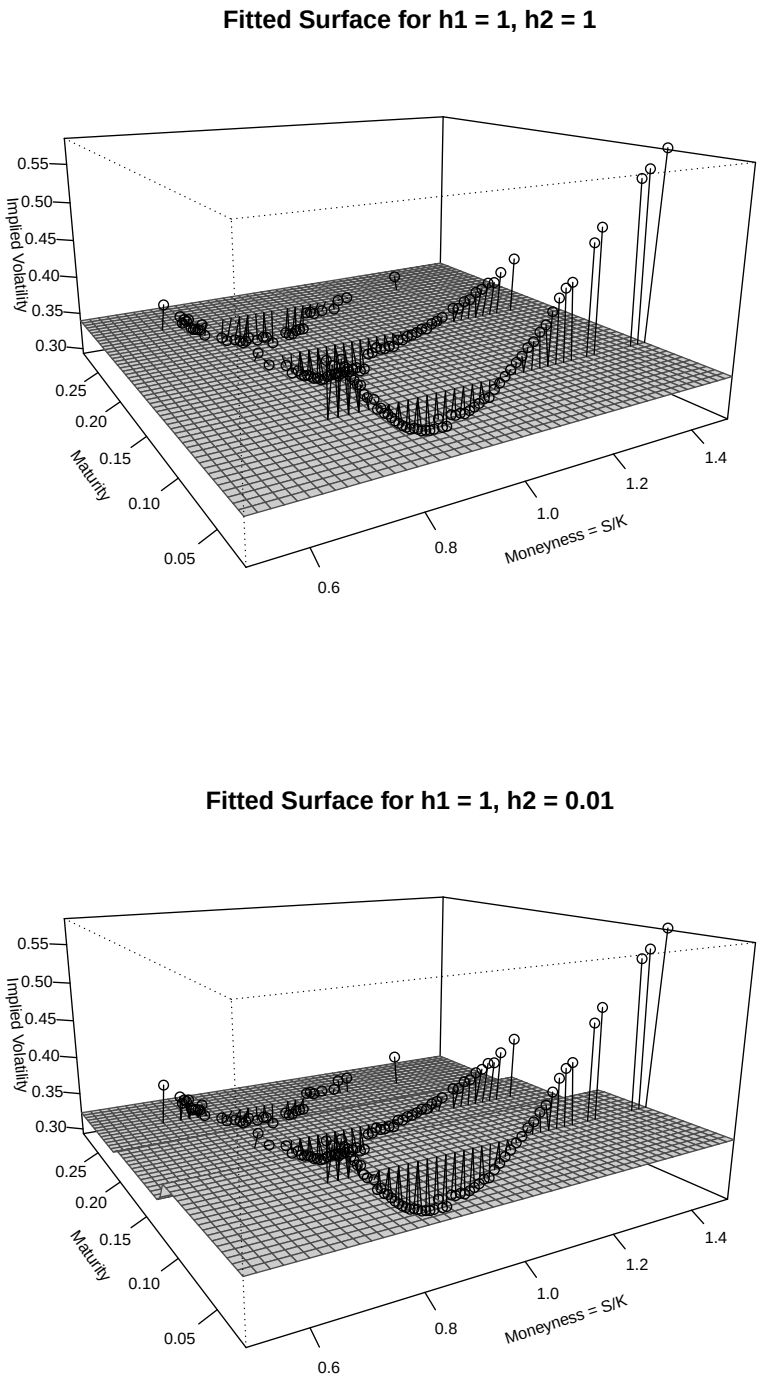


Figure 15: Fitted volatility surface for (top) $h_1 = 1, h_2 = 1$ and (bottom) $h_1 = 1, h_2 = 0.01$.

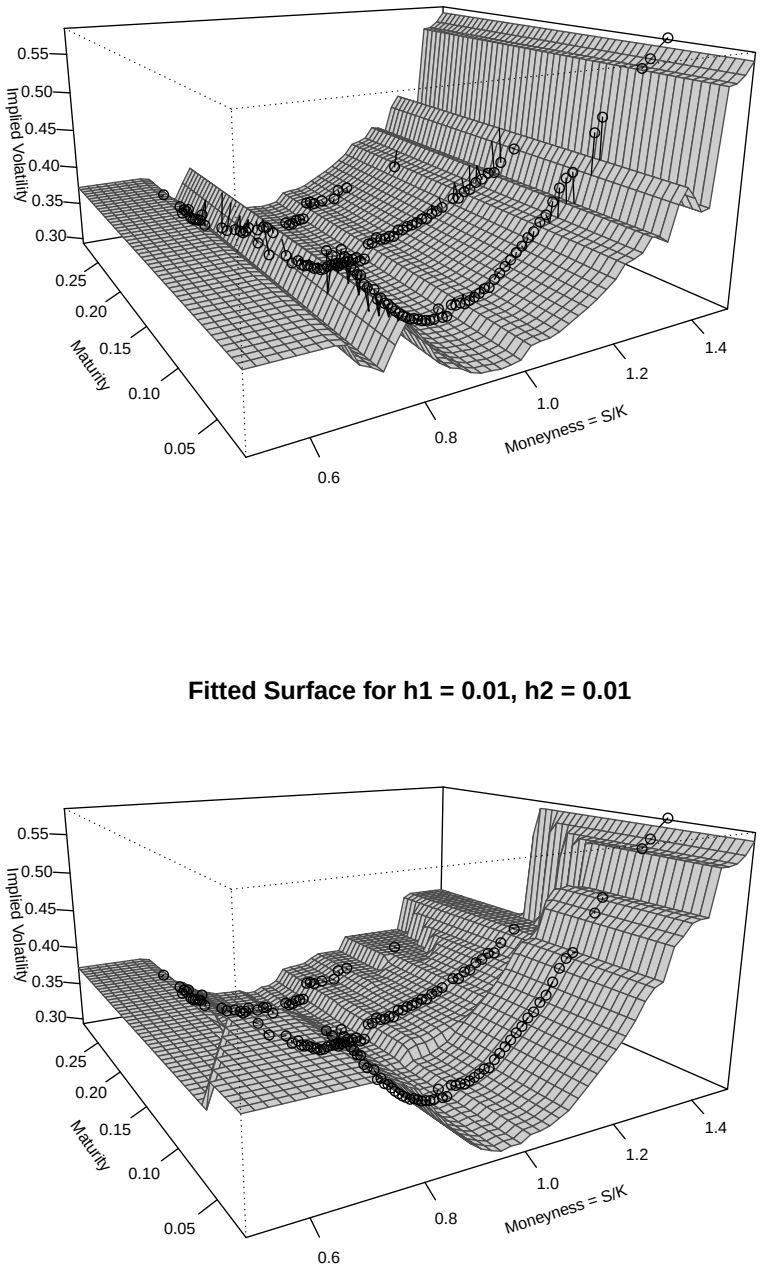


Figure 16: Fitted volatility surface for (top) $h_1 = 0.01, h_2 = 1$ and (bottom) $h_1 = 0.01, h_2 = 0.01$.

