Quantitative Finance and Economics

http://www.aimspress.com/journal/QFE

QFE, 3(4): 661–708
DOI:10.3934/QFE.2019.4.661
Received: 02 September 2019
Accepted: 06 November 2019
Published: 13 November 2019

Research article

Trajectorial asset models with operational assumptions

Sebastian Ferrando[1,*], **Andrew Fleck** [2], **Alfredo Gonzalez**[3] **and Alexey Rubtsov**[1]

[1] Department of Mathematics, Ryerson University, 350 Victoria St., Toronto, Ontario M5B 2K3, Canada

[2] Department of Mathematics and Statistics, York University, Toronto, Ontario, Canada

[3] Departamento de Matemática, Facultad de Ciencias Exactas y Naturales, Universidad Nacional de Mar del Plata, Funes 3350, Mar del Plata 7600, Argentina

* **Correspondence:** Email: ferrando@ryerson.ca; Tel: +1416979500x7415.

1. Appendix 1: Algorithm, Trajectorial Asset Models in Matlab

This Appendix (part of the paper "Trajectorial Asset Models with Operational Assumptions") provides details of a Matlab program implementing trajectorial asset models and pricing functions. As indicated in the main body of the paper, the basic mathematical structure is a set of sequences, here called trajectories, given by a recursive definition. An adjacency matrix encodes the graph structure implied by the recursive definition and an index array stores the content of the trajectory coordinates (nodes). These nodes also contain the required superhedging investment amounts and portfolio values. The pricing algorithm does not depend on any probabilistic assumptions and provides the superhedging and subhedging optimal portfolios as well as the initial investments for contingent claims. Such algorithm has been studied elsewhere but we provide a convenient summary in this appendix. The algorithm evaluates the intersection of a line and an appropriate convex hull and it is here rendered in the context of the specific models. The Matlab code, included in Appendix 2, provides a driver program demonstrating the estimation of the models' parameters, the execution of the main functionality and output based on illustrative data which is also included.

This appendix summarizes the basic functionality of our Matlab software implementation and elaborates on the algorithmic construction of data structures encoding the trajectory sets defined in the paper. These data structures are discussed with some detail in order to illuminate the computational complexity issues which form a main practical problem to overcome when implementing trajectorial

market models. We also provide some details on the implementation of the superhedging algorithm which is introduced mathematically in Degano et al. (2018) and described as an algorithm in Section 1.3. One could also introduce an alternative implementation based on linear programming; we have not pursued this possibility as our goal is to establish a practical implementation of trajectorial asset models that can be used to gauge the financial merits of the proposed models. From this perspective, studying alternative implementations for the pricing algorithm will represent an unnecessary technical detour.

The package with the software distribution, introduced in Appendix 2, also contains a file (`outputOperationalModelsMatlab.pdf`) with displays and values of relevant parameters. That file includes most of the output that we have obtained from the software. Output presented in the paper can be obtained by executing the driver program (`Driver.m`) with a different data set and altered values of the parameters δ, δ_0, T and Δ. How to achieve this is explained in detail below.

1.1. Trajectory Spaces, Parameters and Estimation

Recall that trajectory sets are denoted by $\mathcal{S}$, $\mathbf{S} \in \mathcal{S}$ are of type I or type II. In the former case $\mathbf{S} = \{\mathbf{S}_i \equiv (S_i, T_i)\}_{0\leq i\leq N(\mathbf{S})}$, in the latter case $\mathbf{S} = \{\mathbf{S}_i \equiv (S_i, T_i, W_i)\}_{0\leq i\leq N(\mathbf{S})}$. $S_i \in \delta_0\mathbb{N}$ where $\delta_0 \geq 0.01$ is a positive parameter that provides an approximating grid to the market chart values $x(t) \in \hat{\delta}_0\,\mathbb{N}$ where $\hat{\delta}_0 \equiv 0.01$ is the numeric value of the smallest monetary unit under consideration (i.e. we assume, as is typical in practice, that the risky asset quotes are in multiples of cents). $T_i \in [0, T] \cap \Delta\mathbb{N}$ for $\Delta > 0$, the smallest time resolution associated to the investor. In the type II models $W_i \in \delta_0\mathbb{N}$ and $W_{i+1} - W_i$ is the consumed trajectory variation in a time interval of length $(T_{i+1} - T_i)$. For simplicity, we concentrate mostly in type II trajectory sets as the type I case is covered by neglecting the third coordinate W_i. Recall, that the *conditional* trajectory sets are constructed recursively as follows: given a node (S_i, T_i, W_i) we provide sets $\mathcal{N}_A(S_i, T_i, W_i) = \{(S_{i+1} = s_0 + k_{i+1}\delta_0, t_0 + T_{i+1} = t_0 + n_{i+1}\Delta, , W_{i+1} = j_{i+1}\delta_0) \;:\; \text{with } (k_{i+1}, n_{i+1}, j_{i+1}) \textit{ admissible}\}$. What constitutes an admissible triple $(k_{i+1}, n_{i+1}, j_{i+1})$ has been described algorithmically in the paper. Despite such mathematical description, some of the computations required to evaluate admissible integers j_{i+1} are not straightforward to implement in software. In these cases, we provide additional computational details in Section 1.10.

