A Numerical Exploration of the Local Volatility Model for Option Pricing*

Francisco Rivera† Jiafeng Chen‡

December 18, 2017

1 Introduction

Option pricing is a growing field ever since the groundbreaking work of Black and Scholes (1973). In the Black-Scholes model, the underlying security is assumed to follow a geometric Brownian motion—a process with constant volatility. Such an assumption, while mathematically elegant, does not agree with empirical observations. To remedy this weakness, Dupire (1997) developed the *local volatility model*, where volatility is assumed to be a function of price and time. Dupire (1997)'s equation (2.9) calculates local volatility analytically from option prices. However, in order to put Dupire (1997)'s result into practice, we need to estimate local volatility functions from data and to predict option prices from local volatility functions in a numerically stable and statistically robust manner.

This report proposes and tests two approaches of inference and prediction and analyze the errors both in a theoretical manner and through numerical experiments. We estimate Dupire's equation via finite difference approximations of the derivatives and via local quadratic approximations of the option price surface to estimate derivatives. We then use Monte Carlo methods to price options given a local volatility surface. We find that both approaches generate a similar magnitude of errors. We find that the errors are relatively small for at-the-money options, but are large for deep-money options.

This report proceeds as follows. Section 2 introduces some background on option pricing, local volatility theory, and Dupire's equation. Section 3 outlines our approach to pricing options given known local volatility functions or estimates thereof and discusses theoretical results regarding the errors of these methods. Section 4 describes our approach to fit local volatility functions given observed option price data, and discusses some theoretical difficulties regarding error analysis. Section 5 describes the procedure and results of our numerical experiments testing the approach outlined in the previous sections. Section 6 discusses the results and concludes.

*https://github.com/frtennis1/am205-options

†Harvard College, frivera@college.harvard.edu

‡Harvard College, jiafengchen@college.harvard.edu

2 Background

2.1 Options Terminology

Before delving into the theoretical and numerical results, we briefly summarize options terminology. An option is a *derivative* on some asset, henceforth called the *underlying*—i.e. it's value is *derived* from the value of the underlying. The owner of a call (resp. put) option has the right but not the obligation to buy (resp. sell) the underlying asset at a given price at some date in the future.

The price at which the holder of the option can buy/sell is called the *strike price*, denoted K. Invoking the right to buy or sell is called *exercising* the option. The last time at which the holder can exercise is called the *expiry*, denoted as time T. The value of the underlying asset at time t is denoted as S_t (and sometimes the subscript is dropped whenever it is implied).

The payoff of a call option at expiry is thus given by,

$$\max(S_T - K, 0) \tag{2.1}$$

because the owner will only exercise it if the underlying is worth more than the strike price. This function gives rise the characteristic "hockey-stick" payoff diagram of an option at expiry,

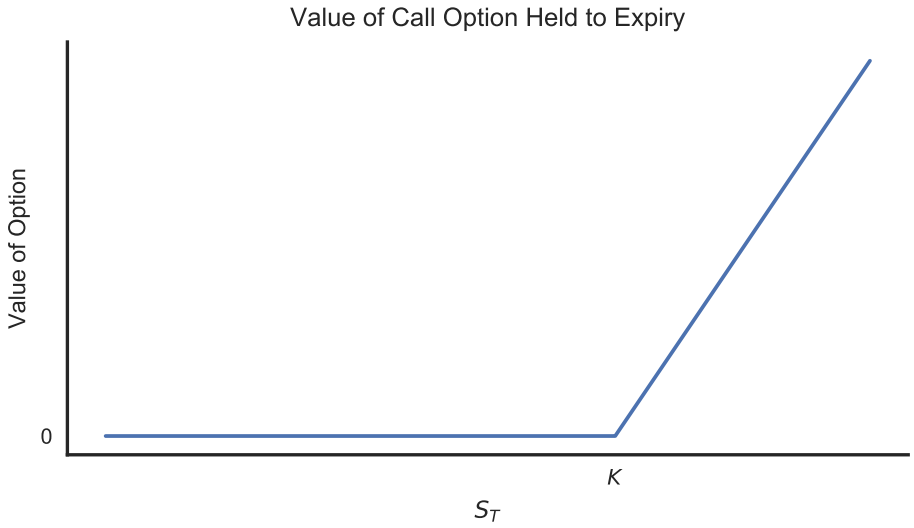

Figure 1: Payoff of call option

The payoff of a put option is very similar to equation 2.1,

$$\max(K - S_T, 0) \tag{2.2}$$

because the owner of the option will only exercise it if the underlying is worth less than the strike price, and the payoff curve looks like its mirror image,

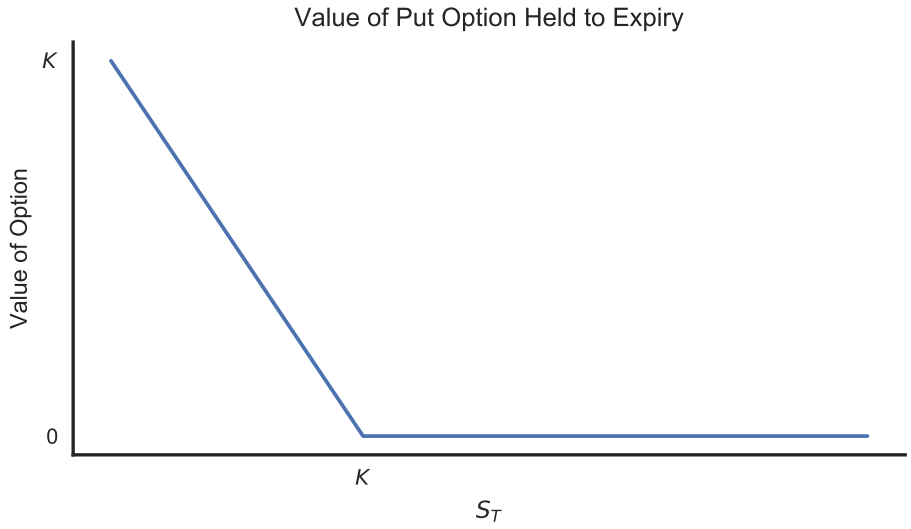

Figure 2: Payoff of put option

2.2 Risk-Neutral Pricing

The value of an option at expiry is easily observable and displayed in the figures in Section 2.1. However, before time T, the value of S_T is a random variable. Asset pricing (change of measure) theorems (Cochrane, 2009) allow us to write the value of a call option as the discounted expectation of the payoff under what we call the *risk-neutral distribution*,

$$\frac{C(S_0, K, T)}{e^{-rT}} = \int_K^\infty (S - K) \underbrace{\phi(S, T; S_0)}_{\text{risk-neutral PDF}} dS \tag{2.3}$$

However, in order to make any progress beyond this, we need to make a further assumption about what this risk-neutral distribution looks like.

2.3 Black-Scholes Model

The Black-Scholes model makes one such assumption by writing how the asset diffuses over time. In particular, the Black-Scholes model treates the asset price as a geometric Brownian motion represented by the following stochastic difference equation:

$$\frac{dS}{S} = rdt + \sigma dW \tag{2.4}$$

where W is Brownian motion and σ is a constant value called the *implied volatility*.

This forces the risk-neutral distribution to be log-normal and makes finding the price analytically tractable. In particular, since the only pricing input into this model that we do not observe is the implied volatility, we can quote price as a function of implied volatility (e.g. an option is said to cost $\sigma = 16\%$ if its price is consistent with the price the Black-Scholes model would predict if $\sigma = 16\%$).