Appendix B Greeks: Option Δ and ν

The Black-Scholes price of a European call/put option are

$$C(S,K,t,T,r,\sigma) = \Phi(d_1)S - \Phi(d_2)Ke^{-r(T-t)}$$
$$P(S,K,t,T,r,\sigma) = \Phi(-d_2)Ke^{-r(T-t)} - \Phi(-d_1)S$$

with

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right]$$
$$d_2 = d_1 - \sigma\sqrt{T-t}$$

We wish to compute the option Delta, Δ_C, Δ_P and option Vega, ν_C, ν_P. First computing the derivatives for the call price we find

$$\begin{aligned}
\Delta_C = \frac{\partial C}{\partial S} &= \frac{\partial}{\partial S}\Big(\Phi(d_1)S - \Phi(d_2)Ke^{-r(T-t)}\Big) \\
&= \Phi'(d_1)\Big(\frac{\partial}{\partial S}d_1\Big)S + \Phi(d_1) - \Phi'(d_2)\Big(\frac{\partial}{\partial S}d_2\Big)Ke^{-r(T-t)} \\
&= \phi(d_1)\Big(\frac{\partial}{\partial S}d_1\Big)S + \Phi(d_1) - \phi(d_2)\Big(\frac{\partial}{\partial S}d_2\Big)Ke^{-r(T-t)} \\
\nu_C = \frac{\partial C}{\partial \sigma} &= \frac{\partial}{\partial \sigma}\Big(\Phi(d_1)S - \Phi(d_2)Ke^{-r(T-t)}\Big) \\
&= \phi(d_1)\Big(\frac{\partial}{\partial \sigma}d_1\Big)S - \phi(d_2)\Big(\frac{\partial}{\partial \sigma}d_2\Big)Ke^{-r(T-t)}
\end{aligned}$$

Computing $\frac{\partial}{\partial S}d_1$ and $\frac{\partial}{\partial S}d_2$,

$$\begin{aligned}
\frac{\partial}{\partial S}d_1 &= \frac{\partial}{\partial S}\frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right] \\
&= \frac{1}{S\sigma\sqrt{T-t}} \\
\frac{\partial}{\partial S}d_2 &= \frac{\partial}{\partial S}(d_1 + \sigma\sqrt{T-t}) = \frac{\partial}{\partial S}d_1 = \frac{1}{S\sigma\sqrt{T-t}}
\end{aligned}$$

Thus

$$\begin{aligned}
\Delta_C = \frac{\partial C}{\partial S} &= \phi(d_1)\frac{1}{S\sigma\sqrt{T-t}}S + \Phi(d_1) - \phi(d_2)\frac{1}{S\sigma\sqrt{T-t}}Ke^{-r(T-t)} \\
&= \Phi(d_1) + \frac{1}{S\sigma\sqrt{T-t}}\Big[S\phi(d_1) - Ke^{-r(T-t)}\phi(d_2)\Big]
\end{aligned}$$

but notice

$$\begin{aligned}
S\phi(d_1) - Ke^{-r(T-t)}\phi(d_2) = 0 &\iff \frac{S}{K}e^{r(T-t)} = \frac{\phi(d_2)}{\phi(d_1)} \\
&\iff \ln\frac{S}{K} + r(T-t) = \frac{1}{2}\big(d_1^2 - d_2^2\big)
\end{aligned}$$

and

$$\begin{aligned}
\frac{1}{2}\big(d_1^2 - d_2^2\big) &= \frac{1}{2}(d_1 - d_2)(d_1 + d_2) \\
&= \frac{1}{2}(d_1 - d_1 + \sigma\sqrt{T-t})(d_1 + d_1 - \sigma\sqrt{T-t}) \\
&= \frac{\sigma\sqrt{T-t}}{2}(2d_1 - \sigma\sqrt{T-t}) \\
&= d_1\sigma\sqrt{T-t} - \frac{\sigma^2(T-t)}{2} \\
&= \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right]\sigma\sqrt{T-t} - \frac{\sigma^2(T-t)}{2} \\
&= \ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t) - \frac{\sigma^2(T-t)}{2} \\
&= \ln\left(\frac{S}{K}\right) + r(T-t)
\end{aligned}$$

Hence

$$\begin{aligned} S\phi(d_1) - Ke^{-r(T-t)}\phi(d_2) = 0 & \\ \implies \Delta_C = \Phi(d_1) + \frac{1}{S\sigma\sqrt{T-t}}\Big[S\phi(d_1) - Ke^{-r(T-t)}\phi(d_2)\Big] & \\ = \Phi(d_1) & \end{aligned}$$

For call Vega we note that

$$\begin{aligned} \frac{\partial}{\partial\sigma}d_2 = \frac{\partial}{\partial\sigma}(d_1 - \sigma\sqrt{T-t}) & \\ \implies \frac{\partial}{\partial\sigma}d_1 - \frac{\partial}{\partial\sigma}d_2 = \sqrt{T-t} & \end{aligned}$$

So

$$\begin{aligned} \nu_C = \frac{\partial C}{\partial\sigma} &= \phi(d_1)\Big(\frac{\partial}{\partial\sigma}d_1\Big)S - \phi(d_2)\Big(\frac{\partial}{\partial\sigma}d_2\Big)Ke^{-r(T-t)} \\ &= \phi(d_1)\Big(\frac{\partial}{\partial\sigma}d_1\Big)S - \phi(d_2)\Big(\frac{\partial}{\partial\sigma}d_2\Big)Ke^{-r(T-t)} - \phi(d_2)\Big(\frac{\partial}{\partial\sigma}d_1\Big)Ke^{-r(T-t)} + \\ &\qquad \phi(d_2)\Big(\frac{\partial}{\partial\sigma}d_1\Big)Ke^{-r(T-t)} \\ &= \Big(\frac{\partial}{\partial\sigma}d_1\Big)\Big[\phi(d_1)S - \phi(d_2)Ke^{-r(T-t)}\Big] + \phi(d_2)Ke^{-r(T-t)}\Big[-\frac{\partial}{\partial\sigma}d_2 + \frac{\partial}{\partial\sigma}d_1\Big] \\ &= \Big(\frac{\partial}{\partial\sigma}d_1\Big)\Big[\phi(d_1)S - \phi(d_2)Ke^{-r(T-t)}\Big] + \phi(d_2)Ke^{-r(T-t)}\sigma\sqrt{T-t} \end{aligned}$$

but

$$S\phi(d_1) - Ke^{-r(T-t)}\phi(d_2) = 0 \iff S\phi(d_1) = Ke^{-r(T-t)}\phi(d_2)$$