It follows from the brief description above that nodes (S_i, T_i, W_i) and $(S'_{i'}, T'_{i'}, W'_{i'})$ are adjacent, as nodes in a graph, if $(S'_{i'}, T'_{i'}, W'_{i'}) \in \mathcal{N}_A(S_i, T_i, W_i)$ (we are taking $i < i'$ without loss of generality); in particular, $\mathbf{S}' \in \mathcal{S}_{(\mathbf{S},i)}$. A desirable property of the model is recombination of trajectories; as it will follow from our explanations below, our implementation takes advantage of any recombination present in the trajectory set. Notice that recombination is not imposed in an ad hoc manner but results from the operational setup (see discussion in Section 1.5).

All models rely on the parameters τ, Γ, the set $\mathcal{N}_E$ and the collection of sets $\{V_\rho\}_{0\leq\rho\leq T}$ (this later collection is only needed for type II models). We refer to this family of parameters and sets generically as *input parameters* and need to be estimated. Specifications and estimations of the input parameters are detailed in the paper. The remaining parameters $\delta, \delta_0, \Delta, T$, are of a different type. The parameters δ and Δ are investor dependent and they should be calibrated, guidelines and a general point of view to set values for these parameters were introduced in the paper. In particular, the parameter δ_0 is a parameter of practical concern fully and its role has been discussed in detail. Finally T depends on the intended financial application.

The collection of sets $\{V_\rho\}_{0\leq\rho\leq T}$ is used to enforce $W_i \in V_{T_i}$ and, for all practical purposes, V_{T_i} should

be considered as a union of intervals (examples are provided in the paper, in those displays each estimated V_ρ could actually be taken to be a single interval). Estimation in this trajectorial setting is *worst case* estimation, this is a simple minded methodology that requires that informative variables have been chosen to define trajectories. *Informative* here means that the parameters that confine the model variables are contained in worst case ranges that reflect observable and possible financial situations that impact the values of the risky asset and the class of investors under consideration.

The Estimation of the aforementioned parameters is carried out in `Estimation.m` located in the source directory (see Appendix 2).

Elements $(m, q) \in \mathcal{N}_E$ give the possible, historically estimated, changes $\Delta_i S = S_{i+1} - S_i = m\delta_0$, $\Delta_i T = T_{i+1} - T_i = q\Delta$. Different usages of the input parameters leads to the construction of four models: `MIOBS`, `MIDP`, `MIIOBS` and `MIIDP`. The class denominations `DP` and `OBS` refer to how the sets $\mathcal{N}_E$ are set to different possibilities: `direct product` in one case and `observable` sets in the other case. The relevant code for each model is kept in the source directory (see Appendix 2) under `Model_1_Observable.m`, `Model_1_Direct_Product.m`, `Model_2_Observable.m` and `Model_2_Direct_Product.m` respectively.