Thus, Black-Scholes predicts that if σ is indeed a constant, then for any option written on the same underlying (regardless of strike price or expiry), that the implied volatility will be (approximately) constant. However, this is directly counterfactual to the options prices that we observe. For example, looking at the prices of AAPL options that expire on April 2018 retrieved on December 17th, 2017 from Yahoo Finance, we get the following curve

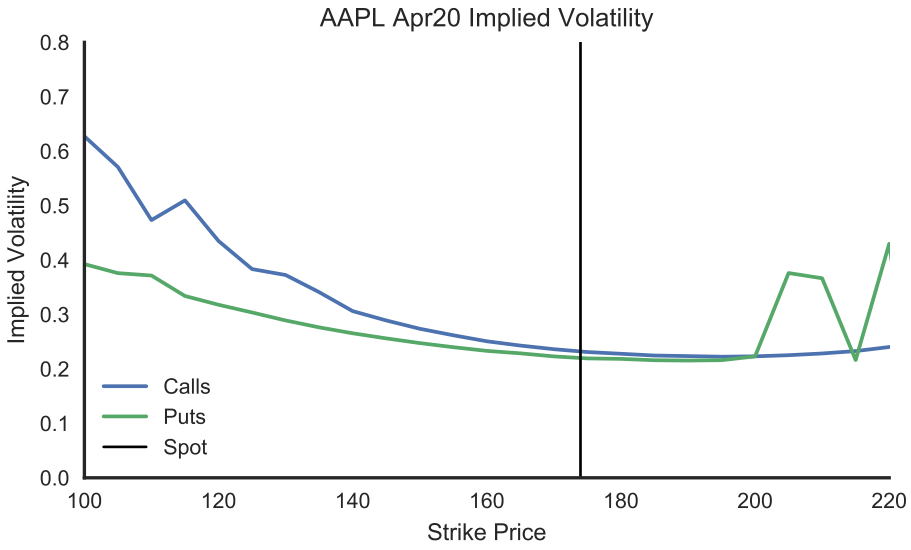

Figure 3: Implied volatility from AAPL options.

While the data is noisy, particularly for deep-in-the money[1] puts which are hardly ever traded, the graph is unmistakably not constant. Moreover, AAPL is not an exception: most graphs of implied volatility versus strike have a similar shape. These deviations from Black-Scholes also correspond to intuitive qualitative ideas. If the price of AAPL has just plunged 50%, it is palatable to think there is a lot of investor uncertainty, and that the future price of AAPL will diffuse with higher volatility than if it is just up 5%. Thus, the assumption of constant σ is suspect.

2.4 Local Volatility Theory

The *local-volatility model* directly responds this shortcoming of the Black-Scholes model by rewriting the diffusion equation as,

$$\frac{dS}{S} = rdt + \sigma(S,t)dW \tag{2.5}$$

such that the volatility is no longer a global constant, but rather a function of spot-price and time. We refer to this function as the *local volatility function*, and the value of the function at any given point as the *local volatility*. Note of course that in the special case when σ is a constant function, we have simply recovered the Black-Scholes model. We will make use of this fact when we need to confirm the numerical results of the local volatility model against theoretical closed-forms.

[1] A call (resp. put) option is said to be *in-the-money* when $S_0 > K$ (resp. $S_0 < K$), *out-of-the-money* when $S_0 < K$ (resp. $S_0 > K$), and at the money if $S_0 = K$.

In the general case, though, we care about what the local volatility surface says about prices and vice-versa. To this end, we can invoke Dupire (1997). The first thing to note is that we can differentiate our risk-neutral pricing formula with respect to K twice to get that,

$$e^{-rT}\phi(K,T;S_0) = \frac{\partial^2 C}{\partial K^2}. \tag{2.6}$$

Moreover, because we have a diffusion rule for the underlying S_t, the probability distributions must satisfy the *Fokker-Planck equation* (Risken, 1996),[2]

$$\frac{\partial C}{\partial T} = \frac{1}{2}e^{rT}\sigma^2 K^2 \frac{\partial^2 C}{\partial K^2} \tag{2.7}$$

Rearranging gives us Dupire's equation,

$$\sigma^2(K,T,S_0) = \frac{\frac{\partial C}{\partial T}e^{-rT}}{\frac{1}{2}K^2\frac{\partial^2 C}{\partial K^2}}. \tag{2.8}$$

We can also rewrite this assuming 0 interest rates, which we will assume going forward for simplicity,

$$\sigma^2(K,T,S_0) = \frac{\frac{\partial C}{\partial T}}{\frac{1}{2}K^2\frac{\partial^2 C}{\partial K^2}}. \tag{2.9}$$

There are a couple notable takeaways from this equation. First of all, if we fully and perfectly observed a continuum of option prices for all strikes and expiries, we would be able to uniquely determine the local volatility surface. Moreover, we can do this *regardless* of what the option prices are[3]. This means that unlike Black-Scholes, the local volatility model can perfectly fit what we see, and we should be apprehensive of overfitting.

3 Monte Carlo Pricing

In this section, we will concern ourselves with what we can do once we have the local volatility surface. In particular, our main objective will be to price options. Note that the prices may exist in the market (e.g. as a sanity check for our model), or they may not (e.g. to price options with expiries that are not quoted, giving our model predictive power).

One way to do this is by realizing that the price is an expectation under the risk-neutral distribution of the pay-off random variable (as in Equation 2.3). Thus, if we can sample from this distribution, we can invoke the law of large numbers and get an estimate for its average with sufficient samples.

In Monte Carlo pricing, we draw from the distribution of S_T and compute option prices. We start with the stochastic difference equation[4]

$$dS_t = S_t\sigma(S_t,t)\,dW_t,$$

[2] We display this without drift. In practice, we can do this for the underlying itself if is driftless, but if we are worried about drift, we can simply use the forward price which is by construction a martingale

[3] We get negative local volatility if arbitrage conditions are violated, but we assume this does not happen.

[4] We assume interest rates are zero, and so there is no drift term for simplicity

and approximate with finite differences

$$\tilde{S}_{t_{k+1}} - \tilde{S}_{t_k} = \tilde{S}_{t_k}\sigma(\tilde{S}_{t_k}, t_k)\left(W_{t_{k+1}} - W_{t_k}\right) = \tilde{S}_{t_k}\sigma(\tilde{S}_{t_k}, t_k)\sqrt{t_{k+1} - t_k}Z_k, \tag{3.1}$$

for some $Z_k = \frac{W_{t_{k+1}} - W_{t_k}}{\sqrt{t_{k+1} - t_k}} \sim \mathcal{N}(0,1)$. This is the well-known *Euler-Maruyama method*, an extension of the forward Euler method in stochastic calculus. It has been shown that the recursively-computed sequence $\tilde{S}_{t_1}, \ldots, \tilde{S}_{t_N}$ converges to a draw from the true stochastic process S_t as the mesh of the partition $\{t_1, \ldots, t_N\}$ tends to zero (Dobrow, 2016). We implement this procedure in Listing 1.

In this implementation, we have two types of numerical error to reason about. The first is simply sampling error. Because we are only taking a finitely many number of draws from the distribution, our sample mean will differ from the true mean. In addition, the distribution that we sample from will not be precisely the distribution of S_T because our partition is finite. We call the difference of means of these two distributions our *discretization error*.

3.1 Sampling Error

Reasoning about sampling error is straightforward: by the Central Limit Theorem, if we sample possible prices $P_1, \ldots, P_N$, then the distribution of their sample mean will be appriximately distributed as,

$$\overline{P} \sim \mathcal{N}\left(\mu_P, \frac{\sigma_P}{\sqrt{N}}\right)$$

for big enough N. In general, we can estimate σ_P from our sample, and we can bring down the error because it is decreases in $\sqrt{n}$.

We can empirically confirm this by using Black-Scholes. If our local volality function is constant, this means that we can analytically price options. It also means that there is no discretization error. Thus, we can run a couple samples of size 10^5 from the price distribution, price call options with each sample, and compare to the Black-Scholes theoretical price and standard error. The results of this are plotted in Figure 4.

The figure shows us that 10^5 samples are enough to get a pricing error of a couple cents, and by consequence, that 10^6 samples are enough to get pricing error on the order of a cent. Moreover, pricing errors are smaller for far out of the money calls, which makes sense because the distribution of payoffs will be dominated by a point-mass at 0 (most far out of the money calls expire out of the money).