Hence

$$\begin{aligned} \nu_C &= \phi(d_2)Ke^{-r(T-t)}\sigma\sqrt{T-t} \\ &= S\phi(d_1)\sigma\sqrt{T-t} \end{aligned}$$

So, for a European call we have

$$\begin{aligned} \Delta_C &= \Phi(d_1) \\ \nu_C &= S\phi(d_1)\sigma\sqrt{T-t} \end{aligned}$$

Using Put-Call Parity we find the put Delta Δ_P

$$\begin{aligned} C &= P + S - e^{-r(T-t)}K \\ \implies \frac{\partial}{\partial S}C &= \frac{\partial}{\partial S}\big(P + S - e^{-r(T-t)}K\big) \\ \implies \Delta_C &= \Delta_P + 1 \\ \implies \Delta_P &= \Delta_C - 1 \\ &= \Phi(d_1) - 1 \\ &= -\Phi(-d_1) \end{aligned}$$

and Vega ν_P

$$\begin{aligned} \frac{\partial}{\partial\sigma}C &= \frac{\partial}{\partial\sigma}\big(P + S - e^{-r(T-t)}K\big) \\ \nu_C &= \nu_P \end{aligned}$$

Appendix C Code

C.1 MainQ1.cpp

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

int main() {
   double bisect_a = 0; // left endpoint for starting bisection
   double bisect_b = 1; // right endpoint for starting bisection
   // double NR_x0 = 0.1; // initial guess for newton-raphson

   double S0 = 4.75; // market spot price
   double R = 0.0492; // con't comp. interest rate

   // 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] = {nan("1"),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,nan("1"),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_vols.open("../../data/output/q1_implied_vols.csv"); // initialize csv

   // write csv columns & write data
   outfile_vols << "strike,moneyness,call_imp_vol,put_imp_vol" << std::endl;

   // 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;

      // Newton-Raphson
      // outfile_vols << strikes[i] << "," << strikes[i]/S0 << "," <<
      // CO -> implied_vol_NR(call_asks[i], NR_x0) << "," <<
      //    PO -> implied_vol_NR(put_asks[i], NR_x0) << std::endl;

      // Bisection
      outfile_vols << strikes[i] << "," << strikes[i]/S0 << "," <<
         CO -> impliedVolBisect(call_asks[i], bisect_a, bisect_b) << "," <<
         PO -> impliedVolBisect(put_asks[i], bisect_a, bisect_b) << std::endl;
   }

   outfile_vols.close();
}
```

C.2 MainQ2.cpp

```
#include <iostream>
#include <fstream>
```

```cpp
#include <cmath>
#include <string>
#include <vector>
#include "MathHelp.h"
#include "Asset.h"
#include "CallOption.h"
#include "PutOption.h"
#include "OptionData.h"

using namespace std;

vector<double> concatenateVec(vector<double> A, vector<double> B) {
   // concatenate vector A with vector B
   vector<double> AB;

   AB.reserve( A.size() + B.size() ); // preallocate memory
   AB.insert( AB.end(), A.begin(), A.end() );
   AB.insert( AB.end(), B.begin(), B.end() );

   return AB;
}

vector<vector<double> > cartProd(vector<vector<double> > x, vector<vector<double> > y) {
   // cartesian product of two vectors
   // (well, technically for vectors of vectors for generality)
   int n = x.size();
   int m = y.size();
   vector<vector<double> > out(n + m);

   for (int i = 0; i < x.at(0).size(); i++) {
      for (int j = 0; j < y.at(0).size(); j++) {
         for (int k = 0; k < n; k++) {
            out.at(k).push_back( x.at(k).at(i) );
         }
         int idx = 0;
         for (int k = n; k < (n + m); k++) {
            out.at(k).push_back( y.at(idx).at(j) );
         }
      }
   }
   return out;
}
vector<vector<double> > cartProdN(vector<vector<double> > data) {
   // generalizes to the cartesian product over N vectors
   int d = data.size(); // d dimensions/vectors to perform the cartesian product

   if (d == 1) { // you're killing me here, come on
      return data;
   }
   else { // if the number of dimensions/vectors >= 2
      vector<vector<double> > x1(1);
      vector<vector<double> > x2(1);
      vector<vector<double> > xn(1);

      x1.at(0) = data.at(0);
      x2.at(0) = data.at(1);

      vector<vector<double> > data_cart_prod = cartProd(x1, x2);

      for (int i = 2; i < d; i++) {
         xn.at(0) = data.at(i);
         data_cart_prod = cartProd(data_cart_prod, xn);
      }
      return data_cart_prod;
   }
}

vector<vector<double> > fitVols(double spot, vector<vector<double> > x_obs,
   vector<double> y_obs, vector<double> h_vec, vector<double> min_vec,
   vector<double> max_vec, int N) {
   // h_vec, min_vec, max_vec, and number of x_obs dimensions must agree

   int d = h_vec.size();
   vector<double> dx(d); // granularity of each x dimension to fit
```

```
   vector<vector<double> > x_unobs(d); // matrix of points to smooth over
   // ideally i would have initialized the dimension of each 'column' in the matrix
   // using N_vec, but i don't know how to do so (thus forced to use push_back)

   for (int i = 0; i < d; i++) { // compute dx for each x var
      dx.at(i) = (max_vec.at(i) - min_vec.at(i))/(N - 1);
      x_unobs.at(i).push_back( min_vec.at(i) ); // place the min as the first step
   }

   for (int i = 0; i < d; i++) { // generate matrix of steps, loop over each x var
      for (int j = 1; j < N; j++) { // increment over corresp. dx
         x_unobs.at(i).push_back( x_unobs.at(i).at(j - 1) + dx.at(i) );
      }
   }

