Monte Carlo Valuation of American Put Options

Fabrizio Amonini, David Brusa and Nicole Lipsky
Group 8

A Term Paper in
Computational Economics and Finance
University of Zurich

July 1, 2016

University of Zurich
Chair of Quantitative Business Administration
Moussonstrasse 15
CH-8032 Zurich

Abstract

This project attempts to introduce a framework to compute the value of American options as well as other options such as Asian, Bermudan and European options in one or more underlyings based on the work in Longstaff and Schwartz (2001) and Jia (2009). This framework is then used for a wide array of examples to gain insights about and verify the implementation as well as showcase the different customizations enabled.

Contents

1 Introduction

Our paper focuses on valuing American put options. Our goal was to create a code that would allow for the valuation of a great different amount of such instruments, with flexible methods, conditions and terms of the option contract itself. We explain what methods we chose for the valuation process and how they perform.

We focus particularly on put options because the price of an American call option is identical to the price of European calls (if there are no dividends) and this is trivial to compute with the closed-form Black and Scholes (1973) formula. For put options, however, in certain situations it can be profitable to exercise the option before the expiry date. Without a defined exercise time, it makes it much harder to evaluate the options. If the option is in-the-money, there exists a dilemma between exercising the option or waiting until later when the payoff might be even bigger. We hence have to rely on open-form methods, which do not yield consistent results and are not as quickly computable. One way of doing so is by using Monte Carlo methods to generate random asset price developments across one or more assets and defining early-exercise policies, according to which option-holders realize their rights. The benefits of using Monte Carlo are as follows:

- It can give a solution to arbitrary payoffs. This allows us to valuate any type of options (e.g. path-dependent options, time-dependent options, options with multiple underlyings etc.).
- The code implementation can easily be changed to different path generators. In our base-version, we use a geometric Brownian motion to model stock price-development: $S_{t+\delta t} = S_t e^{\left(\mu - \frac{1}{2}\sigma^2\right)\sqrt{\delta t}N(0,1)}$, but Monte Carlo allows for an easy switch to other path generation functions, which makes the valuation very flexible and testable under different assumptions, depending on the path-function.

The problem of pricing American put options with Monte Carlo lies within the difficulty of pricing the possibility of early exercise. Because of this possibility, we need to know when an optimal strategy would exercise given each sample path. A naive answer might be to assume exercise at the maximum point of each single path within the Monte Carlo simulation, but this is actually "too good" for the optimal strategy, since it depends on future information. It is therefore necessary

to construct conditional expectations about future movements in each time step. Monte Carlo is already used to construct expectation values and applying another layer of expectations leads to "Monte Carlo on Monte Carlo" which becomes computationally punishing. The solution to this is using the *Least Squares Method* by Longstaff and Schwartz (2001) to approximate inner conditional expectations by fitting existing sample paths, which we will describe in the following section.

2 Algorithm

We implement the Least-Squares-Monte-Carlo approach to pricing of American Options introduced in Longstaff and Schwartz (2001) and introduce various extensions and parameters, including but limited to the different choice of basis for the least-squares regression as in Jia (2009), the application to higher-dimensional and path-dependent options as well as options that can only be exercised at specific points in time. We therefore introduce a more general framework for option pricing using Monte Carlo and embed the previous work therein.

Basic steps We begin by **generating sample paths** of the underlying asset(s) starting at $t = 0$ until expiry T (see Figure 1 for an illustration of 1000 simulations of a Black-Scholes path). We then **compute the payoff** of the product to be priced if it is held until expiry. So far, this approach is identical to the MC approach to pricing European options. Next, we utilize the technique found in Longstaff and Schwartz (2001) of **working backwards from expiry** and deciding for each sample path and each time step - where exercising is allowed - if the **ideal strategy** would continue the option. Finally, we compute the **statistics** for the obtained payoffs on these sample paths to estimate the **expected discounted payoff** of the product given that the continuation decision followed some optimal strategy.

This broad description can now be filled in with the different implementations described below.

2.1 Generating Sample Paths

We require a model for the underlying stock prices of the options we wish to price in order to be able to simulate stock price developments. We therefore quickly introduce the most important

concepts needed to define stock price models such as Black-Scholes, stochastic ordinary differential equations (SODE) as well as the most basic techniques to generate approximate solutions to these SODEs.

2.1.1 Definitions

Brownian Motion For our purposes[1], a Brownian Motion or Wiener Process is a continuous stochastic process W_t which satisfies the following conditions:

$$W_0 = 0 \tag{1}$$

$$W_{t+\Delta t} - W_t \sim \mathcal{N}(0, \Delta t) \text{ independent of } W_s \forall s \leq t \tag{2}$$

Poisson For our purposes[2], a Poisson Process is a stochastic process N_t which satisfies the following conditions:

$$N_0 = 0 \tag{3}$$

$$N_{t+\Delta t} - N_t \sim \mathcal{P}(\lambda \Delta t) \tag{4}$$

Intuitively, this means that the Poisson Process increases in integer steps at random times, which is why it will be used to model market crashes.

A Stochastic Ordinary Differential Equation (SODE) is an equation which relates the differential dX_t of a stochastic process X_t to some other quantities. An example of such an equation is Itô's drift-diffusion process $dX_t = \mu_t dt + \sigma_t dW_t$, where there are some conditions on the processes μ_t and σ_t, which are ignored in this discussion, and W_t is a Brownian Motion (BM).

Itô's Lemma Let $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be twice differential and $dX_t = \mu_t dt + \sigma_t dW_t$ then:

[1] The technical relies on σ-algebras and the concept of 'almost surely', which we chose to avoid at this point because this mathematical rigor doesn't really add much to the application at hand.

[2] Again, we chose to omit technicalities that do not contribute to the intuitive understanding.

$$df(t, X_t) = \frac{\partial f(t, X_t)}{\partial t} dt + \frac{\partial f(t, X_t)}{\partial X_t} dX_t + \frac{\sigma_t^2}{2} \frac{\partial^2 f(t, X_t)}{\partial X_t^2} dt \tag{5}$$

Euler–Maruyama method Is a scheme for numerical integration of ordinary differential equations. For deterministic ordinary differential equations (ODE) there are many numerical integration schemes, which, at their core, make use of the fact that for many functions F the value $F(t + \Delta t)$ can be approximated by $F(t) + \Delta t \frac{dF}{dt}$, propagating the solution forwards in time from some starting value. The deterministic explicit Euler method is the simplest scheme of this form since it simply propagates the solution forward by a fixed distance along the tangent of the current point. Euler–Maruyama is the extension of this simple concept to Itô processes $dX_t = a(t, X_t)dt + b(t, X_t)dW_t$.

2.1.2 Models

Jump-Diffusion Processes are stochastic processes whose propagation is defined by the SODE

$$dS_t = S_t(\mu dt + \sigma dW_t + \phi(dN_t - \lambda dt)) \tag{6}$$

,which can be solved using Itô's Lemma (Chesney (2015)) to

$$S_t = S_0 \exp((\mu - \frac{\sigma^2}{2} - \lambda\phi)t + \sigma W_t + \log(1 + \phi)N_t) \tag{7}$$

From the SODE we can see that the change in stock price is governed by a deterministic interest μ, a random 'diffusion' of magnitude σ and a jump of magnitude ϕ. During a unit interval λ jumps are expected. Setting ϕ or λ to zero results in the plain old Black-Scholes paths which is what is implemented in Longstaff and Schwartz (2001) and Jia (2009).

Heston Models consider the evolution of volatility itself and the method is described by the SODE

$$dS_t = \mu S_t dt + \sqrt{v_t} S_t dW_t^S \tag{8}$$

$$dv_t = \kappa(\theta - v_t)d_t + \xi\sqrt{v_t}dW_t^v, \tag{9}$$

where dv_t is a Cox-Ingersoll-Ross (CIR) process and dW_t^S and dW_t^v are Brownian motions. μ is the rate of return on the asset, θ is the long variance, κ is the rate of reversion from v_t to θ and ξ is the volatility of the volatility. To generate realizations of v_t we rely on Euler-Marayama with 240 iterations per desired time-step - i.e. to create a path of three time-steps we would need to run 780 steps of the numerical integration. The generated v_t's are then injected into a Jump-Diffusion process replacing σ with $\sqrt{v_t}$. This way, we recycle code and may even simulate stochastic volatility for Jump-Diffusion processes.

Constant Elasticity of Variance (CEV) models attempt to capture the leverage effect usually present in a stock. It defines the process:

$$dS_t = \mu S_t dt + \sigma S_t^\gamma dW_t, \tag{10}$$

whereby $\sigma \geq 0$ and $\gamma \geq 0$. The intensity of the relationship between the price of an asset and its volatility is expressed by γ. Therefore, if $\gamma < 1$, the leverage effect becomes apparent: The volatility of an asset's price will increase as it falls. We rely on Euler-Marayama to generate sample paths as we are unaware of a closed-form solution for $\gamma \neq 1$.

2.2 Options

Payoffs This algorithm can consider any option whose payoff can be expressed as a mapping that takes the current time t and a selection of past stock prices $S_\tau \in \mathbb{R}^d, \tau \in \mathrm{T}$. In particular this allows us to consider multi-dimensional options and Asian Options.

Exercise Times The basic steps mention going through all time-steps in which the option could be exercised. For American Options, one would simply set this list to all time-steps of the generated paths but with this additional simulation parameter, we are also free to consider Bermudan Options[3].

[3]Bermudan options can only be exercised on certain dates.

Remarks We show that with this general definition of the basic algorithm, we are free to model a wide variety of well-known products but we are also free to make up other products to price as long as they fit this fairly loose framework.

2.3 Strategy: Expected Payoff

The basic steps call for a strategy which decides if the options should be exercised for each time-step. To avoid an under-priced value of the option, we should consider the best possible strategy. Naïvely, one might be tempted to simply look at the entire path and find the optimal exercise time for that path - in the case of an American put option that would amount to finding the minimum of the discounted sample path. However, this is not a 'possible' strategy since it relies on future information - since the entire path must be known at the exercise time. We therefore chose the strategy of comparing the discounted payoff of exercising with the expected discounted payoff of continuing conditional on the current (and possibly past) stock prices. This is the only strategy we consider and implement - albeit in different flavors - for this paper. Note that we still provide the 'hook' in our framework to implement arbitrary different strategies, should the need arise.
The next question to answer is how to compute the expected payoff, which is where we introduce two different approaches: Least-Squares Regression and Local Regression.

2.3.1 Least-Squares Regression

Monomial Basis We implement Least-Squares Regression onto a monomial basis as in Jia (2009) although we extend this to two dimensional monomials as well.

Weighted Laguerre Polynomials We implement Least-Squares Regression onto a basis of exponentially weighted Laguerre polynomials $\exp(\frac{X}{2})L_k(X)$ as in Longstaff and Schwartz (2001) for one dimension. Because of the exponential term, we need to be careful with the argument range and therefore scale the argument to the basis function with S_0, which is not done in Longstaff and Schwartz (2001), as there, $S_0 = 1$ anyhow. Therefore, we use the basis functions $\exp(\frac{S}{2S_0})L_k(\frac{S}{S_0})$.

To compute the Laguerre polynomials, we use the recursion:

$$L_0(X) = 1$$
$$L_1(X) = 1 - X$$
$$L_{k+1}(X) = \frac{(2k + 1 - X)L_k(X) - kLk - 1(X)}{k + 1}$$

2.3.2 Local Regression

Regressing observations onto a model as is done in Least Squares Regression attempts to distill expression for the conditional expectation from observations. However, this inherently introduces some bias, as the expression is assumed to be from a certain class of functions - i.e. for linear regression, one assumes that the conditional expectation depends linearly on the explanatory variables. This assumption can be weakened using Local Regression, where the conditional expectation is assumed to adhere to some structure *locally*, meaning on a 'small enough' interval. The Local Regression approach consists of constructing the conditional expectation by weighting each observation according to the distance of the explanatory variables to the evaluation point and using these weighted observations for the regression. This means that a regression needs to be done for every point in which the conditional expectation is to be evaluated. The simplest example of this is **Kernel Smoothing** which assumes that the conditional expectation can be expressed locally as a constant, which then amounts to computing a **weighted average** of the samples around the evaluation point as an estimation of the conditional expectation. In other words, if we wish to estimate the conditional expectation $\mathbb{E}(Y|X_0)$ given realizations of $Y|X_i$, we can simply weigh each such observation according to the distance $|X_0 - X_i|$ and then take the average (or fit it onto some basis). The function used to weigh the samples is called the **smoothing kernel** and is often chosen as a Gaussian bell curve $K(x^*, x_i) = exp\left(- \frac{(x^* - x_i)^2}{2b^2} \right)$. The width of the curve is a parameter which might need to be fine-tuned for the application. If the bandwidth is too small, we might model noise as only a few samples will be considered per evaluation point. If the bandwidth is too large, the weight is uniform and the local regression degenerates to a global regression.

3 Experiments

3.1 Reference Solution

In order to verify our implementation we compute a reference solution for an American put option with an underlying following a Black-Scholes path with the parameters shown in Table 1 using a third-party solver based on finite differences.

Parameter Name		Value
Initial price of the underlying	S_0	80
Strike price	K	100
Risk free interest rate	r	0.1
Time to expiry of the option	T	1
Volatility of the underlying	σ	0.25

Table 1: Option parameters in our simulations (set as default in our code).

Given these parameters, the resulting put option price is 21.6.

3.2 Performance Evaluation & Comparison

In this section we present and compare the performance of some of the extensions that we implemented in our code.

First, let us give an explanation of the schematic the graphs are built with: The left one is always a pricing outcome (y-axis) of the option, using Monte Carlo with different numbers of simulations (x-axis), represented by the blue stars in the plot. The green line represents the reference solution from Section 3.1. The image in the middle is a histogram of the points in time (out of the 50 time steps) where the option has been exercised. The data is taken from the last simulation run with the 150 000 paths. The rightmost graph, if contained, shows the error of the simulation run compared to the reference solution (the absolute difference between a point and the green line in the leftmost graph).

3.2.1 Regression Choices

Polynomial Fit We run several simulations with varying degrees of the polynomial regression and observe an interesting behavior. The polynomial of the first degree constantly finds a solution that is lower than the reference solution (which we regard as the "true" solution), for any number of simulation runs (Figure 2). When increasing the degree to two or three, we obtain solutions, which converge very early and nicely to the true value (Figure 3 and Figure 4). But when increasing the polynomial degree even more, we start getting unreasonable results, with exercise times becoming heterogeneous (Figure 5), price convergence towards a too low number (Figure 6), and absolute broken behavior regarding the optimal exercise time. The problem is that the regression matrix develops a singularity problem, as also described in the Mathworks Documentation (2016): "Since the columns in the Vandermonde matrix are powers of the vector x, the condition number of [the matrix] is often large for high-order fits, resulting in a singular coefficient matrix." Particularly, when choosing the degree to be 12, the predictor of future payoffs breaks down, leading to an absolutely distorted image, and resulting in all options being exercised at their inception. Thus, the option price is just its intrinsic value at $t = 0$, i.e. $K - S_0$.
Therefore, we choose the third degree to set as default.

Weighted Laguerre polynomial As described above, we also implemented a Least-Squares Regression onto a basis of exponentially weighted Laguerre polynomials. Unlike in Longstaff and Schwartz (2001), we however scale the argument to the basis function by dividing by S_0. This way, we obtain nicely converging solutions for varying orders of the polynomial, even when increasing the degree to twelve (Figure 8 - Figure 13). The fact that the matrix does not break down when using a high-order polynomial shows that this method is much more stable and therefore favorable for our purposes compared to the polynomial fit on a monomial base described above.