3.2 Discretization Error (Theory)

Having understood the behavior of the sampling error, we still have to bound the discretization error. Let $\Delta t = t_{k+1} - t_k$ and suppose that S_{t_k} is a draw from the true distribution. Let $S_{t_{k+1}}$ be drawn from the true distribution conditional on S_{t_k}, and let $\tilde{S}_{t_{k+1}}$ be the corresponding finite difference approximation via (3.1) with respect to the same draw from the Brownian motion W_t. We have the following bounds on convergence (Higham, 2001):

$$E\left|S_{t_{k+1}} - \tilde{S}_{t_{k+1}}\right| \leq C_1(\Delta t)^{1/2} \qquad \text{(Strong convergence)}$$

$$\left|E(g(\tilde{S}_{t_{k+1}})) - E(g(S_{t_{k+1}}))\right| \leq C_2\Delta t \qquad \text{(Weak convergence)}$$

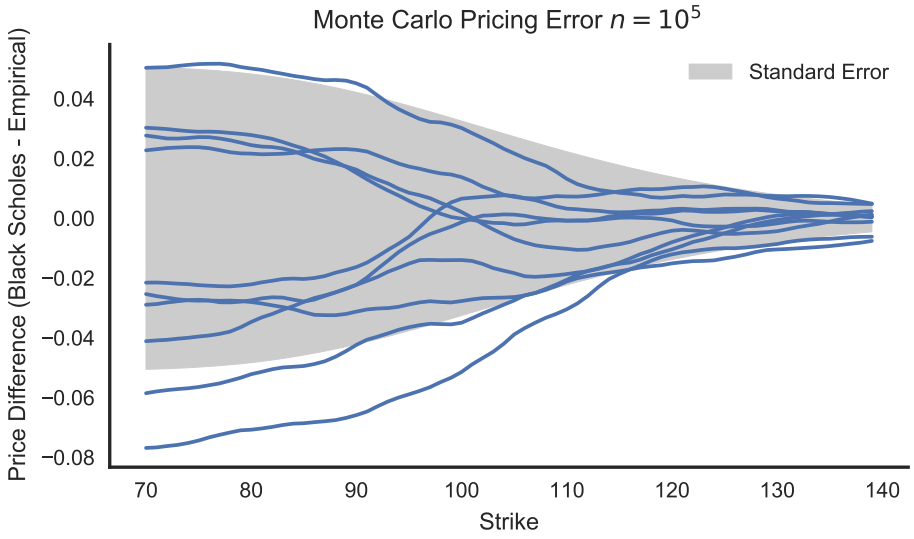

Figure 4: Monte Carlo sampling error $n = 10^5$

for a g that satisfies certain regularity conditions. Applying an equally-spaced partition to $[0, T]$ with step-size Δt, we immediately observe that the draws from the distribution at expiry must obey the same bounds of convergence:

$$E\left|S_T - \tilde{S}_T\right| = O\left((\Delta t)^{1/2}\right)$$
$$\left|E(g(\tilde{S}_T)) - E(g(S_T))\right| = O(\Delta t).$$

Since we apply Euler-Murayama method to option pricing, we are more interested in the error $\left|E(g(\tilde{S}_T)) - E(g(S_T))\right|$, where g is a function of the form $g(S_T) = \max(S_T - K, 0)$ (for call options). The nondifferentiability of g may pose some concern, as Higham (2001) notes that the weak convergence bound works for g smooth. However, since there exists a sequence of smooth functions $\tilde{g}$ that uniformly converges to $g(S_T) = \max(S_T - K, 0)$, we have

$$\left|E(g(\tilde{S}_T)) - E(g(S_T))\right| \leq 2||g - \tilde{g}||_\infty + \left|E(\tilde{g}(\tilde{S}_T)) - E(\tilde{g}(S_T))\right|,$$

where $||g - \tilde{g}||_\infty$ can be arbitrarily small.

3.3 Discretization Error (Numerical Convergence)

We perform the following numerical experiment to supplement the theoretical analysis above. Let $S_0 = 1, K = 1.1$, and

$$\sigma(S, t) = \min\left(0.1 + (S - 1)^2, 0.5\right).$$

Consider a call option at expiry $T = 1$ with strike K, whose payoff is $\max(S_T, K) - K$. Assume zero interest rate. We approximate the expectation with sample mean with sample size 10^6. We approximate the true value of the option by computing the Euler-Murayama approximation with

step-size equalling $1/200$. We then plot the pricing errors of Euler-Murayama approximations with step size $1/n$ for $n = 1, \ldots, 40$ in Figure 5.

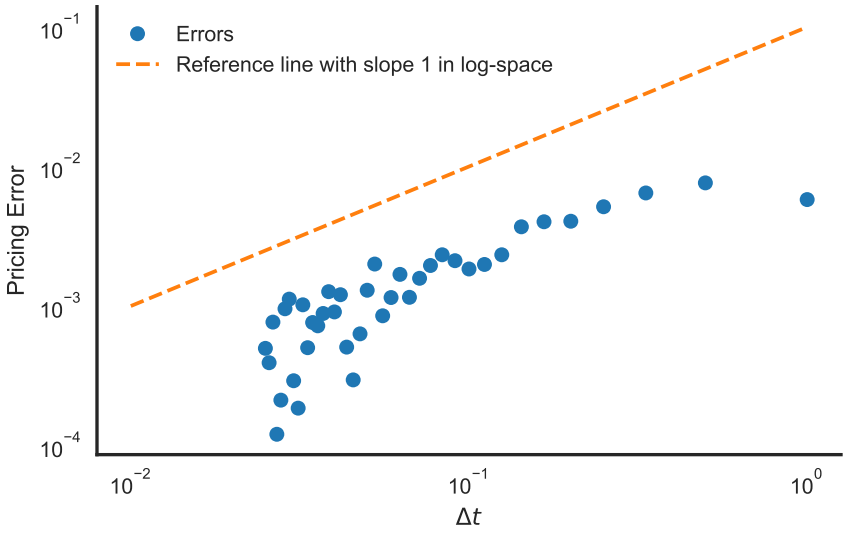

Figure 5: Log-log plot of pricing error against Δt

It is difficult to control the Monte Carlo sampling error, due to computational resource constraints, but we do observe approximately a first-order convergence pattern.

Thus, for a known σ^2, we can limit the size of the discretization error at rate $O(\Delta t)$, and the size of the sampling error at rate $O(n^{-1/2})$ where n is the sample size. The theory and the numerical experimentation suggests that the Monte Carlo pricing method is numerically robust.

3.4 Discretization Error (Across Strikes)

In the previous numerical experiment, we explored the asymptotics of discretization error. However, this is only half the story: we are also interested in how discretization error varies across strike. To this end, we will use an artificial local volatility function,

$$\sigma(S_t, t) = \min(\max(0.16 + 10^{-4}(S_t - 100)^2, 0), 0.3)$$

and price one-year-dated call options with a different number of discretizations ranging from 1 to 25. We plot the 1-standard-error intervals minus our most accurate point estimate (10^6 samples with 100 discretizations) for each of these pricings in Figure 6.

3.5 Sensitivity to Local Volatility

Thus far, we have talked about pricing errors: differences between an option price we calculate and the true option price under asset diffusion as specified by the local volatility function. In practice, though, we will not know the true local volatility function since we must estimate it. This means that we also care about the pricing *sensitivy*, i.e. how prices change for a change in the local volatility function.

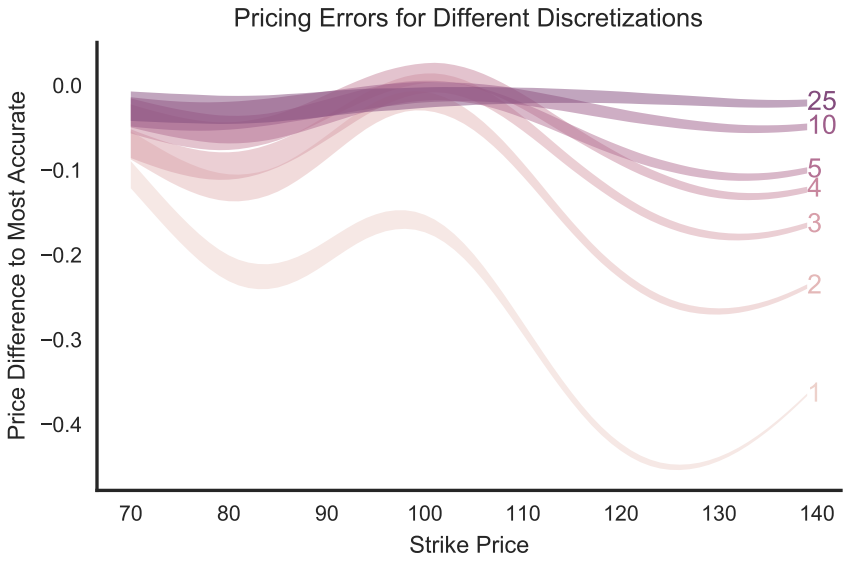

Figure 6: Pricing errors for different discretizations

In general, the sensitivity will be contingent both on the local volatility function as well as which parts of the function are changing. Since we are only interested in a gauge of how sensitive these quantities are, we will restrict ourselves to the case where the local volatility function is constant and we shift the entire function up. In this case, we are in a Black-Scholes world and this sensitivity—called Vega—has a closed form. We can plot the values of vega for two $100-stocks with 16% and 24% volatility respectively in Figure 7,