   // create the cartesian product of our x coordinates to be smoothed over
   vector<vector<double> > x_unobs_cart_prod = cartProdN(x_unobs);

   vector<vector<double> > data_out;
   vector<double> row(d + 1); // create vector for each 'row' of the data
   vector<double> x_unobs_row(d); // single row of our cartesian product

   for (int i = 0; i < x_unobs_cart_prod.at(0).size(); i++) {
      for (int j = 0; j < x_unobs_cart_prod.size(); j++) {
         row.at(j) = x_unobs_cart_prod.at(j).at(i);
         x_unobs_row.at(j) = x_unobs_cart_prod.at(j).at(i);
      }
      row.at(d) = MathHelp::nadWatEstimator(h_vec, x_unobs_row, x_obs, y_obs);

      data_out.push_back( row ); // append row to data output
   }
   return data_out;
}

vector<vector<double> > fitVols(double spot, vector<double> x_obs, vector<double> y_obs,
   double h, double min, double max, int N) {
   // 1 dimensional wrapper
   // we are storing our 1-D parameters as vectors since we have defined
   // our smoothing fxns to be sufficiently general

   vector<double> h_vec(1); h_vec.at(0) = h; // smoothing parameter
   vector<double> min_vec(1), max_vec(1);
   min_vec.at(0) = min; max_vec.at(0) = max; // min/max x-variable to fit


   // independent variables: place observed x variables in matrix
   vector<vector<double> > x_obs_vec(1);
   x_obs_vec.at(0) = x_obs;

   vector<vector<double> > fitted_vols = fitVols(spot, x_obs_vec,
      y_obs, h_vec, min_vec, max_vec, N);

   return fitted_vols;
}

int main() {
   // ----------------------
   // set up some parameters
   // ----------------------

   double spot = 502.12; // market spot
   string start_date = "2012-02-17"; // t0
   double r = 0.0175; // con't comp. interest rate

   Asset AAPL(spot, r); // create our asset

   double bisect_a = 0; // bisection left endpoint
   double bisect_b = 2; // bisection right endpoint

   // ---------
   // load data
   // ---------

   // filenames to load data
```

```
string filename_mar2012 = "../../data/cleaned/cleaned_mar2012.csv";
string filename_apr2012 = "../../data/cleaned/cleaned_apr2012.csv";
string filename_may2012 = "../../data/cleaned/cleaned_may2012.csv";

// load/create datasets
OptionData mar2012, apr2012, may2012;
mar2012.readCsv(filename_mar2012);
apr2012.readCsv(filename_apr2012);
may2012.readCsv(filename_may2012);

// -----------
// filter data
// -----------

// keep only data with volume >= 100, last >= 0.1
double inf = numeric_limits<double>::infinity();
mar2012.filterLasts(0.1, inf); mar2012.filterVolumes(100, inf);
apr2012.filterLasts(0.1, inf); apr2012.filterVolumes(100, inf);
may2012.filterLasts(0.1, inf); may2012.filterVolumes(100, inf);

// generate maturities & implied vols
mar2012.computeTaus(start_date);
apr2012.computeTaus(start_date);
may2012.computeTaus(start_date);
mar2012.computeImpliedVols(AAPL, bisect_a, bisect_b);
apr2012.computeImpliedVols(AAPL, bisect_a, bisect_b);
may2012.computeImpliedVols(AAPL, bisect_a, bisect_b);

// ---------------------------------------
// write unsmoothed data to csv
// ---------------------------------------

// prepare csv files
ofstream mar_outfile, apr_outfile, may_outfile;
mar_outfile.open("../../data/output/q2_imp_vols_mar.csv"); // initialize csv
apr_outfile.open("../../data/output/q2_imp_vols_apr.csv"); // initialize csv
may_outfile.open("../../data/output/q2_imp_vols_may.csv"); // initialize csv

// write csv columns & write data
mar_outfile << "date,maturity,last,strike,moneyness,type,imp_vol" << endl;
apr_outfile << "date,maturity,last,strike,moneyness,type,imp_vol" << endl;
may_outfile << "date,maturity,last,strike,moneyness,type,imp_vol" << endl;

// loop over march 2012 rows
for (int i = 0; i < mar2012.getLasts().size(); i++) {
   string date = mar2012.getDates().at(i);
   double tau = mar2012.getTaus().at(i);
   double last = mar2012.getLasts().at(i);
   double vol = mar2012.getVolumes().at(i);
   double strike = mar2012.getStrikes().at(i);
   string type = mar2012.getTypes().at(i);
   double imp_vol = mar2012.getImpliedVols().at(i);

   mar_outfile << date << "," << tau << "," << last << "," << strike << "," <<
      spot/strike << "," << type << "," << imp_vol << endl;
}

// loop over april 2012 rows
for (int i = 0; i < apr2012.getLasts().size(); i++) {
   string date = apr2012.getDates().at(i);
   double tau = apr2012.getTaus().at(i);
   double last = apr2012.getLasts().at(i);
   double vol = apr2012.getVolumes().at(i);
   double strike = apr2012.getStrikes().at(i);
   string type = apr2012.getTypes().at(i);
   double imp_vol = apr2012.getImpliedVols().at(i);

   apr_outfile << date << "," << tau << "," << last << "," << strike << "," <<
      spot/strike << "," << type << "," << imp_vol << endl;
}

// loop over may 2012 rows
for (int i = 0; i < may2012.getLasts().size(); i++) {
   string date = may2012.getDates().at(i);
```

```
    double tau = may2012.getTaus().at(i);
    double last = may2012.getLasts().at(i);
    double vol = may2012.getVolumes().at(i);
    double strike = may2012.getStrikes().at(i);
    string type = may2012.getTypes().at(i);
    double imp_vol = may2012.getImpliedVols().at(i);

    may_outfile << date << "," << tau << "," << last << "," << strike << "," <<
       spot/strike << "," << type << "," << imp_vol << endl;
}

mar_outfile.close(); apr_outfile.close(); may_outfile.close();

// -------------
// 1-D Smoothing
// -------------
double h_1d = 0.075; // smoothing parameter
double min_1d = 0.5; // min moneyness to fit
double max_1d = 1.5; // max moneyness to fit
int N_1d = 200; // granularity of moneyness steps to fit

// --------------
// Fit March Data
// --------------
// prepare march moneyness
vector<double> moneyness_mar(mar2012.getStrikes().size());
for (int i = 0; i < moneyness_mar.size(); i++) {
    moneyness_mar.at(i) = spot / mar2012.getStrikes().at(i);
}

// fit vols
vector<vector<double> > fitted_vols_1d_mar = fitVols(spot, moneyness_mar,
    mar2012.getImpliedVols(), h_1d, min_1d, max_1d, N_1d);

// --------------
// Fit April Data
// -------------- // prepare april moneyness
vector<double> moneyness_apr(apr2012.getStrikes().size());
for (int i = 0; i < moneyness_apr.size(); i++) {
    moneyness_apr.at(i) = spot / apr2012.getStrikes().at(i);
}