Kernel Smoother Figure 14 shows a slight convergence that is below the reference solution. The (only) good thing about the Kernel Smoothing technique we employ is that it can handle arbitrary dimensions. It is however extremely slow to calculate, which lies in the nature of how

Kernel Smoothers work: A regression needs to be done for every point in which the conditional expectation is to be evaluated, for which many intensive calculations have to be made. For this reason, in order to compute the convergence to a price, we could only raise the number of simulations to 15 000 (instead of 150 000 as for the other methods), as the computation would have become too intense and would have taken too long.

3.2.2 Path Choices

Heston We now alter the path generation to follow the Heston Model (instead of Black-Scholes). We choose $\theta = \sigma^2$ and the starting point to also be σ^2 because with these parameters Heston is "similar" to the reference Black-Scholes paths (expected volatility is the same for both). Figure 15 shows that on the one hand, the price nicely converges, but that on the other hand, it is also slightly higher than the reference solution. The crucial difference of the Heston path generation compared to the Black-Scholes path is ξ, the parameter that describes the volatility of the volatility. When increasing this parameter, the option price slightly increases in our experiments, indicating that one gets a risk premium for the uncertainty in the volatility.

CEV In the CEV model we also chose parameters such that the path generation is "similar" to the reference paths, i.e. the volatility at S_0 is equal to the volatility of the Black-Scholes paths. The resulting price converges nicely, but seems to be consistently larger than the reference solution, as can be seen in Figure 16. The difference of CEV compared to Black-Scholes is that the volatility is innately different in CEV methods. The volatility depends on the stock price, and as the stock price changes in every time-step, so does the volatility, which results in a slightly different option price in the end. It also appears that the exercise times are more homogeneously distributed than in other examples, although the majority of paths are still exercised at the end of the cycle.

Jump-Diffusion Figure 17 incorporates a Jump-Diffusion, with $\lambda = 1$, and $\phi = -0.6$ meaning that the underlying is expected to crash once during the observed time interval, and when it crashes, 60% of its value will be lost.[4] Since put options benefit from collapsing prices, the resulting value

[4]Note that a Black-Scholes path is a version of a Jump-Diffusion Process, with λ set to zero.

is much higher in this example compared to the previous ones. Furthermore, as the probability of crashes at each new time-step is not subject to change, even if the underlying has already crashed in the past of the same path, it makes sense for market participants to wait and not to exercise early, as the market could still collapse until expiry. The market's loss (60%) and the expected number of crashes is too huge compared to the interest rate that early exercise is almost never a good idea. This is unless the market has repeatedly collapsed and the asset's value is approaching zero. In this case, the marginal utility of an additional crash is less than exercising early and investing the gains at the risk-less rate.

Sobol Sequences Instead of generating the paths randomly (using `MATLAB`'s *randn* function), one can choose the points more strategically in order to "fill out" the space better. This way, the selections of points is a low-discrepancy sequence, or Sobol sequence. However, there seems to be a problem with this method. While *randn* generated paths as e.g. in Figure 1, Quasi-Monte Carlo generates paths as in Figure 19. The top left image depicts 1 000 paths, starting at $S_0 = 80$. It is obvious that this is not even close to random anymore, as there emerge only eight possible ways the stocks follow with nothing in between. When slightly lowering the number of simulations to 987, the result is what can be seen in the top right window, and this is even more mysterious. Not only is a structure in the paths present (with the paths still being quite predictable), the structure is very different now. Even when using tenfold as many paths, this odd behavior remains. Therefore, the calculated value of the option is obviously worthless. Its abnormal structure can be seen in Figure 20.

We present in Figure 21 a histogram and a QQ-plot for pseudo-random sequences, Sobol sequences, and scrambled Sobol sequences in order to verify that the odd paths do not stem from a faulty generation of quasi-normally distributed samples. We clearly see that, as far as the frequency of observations are concerned, the used sequences "look very normally distributed". The same is true if we consider the first few sample moments. We therefore hypothesize that the path generator exhibits some subtle dependence on the sequence of random numbers that is not captured by the usual statistical properties. Therefore, we see Sobol sequences unfit to be used with our path generators.

3.2.3 Choice in the Characteristics of the Options

Two Dimensions with Monomial Basis We now explore the case of two underlying assets. As the weighted Laguerre method only works for one dimension, we choose a two-dimensional monomial basis. Furthermore, we purposely set the volatility of each of the two assets to $\sqrt{2}$ times the σ of the previous (one-dimensional) examples in order to obtain a portfolio volatility of the same value as in the one-dimensional case. In Figure 22 one can see a nice convergence to almost the exact price as our one-dimensional reference solution. As in the low-degree one-dimensional polynomial examples, most options are being exercised at the end of their life-span and execution also becomes less frequent as the expiration date approaches.

Three Dimensions with Kernel Smoothing We now extend the option to have three underlying assets (Figure 23). Since we only have Kernel Smoother available for dimensions larger than two, we have to rely on that. Unfortunately, this exhibits horrible convergence (if at all). This is somewhat to be expected considering that the method converges much slower with Kernel Smoother and the conditional expectation now depends on three explanatory values. We can therefore not conclude whether this experiment runs correctly.

Time Dependency Let us now modify our American option to become a time-dependent option, i.e. a so-called Asian option. Figure 24 illustrates the resulting prices, when we set the payoff to the strike price K minus the mean of the stock price of the last τ days, where we set $\tau = 5$. Now, the option will almost always be executed on the very first possible time-step (which is $t = \tau$ in our example, as it does not make sense to take a mean of the past τ days if that many days have not even passed yet). The intuition behind the early exercise times is that when taking a mean of the stock price over several days, the volatility is smoothed out. However, a lower volatility of an underlying implies a lower price of the option. The reason for this is the absence of a downside-risk, i.e. a high volatility means a chance of large gains without the risk of suffering from large losses, as the lowest possible payoff is capped at zero.

Now, the volatility is too low and the risk-free interest rate too high to have an incentive to keep

the option. Therefore, the value of the option is in the ballpark of the expected stock price at $t = \tau$, discounted with the risk-free interest rate: $exp(-r\tau)$.

3.3 Concluding Remarks

The implementation of the weighted Laguerre method with the polynomial fit seems to function well for our purposes. The calculated solution matches the one given in the reference. Both Heston and CEV methods also seem to be consistent with our expectations and are well in-line with the reference solution, thus providing a good approach to be used in valuation. However, the code we chose to implement shows great sensitivity to polynomial degrees. If at large, these imply a breakdown of the computation of expected future payoffs as the regression-matrix becomes rank-deficient. Time-dependent as well as multi-asset options yield respectable and intuitively consistent results. Their price-convergence with higher numbers of simulation-runs is also observed.

Unfortunately, the Kernel Smoother was not a satisfactory technique for our purposes. While on the plus side, it can handle arbitrary dimensions, the down side is that it is extremely slow in computing the value of our example options. For illustration, the generation of a figure as Figure 14, but with up to 150 000 simulation runs, has not even been half way through yet after 24 hours of computation, while the other regression methods only took a couple of minutes. The fact that the accuracy of the price prediction is worse than when using other methods makes it an even less useful method to regress the future payoffs of American options.

As the Sobol method generates weird paths, its solution is worthless. Its behavior when generating paths is puzzling to us, considering that the implemented Sobol code seems to be correct, as discussed above.

4 Discussion

In this section we would like to discuss a possible shortcoming of the least-squared method introduced in Longstaff and Schwartz (2001), and by extension all the other variants of regressing future discounted payoffs to use it as a predictor.

As described in Section 2, we take a regression at each time step to predict the payoff when keeping

the option instead of exercising. When it is beneficial to exercise, we update the payoff vector. Then, when going on to the time step prior to that, we use the updated payoff vector to make a new regression. Now, let us take a closer look how the discounted payoffs at time step $t = 25$ look like in Figure 25. For every simulation run there is a blue scatter point that represents the realized discounted payoff when keeping the option at that time step conditional on the current stock price. However, an artifact in this plot is conspicuous: For stock prices between approximately 60 and 80, a clear line with zero variance instead of a cloud of scatter points is depicted. In order to understand what is going on, let us take a look at Figure 26, a simple model of some form of regression on discounted future payoffs (red line). This regression line is compared to the payoff when exercising, which is a linear relationship to the current stock price (blue line). Every time when the blue line is above the red line, exercising now is beneficial, and thus, the payoff vector is updated to new values. Note that the values which are updated contain no variance but a clear linear relationship to the size of S_t. This phenomenon is apparent when looking back at Figure 25.

This behavior is inevitable. For any type of regression that we use, the regression line is curvy and crosses the straight line at some point. Nevertheless, this does not seem to impair the option price valuation in the end in any way.

5 Conclusion

In the scope of our project, we built a software package to compute the prices of broad types of options based on Monte Carlo simulations. We allow for great flexibility regarding three aspects: Firstly, the type of options can take an arbitrary form, e.g. options that are dependent on a single or multiple underlyings, options whose value depends on past outcomes or options that are exercisable only on particular dates etc. Secondly, we allow for flexibility in the choice of path generation parameters, i.e. we implement different types of methods that model an asset-paths following a random process. Thirdly, we allow for parameter tweaking regarding the market or the underlying, e.g. risk of crashing markets, risk-free interest rates, variance of the stock price, characteristics of the variance of the stochastic process etc.

We observe that some models seem to be more suitable for our purposes than others. In particular,

we regard the Kernel Smoother as a poor technique to regress the payoffs because it is computationally unfeasible, and the solution in the end did not turn out to be more precise than other techniques' solutions. As a result, we faced difficulties in valuating options dependent on more than two underlyings, as we only have the Kernel Smoother available for dimensions upward of two. Furthermore, we see Sobol sequences unfit to be used with our path-generators as they seem to exhibit some subtle dependence on the sequence of random numbers and hence fail to generate unstructured stock paths.

However, the other methods that we implemented (e.g. polynomial fit (with low-order degrees) on a monomial base and the weighted Laguerre polynomial for the regression problem and Heston, CEV, and Jump-Diffusion for the path generation) worked nicely and provided satisfactory results.

References

Black, F. and M. Scholes (1973, May-June). The pricing of options and corporate liabilities. *The Journal of Political Economy 81*(3).

Chesney, M. (2015). Continuous time quantitative finance. Lecture Slides FS15.

Jia, Q. (2009, June). Pricing American options using Monte Carlo methods. *U.U.D.M Project Report, University of Uppsala (Sweden)*.

Longstaff, F. and E. Schwartz (2001, Spring). Valuing American options by simulation: A simple least squares approach. *The Review of Financial Studies 14*(1).

Mathworks Documentation (2016). "polyfit" - polynomial curve fitting. Accessed: July 1, 2016.

A Disclaimer

One of us took a seminar, "Applied Topics in Finance" at the University of Zurich, where the seminar project contained the pricing of American put options. Therefore, a minor part of the code presented in this paper resembles the code in there. More precisely, our 800+ lines of code stem on the 85 lines of code written in the scope of said seminar project. These 85 lines contain the bare basic structure of Monte Carlo simulations for American put options (i.e. without any of the extensions introduced in this paper), which, in turn, have been inspired by the code presented in Jia (2009).

B Listings

List of Figures

List of Tables

Listings

C Figures

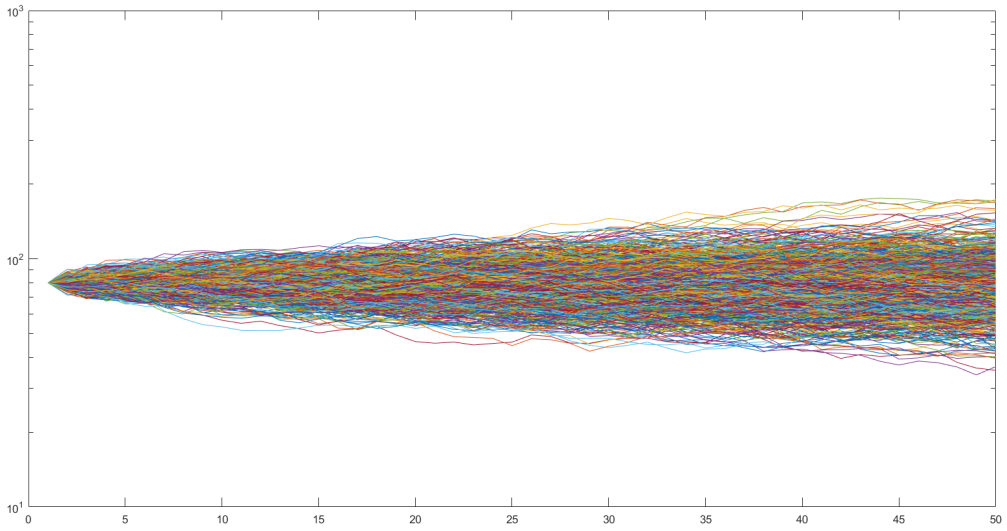

Figure 1: 1000 simulations of a Black-Scholes stock path, with starting value $S_0 = 80$.