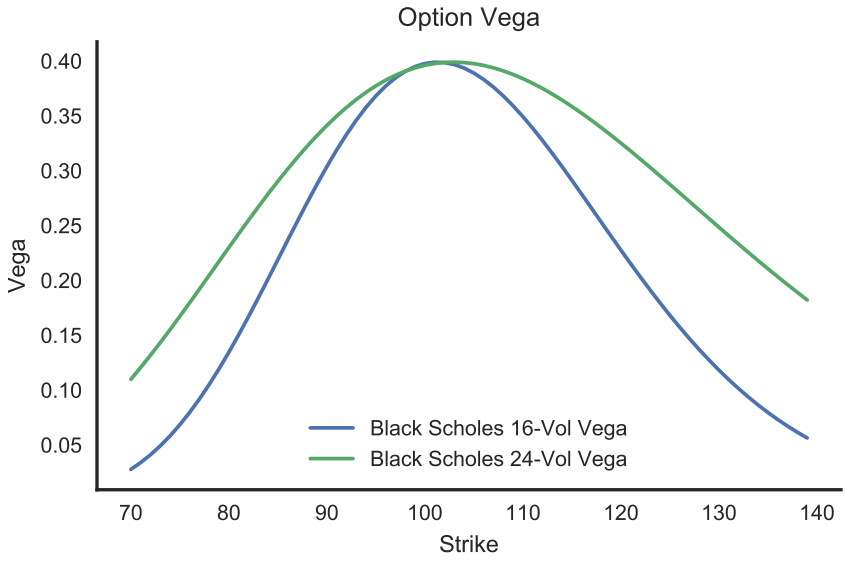

Figure 7: Option Vega

noting that 0.40 vega means that if volatility goes up by 1%, then the price of the option goes up

by $0.40.

4 Fitting Local Volatility

Given a local volatility function, we have seen how we can price options. Also, given a continuum of call option prices, we can uniquely identify the local volatility function via Equation 2.9. However, we do not in practice get to observe a continuum of call option, since options are quoted only for discrete expiries and strikes. Thus, we must numerically approximate the local volatility function from what we observe.

4.1 Finite Differences

In order to estimate the local volatility from discrete option data, we will replace the derivatives in Equation 2.9 with finite differences in order to get point estimates of the local volatility function. In particular, if we observe a grid of call option prices as in Figure 8, then we will get point estimates for all points on the inside of the grid.

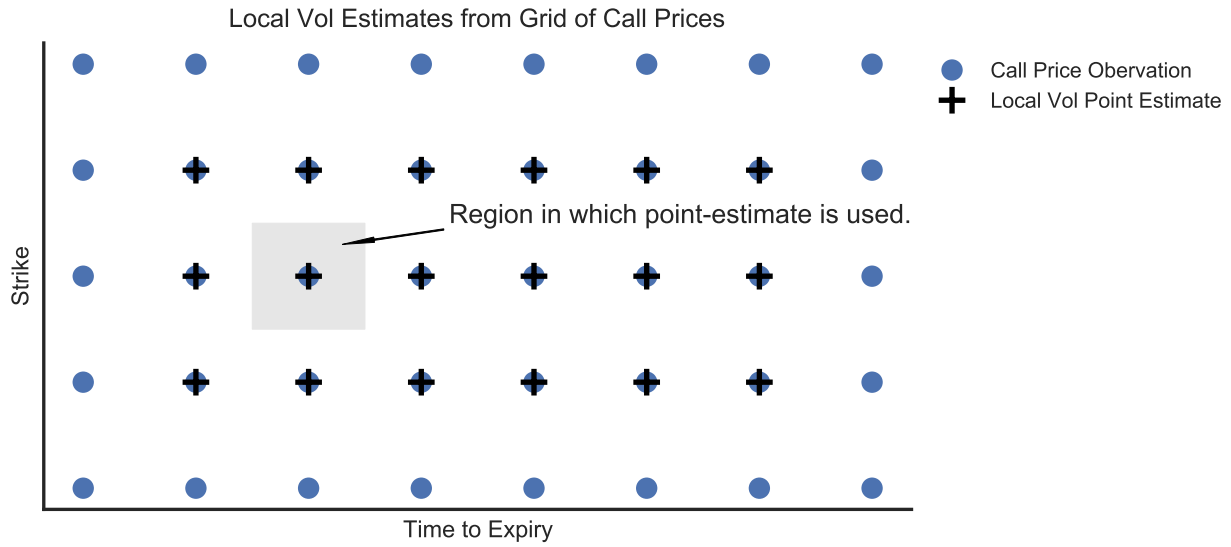

Figure 8: Local vol estimates from grid of call prices

Then, we can estimate the first partial derivative with respect to expiry using a central difference, and the second partial derivative with respect to strike using the second-order central finite difference. However, when pricing options, we do not just need the local volatility at a discrete grid of points. Thus, we need a way to use our grid of point estimates to get estimates at any point. To this end, we employ two main approaches. The first is to simply use the closest point-estimate (for instance, the shaded area represents the expanse of one point estimate), to get a piece-wise constant function. The second approach is to perform a bilinear interpolation on the grid.

4.2 Local Quadratic Regression

Another method to estimate local volatility that we test is by approximating C locally quadratically in order to get analytic first- and second-derivatives. For any point in the interior of the grid as in Figure 8, we consider the stencil formed by the point and its eight surrounding neighbors. We then fit a regression on the feature vector $(1, K, T, K^2, KT, T^2)$ for each of the nine points. Since we need to estimate six coefficients and we have nine data points, the linear system is over-identified, lending well to fitting a linear regression. We estimate $\frac{\partial C}{\partial T}$ and $\frac{\partial^2 C}{\partial K^2}$ by differentiating the best-fit quadratic surface.

The approach is motivated by a concern that finite differences may give estimates that are too volatile, where we attempt to enforce some continuity by estimating a quadratic function locally on the option price surface.

4.3 Theoretical Difficulties

However, computing σ^2 from observed data accurately is a much more difficult task. We illustrate the main difficulty below.

Let $A = \frac{\partial C}{\partial T} + e_1$ and $B = \frac{\partial^2 C}{\partial K^2} + e_2$ be two approximations. Then

$$\frac{A}{\frac{1}{2}K^2 B} = \sigma^2 \frac{\frac{\partial^2 C}{\partial K^2}}{\frac{\partial^2 C}{\partial K^2} + e_2} + \frac{e_1}{\frac{1}{2}K^2 B}.$$

Thus the absolute error

$$\left| \frac{A}{\frac{1}{2}K^2 B} - \sigma^2 \right| \leq \sigma^2 \left| \frac{e_2}{\frac{\partial^2 C}{\partial K^2} + e_2} \right| + \left| \frac{e_1}{\frac{1}{2}K^2 B} \right|.$$

Since $\frac{\partial^2 C}{\partial K^2} > 0$, it should be unsurprising that the error vanishes as $e_1, e_2 \to 0$. However, note that deep-in-the-money options are virtually indistinguishable from the underlying asset, whereas deep-out-of-the-money options are virtually worthless. Thus C is almost linear as a function of K when K is away from S_0.

Even though $\frac{\partial^2 C}{\partial K^2} > 0$, the infimum of this second derivative is zero. This presents a first difficulty when attempting to bound the error of the approximations. A second challenge arises from the discreteness of empirical prices. We cannot evaluate the function $C(K, T)$ anywhere we wish, but rather we only observe its values on a grid whose fineness is capped at the finest intervals that prices are quoted in. To make matters worse, the prices quoted are accurate to \$0.01, and so it is as if we are working in a world where machine precision is nontrivially large.