// fit vols
vector<vector<double> > fitted_vols_1d_apr = fitVols(spot, moneyness_apr,
    apr2012.getImpliedVols(), h_1d, min_1d, max_1d, N_1d);

// --------------
// Fit May Data
// -------------- // prepare may moneyness
vector<double> moneyness_may(may2012.getStrikes().size());
for (int i = 0; i < moneyness_may.size(); i++) {
    moneyness_may.at(i) = spot / may2012.getStrikes().at(i);
}

// fit vols
vector<vector<double> > fitted_vols_1d_may = fitVols(spot, moneyness_may,
    may2012.getImpliedVols(), h_1d, min_1d, max_1d, N_1d);

// -------------
// Write to CSV
// -------------
// prepare csv
ofstream fitted_1d_mar_outfile, fitted_1d_apr_outfile, fitted_1d_may_outfile;
fitted_1d_mar_outfile.open("../../data/output/q2_fitted_1d_mar_data.csv");
fitted_1d_apr_outfile.open("../../data/output/q2_fitted_1d_apr_data.csv");
fitted_1d_may_outfile.open("../../data/output/q2_fitted_1d_may_data.csv");

// write headers
fitted_1d_mar_outfile << "moneyness,fitted_vol" << endl;
fitted_1d_apr_outfile << "moneyness,fitted_vol" << endl;
fitted_1d_may_outfile << "moneyness,fitted_vol" << endl;

// write rows
for (int i = 0; i < fitted_vols_1d_mar.size(); i++) { // loop over march rows
```

```
   fitted_1d_mar_outfile << fitted_vols_1d_mar.at(i).at(0) << "," <<
      fitted_vols_1d_mar.at(i).at(1) << endl;
}
for (int i = 0; i < fitted_vols_1d_apr.size(); i++) { // loop over april rows
   fitted_1d_apr_outfile << fitted_vols_1d_apr.at(i).at(0) << "," <<
      fitted_vols_1d_apr.at(i).at(1) << endl;
}
for (int i = 0; i < fitted_vols_1d_may.size(); i++) { // loop over may rows
   fitted_1d_may_outfile << fitted_vols_1d_may.at(i).at(0) << "," <<
      fitted_vols_1d_may.at(i).at(1) << endl;
}

// close files
fitted_1d_mar_outfile.close();
fitted_1d_apr_outfile.close();
fitted_1d_may_outfile.close();

// -------------
// 2-D Smoothing
// -------------
vector<double> h_2d(2); // smoothing parameters
h_2d.at(0) = 0.05;
h_2d.at(1) = 0.05;
vector<double> min_2d(2), max_2d(2);
min_2d.at(0) = spot/770; min_2d.at(1) = 29.0/365; // min moneyness/time to fit
max_2d.at(0) = spot/330; max_2d.at(1) = 90.0/365; // max moneyness/time to fit
int N_2d = 50;

// concatenate our vectors of strikes for use in the smoother
vector<double> mar_apr_strikes = concatenateVec(mar2012.getStrikes(),
   apr2012.getStrikes());
vector<double> strikes = concatenateVec(mar_apr_strikes, may2012.getStrikes());

// compute moneyness
vector<double> moneyness(strikes.size());
for (int i = 0; i < strikes.size(); i++) {
   moneyness.at(i) = spot/strikes.at(i);
}

// concatenate our vectors of maturities for use in the smoother
vector<double> mar_apr_taus = concatenateVec(mar2012.getTaus(),
   apr2012.getTaus());
vector<double> taus = concatenateVec(mar_apr_taus, may2012.getTaus());

// independent variable: place all x variables in a matrix
vector<vector<double> > x_obs(2);
x_obs.at(0) = moneyness;
x_obs.at(1) = taus;

// dependent variable: create vector of implied vols
vector<double> mar_apr_vols = concatenateVec(mar2012.getImpliedVols(),
   apr2012.getImpliedVols());
vector<double> vols = concatenateVec(mar_apr_vols, may2012.getImpliedVols());

// --------
// Fit Data
// --------
vector<vector<double> > fitted_vols_2d = fitVols(spot, x_obs, vols,
   h_2d, min_2d, max_2d, N_2d);

// -------------
// Write to CSV
// -------------
// prepare to write 2d fitted data to csv
ofstream fitted_2d_outfile;
fitted_2d_outfile.open("../../data/output/q2_fitted_2d_data.csv");
fitted_2d_outfile << "moneyness,maturity,fitted_vol" << endl;

// write rows to csv
for (int i = 0; i < fitted_vols_2d.size(); i++) {
   fitted_2d_outfile << fitted_vols_2d.at(i).at(0) << "," <<
      fitted_vols_2d.at(i).at(1) << "," <<
      fitted_vols_2d.at(i).at(2) << endl;
}
```