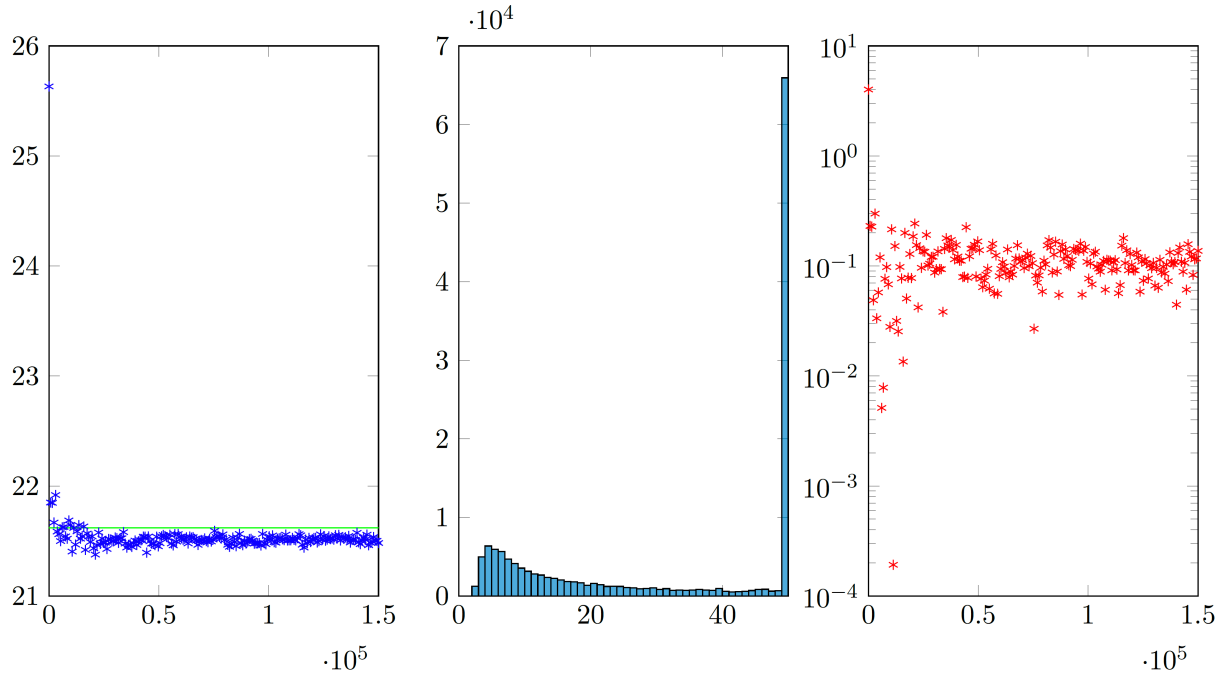

Figure 2: First-degree Polynomial

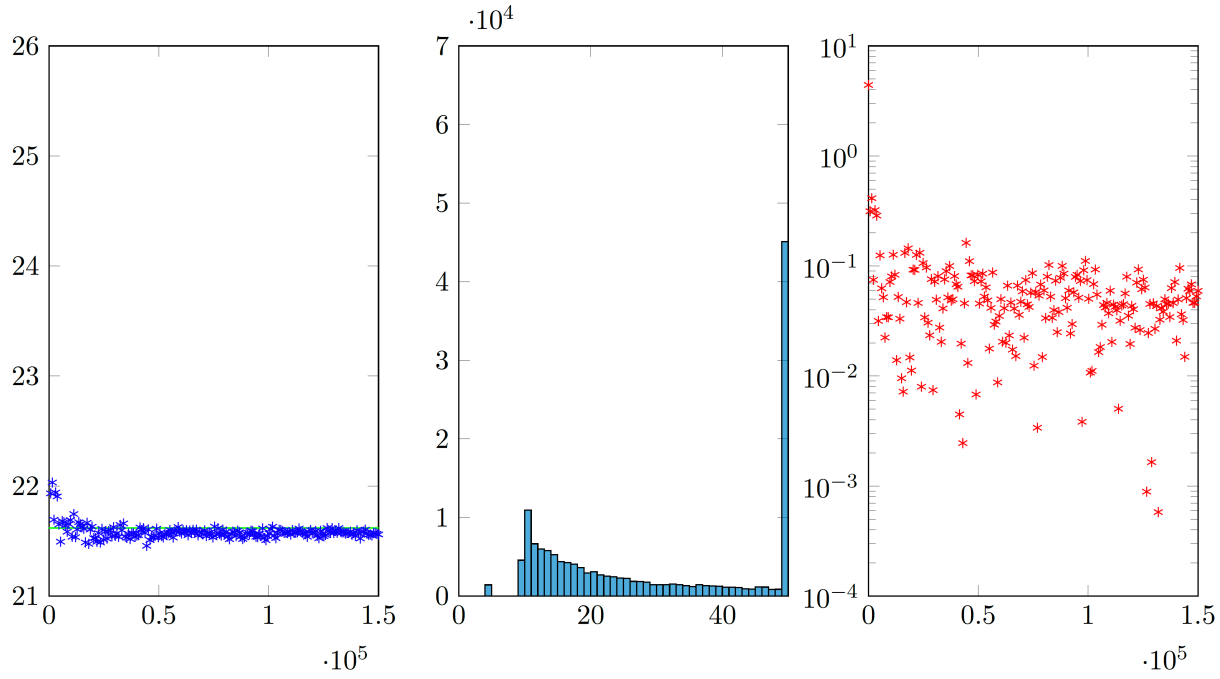

Figure 3: Second-degree Polynomial

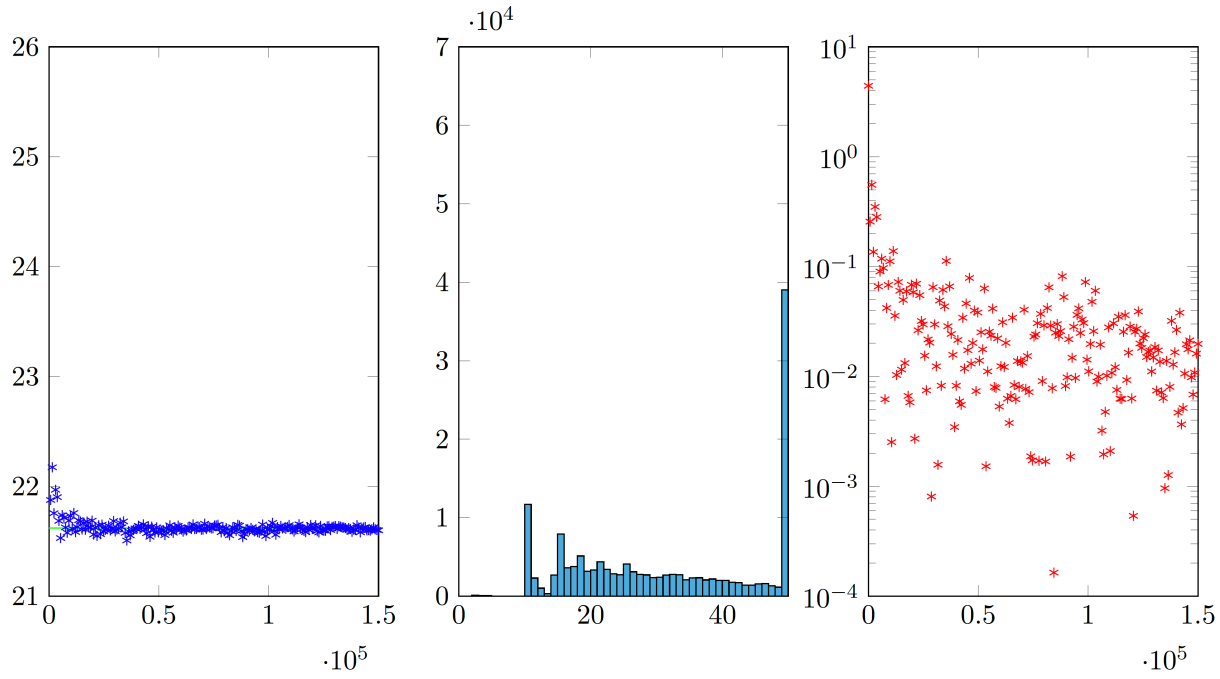

Figure 4: Third-degree Polynomial

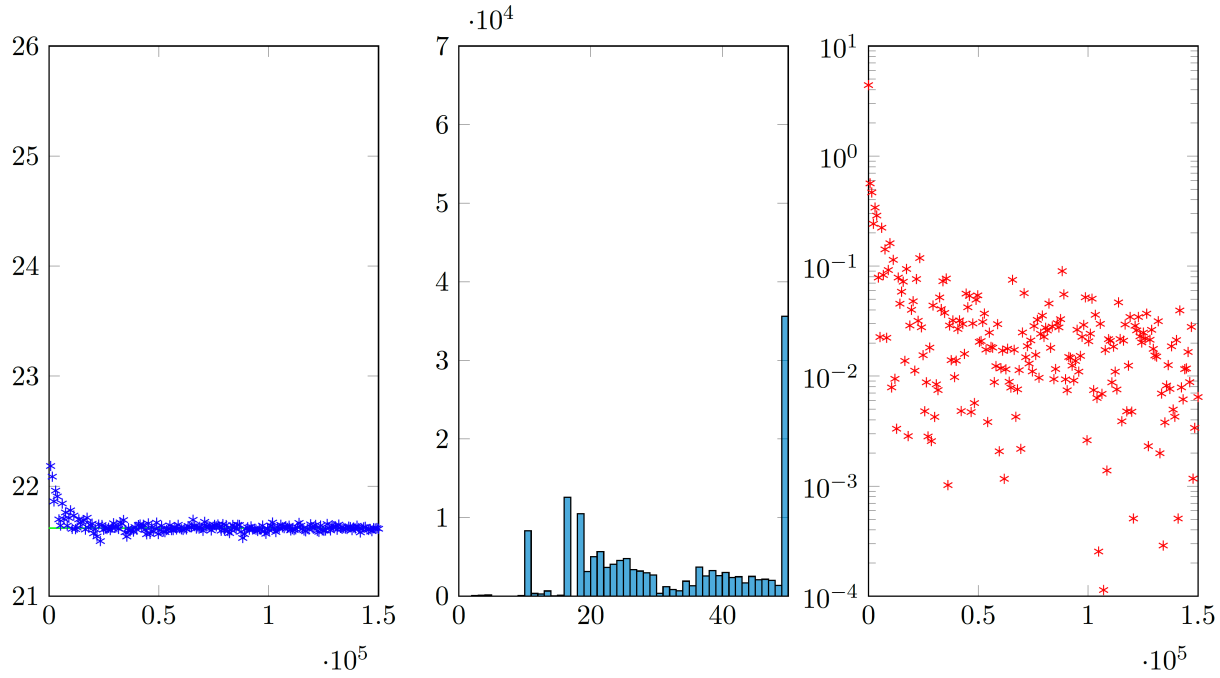

Figure 5: Fourth-degree Polynomial, close to singularity.

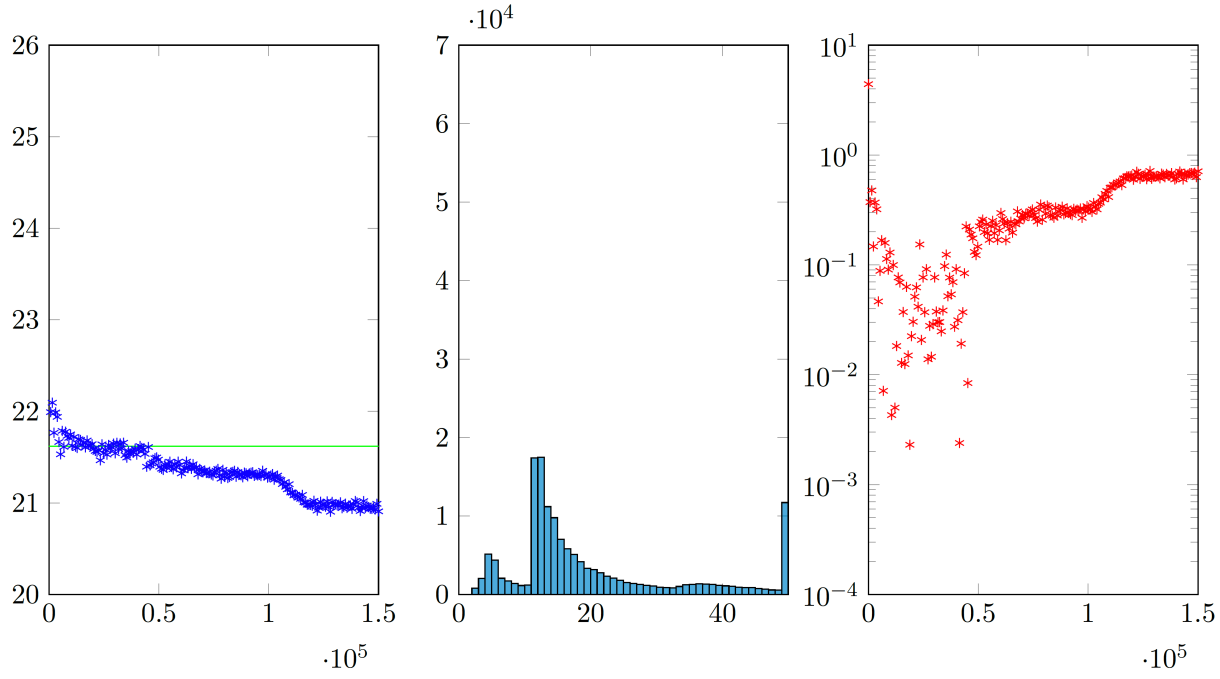

Figure 6: Sixth-degree Polynomial, close to singularity.

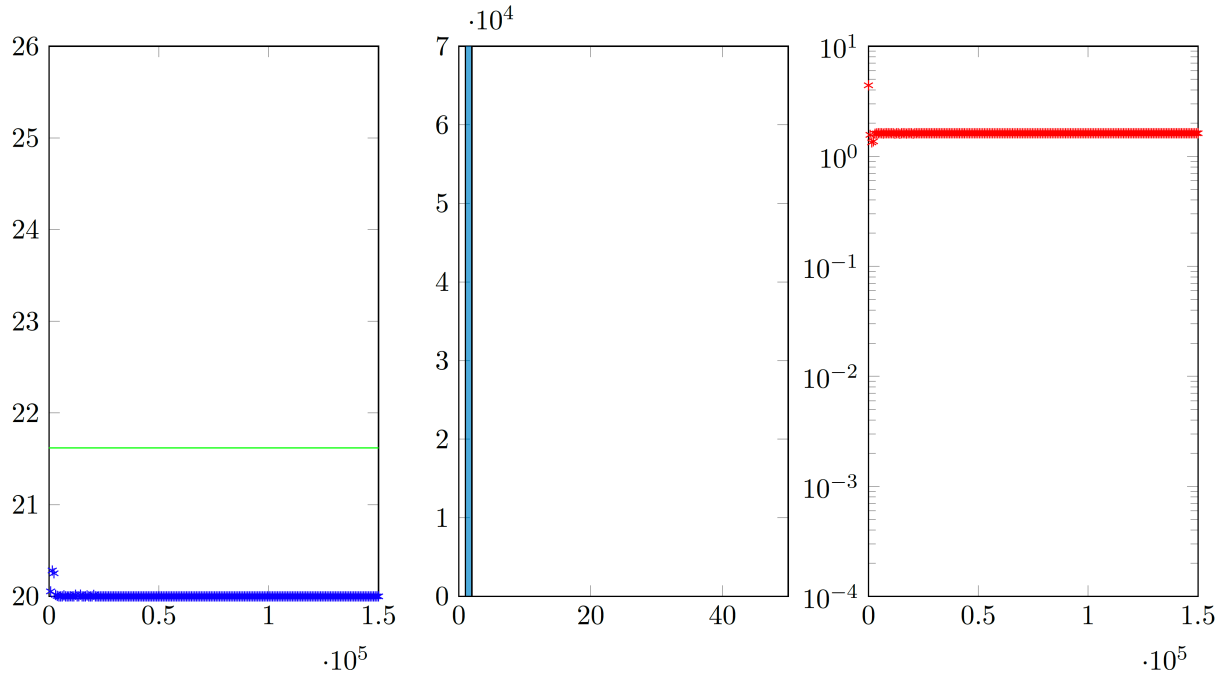

Figure 7: Twelfth-degree Polynomial. Note that the regression matrix is now rank-deficient.

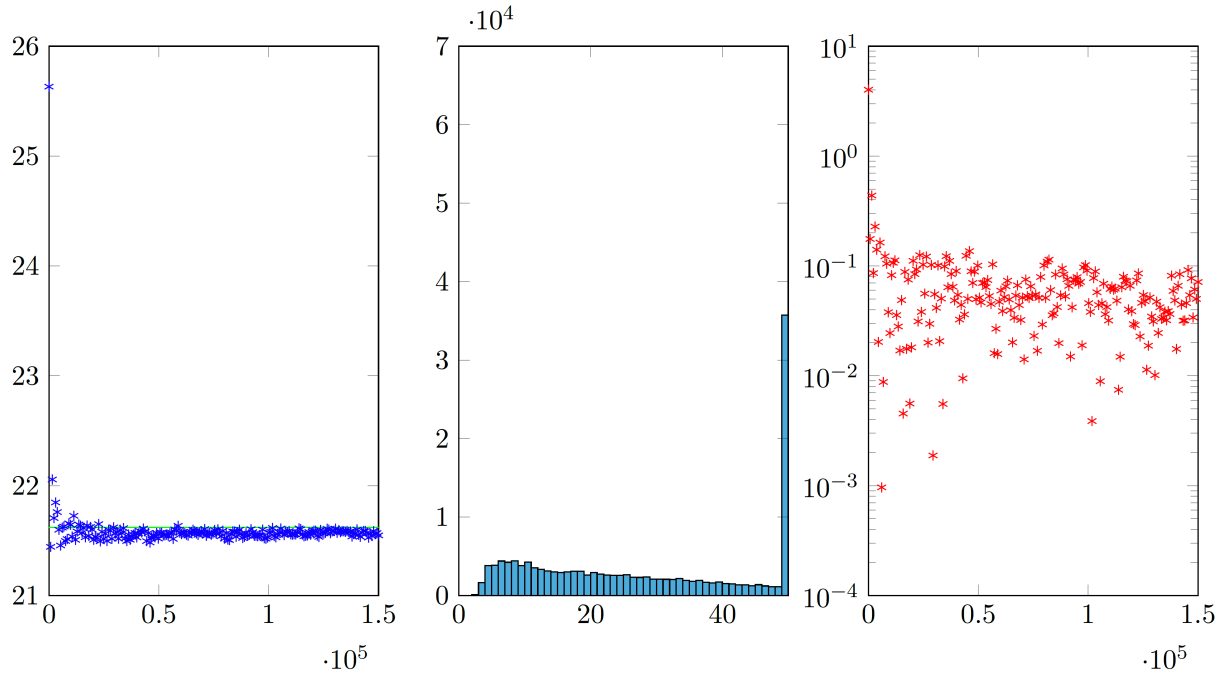

Figure 8: Weighted Laguerre-Polynomial of first degree.