The usual centered difference approximations have

$$|e_1| \leq \frac{h_1^2}{6} \sup \left| \frac{\partial^3 C}{\partial T^3} \right| + \frac{\epsilon}{2h_1} \qquad |e_2| \leq \frac{h_2^2}{12} \sup \left| \frac{\partial^4 C}{\partial K^4} \right| + \frac{2\epsilon}{h_2^2}.$$

For values of $\frac{\partial^2 C}{\partial K^2}$ sufficiently large, it is plausible that these approximations are sufficiently accurate. However, the approximation is certainly not accurate for values of $\frac{\partial^2 C}{\partial K^2}$ close to zero. Yet for these values, which correspond to deep-money options, the option price is virtually known—the large errors in σ^2 for K far away from S_0 is weighted by the extremely small probability that such errors matter. Such weighting is analytically intractable, and we mainly focus on numerical experimentation.

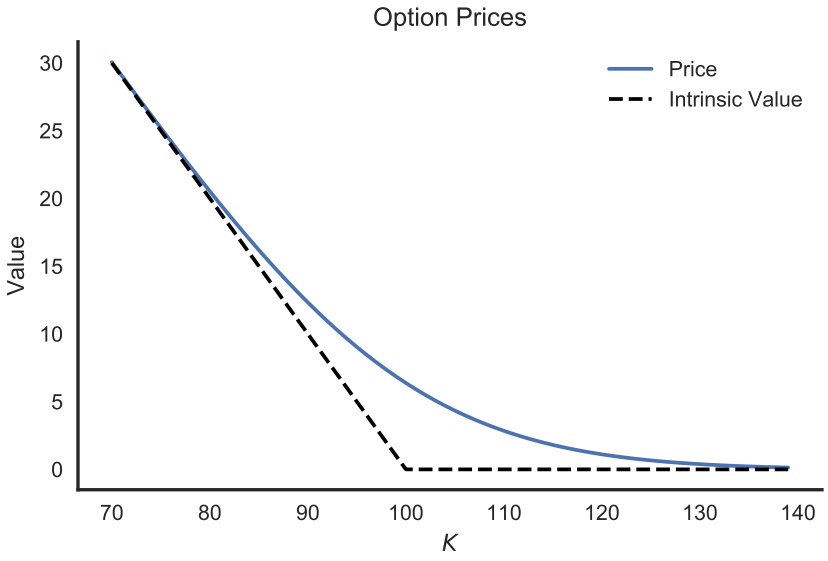

Figure 9: Option Intrinsic and Time Value

5 Numerical Experiment

5.1 Methods

We conduct numerical experiments with local volatility functions in the following analytic forms, which are plotted in Figure 10:

$$\sigma_1(S,t) = \min(.16 + 10^{-4}(S-100)^2, .5) \quad \text{(Quadratic)}$$
$$\sigma_2(S,t) = 0.2\cos^2(0.1(S-100)) + 0.16 \quad \text{(Sinusoidal)}$$
$$\sigma_3(S,t) = 0.24 + \frac{0.1}{1+\exp(0.2(S-100))} \quad \text{(Logistic)}$$
$$\sigma_4(S,t) = \text{clip}(0.16 + 300^{-1}(S-100), 0, 1) \quad \text{(Linear)}$$

For simplicity, we assume none of these are dependent on t. For each local volatility function, we simulated the movement of the underlying for one year with 100 intervals per year, and calculated simulated *true* option prices for each month, with \$1 apart strike prices based on 10^7 samples (The $t = 1$ prices for each local volatility function are plotted in Figure 11.). We apply the procedures with the following caveat. We clip estimates of $\frac{\partial C}{\partial T}$ to zero if they become negative, since $\frac{\partial C}{\partial T} < 0$ violates a no-arbitrage condition. We assume estimates of $\frac{\partial^2 C}{\partial K^2}$ as to be `NaN` if it is smaller than some threshold of tolerance. We use a tolerance of 0.01 for the quadratic model, and 0.005 for other σs.

Then, we apply the methods discussed in Section 4 to each observed true prices. Since the true prices form a grid, we obtain a grid of estimated local volatility. We construct a function $\tilde{\sigma}$ over all values of S, t from its observed values on a grid by taking the nearest neighbor, i.e., the $\tilde{\sigma}$-image of a point S, t not in the grid is the closest observed local volatility estimate. This is implemented via the library function `NearestNDInterpolator` in `scipy.interpolate`. We then use the interpolated $\tilde{\sigma}$ to price options via Monte Carlo pricing, as in Section 3. We then compare the predicted prices to the actual prices.

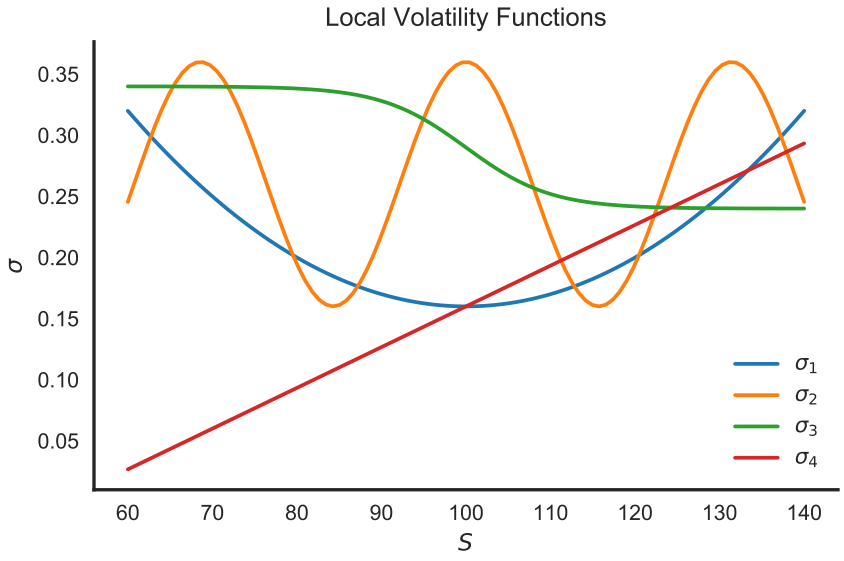

Figure 10: Local Volatility Functions Considered in Numerical Experiments

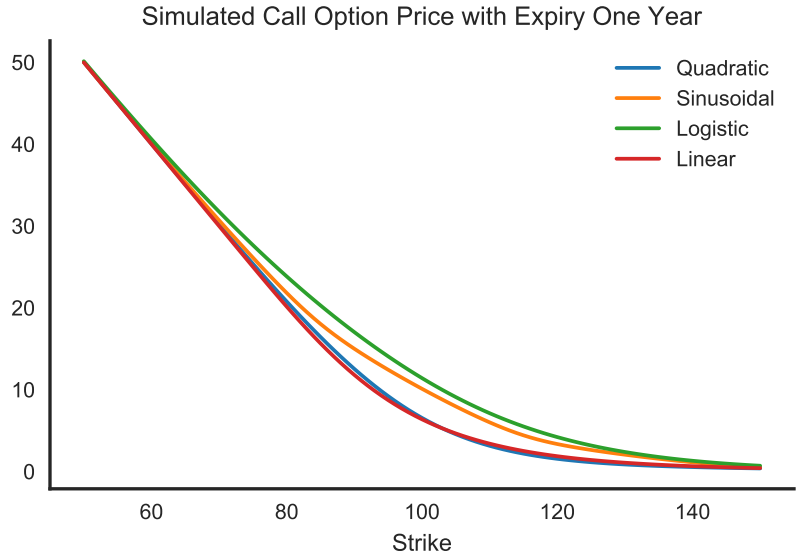

Figure 11: Simulated Option Price

5.2 Results

We plot in Figure 12 the true local volatility function and the fitted local volatility function at $t = 0$ (initial) and $t = 1$ (final). At values far away from $S_0 = 100$, all estimates of $\frac{\partial^2 C}{\partial K^2} \approx 0$, and so we discard the estimates. Since we are using the nearest-neighbor interpolation method, $\tilde{\sigma}$ values outside a band around $S_0 = 100$ are clipped to the boundary values around the band. These are values far away from the strike, and so they are unlikely to be incurred. We see that the errors are visible when $t = 0$, but small when $t = 1$. We hypothesize that this is due to the fact that the

option prices near $t = 0$ are close to piecewise linear, making the estimates of $\frac{\partial^2 C}{\partial K^2}$ imprecise.