```
   fitted_2d_outfile.close();
}
```

C.3 MainQ4.cpp

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <string>
#include <vector>
#include "MathHelp.h"
#include "Asset.h"
#include "CallOption.h"
#include "PutOption.h"
#include "OptionData.h"

using namespace std;

vector<vector<double> > readFittedVols(string filename) {
   string fileline;
   vector<vector<double> > data_out(3);

   ifstream data;
   data.open(filename);

   if (!data.is_open()) { // abort program if file doesn't exist
      exit(EXIT_FAILURE); // EXIT_FAILRuE comes from <cstdlib>
   }

   data >> fileline; // read first line (first line is header data)
   data >> fileline; // read next line
   string last_line;
   vector<string> tmp_line; // temporary storage for each line

   // ASSUMES NO CONSECUTIVE LINES OUGHT TO BE INDENTICAL
   // this solves an issue where the final line was being read twice
   while (!data.eof() || (fileline != last_line)) { // loop over each line of the csv
      last_line = fileline;

      boost::split(tmp_line, fileline, boost::is_any_of(",")); // split line by comma

      data_out.at(0).push_back( stod(tmp_line.at(0)) );
      data_out.at(1).push_back( stod(tmp_line.at(1)) );
      data_out.at(2).push_back( stod(tmp_line.at(2)) );

      data >> fileline; // read next line
   }
   return data_out;
}

int main() {
   // ----------------------
   // set up some parameters
   // ----------------------

   double spot = 502.12; // market spot
   double r = 0.0175; // con't comp. interest rate
   double strike_wanted = 572.50;
   double maturity_wanted = 60.0/365;
   double moneyness_wanted = spot/strike_wanted;

   // ---------
   // load data
   // ---------

   // filenames to load data
   string fitted_vols_infile = "../../data/output/q2_fitted_2d_data.csv";

   // load data
   vector<vector<double> > fitted_vols = readFittedVols(fitted_vols_infile);
```

```
   // --------------------------------
   // find closest fit to desired data
   // --------------------------------
   double closest_moneyness, closest_maturity, sigma_hat,
      tmp_moneyness_err, tmp_maturity_err;

   double moneyness_err = numeric_limits<double>::infinity();
   double maturity_err = numeric_limits<double>::infinity();

   for (int i = 0; i < fitted_vols.at(0).size(); i++) { // loop over all values to find
      // the one with the lowest error in both dimensions
      tmp_moneyness_err = fabs(fitted_vols.at(0).at(i) - moneyness_wanted);
      tmp_maturity_err = fabs(fitted_vols.at(1).at(i) - maturity_wanted);

      if ( (moneyness_err >= tmp_moneyness_err) & (maturity_err >= tmp_maturity_err) ) {
         moneyness_err = tmp_moneyness_err;
         maturity_err = tmp_maturity_err;

         closest_moneyness = fitted_vols.at(0).at(i);
         closest_maturity = fitted_vols.at(1).at(i);

         sigma_hat = fitted_vols.at(2).at(i);
      }
   }

   // ------------
   // price option
   // ------------
   Asset AAPL(spot, r, sigma_hat); // create our asset with the sigma_hat
   CallOption C(AAPL, strike_wanted, maturity_wanted); // create our option
   CallOption* CO = &C; // create a pointer to the option

   double bs_price = CO -> blackScholesPrice();
   double bs_hedge_ratio = CO -> blackScholesDelta();

   cout << closest_moneyness << "\t\t" << closest_maturity << "\t\t" << sigma_hat << endl;

   cout << bs_price << endl; // price
   cout << bs_hedge_ratio << endl; // hedge ratio
}
```

C.4 OptionData.h

```
/*
   Header file for a OptionData structure
   Very rudimentary data type to store our option data
*/
#ifndef OPTIONDATA_H
#define OPTIONDATA_H

#include "Option.h"
#include <ctime> // calculate time between dates
#include <string>
#include <vector>
#include <boost/algorithm/string.hpp>
#include "Asset.h"

using namespace std;

class OptionData {
   private:
      vector<string> dates;
      vector<double> lasts;
      vector<double> volumes;
      vector<double> strikes;
      vector<string> types;

      vector<double> taus;
      vector<double> implied_vols;
```

```
      vector<int> stringToDate(string string_date);
      tm makeTm(vector<int> date);
      double stringDateDiff(string date_a, string date_b);

   public:
      OptionData(); // default constructor
      OptionData(vector<string> dates_in, vector<double> lasts_in,
         vector<double> volumes_in, vector<double> strikes_in, vector<string> types_in);
      vector<string> getDates();
      vector<double> getLasts();
      vector<double> getVolumes();
      vector<double> getStrikes();
      vector<string> getTypes();
      vector<double> getTaus();
      vector<double> getImpliedVols();

      void readCsv(string filename);
      void filterVolumes(double min, double max);
      void filterLasts(double min, double max);

      void computeTaus(string start_date);
      void computeImpliedVols(Asset A, double bisect_a, double bisect_b);
};
#endif
```

C.5 OptionData.cpp

```
#include "OptionData.h"
#include "CallOption.h"
#include "PutOption.h"
#include <iostream>
#include <fstream>

using namespace std;

OptionData::OptionData() {}
OptionData::OptionData(vector<string> dates_in, vector<double> lasts_in,
   vector<double> volumes_in, vector<double> strikes_in, vector<string> types_in) {

   dates = dates_in;
   lasts = lasts_in;
   volumes = volumes_in;
   strikes = strikes_in;
   types = types_in;
}
vector<std::string> OptionData::getDates() {
   return dates;
}
vector<double> OptionData::getLasts() {
   return lasts;
}
vector<double> OptionData::getVolumes() {
   return volumes;
}
vector<double> OptionData::getStrikes() {
   return strikes;
}
vector<string> OptionData::getTypes() {
   return types;
}
vector<double> OptionData::getTaus() {
   return taus;
}
vector<double> OptionData::getImpliedVols() {
   return implied_vols;
}

void OptionData::readCsv(string filename) {
   string fileline;

   ifstream data;
```

```
   data.open(filename);

   if (!data.is_open()) { // abort program if file doesn't exist
      exit(EXIT_FAILURE); // EXIT_FAILRuE comes from <cstdlib>
   }

   data >> fileline; // read first line (first line is header data)
   data >> fileline; // read next line
   string last_line;
   vector<string> tmp_line; // temporary storage for each line

   // ASSUMES NO CONSECUTIVE LINES OUGHT TO BE INDENTICAL
   // this solves an issue where the final line was being read twice
   while (!data.eof() || (fileline != last_line)) { // loop over each line of the csv
      last_line = fileline;

      boost::split(tmp_line, fileline, boost::is_any_of(",")); // split line by comma

      dates.push_back(tmp_line[0]); // place date in its vector

      // some cells are NA for options that do not exist
      // we must handle these cases
      if (tmp_line[1] != "NA") { // place last in its vector
         lasts.push_back(stod(tmp_line[1]));
      } else {
         lasts.push_back(nan("1"));
      }

      if (tmp_line[2] != "NA") { // place vol in its vector
         volumes.push_back(stod(tmp_line[2]));
      } else {
         volumes.push_back(nan("1"));
      }

      if (tmp_line[3] != "NA") { // place strike in its vector
         strikes.push_back(stod(tmp_line[3]));
      } else {
         volumes.push_back(nan("1"));
      }

      // remove the prepended & appended quotations to the option type
      string tp = tmp_line[4].substr(1, tmp_line[4].length() - 2);
      types.push_back(tp); // type

      data >> fileline; // read next line
   }
}

void OptionData::filterVolumes(double min, double max) {
   vector<string> new_dates, new_types;
   vector<double> new_lasts, new_volumes, new_strikes;

   for (int i = 0; i < volumes.size(); i++) {
      if ((volumes.at(i) >= min) & (volumes.at(i) <= max)) {
         new_dates.push_back( dates.at(i) );
         new_lasts.push_back( lasts.at(i) );
         new_volumes.push_back( volumes.at(i) );
         new_strikes.push_back( strikes.at(i) );
         new_types.push_back( types.at(i) );
      }
   }

   dates = new_dates;
   lasts = new_lasts;
   volumes = new_volumes;
   strikes = new_strikes;
   types = new_types;
}
void OptionData::filterLasts(double min, double max) {
   vector<string> new_dates, new_types;
   vector<double> new_lasts, new_volumes, new_strikes;

   for (int i = 0; i < lasts.size(); i++) {
      if ((lasts.at(i) >= min) & (lasts.at(i) <= max)) {
```

```
            new_dates.push_back( dates.at(i) );
            new_lasts.push_back( lasts.at(i) );
            new_volumes.push_back( volumes.at(i) );
            new_strikes.push_back( strikes.at(i) );
            new_types.push_back( types.at(i) );
         }
      }

      dates = new_dates;
      lasts = new_lasts;
      volumes = new_volumes;
      strikes = new_strikes;
      types = new_types;
}

void OptionData::computeTaus(string start_date) {
      for (int i = 0; i < dates.size(); i++) {
         taus.push_back( stringDateDiff(start_date, dates.at(i)) );
      }
}

void OptionData::computeImpliedVols(Asset A, double bisect_a, double bisect_b) {
      for (int i = 0; i < lasts.size(); i++) {
         double imp_vol;

         if (types.at(i) == "Call") {
            CallOption C(A, strikes.at(i), taus.at(i));
            CallOption* CO = &C;

            imp_vol = CO -> impliedVolBisect(lasts.at(i), bisect_a, bisect_b);
         }
         else if (types.at(i) == "Put") {
            PutOption P(A, strikes.at(i), taus.at(i));
            PutOption* PO = &P;

            imp_vol = PO -> impliedVolBisect(lasts.at(i), bisect_a, bisect_b);
         }
         else {
            cout << "WARNING in compute_implied_vol: Neither Call nor Put selected." << endl;
         }

         implied_vols.push_back(imp_vol);
      }
}

vector<int> OptionData::stringToDate(string str_date) {
      // a string of format "YYYY-MM-DD" to a date
      vector<string> v_str_date;
      boost::split(v_str_date, str_date, boost::is_any_of("-"));

      vector<int> v_int_date(3);
      v_int_date[0] = stod(v_str_date[0]); // year
      v_int_date[1] = stod(v_str_date[1]); // month
      v_int_date[2] = stod(v_str_date[2]); // day

      return v_int_date;
}

tm OptionData::makeTm(vector<int> date) {
      // make tm structure representing a date
      int year = date.at(0);
      int month = date.at(1);
      int day = date.at(2);

       std::tm tm = {0};
       tm.tm_year = year - 1900; // years count from 1900
       tm.tm_mon = month - 1;  // months count from January=0
       tm.tm_mday = day;       // days count from 1

       return tm;
}
double OptionData::stringDateDiff(string date_a, string date_b) {
      // computes time difference in years
      vector<int> int_date_a = stringToDate(date_a);
```

```
   vector<int> int_date_b = stringToDate(date_b);

   tm tm_a = makeTm(int_date_a);
   tm tm_b = makeTm(int_date_b);

   time_t time_a = mktime(&tm_a);
   time_t time_b = mktime(&tm_b);

   double tau = difftime(time_b, time_a)/(60 * 60 * 24 * 365);

   return tau;
}
```