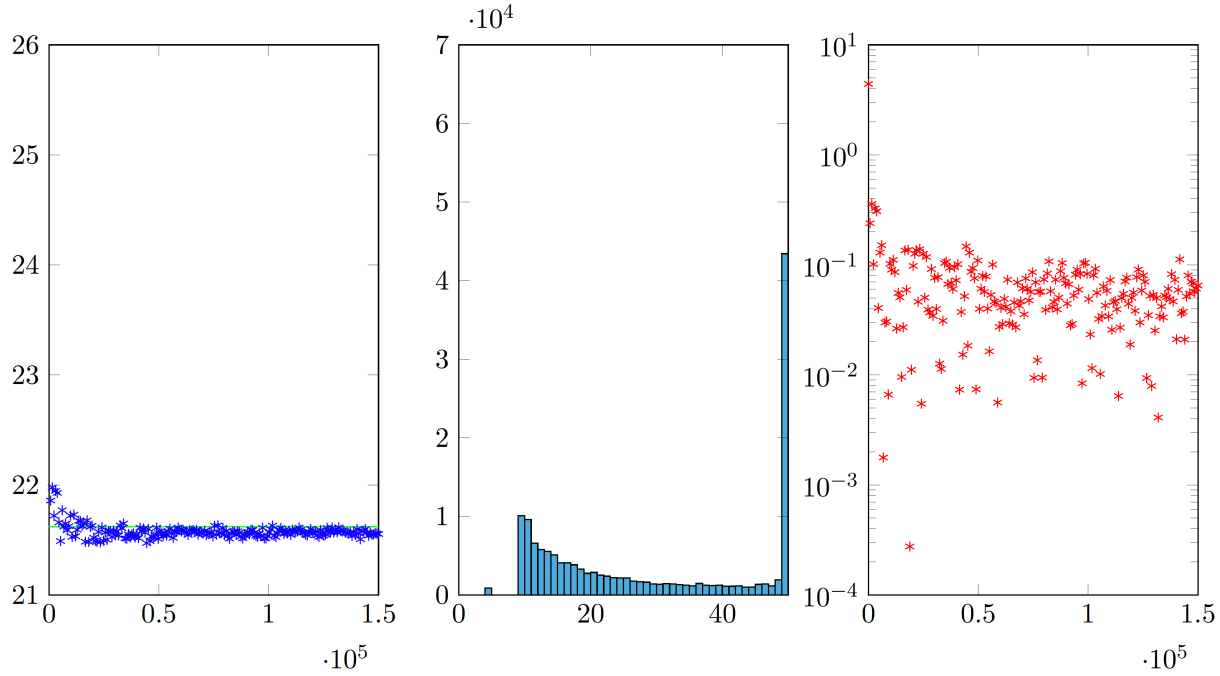

Figure 9: Weighted Laguerre-Polynomial of second degree.

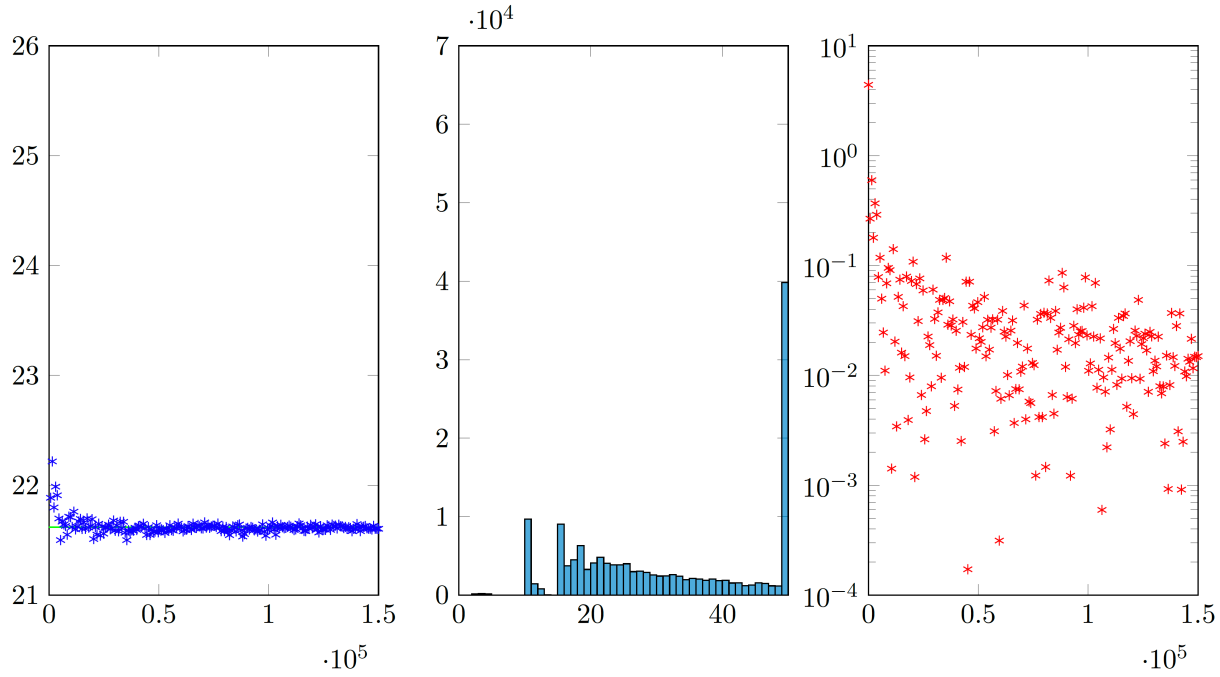

Figure 10: Weighted Laguerre-Polynomial of third degree.

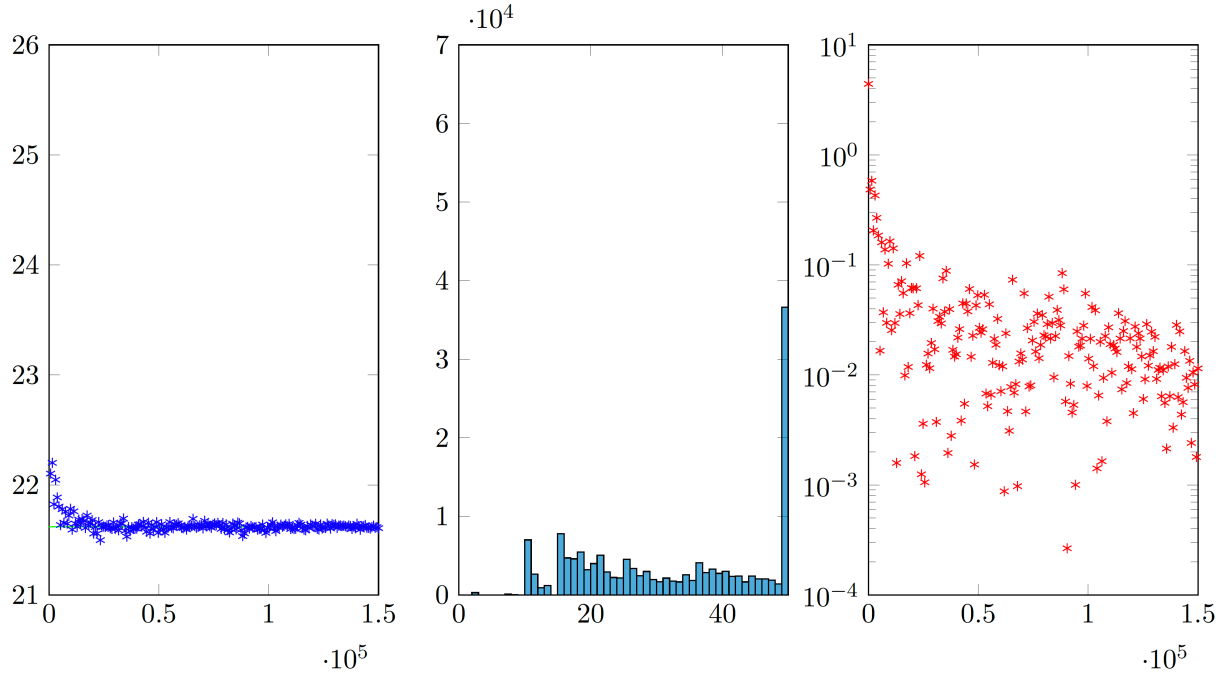

Figure 11: Weighted Laguerre-Polynomial of fourth degree.

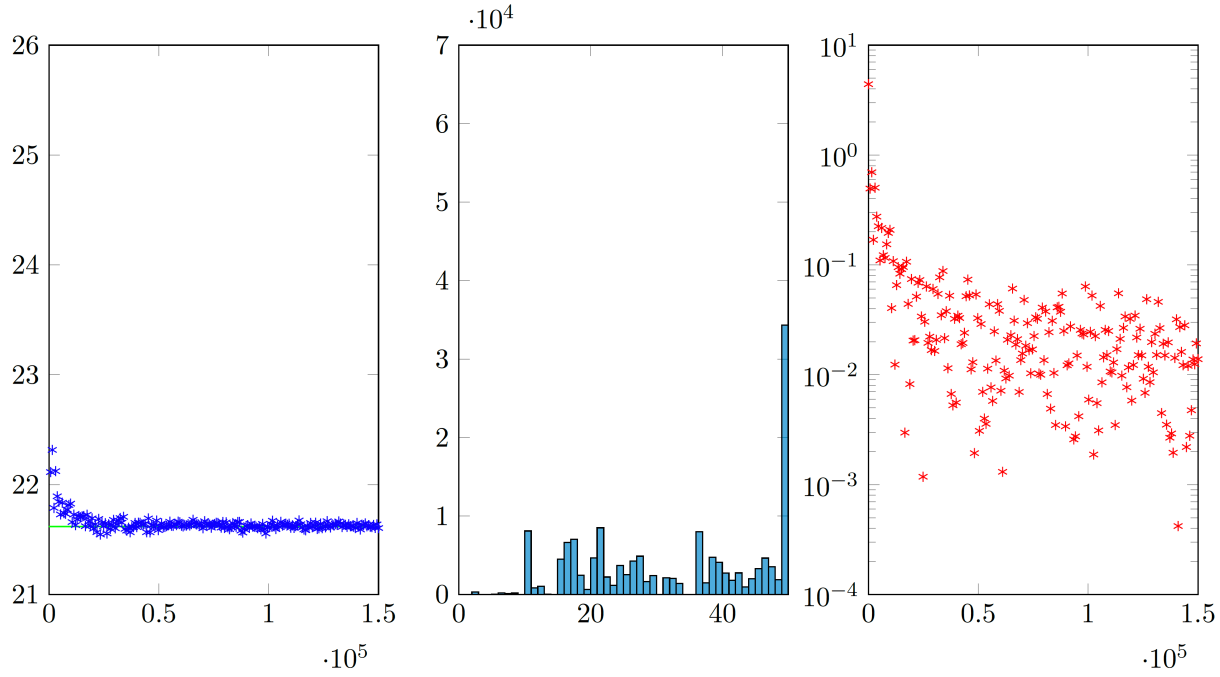

Figure 12: Weighted Laguerre-Polynomial of sixth degree.

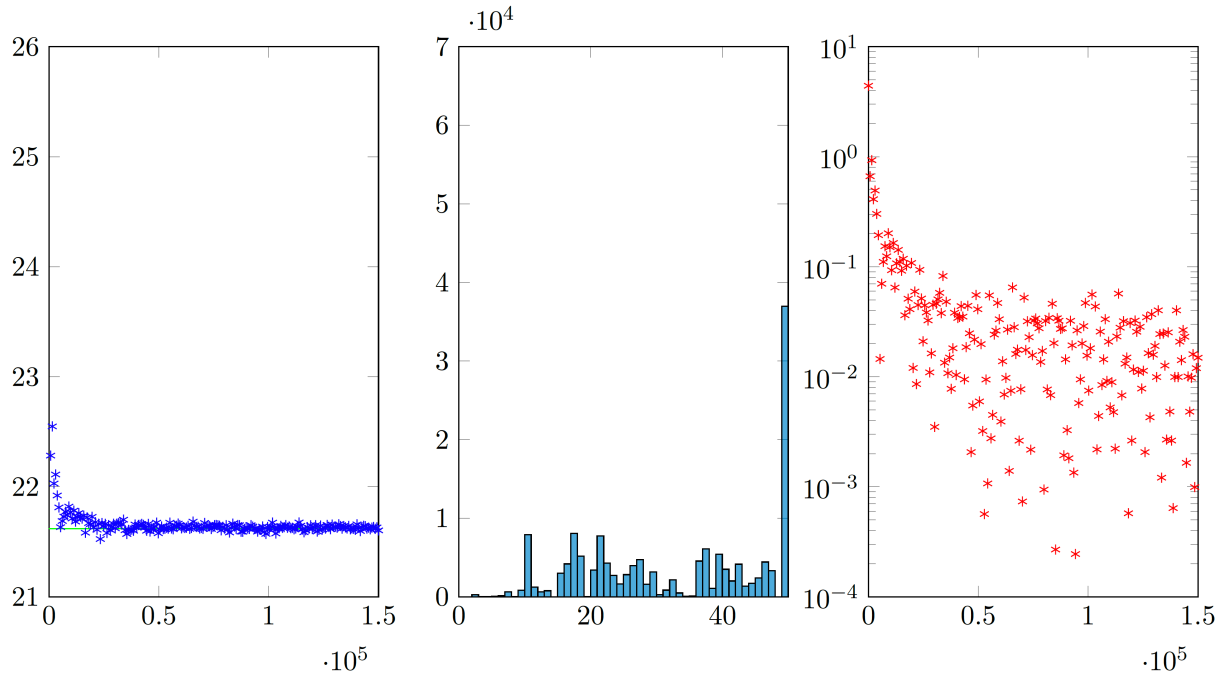

Figure 13: Weighted Laguerre-Polynomial of twelfth degree.