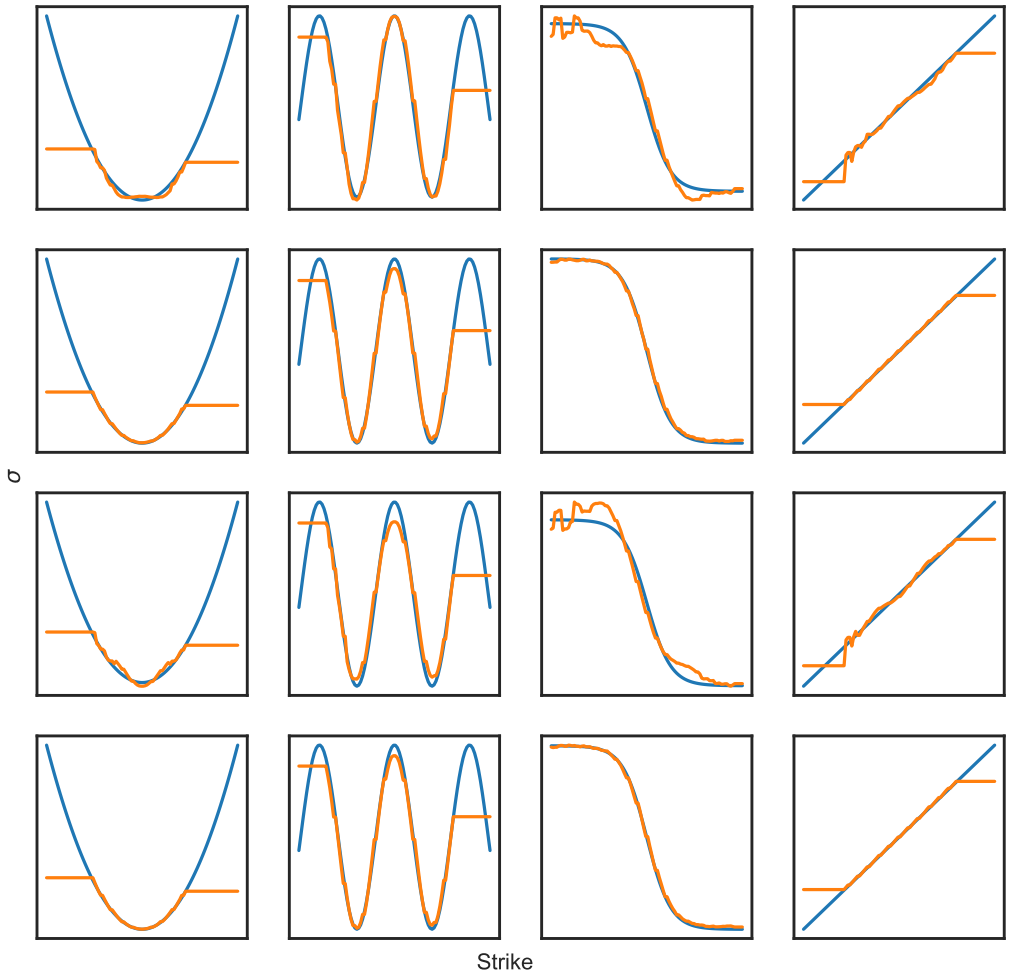

Figure 12: The fitted local volatility functions; the top two rows are finite-difference fits and the bottom two rows are local quadratic regression fits; the odd rows are $t = 0$ and the even rows are $t = 1$ year.

We plot the pricing errors in Figure 13. We observe that the horizontal stripes, which represent Monte Carlo error, are on the order of 10^{-2}. Pricing errors, except for far-out-of-money calls, appear to be on the order of \$0.05–\$0.10, which is about 1% error for options that is near at-the-money.

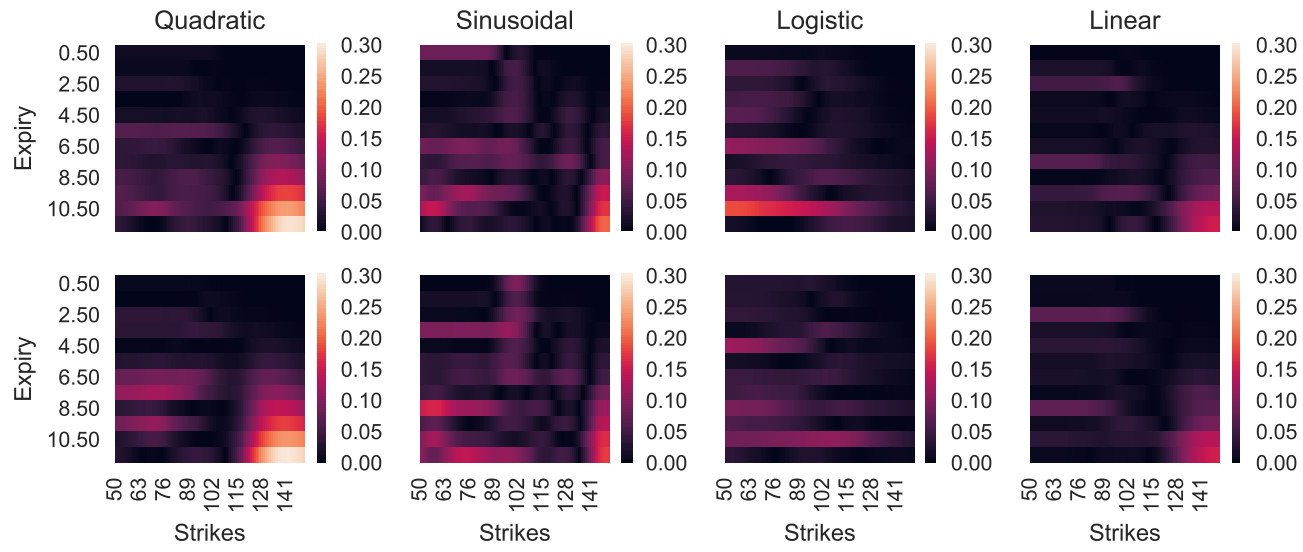

Figure 13: Errors obtained on four local volatility functions; the top row is errors using finite difference and the bottom row is errors using local quadratic regressions

6 Discussion and Conclusion

From our results, we can recover some stylized facts about the effectiveness and shortcomings of our approaches.

First of all, our numeric approximations of the local volatility function track most closely to the actual local volatility function near the current spot value. This is consistent with with both derivatives in equation 2.9 approaching 0 for options that are far out of the money or deep in the money. Thus, small absolute errors in pricing these options (such as from Monte Carlo pricing) can become amplified. Similarly, we observe that the estimations of the local volatility function at times that are further away track more closely. This is similarly consistent with the price of a short-dated option approaching its intrinsic value, which is a linear function with zero second partial derivative. Our results appear robust to the particular functional form chosen for the local volatility function.

Second of all, we can observe that different local volatility functions lead to different levels of pricing error. While the exact dynamics at play are hard to pin down, we identify two possible heuristics for the difficulty of pricing options given a certain local volatility function. A large average value of the local volatility function increases asset dispersion and makes the discretization error of our pricer become more relevant. Additionally, a local volatility function that changes rapidly (has steep slope) makes it such that small errors in an asset's diffusion affect the future significanty. In this light, it makes sense that the linear function can be priced accurately (because it has the lowest average value), and that the logistic function can be priced accurately as well (because it changes gradually). On the other hand, the quadratic and sinusoidal local volatility functions appear harder to price.

Regarding the finite differences approach versus local quadratic regression, both appear to be

doing quite similarly. They provide estimated local volatility functions that track approximately about as well in all experimented situations, and their pricing errors are comparable as well.

From all our prices, it appears that pricing far out-of-the money long-dated options given the quadratic local volatility causes the most error. While this phenomena is not *a priori* evident, we attribute it to the way we fit the local volatility curve. In particular, in regions where we do not have enough data, we simply take the closest reliable estimate. For the logistic curve, this is not a problem, since that accurately describes the tail dynamic of the curve. However, for the quadratic curve, this causes us to tremendously underestimate the local volatility for high values of spot. An implication of this is that we perceptibly underestimate the likelihood of tail events, causing us especially misprice the options whose value is entirely derived from the probability of these right tail events.