C.6 MathHelp.h

```
/*
   Header file for MathHelp
   Contains some useful functions for our purposes
*/
#ifndef MATHHELP_H
#define MATHHELP_H

#include <vector>

using namespace std;

class MathHelp {
   public:
      static double stdNormPdf(double x);
      static double stdNormCdf(double x);
      static double kernelSmoother(vector<double> h, vector<double> data_point);
      static double nadWatEstimator(std::vector<double> h_vec,
         vector<double> x_unobs, vector<vector<double> > x_obs, vector<double> y_obs);
};
#endif
```

C.7 MathHelp.cpp

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

using namespace std;

double MathHelp::stdNormPdf(double x) {
   return 1/sqrt(2 * M_PI) * exp(-1/2.0 * pow(x, 2));
}
double MathHelp::stdNormCdf(double x_in) {
   // we notice that the approximation is much better for positive x
   // use symmetry of the normal CDF to compute the negative x values
   double x = fabs(x_in);

   double b0 = 0.2316419;
   double b1 = 0.31938530;
   double b2 = -0.356563782;
   double b3 = 1.781477937;
   double b4 = -1.821255978;
   double b5 = 1.330274429;
   double t = 1/(1 + b0 * x);

   double y = 1 - stdNormPdf(x) * (b1*t + b2*pow(t,2) + b3*pow(t,3) + b4*pow(t,4) + b5*pow(t,5));

   if (x_in < 0) {
      y = 1 - y;
   }

   return y;
}
```

```
double MathHelp::kernelSmoother(vector<double> h, vector<double> data_point) {
   // normal density kernel smoother for arbitrary dimensional point
   int dim = data_point.size();

   double K = 1/(h.at(0) * sqrt(2 * M_PI)) * exp(-1/2.0 *
      pow(data_point.at(0)/h.at(0), 2));

   for (int i = 1; i < dim; i++) {
      K *= 1/(h.at(i) * sqrt(2 * M_PI)) * exp(-1/2.0 * pow(data_point.at(i)/h.at(i), 2));
   }

   return K;
}
double MathHelp::nadWatEstimator(vector<double> h_vec, vector<double> x_unobs,
   vector<vector<double> > x_obs, vector<double> y_obs) {

   if (h_vec.size() != x_unobs.size()) {
      cout << "WARNING in nad_wat_estimator: Dimensions to not agree" << endl;
   }
   int n = x_unobs.size(); // number of x variables to smooth over

   double num = 0;
   double den = 0;

   vector<double> data_point(n); // stores n dimensional difference of unobs - obs

   for (int i = 0; i < x_obs.at(0).size(); i++) { // loop over observed data
      for (int j = 0; j < n; j++) { // loop over dimensions to store difference
         data_point.at(j) = x_unobs.at(j) - x_obs.at(j).at(i);
      }

      num += kernelSmoother(h_vec, data_point) * y_obs.at(i);
      den += kernelSmoother(h_vec, data_point);
   }

   return num/den;
}
```

C.8 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
```