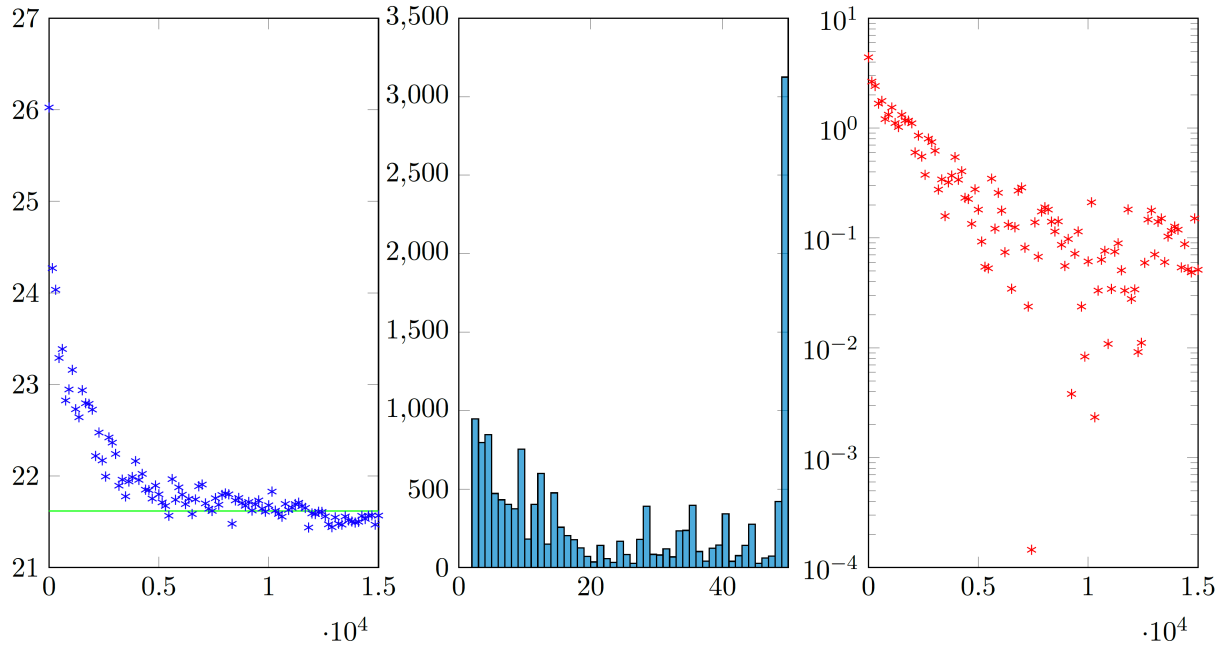

Figure 14: Kernel Smoother (Local Linear Regression)

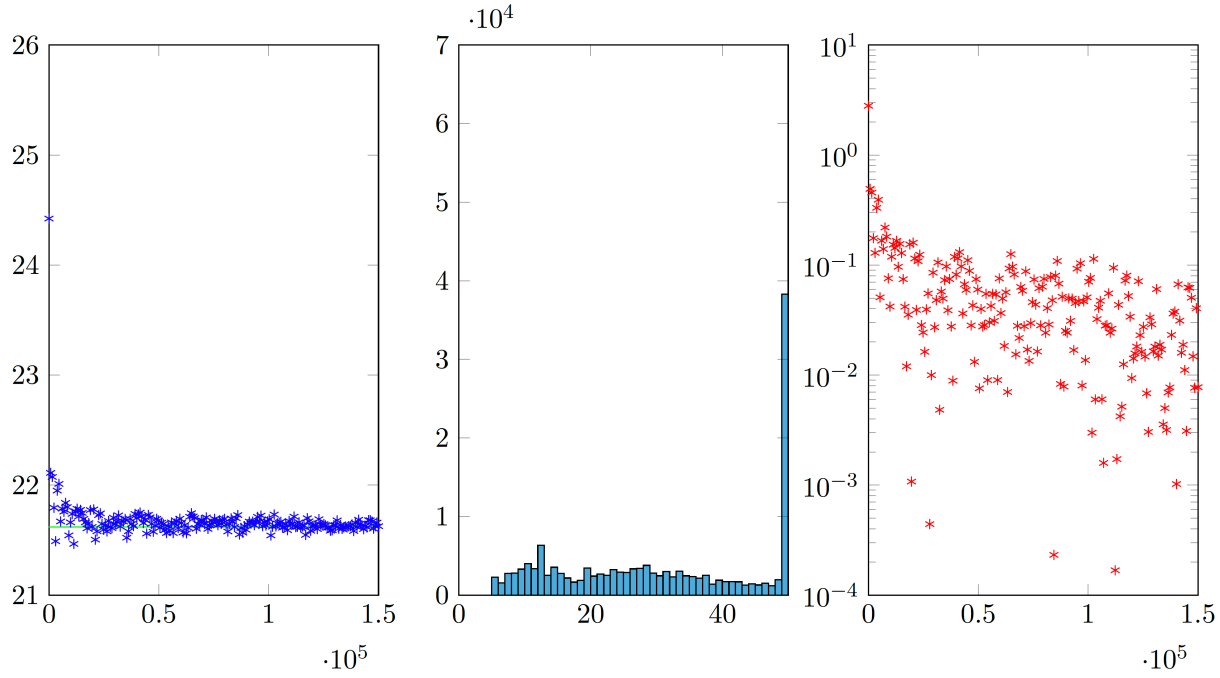

Figure 15: Heston Process

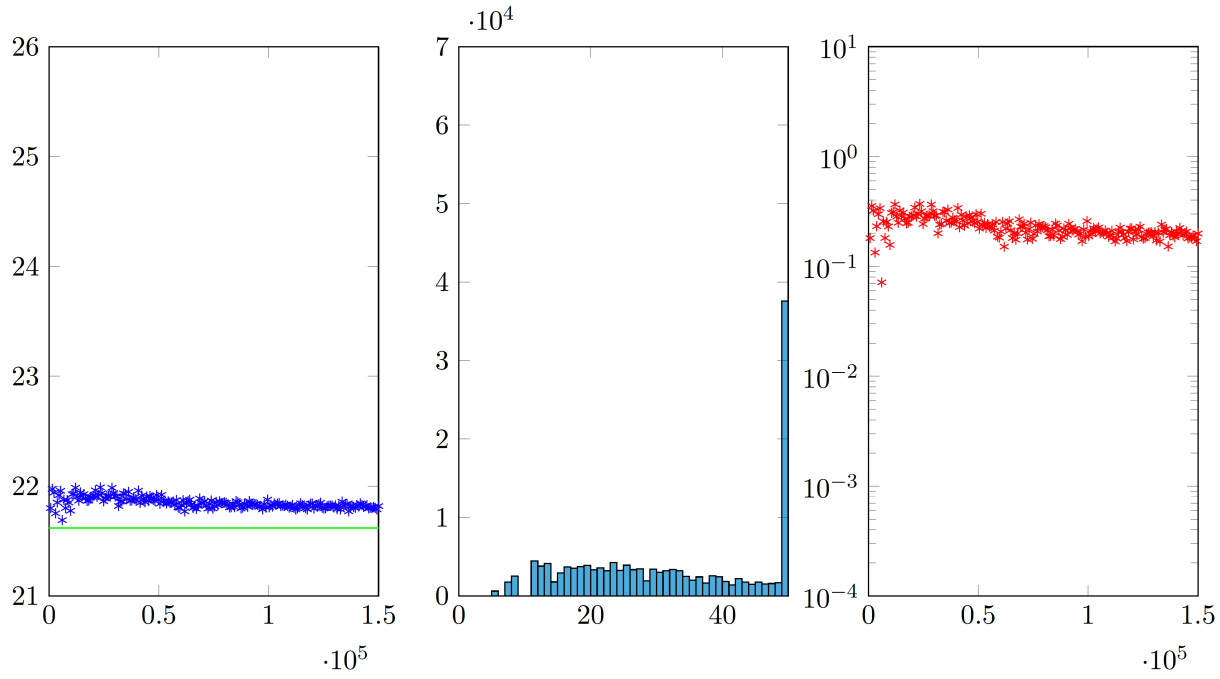

Figure 16: CEV

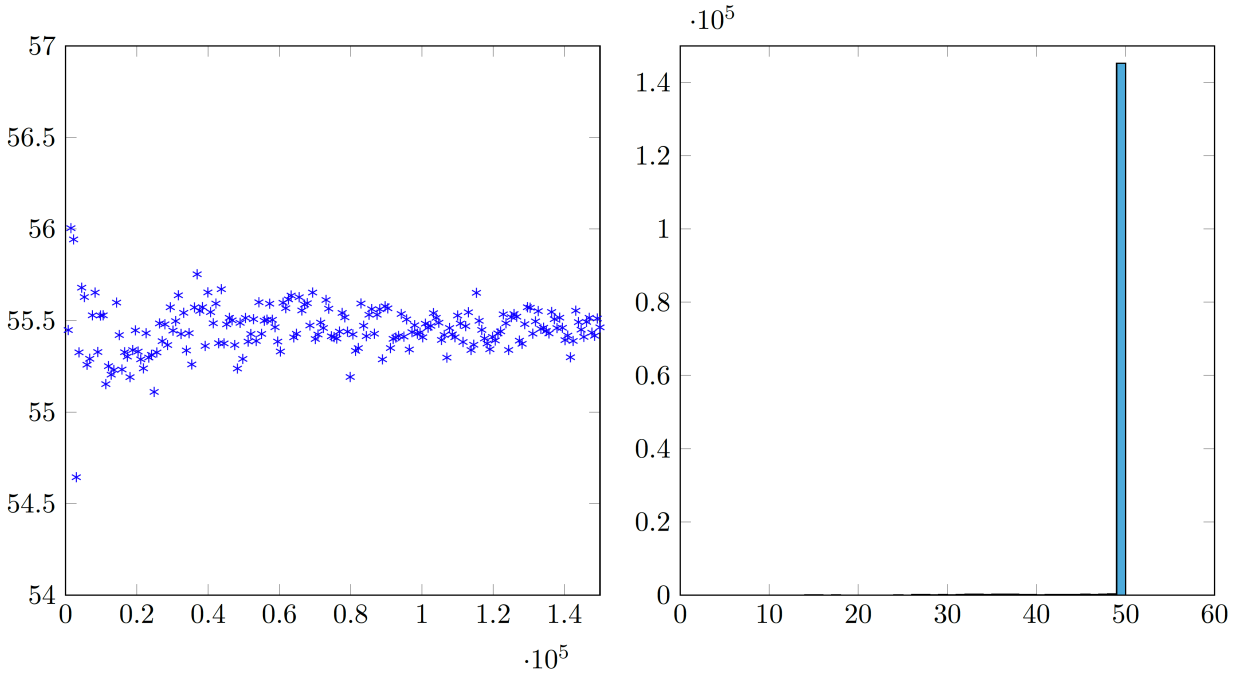

Figure 17: Jump Diffusion with $\lambda = 1$ and $\phi = -0.6$.

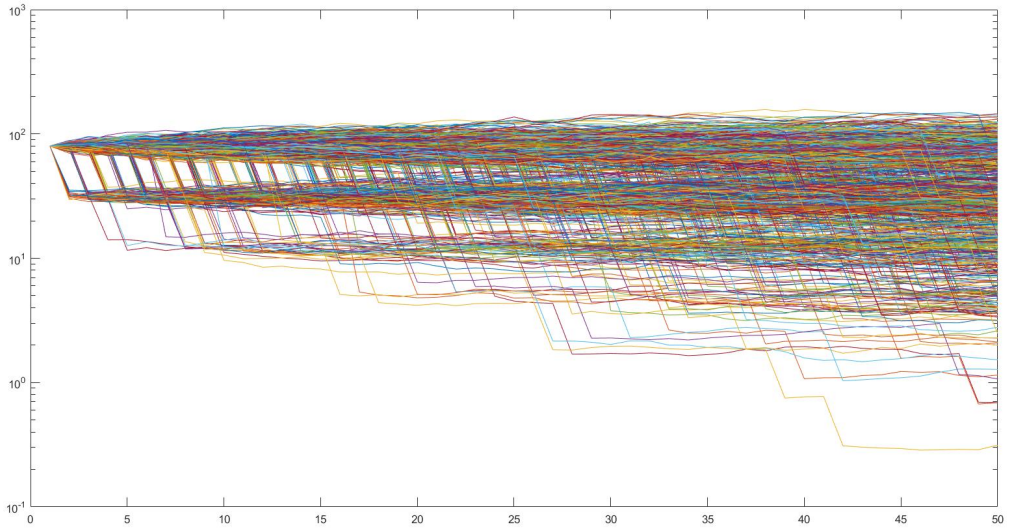

Figure 18: Jump Diffusion stock paths with $\lambda = 1$ and $\phi = -0.6$.

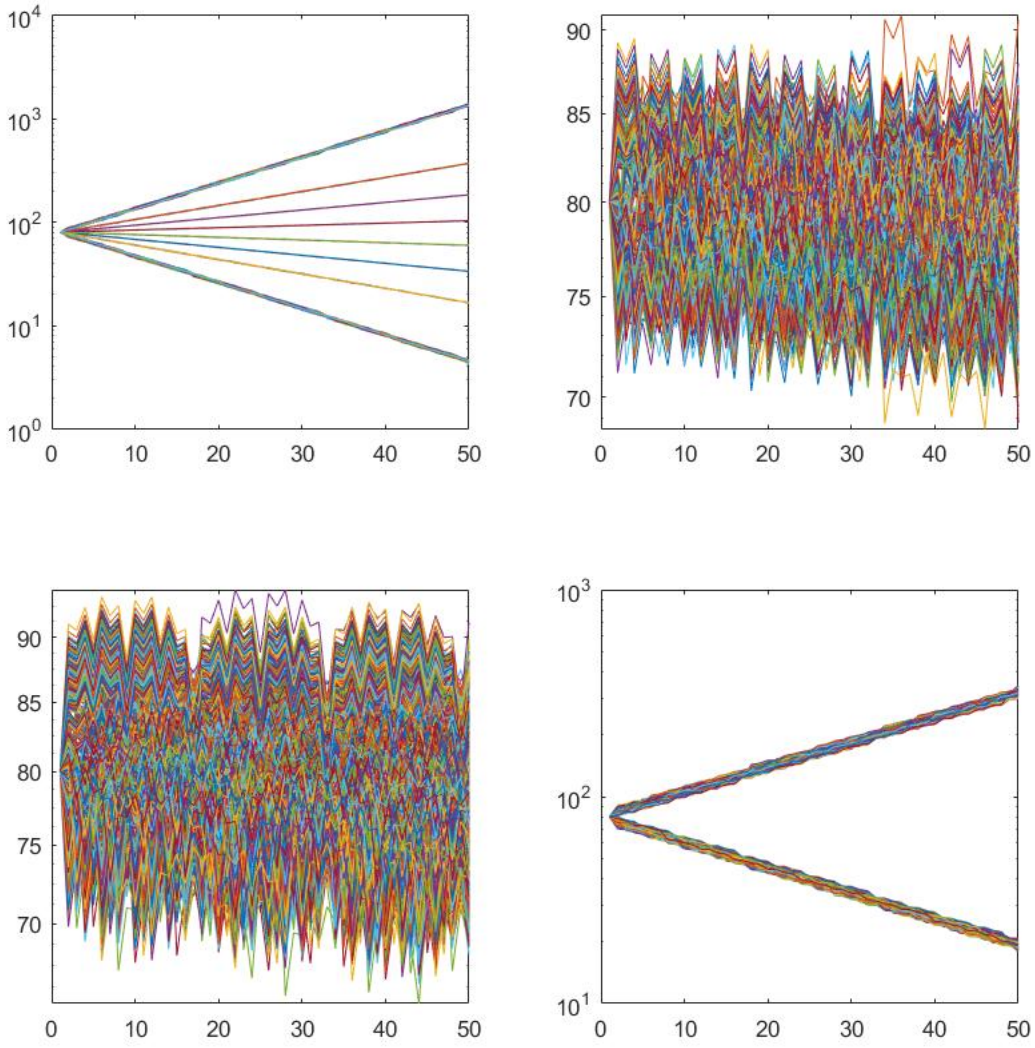

Figure 19: Stock paths generated with Sobol sequences with starting value $S_0 = 80$; the number of simulations are set to 1000 and 987 for the top left and right windows, respectively, and 9871 and 9878 for the bottom left and right windows, respectively.