6.1 Future Work

On the back of our encouraging results, it would be interesting to apply our methods to market data. This would be problematic for data sources such as Yahoo Finance (see Figure 3) which manage illiquid strikes poorly (the jagged ends of the curve would be fit poorly by local volatility). However, with a clean data-source, it would be possible to fit a local volatility surface with in-sample prices, and attempt to predict out-of-sample prices (since there is no ground-truth local volatility function to compare against).

Numerically, it would also be interesting to explore in depth the features of a local volatility function that make our fit local volatility curve and subsequent prices diverge the most from the truth. While we price four different local volatility functions in the above work, exploring more functional forms, as well as approaching the problem from a theory standpoint could prove fruitful. Furthermore, all the local volatility functions we use are time-independent, and while we believe these are representative (because volatility is local and does not depend on the past or future), it would also be interesting to consider time-dependent local volatility functions.

Finally, we could explore additional methods both for fitting the local volatility surface as well as for pricing. While Monte Carlo pricing works well for our needs, it may be possible to achieve comparable accuracy in less time through a tree-based pricing model. Perhaps more interestingly, we may be able to fit the local volatility by optimizing over parameteric functional forms. This approach has the benefit of characterizing the behavior of the local volatility function in a way *a priori* encoded by the functional form, which may beat our non-parameteric solution when the prices are particularly noisy.

References

Black, F. and M. Scholes (1973). The pricing of options and corporate liabilities. *Journal of political economy 81*(3), 637–654.

Cochrane, J. H. (2009). *Asset Pricing:(Revised Edition)*. Princeton university press.

Dobrow, R. P. (2016). *Introduction to Stochastic Processes with R*. John Wiley & Sons.

Dupire, B. (1997). Pricing and hedging with smiles. *Mathematics of derivative securities 1*(1), 103–111.

Higham, D. J. (2001). An algorithmic introduction to numerical simulation of stochastic differential equations. *SIAM review 43*(3), 525–546.

Risken, H. (1996). Fokker-planck equation. In *The Fokker-Planck Equation*, pp. 63–95. Springer.

Appendices

Listing 1: Sampling from risk-neutral S_T distribution

```python
import numpy as np
import pandas as pd

def sample_end_price(S0, local_vol_f, duration, n_intervals, n_samples):
    """
    Draw samples from the end price of an asset diffusion.

    Inputs
    ------
    S0 : float
        The initial spot price of the underlying at time t=0

    local_vol_f : float -> float (vectorized)
        The local volatility at a given spot price (assumed constant over time)

    duration : float
        The time to expiry, i.e. T.

    n_intervals : float
        Number of intervals into which to break up the numerical simulation

    n_samples : int
        Number of simulations to run

    Output
    -----
    S : NumPy float vector of length n_samples
        The ending spot prices of the asset diffusion for each simulation
    """
    if duration == 0:
        return S0 * np.ones(n_samples)


    dt = duration / n_intervals
    scaling_factor =  np.sqrt(duration / n_intervals)
    S = S0 * np.ones(n_samples)
    for i in range(1,n_intervals+1):
        time = i / duration
        local_vols = local_vol_f(S, time).flatten()
        growth_factor = np.exp(local_vols * np.random.randn(n_samples) * scaling_factor
                               - dt * local_vols**2/2)
        S = S * growth_factor
    return S

def sample_end_prices(S0, local_vol_f, durations, intervals_per_year, n_samples):
    S = np.zeros((n_samples, len(durations)))
    for i,duration in enumerate(durations):
        n_intervals = max(1,int(duration * intervals_per_year))
        S[:,i] = sample_end_price(S0, local_vol_f, duration, n_intervals, n_samples)
    df = pd.DataFrame(S, columns=durations)
    df.columns.name = 'Expiries'
    return df
```

Listing 2: Pricing call option from Monte-Carlo samples

```python
import numpy as np
import pandas as pd
```

```

def price_call(K, end_samples):
    """
    Price a call option using Monte Carlo samples.

    Inputs
    ------
    K : float:
        The strike price of the call option we are pricing.

    end_samples : np.array[float]
        A NumPy array of draws from the distribution of prices
        of the underlying at expiry.

    Outputs
    -------
    px : float
        A point estimate for the price of the call option
    sd : float
        An estimate of the standard error incurred from this pricing.
    """

    payoffs = np.clip(np.subtract.outer(end_samples, K), 0, np.inf)
    px = payoffs.mean(axis=0)
    sd = payoffs.std(axis=0) / np.sqrt(len(end_samples))
    return px, sd

def price_calls(Ks, samples):
    payoffs = np.clip(np.subtract.outer(samples, Ks), 0, np.inf)
    px = payoffs.mean(axis=0)
    df = pd.DataFrame(px, index=samples.columns, columns=Ks)
    df.index.name = 'Expiry'
    df.columns.name = 'Strikes'
    return df
```

Listing 3: Price using Black-Scholes formula

```
import numpy as np
from scipy.stats import norm

def black_scholes_price(S, vol, T, K):
    """
    A Black Scholes pricer for European call options which assumes
    interest rates are 0 and an underlying that doesn't pay out dividends.

    Inputs
    ------
    S : float
        The initial price of the underlying
    vol : float
        The annualized volatility of the underlying
    T : float
        The time to expiry (in years)
    K : float
        The strike price of the call option

    Outputs
    -------
    px : float
        The price of the call option under the Black Scholes model with 0
        risk-free interest rate.
    """
```

```
    d1 = (np.log(S / K) + (vol**2/2)*T) / (vol*np.sqrt(T))
    d2 = d1 - vol*np.sqrt(T)
    return S * norm.cdf(d1) - K*norm.cdf(d2)
```

Listing 4: Create interpolating objects to represent fit local vol surfaces

```
import numpy as np
import scipy.interpolate as interp

class InterpolatedLocalVol:

    def __init__(self, point_estimates, dates, strikes, uses_dates=False):
        self.uses_dates = uses_dates
        self.point_estimates = point_estimates
        if uses_dates:
            self.raw_dates = np.array(dates)
            self.dates = np.array([d.toordinal() for d in dates])
        else:
            self.raw_dates = np.array(dates)
            self.dates = np.array(dates)

        self.strikes = np.array(strikes)
        self.f = interp.interp2d(self.strikes, self.dates,
            point_estimates, bounds_error=False, kind='linear')

    def __call__(self, raw_dates, strikes):
        if self.uses_dates:
            try:
                dates = np.array([d.toordinal() for d in raw_dates])
            except:
                dates = raw_dates.toordinal()
        else:
            dates = raw_dates
        return self.f(strikes, dates)

class NNLocalVol(InterpolatedLocalVol):

    def __init__(self, point_estimates, dates, strikes, uses_dates=False):
        super().__init__(point_estimates, dates, strikes, uses_dates)
        strikes, expiries = np.meshgrid(self.strikes, self.dates)

        self.f = interp.NearestNDInterpolator(
                np.vstack([strikes.flatten(), expiries.flatten()]).T,
                self.point_estimates.flatten(),
                rescale=True)
```

Listing 5: Local Volatility Pricer

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm_notebook as tqdm
import scipy.interpolate as interp
from sklearn.preprocessing import PolynomialFeatures
from copy import copy

# Requirements in notebook
# - Requires sampleendprice, price_calls, and InterpolatedLocalVol
# %matplotlib inline
# %config InlineBackend.figure_format = 'retina'
# %run report / scripts / sampleendprice.py
```

```
# %run report / scripts / pricecall.py
# %run interpolated_local_vol.py

class LocalVolatilityPricer:
    """An object that saves true prices and handles fitting and prediction"""

    def __init__(self, local_vol, expiries, strikes):
        """
        Instantiate a LocalVolatilityPricer object

        Inputs
        ------
        local_vol: float, float -> float, vectorized
            local volatility function at a given spot price and time

        expiries: np.array
            the dates of expiries simulated

        strikes: np.array
            the strike prices interested
        """
        self.local_vol = local_vol
        self.expiries = expiries
        self.strikes = strikes

    def get_true_prices(self, S0=100, interval_per_year=100, n_samples=int(1e5)):
        """
        Perform Monte Carlo pricing to simulate true prices of a call option