C.9 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;
}
```

C.10 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'
*/
#ifndef OPTION_H
#define OPTION_H

#include "Asset.h"
#include <cmath>

class Option {
   protected:
      Asset A; // underlying asset
      double strike; // strike price
      double tau; // time to expiry
      double d1;
      double d2;

   public:
      Option(); // default constructor
      Option(Asset A_in, double strike_in, double tau_in);
      void updateOption();
      void setAsset(Asset A_in);
      Asset getAsset();
      void setStrike(double strike_in);
      double getStrike();
      void setTau(double tau_in);
      double getTau();
      void setD1(double d1_in);
      double getD1();
      void setD2(double d2_in);
      double getD2();

      double computeD1(double S, double K, double tau, double R, double sigma);
      double computeD2(double S, double K, double tau, double R, double sigma);
      double blackScholesVega();
      double blackScholesD2K2();
```

```
    double impliedVolNR(double V_obs, double x_0, double eps = pow(10,-6));
    double impliedVolBisect(double V_obs, double a, double b, double eps = pow(10,-6));

    virtual double payoff() = 0;
    virtual double payoff(double some_strike) = 0;
    virtual double blackScholesPrice() = 0;
    virtual double blackScholesDelta() = 0;
    virtual double blackScholesDK() = 0;
    virtual double blackScholesDT() = 0;
};
#endif
```

C.11 Option.cpp

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

Option::Option() {}
Option::Option(Asset A_in, double strike_in, double tau_in) {
   A = A_in;
   strike = strike_in;
   tau = tau_in;

   double spot = A.getSpot();
   double sigma = A.getSigma();
   double R = A.getR();

   d1 = computeD1(spot, strike, tau, R, sigma);
   d2 = computeD2(spot, strike, tau, R, sigma);
}
void Option::updateOption() {
   double spot = A.getSpot();
   double sigma = A.getSigma();
   double R = A.getR();

   d1 = computeD1(spot, strike, tau, R, sigma);
   d2 = computeD2(spot, strike, tau, R, sigma);
}
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;
}
void Option::setD1(double d1_in) {
   d1 = d1_in;
}
double Option::getD1() {
   return d1;
}
void Option::setD2(double d2_in) {
   d2 = d2_in;
}
double Option::getD2() {
   return d2;
}
double Option::computeD1(double S, double K, double tau, double R, double sigma) {
```

```
   return 1/(sigma * sqrt(tau)) * (log(S/K) + (R + 0.5 * pow(sigma,2)) * tau);
}
double Option::computeD2(double S, double K, double tau, double R, double sigma) {
   return computeD1(S, K, tau, R, sigma) - sigma * sqrt(tau);
}

double Option::blackScholesVega() {
   if (A.getSigma() == 0) {
      std::cout << "WARNING in black_scholes_vega(): sigma = 0, is this correct?" << std::endl;
      return 0;
   }
   double spot = A.getSpot();
   return spot * MathHelp::stdNormPdf(d1) * sqrt(tau);
}
double Option::blackScholesD2K2() { // second derivative wrt strike
   double sigma = A.getSigma();
   double r = A.getR();

   return MathHelp::stdNormPdf(d2) * exp(-r * tau) / ( strike * sigma * sqrt(tau));
}

double Option::impliedVolNR(double V_obs, double x_0, double eps) {
   if (V_obs != V_obs) { // exploit the fact that nan is not equal to itself
      return std::nan("1"); // this is for simplicity when dealing with options w/o asks
   }

   double sigma_guess = x_0;

   setAsset(Asset(A.getSpot(), A.getR(), sigma_guess));
   updateOption();

   double V_guess = blackScholesPrice();

   while(fabs(V_guess - V_obs) >= eps) {
      double f = blackScholesPrice() - V_obs;
      double f_prime = blackScholesVega();

      if (f_prime == 0) {
         std::cout << "WARNING in newton_raphson: computed vega = 0, divide by zero" << std::endl;
         break;
      }

      sigma_guess = sigma_guess - f/f_prime;

      setAsset(Asset(A.getSpot(), A.getR(), sigma_guess));
      updateOption();

      V_guess = blackScholesPrice();
   }
   return sigma_guess;
}
double Option::impliedVolBisect(double V_obs, double a, double b, double eps) {
   if (V_obs != V_obs) { // exploit the fact that nan is not equal to itself
      return std::nan("1"); // this is for simplicity when dealing with options w/o asks
   }

   double sigma_guess = (a + b)/2;

   setAsset(Asset(A.getSpot(), A.getR(), sigma_guess));
   updateOption();

   double V_guess = blackScholesPrice();

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

      setAsset(Asset(A.getSpot(), A.getR(), sigma_guess));
```

```
      updateOption();

      V_guess = blackScholesPrice();

   }
   return sigma_guess;
}
```

C.12 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
*/
#ifndef CALLOPTION_H
#define CALLOPTION_H

#include "Option.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);
      double blackScholesPrice();
      double blackScholesDelta();
      double blackScholesDK();
      double blackScholesDT();
};
#endif
```

C.13 CallOption.cpp

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

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);
}
double CallOption::blackScholesPrice() {
   double R = A.getR();
   double spot = A.getSpot();

   return MathHelp::stdNormCdf(d1) * spot - MathHelp::stdNormCdf(d2) * strike *
      exp(-R * tau);
}
double CallOption::blackScholesDelta() {
   return MathHelp::stdNormCdf(d1);
}
double CallOption::blackScholesDK() { // derivative wrt strike
   double r = A.getR();
   return -MathHelp::stdNormCdf(d2) * exp(-r * tau);
}
double CallOption::blackScholesDT() { // derivative wrt T
   double r = A.getR();
   double sigma = A.getSigma();

   return strike * exp(-r * tau) * (r * MathHelp::stdNormCdf(d2) +
      sigma * MathHelp::stdNormPdf(d2) / (2 * sqrt(tau)) );
```

```
}
```

C.14 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
*/
#ifndef PUTOPTION_H
#define PUTOPTION_H

#include "Option.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);
      double blackScholesPrice();
      double blackScholesDelta();
      double blackScholesDK();
      double blackScholesDT();
};
#endif
```

C.15 PutOption.cpp

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

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);
}
double PutOption::blackScholesPrice() {
   double R = A.getR();
   double spot = A.getSpot();

   return -MathHelp::stdNormCdf(-d1) * spot + MathHelp::stdNormCdf(-d2) * strike *
      exp(-R * tau);
}
double PutOption::blackScholesDelta() {
   return -MathHelp::stdNormCdf(-d1);
}
double PutOption::blackScholesDK() { // derivative wrt strike
   double r = A.getR();
   return MathHelp::stdNormCdf(-d2) * exp(-r * tau);
}
double PutOption::blackScholesDT() { // derivative wrt T
   double r = A.getR();
   double sigma = A.getSigma();

   return strike * exp(-r * tau) * (-r * MathHelp::stdNormCdf(-d2) +
      sigma * MathHelp::stdNormPdf(d2) / (2 * sqrt(tau)) );
}
```