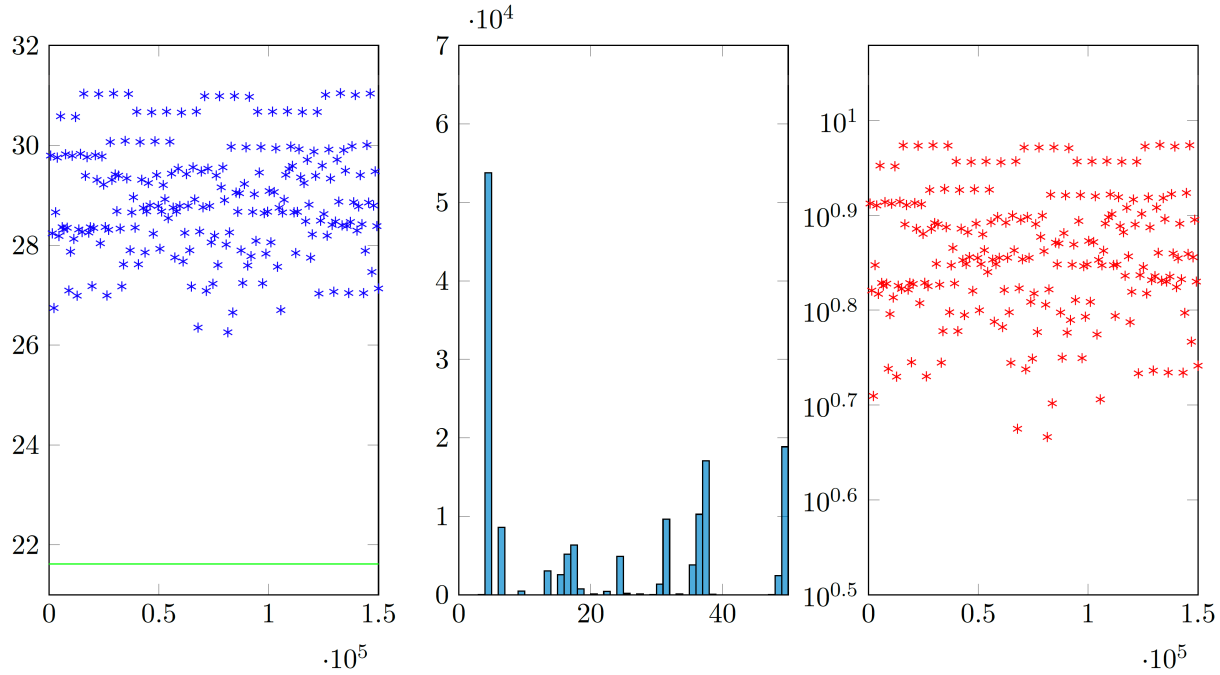

Figure 20: Stock paths generated with a Sobol sequence.

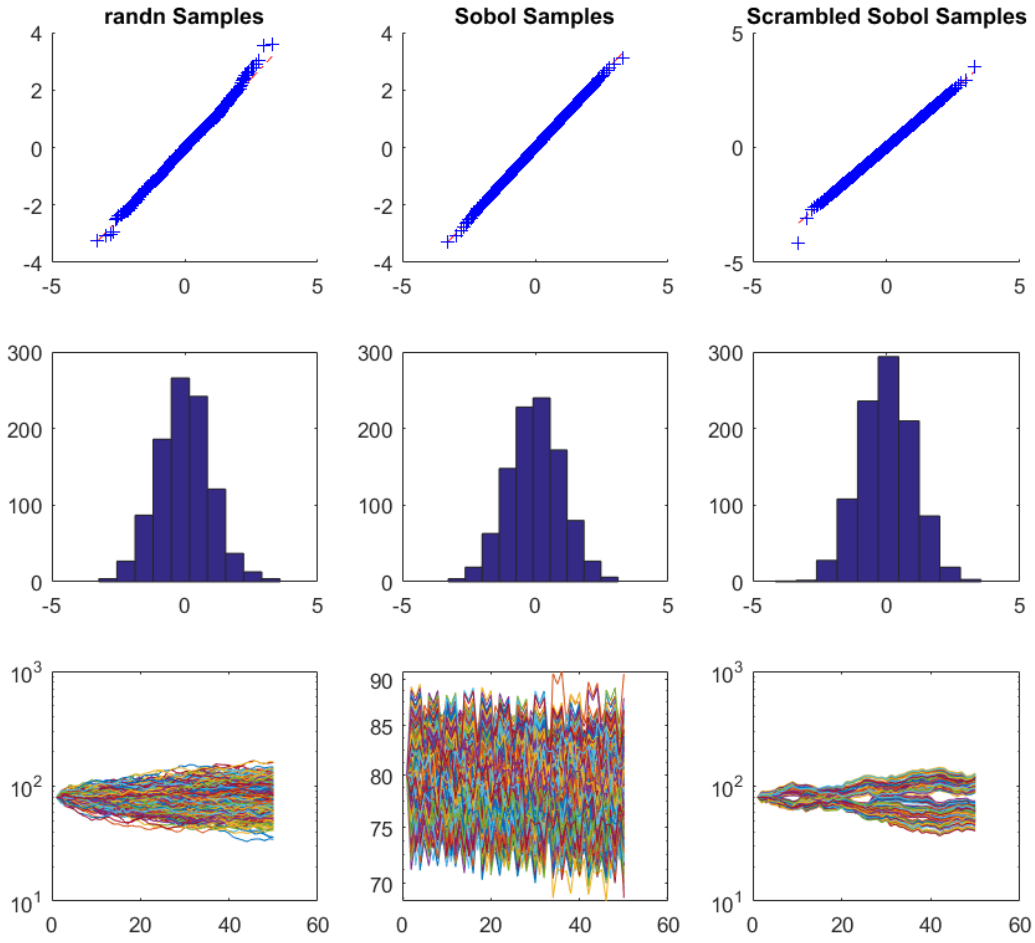

Figure 21: QQ-Plot, histogram and resulting stock paths for pseudo-random, Sobol and Scrambled Sobol Sequences.

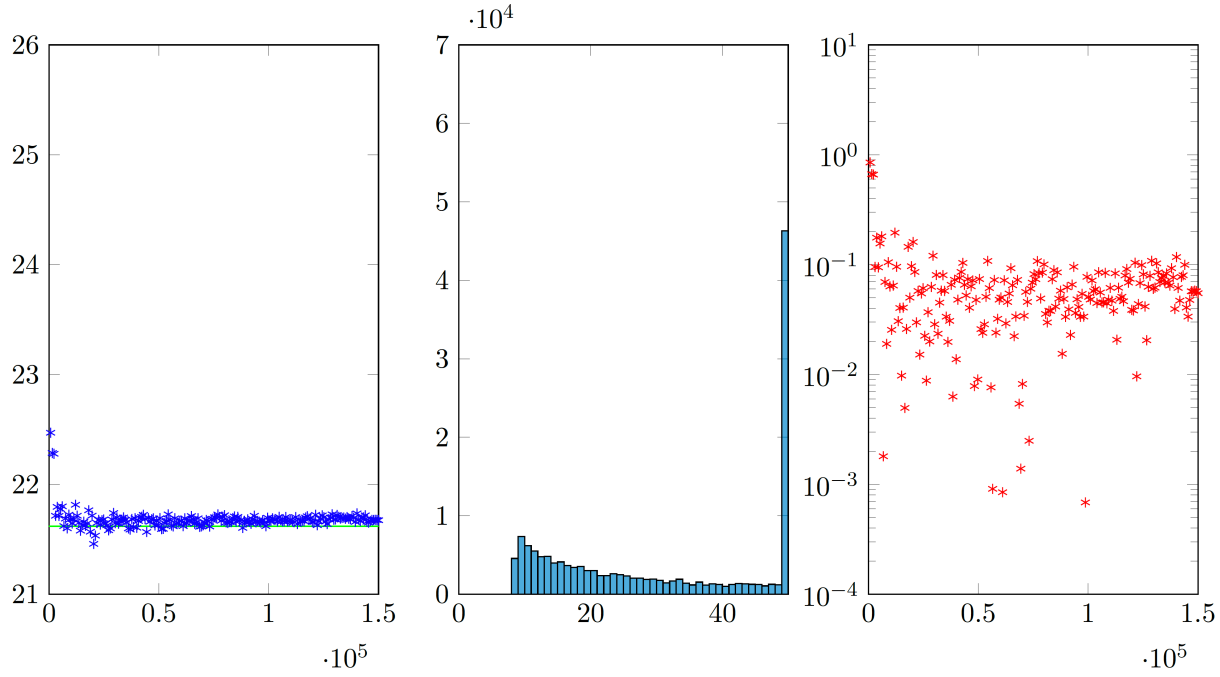

Figure 22: Two underlyings, both with starting value $S_0 = 80$, completely uncorrelated, and each with a volatility of $\sqrt{2}$ times the volatility of an underlying of the one-dimensional cases.

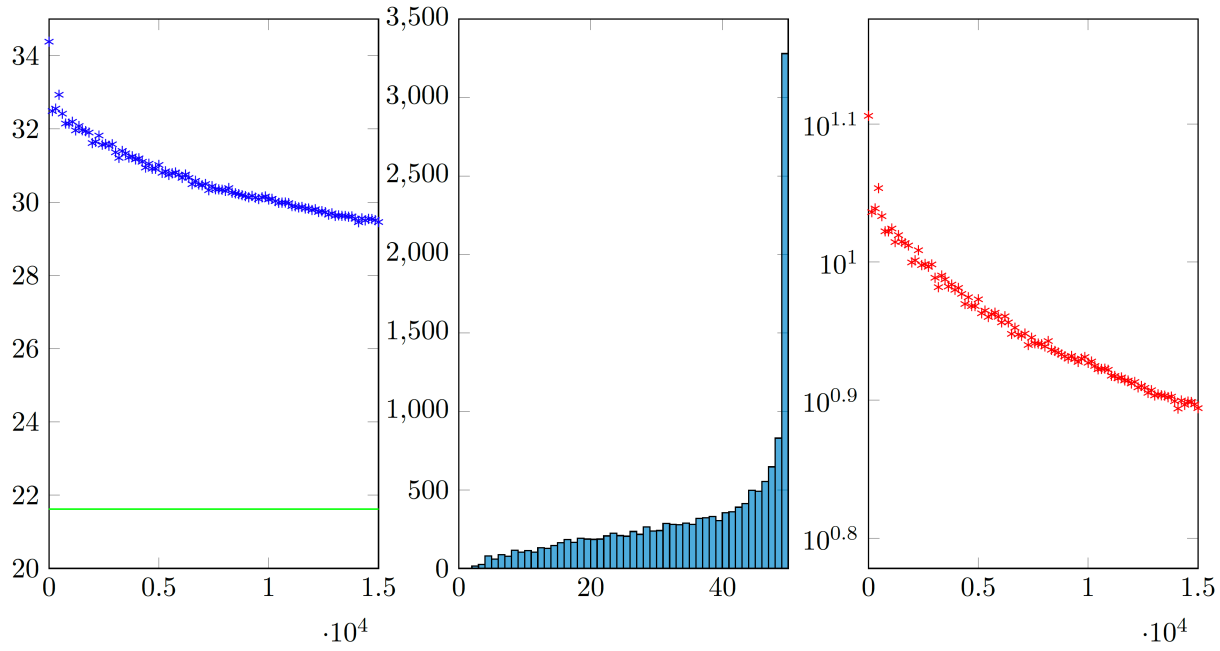

Figure 23: Three underlyings, all with starting value $S_0 = 80$, completely uncorrelated, and each with a volatility of $\sqrt{3}$ times the volatility of an underlying of the one-dimensional cases.

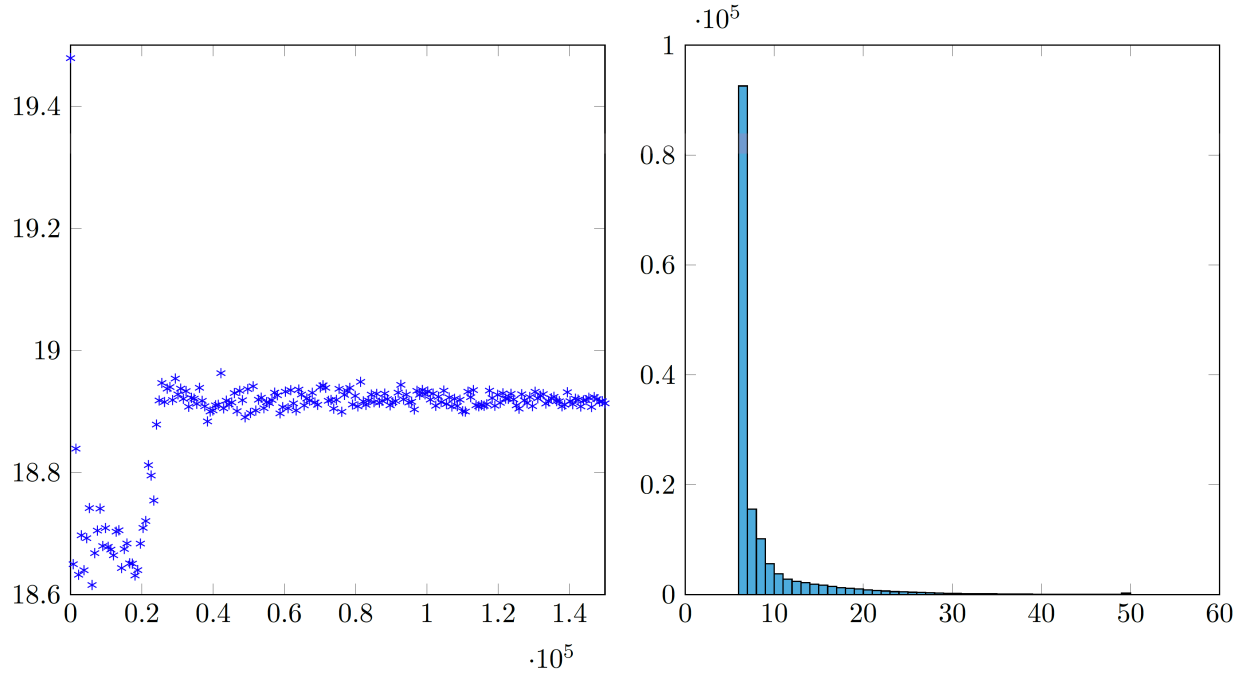

Figure 24: Option with a time-dependent payoffs, where the stock price is average over the last $\tau = 5$ days.

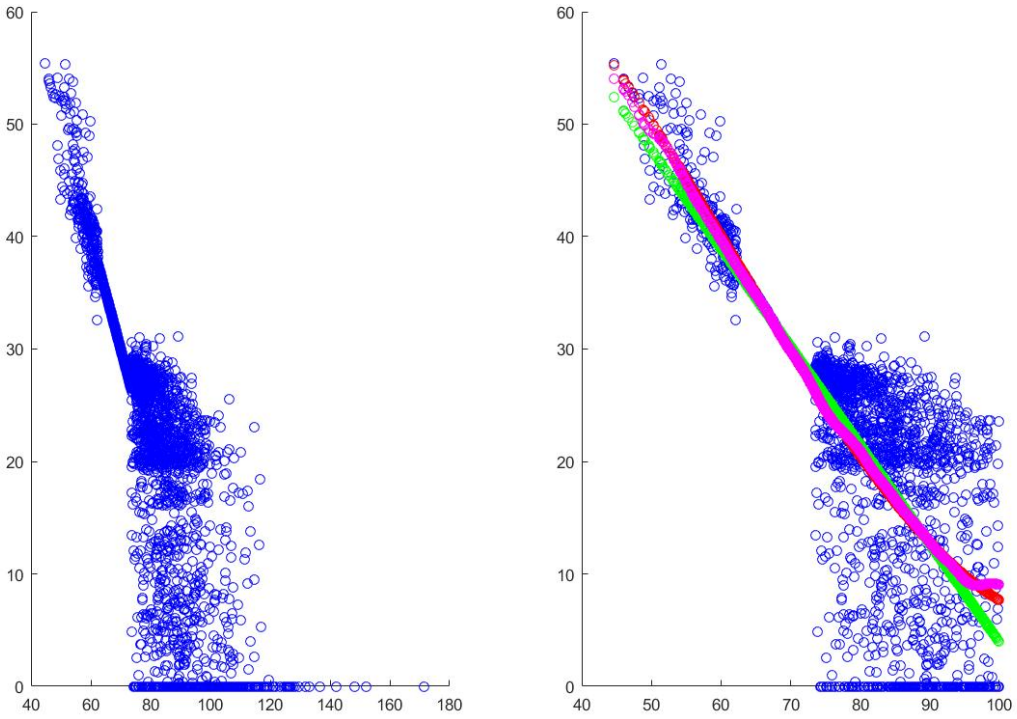

Figure 25: Left: Actual discounted payoffs at $t = 25$ conditional on the stock price at time step t. Right: Regression on the discounted payoffs with third-degree polynomial fit (red points), linear fit (green points), and Kernel Smoother (mangenta points).