        Inputs
        ------
        S0: float
            initial price of the underlying

        interval_per_year: int
            the number of intervals per year simulated; dt = 1/intervals_per_year

        n_samples: int
            the number of samples simluted
        """
        self.S0 = S0
        true_prices = 0

        # price 100 times and take the average price to reduce sampling error
        for _ in tqdm(range(100)):
            sample = sample_end_prices(self.S0, self.local_vol, self.expiries,
                                       interval_per_year, n_samples)
            true_prices += price_calls(self.strikes, sample)

        self.true_prices = true_prices / 100
        return self.true_prices

    def fit_local_vol(self, fitter, interp_option='linear', *args, **kwargs):
        """
        Perform fitting using a given fitter

        Inputs
        ------
        fitter: function
            fitter's first argument takes in a DataFrame of true prices
            fitter returns a tuple (local_volatility, dCdT, d2CdK2) estimated

```

```
        interp_option: string
            option for interpolation passed to _interpolate_local_vol
            'linear' means linear interpolation
            'nn' means using scipy.interpolate.NearestNDInterpolator
        """

        lvol, dCdT, d2CdK2 = fitter(self.true_prices, *args, **kwargs)
        self.fitted_local_vol = lvol
        self.dCdT = dCdT
        self.d2CdK2 = d2CdK2
        self._interpolate_local_vol(interp_option)
        return self.fitted_local_vol

    def _interpolate_local_vol(self, interp_option):
        """Interpolate the grid of local volatility values using interp_option"""
        if interp_option == 'linear':
            data = self.fitted_local_vol.T.stack().reset_index().as_matrix()
            stks, exps, lv = data[:, 0], data[:, 1], data[:, 2]
            self.interp_fitted_local_vol = InterpolatedLocalVol(lv, stks, exps)
        elif interp_option == 'nn':
            data = self.fitted_local_vol.T.stack().reset_index().as_matrix()
            args, lv = data[:, :2], data[:, 2]
            self.interp_fitted_local_vol = interp.NearestNDInterpolator(
                args, lv)
        else:
            raise ValueError('interp_option not one of "linear", "nn"')

    def generate_prediction(self, interval_per_year=100, n_samples=int(1e5)):
        """
        Use interpolated local volatility estimates to compute option prices
        via Monte Carlo pricing
        """
        f_samples = sample_end_prices(self.S0, self.interp_fitted_local_vol,
                                      self.expiries, interval_per_year, n_samples)
        self.predicted_prices = price_calls(self.strikes, f_samples)
        return self.predicted_prices

    def plot_errors(self):
        """Plot errors obtained by comparing predicted prices to true prices"""
        f, ax = plt.subplots(nrows=2, figsize=(5, 2 * 5 / 1.618))
        sns.heatmap(self.predicted_prices
                    - self.true_prices, ax=ax[0], fmt='%.2f')
        sns.heatmap(np.abs(self.predicted_prices - self.true_prices), ax=ax[1])

    def plot_fit(self, time, xs=np.arange(70, 130, .1)):
        """Plot comparison of local vol estimates against true local vol"""
        plt.plot(xs, self.local_vol(xs, time))
        plt.plot(xs, self.interp_fitted_local_vol(xs, time))
```

Listing 6: Fitters Used

```
from sklearn.preprocessing import PolynomialFeatures


def fd_fitter(C_grid, tol=0.):
    """Implements finite difference fitting"""
    dts = np.diff(C_grid.index)

    # Compute dCdt
    dCdt = np.clip((C_grid.as_matrix()[1:, :]
                    - C_grid.as_matrix()[:-1, :])
                   / dts[:, np.newaxis], a_min=0, a_max=None)
```

```
12
13      # Centered difference dcdt
14      dCdt_compatible = (dCdt[1:, 1:-1] + dCdt[:-1, 1:-1]) / 2
15
16      # Assume equal strikes
17      Kdiff = np.diff(C_grid.columns)[0]
18      d2CdK2 = (C_grid.as_matrix()[:, :-2] + C_grid.as_matrix()[:, 2:]
19                - 2 * C_grid.as_matrix()[:, 1:-1]) / Kdiff**2
20      d2CdK2_compatible = d2CdK2[1:-1, :]
21
22      # set second derivative of < tol to np.nan
23      d2CdK2_compatible[d2CdK2_compatible < tol] = np.nan
24
25      new_expiries = C_grid.index[1:-1]
26      new_strikes = np.array(C_grid.columns[1:-1])
27
28      local_vol2 = (dCdt_compatible / (1 / 2 * np.array(new_strikes **
29                                                        2).reshape(1, -1) * d2CdK2_compatible))
30
31      # convert to DataFrame and return
32      return (pd.DataFrame(np.sqrt(local_vol2), index=new_expiries, columns=new_strikes),
33              pd.DataFrame(dCdt_compatible, index=new_expiries,
34                           columns=new_strikes),
35              pd.DataFrame(d2CdK2_compatible, index=new_expiries, columns=new_strikes))
36
37
38  def local_quadratic_reg_fit(pxs, tol=0):
39      """Implements local quadratic regression fitting"""
40      def compute_derivatives(coeffs, strike, expiry):
41          """Computes derivatives of a quadratic given coeff"""
42          return (coeffs[2] + coeffs[4] * strike + 2 * coeffs[5] * expiry,
43                  2 * coeffs[3])
44
45      # Initialize
46      local_vol2 = np.zeros((len(pxs) - 2, len(pxs.columns) - 2))
47      dCdTs = np.zeros((len(pxs) - 2, len(pxs.columns) - 2))
48      d2CdK2s = np.zeros((len(pxs) - 2, len(pxs.columns) - 2))
49
50      for i in range(1, len(pxs) - 1):
51          for j in range(1, len(pxs.columns) - 1):
52
53              # Stencil of point at position [i,j]
54              local = pxs.iloc[i - 1:i + 2, j - 1:j + 2]
55
56              # K, T of point [i,j]
57              strike = pxs.columns[j]
58              expiry = pxs.index[i]
59
60              strikes, expiries = np.meshgrid(local.columns, local.index)
61
62              # Transform K,T to 1, K, T, K^2, KT, T^2
63              poly = PolynomialFeatures(degree=2)
64              features = poly.fit_transform(
65                  np.vstack([strikes.flatten(), expiries.flatten()]).T)
66
67              # Regression
68              coeffs, _, _, _ = np.linalg.lstsq(
69                  features, local.as_matrix().flatten())
70
71              dCdT, d2CdK2 = compute_derivatives(coeffs, strike, expiry)
72
73              # Cleaning of resulting derivatives
```

```
            dCdT = np.clip(dCdT, a_min=0, a_max=None)
            d2CdK2 = d2CdK2 if d2CdK2 >= tol else np.nan

            local_vol2[i - 1, j - 1] = dCdT / (1 / 2 * (strike ** 2) * d2CdK2)
            dCdTs[i - 1, j - 1] = dCdT
            d2CdK2s[i - 1, j - 1] = d2CdK2

    # Construct DataFrames and returning
    local_vol = pd.DataFrame(np.sqrt(local_vol2), columns=pxs.columns[
                              1:-1], index=pxs.index[1:-1])
    dCdTs = pd.DataFrame(dCdTs, columns=pxs.columns[
                         1:-1], index=pxs.index[1:-1])
    d2CdK2s = pd.DataFrame(d2CdK2s, columns=pxs.columns[
                           1:-1], index=pxs.index[1:-1])
    return local_vol, dCdTs, d2CdK2s
```

Listing 7: Example of Using `LocalVolatilityPricer`

```
# Example of using LocalVolatilityPricer
expiries = np.arange(1 / 12, 13 / 12, 1 / 12)
K = np.arange(50, 151, 1)


def quadratic_vol(px, time):
    return np.clip(.16 + 1e-4 * (px - 100) ** 2, 0, .5)

# Instantiate object
qv = LocalVolatilityPricer(quadratic_vol, expiries, K)
_ = qv.get_true_prices(n_samples=int(1e5))

_ = qv.fit_local_vol(fd_fitter, interp_option='nn', tol=0.01)

# Check fitting at t = 0
qv.plot_fit(0)

# Check errors
_ = qv.generate_prediction(n_samples=int(1e5))
qv.plot_errors()
```