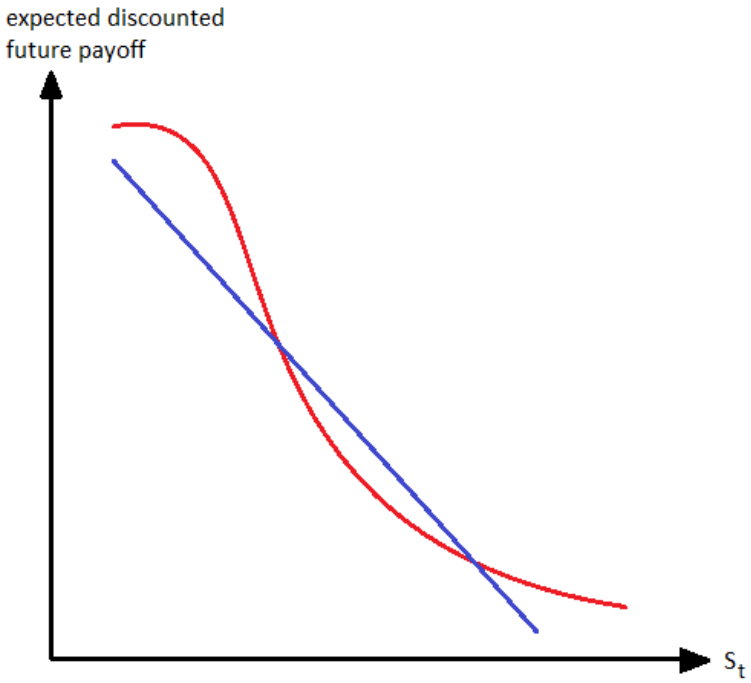

Figure 26: Red line: Expected discounted future payoff (regression line) conditional on the current stock price S_t. Blue line: payoff when exercising today.

D MATLAB Code

D.1 Asset Pricing with Monte Carlo

Listing 1: tests.m

```
function tests(testnumbers)
    M = 3; % Total number of tests
    if nargin == 0
        testnumbers = 1:M;
    end

    for test = testnumbers
        fprintf('Running test number: %i\n',test)
        switch (test)
            case 1
                figure()
                fprintf('\tAmerican Put option, default settings:')
                [value,~,~,~,S,~] = MCAssetPricing();
                value
                semilogy(S')

            case 2
                % American Call option.
                fprintf('\tAmerican Call option price:')
                MCAssetPricing(...
                    'computePayoffs', @(S,t,options) (max(S(:,t) - options.K,0)), ...
                    'r',0.04,'K', 80, 'S0', 100,'seed',42,'basisDegree',3)

            case 3

                % Multidimensional option with memory.
                fprintf('\tMultidimensional option with memory:')
```

```
                tau = 2;
                options = struct( ...
                    'seed',                     0.42, ...
                    'basisDegree',              0, ...
                    'S0',                       [80,80], ...
                    'K',                        100, ...
                    'r',                        0.04, ...
                    'dividends',                [0 0], ...
                    'sigma',                    [0.2,0.2], ...
                    'correlationMatrix',        eye(2), ...
                    'computePayoffs',           @(S,t,options) (max(options.K - ...
                        sum(S(:,((t-tau):t),1),2)/(tau + 1),0)), ...
                    'computeExpectedPayoffs',   @kernelSmoother,...
                    'isExercisable',            @(t,options) (t > tau), ...
                    'getMemory',                @(t) (t-tau):t ...
                );
                MCAssetPricing(options)

        end
    end
end
```

Listing 2: MCAssetPricingOptions.m

```
function options = MCAssetPricingOptions(varargin)

    % ------------------------- Default Options ---------------------------- %

    rng shuffle % To ensure that defaults.seed is chosen differently everytime.
    defaults = struct( ...
        'seed',                     randi(intmax), ...
        'numberOfSamples',          1000, ...
        'computeBasisVector',       @(St, options) cumprod([ones(size(St)) ...
            repmat(St,[1 options.basisDegree])],2), ... % also try laguerre
```

```
        'basisDegree',              3, ...
        'dt',                       1/50, ...
        'T',                        1, ...
        'numberOfTimesteps',        50, ...
        'generatePaths',            @generateJumpDiffusionPaths, ...
        'S0',                       80, ...
        'K',                        100, ...
        'r',                        0.01, ...
        'dividends',                0, ...
        'sigma',                    0.25, ...
        'correlationMatrix',        1, ...
        'generateNormalSamples',    @randn, ... % Try out sobol sequences
        'phi',                      -0.1, ...
        'lambda',                   0, ...
        'computePayoffs',           @(S,t,options) (max(options.K-S(:,t,1),0)), ...
        'isInTheMoney',             @(S,t,options) ...
            (options.computePayoffs(S,t,options) > 0), ...
        'isExercisable',            @(t,options) true, ...
        'computeExpectedPayoffs',   @LSMExpectedPayoffs, ...
        'computeWhoExercises',      @whoExercisesExpectedPayoff, ...
        'getMemory',                @(t) t, ...
        'smoothingBandwidth',       3, ...
        'smoothingKernel',          @(X0,X,options) ...
            exp(-sum((repmat(X0,[size(X,1),1]) - ...
            X).^2./(2*options.smoothingBandwidth^2),2)), ...
        'kappa',                    2, ...
        'theta',                    0.25^2, ...
        'eta',                      0.02, ...
        'gamma',                    1.2 ...
    );

    % -------------------------- Setup Parser ------------------------------ %

    % Custom conditions.
    isfunction = @(x) isa(x,'function_handle');
```

```
42      ispositiveInteger = @(x) isnumeric(x) && (x > 0) && (x == floor(x));
43      isnonnegativeInteger = @(x) isnumeric(x) && (x >= 0) && (x == floor(x));
44      ispositiveNumber = @(x) isnumeric(x) && (x > 0);
45      iscorrelationMatrix = @(x) isnumeric(x) && all(diag(x) == 1) && ...
            ispositiveDefinite(x) && issymmetric(x);
46
47      % Initialize parser.
48      p = inputParser();
49
50      % Product Parameters.
51      addParameter(p,'computePayoffs',defaults.computePayoffs,isfunction);
52      addParameter(p,'isInTheMoney',defaults.isInTheMoney,isfunction);
53      addParameter(p,'isExercisable',defaults.isExercisable,@(x) isfunction(x) || ...
            isnumeric(x));
54      addParameter(p,'K',defaults.K,@isnumeric);
55      addParameter(p,'dividends',defaults.dividends,@isnumeric);
56      addParameter(p,'getMemory',defaults.getMemory,isfunction);
57
58      % Simulation Parameters.
59      addParameter(p,'seed',defaults.seed);
60      addParameter(p,'generateNormalSamples',defaults.generateNormalSamples,isfunction);
61      addParameter(p,'computeBasisVector',defaults.computeBasisVector,isfunction);
62      addParameter(p,'computeExpectedPayoffs',defaults.computeExpectedPayoffs,isfunction);
63      addParameter(p,'computeWhoExercises',defaults.computeWhoExercises,isfunction);
64      addParameter(p,'smoothingBandwidth',defaults.smoothingBandwidth,@isnumeric);
65      addParameter(p,'smoothingKernel',defaults.smoothingKernel,isfunction);
66      addParameter(p,'basisDegree',defaults.basisDegree, isnonnegativeInteger);
67      addParameter(p,'numberOfSamples',defaults.numberOfSamples, ispositiveInteger);
68      addParameter(p,'numberOfTimesteps',defaults.numberOfTimesteps, ispositiveInteger);
69
70      % Path Generation Parameters.
71      addParameter(p,'T',defaults.T,ispositiveNumber);
72      addParameter(p,'dt',defaults.dt,ispositiveNumber);
73      addParameter(p,'S0',defaults.S0,@isnumeric);
74      addParameter(p,'r',defaults.r,@isnumeric);
```

```
75      addParameter(p,'sigma',defaults.sigma,@isnumeric);
76      addParameter(p,'phi',defaults.phi,@isnumeric);
77      addParameter(p,'lambda',defaults.lambda,@isnumeric);
78      addParameter(p,'kappa',defaults.kappa,@isnumeric);
79      addParameter(p,'theta',defaults.theta,@isnumeric);
80      addParameter(p,'eta',defaults.eta,@isnumeric);
81      addParameter(p,'gamma',defaults.gamma,@isnumeric);
82      addParameter(p,'generatePaths',defaults.generatePaths,isfunction);
83      addParameter(p,'correlationMatrix',defaults.correlationMatrix,iscorrelationMatrix);
84
85      % Parse input.
86      parse(p,varargin{:});
87      options = p.Results;
88
89      % ----------------------- Check Dimensions ------------------------------ %
90      [S0, sigma, correlationMatrix, dividends, lambda, phi] = ...
91          deal(options.S0, options.sigma, options.correlationMatrix, ...
                options.dividends, options.lambda, options.phi);
92      D = numel(S0);
93      assert(all(size(S0) == [1,D]), 'S0 must have size [1,D]');
94      assert(all(size(sigma) == [1,D]), 'sigma must have size [1,D]');
95      assert(all(size(dividends) == [1,D]), 'dividends must have size [1,D]');
96      assert(all(size(correlationMatrix) == [D,D]), 'correlationMatrix must have ...
            size [D,D]');
97      if any(lambda ~= 0 && phi ~= 0)
98          assert(all(size(lambda) == [1,D]), 'lambda must have size [1,D]');
99          assert(all(size(phi) == [1,D]), 'phi must have size [1,D]');
100     end
101     % ----------------------- Make Consistent ------------------------------- %
102
103     if numel(options.r) ~= (options.numberOfTimesteps - 1)
104         options.r = repmat(options.r(1),[1 options.numberOfTimesteps-1]);
105     end
106
107     % Extract relevant options to avoid retyping 'options.'
```

```
108     [T, dt, N] = deal(options.T, options.dt, options.numberOfTimesteps);
109     [Td, dtd] = deal(defaults.T,defaults.dt);
110 
111     % If parameters are not consistent
112     if (N * dt ~= T)
113         % If T was set differently from the default.
114         if (T ~= Td)
115             % If dt was set.
116             if (dt ~= dtd)
117                 N = T/dt;
118             else
119                 dt = T/N;
120             end
121         % If dt was set.
122         elseif (dt ~= dtd)
123             T = N*dt;
124         else
125             dt = T/N;
126         end
127     end
128 
129     [options.T, options.dt, options.numberOfTimesteps] = deal(T,dt,N);
130 end
131 
132 function flag = ispositiveDefinite(x)
133     [~,p] = chol(x);
134     flag = p == 0;
135 end
```

D.2 Asset Pricing with Monte Carlo

Listing 3: MCAssetPricing.m

```
1 function [value, stderr, discountedCashFlows, exerciseTimes, S, options] = ...
```

```
MCAssetPricing(varargin)

% -------------------------- Add  Library ------------------------------ %
addpath('../code_basis','../code_paths','../code_regressor','../code_strategy');

% -------------------------- Handle Input ------------------------------ %

% Parse input arguments.
options = MCAssetPricingOptions(varargin{:});

% Set seed.
rng(options.seed);

% Simulation parameters.
[dt, r, numberOfTimesteps, numberOfSamples] = deal(options.dt, options.r, ...
    options.numberOfTimesteps, options.numberOfSamples);

% -------------------------- Simulation -------------------------------- %

% Generate sample paths. Default: generateBlackScholesPaths
S = options.generatePaths(options);

% Compute payoff at maturity. Default: @(S,t,options) (max(options.K-S(:,t),0))
discountedCashFlows = zeros(numberOfSamples,numberOfTimesteps);
discountedCashFlows(:,end) = options.computePayoffs(S,numberOfTimesteps,options);

% Prepare container for exercise times.
exerciseTimes = numberOfTimesteps * ones(numberOfSamples,1);

% Work backwards from maturity.
for t=(numberOfTimesteps-1):-1:1

    % Discount cash flows.
    discountedCashFlows(:,t) = exp(-r(t)*dt) * discountedCashFlows(:,t+1);

```

```
35          % Can the option be exercised in this timestep? Default: @(t,options) true
36          if (options.isExercisable(t,options))
37
38              % Compute payoffs for exercising in this time step. Default: ...
                    @(S,t,options) (max(options.K-S(:,t),0))
39              payoffs = options.computePayoffs(S,t,options);
40
41              % Find samples where exercising is worthwhile.
42              whoExercises = options.computeWhoExercises(S, t, ...
                    discountedCashFlows(:,t), payoffs, options);
43
44              % Exercise if worthwhile.
45              exerciseTimes(whoExercises) = t;
46              discountedCashFlows(whoExercises,t) = payoffs(whoExercises);
47          end
48      end
49
50      % ---------------------------- Statistics ------------------------------- %
51
52      value = mean(discountedCashFlows(:,1));
53      stderr = std(discountedCashFlows(:,1)   )/sqrt(numberOfSamples);
54  end
```

Listing 4: MCAssetPricingOptions.m

```
1   function options = MCAssetPricingOptions(varargin)
2
3       % -------------------------- Default Options ---------------------------- %
4
5       rng shuffle % To ensure that defaults.seed is chosen differently everytime.
6       defaults = struct( ...
7           'seed',                     randi(intmax), ...
8           'numberOfSamples',          1000, ...
9           'computeBasisVector',       @(St, options) cumprod([ones(size(St)) ...
```

```
            repmat(St,[1 options.basisDegree])],2), ... % also try laguerre
10          'basisDegree',              3, ...
11          'dt',                       1/50, ...
12          'T',                        1, ...
13          'numberOfTimesteps',        50, ...
14          'generatePaths',            @generateJumpDiffusionPaths, ...
15          'S0',                       80, ...
16          'K',                        100, ...
17          'r',                        0.01, ...
18          'dividends',                0, ...
19          'sigma',                    0.25, ...
20          'correlationMatrix',        1, ...
21          'generateNormalSamples',    @randn, ... % Try out sobol sequences
22          'phi',                      -0.1, ...
23          'lambda',                   0, ...
24          'computePayoffs',           @(S,t,options) (max(options.K-S(:,t,1),0)), ...
25          'isInTheMoney',             @(S,t,options) ...
              (options.computePayoffs(S,t,options) > 0), ...
26          'isExercisable',            @(t,options) true, ...
27          'computeExpectedPayoffs',   @LSMExpectedPayoffs, ...
28          'computeWhoExercises',      @whoExercisesExpectedPayoff, ...
29          'getMemory',                @(t) t, ...
30          'smoothingBandwidth',       3, ...
31          'smoothingKernel',          @(X0,X,options) ...
              exp(-sum((repmat(X0,[size(X,1),1]) - ...
              X).^2./(2*options.smoothingBandwidth^2),2)), ...
32          'kappa',                    2, ...
33          'theta',                    0.25^2, ...
34          'eta',                      0.02, ...
35          'gamma',                    1.2 ...
36      );
37
38      % --------------------------- Setup Parser ------------------------------ %
39
40      % Custom conditions.
```

```
41      isfunction = @(x) isa(x,'function_handle');
42      ispositiveInteger = @(x) isnumeric(x) && (x > 0) && (x == floor(x));
43      isnonnegativeInteger = @(x) isnumeric(x) && (x >= 0) && (x == floor(x));
44      ispositiveNumber = @(x) isnumeric(x) && (x > 0);
45      iscorrelationMatrix = @(x) isnumeric(x) && all(diag(x) == 1) && ...
            ispositiveDefinite(x) && issymmetric(x);
46
47      % Initialize parser.
48      p = inputParser();
49
50      % Product Parameters.
51      addParameter(p,'computePayoffs',defaults.computePayoffs,isfunction);
52      addParameter(p,'isInTheMoney',defaults.isInTheMoney,isfunction);
53      addParameter(p,'isExercisable',defaults.isExercisable,@(x) isfunction(x) || ...
            isnumeric(x));
54      addParameter(p,'K',defaults.K,@isnumeric);
55      addParameter(p,'dividends',defaults.dividends,@isnumeric);
56      addParameter(p,'getMemory',defaults.getMemory,isfunction);
57
58      % Simulation Parameters.
59      addParameter(p,'seed',defaults.seed);
60      addParameter(p,'generateNormalSamples',defaults.generateNormalSamples,isfunction);
61      addParameter(p,'computeBasisVector',defaults.computeBasisVector,isfunction);
62      addParameter(p,'computeExpectedPayoffs',defaults.computeExpectedPayoffs,isfunction);
63      addParameter(p,'computeWhoExercises',defaults.computeWhoExercises,isfunction);
64      addParameter(p,'smoothingBandwidth',defaults.smoothingBandwidth,@isnumeric);
65      addParameter(p,'smoothingKernel',defaults.smoothingKernel,isfunction);
66      addParameter(p,'basisDegree',defaults.basisDegree, isnonnegativeInteger);
67      addParameter(p,'numberOfSamples',defaults.numberOfSamples, ispositiveInteger);
68      addParameter(p,'numberOfTimesteps',defaults.numberOfTimesteps, ispositiveInteger);
69
70      % Path Generation Parameters.
71      addParameter(p,'T',defaults.T,ispositiveNumber);
72      addParameter(p,'dt',defaults.dt,ispositiveNumber);
73      addParameter(p,'S0',defaults.S0,@isnumeric);
```

```
74      addParameter(p,'r',defaults.r,@isnumeric);
75      addParameter(p,'sigma',defaults.sigma,@isnumeric);
76      addParameter(p,'phi',defaults.phi,@isnumeric);
77      addParameter(p,'lambda',defaults.lambda,@isnumeric);
78      addParameter(p,'kappa',defaults.kappa,@isnumeric);
79      addParameter(p,'theta',defaults.theta,@isnumeric);
80      addParameter(p,'eta',defaults.eta,@isnumeric);
81      addParameter(p,'gamma',defaults.gamma,@isnumeric);
82      addParameter(p,'generatePaths',defaults.generatePaths,isfunction);
83      addParameter(p,'correlationMatrix',defaults.correlationMatrix,iscorrelationMatrix);
84
85      % Parse input.
86      parse(p,varargin{:});
87      options = p.Results;
88
89      % ---------------------- Check Dimensions ------------------------------ %
90      [S0, sigma, correlationMatrix, dividends, lambda, phi] = ...
91          deal(options.S0, options.sigma, options.correlationMatrix, ...
                options.dividends, options.lambda, options.phi);
92      D = numel(S0);
93      assert(all(size(S0) == [1,D]), 'S0 must have size [1,D]');
94      assert(all(size(sigma) == [1,D]), 'sigma must have size [1,D]');
95      assert(all(size(dividends) == [1,D]), 'dividends must have size [1,D]');
96      assert(all(size(correlationMatrix) == [D,D]), 'correlationMatrix must have ...
            size [D,D]');
97      if any(lambda ~= 0 && phi ~= 0)
98          assert(all(size(lambda) == [1,D]), 'lambda must have size [1,D]');
99          assert(all(size(phi) == [1,D]), 'phi must have size [1,D]');
100     end
101     % ---------------------- Make Consistent ------------------------------- %
102
103     if numel(options.r) ~= (options.numberOfTimesteps - 1)
104         options.r = repmat(options.r(1),[1 options.numberOfTimesteps-1]);
105     end
106
```

```
107     % Extract relevant options to avoid retyping 'options.'
108     [T, dt, N] = deal(options.T, options.dt, options.numberOfTimesteps);
109     [Td, dtd] = deal(defaults.T,defaults.dt);
110 
111     % If parameters are not consistent
112     if (N * dt ~= T)
113         % If T was set differently from the default.
114         if (T ~= Td)
115             % If dt was set.
116             if (dt ~= dtd)
117                 N = T/dt;
118             else
119                 dt = T/N;
120             end
121         % If dt was set.
122         elseif (dt ~= dtd)
123             T = N*dt;
124         else
125             dt = T/N;
126         end
127     end
128 
129     [options.T, options.dt, options.numberOfTimesteps] = deal(T,dt,N);
130 end
131 
132 function flag = ispositiveDefinite(x)
133     [~,p] = chol(x);
134     flag = p == 0;
135 end
```

D.3 Basis Vector

Listing 5: weightedLaguerrePolynomial.m

```
%% weightedLaguerrePolynomial
% Evaluates weighted Laguerre basis in points St (in one dimension)
%
%   Input arguments:
%       X: Evaluation points
%       options: MCAssetPricing options object.
%   Output arguments:
%       A: [N,basisDegree] matrix representing evaluation of basis function for ...
    each of the N samples
function A = weightedLaguerrePolynomial(XX,options)
    D = options.basisDegree;
    X = XX./options.S0;

    A = zeros(numel(X),D + 1);
    A(:,1) = 1;
    A(:,2) = 1 - X;
    for k = 2:D
        A(:,1+k) = ((2*k - 1 - X).*A(:,k) - (k-1).*A(:,k-1))/(k);
    end
    A = repmat(exp(-X./2),[1 D+1]).*A;
end
```

Listing 6: monomials2D.m

```
%% monomials2D
% Evaluates 2D monomial basis in points St
%
%   Input arguments:
%       St: Evaluation points
```

```
6 %       options: MCAssetPricing options object.
7 %   Output arguments:
8 %       S: [N,nchoosek(basisDegree + D,D)] matrix representing evaluation of basis ...
      function for each of the N samples
9 function result = monomials2D(St, options)
10      [N,D] = size(St);
11      assert(D == 2);
12      x = St(:,1);
13      y = St(:,2);
14
15      basisDegree = options.basisDegree;
16      offset = 0;
17      result = zeros(N,nchoosek(basisDegree + D,D));
18      for order = 0:basisDegree
19          numberOfMonomials = order + 1;
20          xx = fliplr(cumprod([ones(size(x)) repmat(x,[1 order])],2));
21          yy = cumprod([ones(size(y)) repmat(y,[1 order])],2);
22          result(:,offset + (1:numberOfMonomials)) = xx.*yy;
23          offset = offset + numberOfMonomials;
24      end
25 end
```

D.4 Path Generation

Listing 7: generateJumpDiffusionPaths.m

```
%% generateJumpDiffusionPaths
% Generates sample paths of a D-dimensional geometric brownian motion with jumps.
%
%   Input arguments:
%       options: MCAssetPricing options object.
%       volatility: [1,M] matrix to be used in place of sigma (for stochastic ...
    volatility)
%   Output arguments:
%       S: [N,M,D] matrix representing N samples of M timesteps of a D
%           dimensional geometric brownian motion.
function S = generateJumpDiffusionPaths(options, volatility)

    % Dimensions.
    [N, M] = deal(options.numberOfSamples, options.numberOfTimesteps);

    % Black-Scholes parameters.
    [r, dividends, sigma, S0, Corr] = deal(options.r, options.dividends, ...
        options.sigma, options.S0, options.correlationMatrix);

    % Jump process parameters.
    [phi, lambda] = deal(options.phi, options.lambda);

    % Generator parameters.
    dt = options.dt;

    % Initialize matrix.
    D = numel(S0);
    S = zeros(N,M,D);
    S(:,1,:) = repmat(S0,[N,1]);

```

```
29      if (nargin == 1)
30          volatility = zeros(N,M,D);
31          for d = 1:D
32              volatility(:,:,d) = sigma(d);
33          end
34      end
35
36      % Cholesky decomposition.
37      Rho = chol(Corr);
38
39      % Propagate.
40      for t = 2:M
41          sigma = squeeze(volatility(:,t,:));
42
43          % Uncorrelated Normal samples. Default: randn
44          B = options.generateNormalSamples(N,D);
45
46          % Correlate Normal samples.
47          W = B * Rho;
48
49          % Scale Normal samples.
50          W = W .* sigma;
51
52          % Propagation step.
53          S(:,t,:) = squeeze(S(:,t-1,:)) .* exp(repmat(r(t-1)-dividends-.5, [N ...
                1]).*(sigma.^2)*dt + sqrt(dt).*W);
54
55          % Jump step.
56          if any(lambda ~= 0 && phi ~= 0)
57              S(:,t,:) = squeeze(S(:,t,:)) .* exp(log(1 + ...
                    phi).*poissrnd(repmat(lambda.*dt,[N,1])));
58          end
59      end
60  end
```

Listing 8: generateHestonPaths.m

```
%% generateHestonPaths
% Generates sample paths of a 1-dimensional Heston path (stochastic volatility).
%   Input arguments:
%       options: MCAssetPricing options object.
%   Output arguments:
%       S: [N,M] matrix representing N samples of M timesteps of a 1
%           dimensional geometric brownian motion.
function S = generateHestonPaths(options)
    % Dimensions.
    [N, M] = deal(options.numberOfSamples, options.numberOfTimesteps);

    % Parameters.
    [kappa,theta, eta, sigma, T] = deal(options.kappa, options.theta, options.eta, ...
        options.sigma, options.T);

    a = @(t,v) kappa*(theta - v);
    b = @(t,v) (eta * sqrt(v));
    dt = (T/M)/240; % Simulate in 6 minute intervals
    volatility = sqrt(eulerMethod(a,b,sigma.^2,M,T,N,dt));

    S = generateJumpDiffusionPaths(options, volatility);

end
```

Listing 9: generateCEVPaths.m

```
%% generateCEVPaths
% Generates sample paths of a 1-dimensional of the solution to dSt = St (rdt + ...
    sigma S^(1-gamma)dWt).
%
%   Input arguments:
%       options: MCAssetPricing options object.
%   Output arguments:
```

```
%       S: [N,M] matrix representing N samples of M timesteps of a 1
%           dimensional CEV path.
function S = generateCEVPaths(options)
    % Dimensions.
    [N, M] = deal(options.numberOfSamples, options.numberOfTimesteps);

    % Parameters.
    [r, dividends, sigma, S0, T, gamma] = deal(options.r, options.dividends, ...
        options.sigma, options.S0, options.T, options.gamma);

    a = @(t,s) (r(1) - dividends)*s;
    b = @(t,s) sigma*s.^gamma;
    dt = (T/M)/240; % Simulate in 6 minute intervals
    S = eulerMethod(a,b,S0,M,T,N,dt);

end
```

Listing 10: eulerMethod.m

```
%% eulerMethod
% Implementation of Euler-Maruyama method for numerical integration of SODE dSt = ...
    a(t,St)dt + b(t,St)dWt
%
%   Input arguments:
%       a: function handle for drift term
%       b: function handle for diffusion term
%       X0: initial value
%       M: number of samples to produce in temporal dimension
%       T: final time
%       N: number of samples to produce
%       dt: timestep
%   Output arguments:
%       X: [N,M] matrix representing N samples of the process S (each M ...
    equidistant samples on [0,T])
```

```
14 function X = eulerMethod(a,b,X0,M,T,N,dt)
15     X = zeros(N,M);
16     X(:,1) = X0;
17     t = 0;
18     for k = 1:(M-1)
19         Xk = X(:,k);
20         for i = 1:((T/M)/dt)
21             dW = sqrt(dt)*randn(N,1);
22             t = t + dt;
23             Xk = Xk + a(t,Xk)*dt + b(t,Xk).*dW;
24         end
25         X(:,k+1) = Xk;
26     end
27 end
```

D.5 Regressors

Listing 11: LSMExpectedPayoffs.m

```
%% LSMExpectedPayoffs
% Implements least squares monte carlo computation of expected payoffs
%
%   Input arguments:
%       St: Sample path at a given timestep.
%       discountedCashFlows: TODO
%       options: MCAssetPricing options object.
%   Output arguments:
%
function expectedPayoffs = LSMExpectedPayoffs(St, discountedCashFlows, options)

    % Computes model matrix. Default: Polynomial
    A = options.computeBasisVector(St,options);

    % Least squares fit.
    coeffs = A\discountedCashFlows;

    % Apply model.
    expectedPayoffs = A*coeffs;
end
```

Listing 12: kernelSmoother.m

```
%% kernelSmoother
% Implements kernel smoothing for non-parametric regression.
%
%   Input arguments:
%       X:  X-Samples
%       Y:  g(X)-Samples
```

```
7  %       options: MCAssetPricing options object.
8  %       X0: Evaluation points of regressed function (default: X)
9  %   Output arguments:
10 %       YEstimates: Estimates of f(X0) based on kernel smoothing.
11 function YEstimates = kernelSmoother(X, Y, options, X0)
12     if nargin < 4
13         X0 = X;
14     end
15
16     % Always reshape input arrays to matrices.
17     S = size(X0);
18     M = S(1);
19     D = prod(S(2:end));
20     X0 = reshape(X0,[M,D]);
21     X = reshape(X,[size(X,1),D]);
22     YEstimates = zeros(M,1);
23
24     % Kernel Selection. Default: Gaussian kernel.
25     K = options.smoothingKernel;
26
27     % For each estimation point.
28     for m = 1:M
29         % Evaluate kernel function for each X(i) and compute weight.
30         W = K(X0(m,:),X,options);
31
32         % Fitting constant function (Kernel Smoother).
33         if (options.basisDegree == 0)
34             YEstimates(m) = (W'*Y)/sum(W);
35
36         % Local regression onto basis.
37         else
38             % Compute weighted model matrix. Default: Polynomial
39             A = diag(sqrt(W)) * options.computeBasisVector(X,options);
40
41             % Fit to model.
```

```
42              c = A\(diag(sqrt(W))*Y);
43
44              % Estimate Y(X0(m,:)).
45              YEstimates(m) = options.computeBasisVector(X0(m,:),options) * c;
46          end
47      end
48  end
```

D.6 Strategy

Listing 13: whoExercisesExpectedPayoffs.m

```
%% whoExercisesExpectedPayoff
% Decides what path would exercise based on the estimation of the future expected ...
    payoff.
function [whoExercises, whoIsInTheMoney, payoffs, expectedPayoffs] = ...
    whoExercisesExpectedPayoff(S, t, discountedCashFlows, payoffs, options)

    % Find paths that are in the money. Default: @(S,t,options) ...
        (options.computePayoffs(S,t,options) > 0)
    whoIsInTheMoney = find(options.isInTheMoney(S,t,options));

    % If nobody is in the money, nobody exercises.
    if (numel(whoIsInTheMoney) == 0)
       whoExercises = [];

    % Estimate expected payoffs conditional on S.
    else
        % Default @(t) t.
        tau = options.getMemory(t);

        % Only consider in-the-money paths of the current timestep since
        % expected payoffs are irrelevant for the other paths which will
        % certainly not exercise.
        St = S(whoIsInTheMoney,tau,:);
        payoffs = payoffs(whoIsInTheMoney);
        discountedCashFlows = discountedCashFlows(whoIsInTheMoney);

        % Are all samples equal?
        if (all(St == repmat(St(1,:,:),[size(St,1),1,1])))
            % One cannot fit basis with only one sample for explanatory
            % variable. Furthermore, conditional expectetation estimate
```

```
28              % degenerates to mean of dependent variable samples in this case.
29              expectedPayoffs = mean(discountedCashFlows);
30          else
31              % Use regression to estimate conditional expectation. Default: ...
                    LSMExpectedPayoffs
32              expectedPayoffs = ...
                    options.computeExpectedPayoffs(St,discountedCashFlows, options);
33          end
34
35          % Find paths for which exercising seems better than continuing.
36          whoExercises = whoIsInTheMoney(payoffs > expectedPayoffs);
37      end
38  end
```