1.2. Dynamic Minmax Bounds

In our finite case we do require $N_H(\mathbf{S}) \leq N(\mathbf{S}) \leq M_T$, where, we recall, M_T is the number of data points in a time window $[t, t+T]$. A direct evaluation of the conditional bounds is a daunting task. Additionally, given the formulation of the problem it is not clear how to construct the hedging values $H_i(\mathbf{S})$, for a given payoff Z, by means of the unfolding path values $\mathbf{S}_0, \mathbf{S}_1, \mathbf{S}_2, \ldots$. Consider next another pair of numbers namely $\underline{U}_0(S_0, Z, \mathcal{M})$ and $\overline{U}_0(S_0, Z, \mathcal{M})$; these new bounds are defined in a dynamic, or iterative, manner each instance involving a local optimization akin to solving a local sub problem. We discuss here part of the argument for why these dynamic minmax bounds are equal to our globally defined conditional ones.

Consider an n-bounded, discrete market $\mathcal{M}$; for a given function Z defined on $\mathcal{S}$, any $\mathbf{S} \in \mathcal{S}$, and $0 \leq i \leq n$ set

$$\overline{U}_i(\mathbf{S}, Z, \mathcal{M}) = \begin{cases} \inf\limits_{H \in \mathcal{H}} \sup\limits_{\mathbf{S}' \in \mathcal{S}_{(\mathbf{S},i)}} [\overline{U}_{i+1}(\mathbf{S}', Z, \mathcal{M}) - H_i(\mathbf{S})\Delta_i S'] & \text{if} \quad 0 \leq T_i < M_T\, \Delta \\ Z(\mathbf{S}) & \text{if} \quad T_i = M_T\, \Delta. \end{cases} \tag{1}$$

Also define $\underline{U}_i(\mathbf{S}, Z, \mathcal{M}) = -\overline{U}_i(\mathbf{S}, -Z, \mathcal{M})$.

Denote by $I^k_{\mathbf{S}}$ the set of ranges of portfolio values at node $\mathbf{S}_k$, in other words

$$I^k_{\mathbf{S}} \equiv \{H_k(\mathbf{S}) : H \in \mathcal{H}\} \subseteq \mathbb{R}.$$

Thus we can rewrite the expression in (1) for $0 \leq k < M(\mathbf{S})$,

$$\overline{U}_k(\mathbf{S}, Z, \mathcal{M}) = \inf_{u \in I^k_{\mathbf{S}}} \sup_{\mathbf{S}' \in \mathcal{S}_{(\mathbf{S},k)}} [\overline{U}_{k+1}(\mathbf{S}', Z, \mathcal{M}) - u\, \Delta_k S']. \tag{2}$$

Theorem 1.1. *For any function Z defined on a discrete n-bounded market $\mathcal{M} = \mathcal{S} \times \mathcal{H}$, the following inequality holds:*

$$\overline{U}_0(S_0, Z, \mathcal{M}) \leq \overline{V}(S_0, Z, \mathcal{M}),$$

and hence $\underline{U}_0(S_0, Z, \mathcal{M}) \geq \underline{V}(S_0, Z, \mathcal{M})$ is also valid.

The reverse inequality, establishing $\overline{U}_0(S_0, Z, \mathcal{M}) = \overline{V}(S_0, Z, \mathcal{M})$ and $\underline{U}_0(S_0, Z, \mathcal{M}) = \underline{V}(S_0, Z, \mathcal{M})$, requires some machinery and minimal assumptions, details and proofs can be found in Degano et al. (2018).

The option bounds $\underline{V}(\mathbf{S}_0, Z, \mathcal{M})$ and $\overline{V}(\mathbf{S}_0, Z, \mathcal{M})$ can be computed by an algorithm evaluating $\underline{U}(\mathbf{S}_0, Z, \mathcal{M})$ and $\overline{U}(\mathbf{S}_0, Z, \mathcal{M})$ which we briefly detail below. As indicated, the fact that the proposed algorithm delivers the global minmax bounds is established in Degano et al. (2018).

1.3. Convex Hull Algorithm

This section outlines how to calculate the dynamic bounds $\overline{U}_i(\mathbf{S}, Z, \mathcal{M})$.

Again considering a n-bounded discrete market $\mathcal{M} = \mathcal{S} \times \mathcal{H}$. For $\mathbf{S} \in \mathcal{S}$, and $0 < i < M(\mathbf{S})$ we are going to expand on a method in order to resolve (2). For an arbitrary, but momentarily fixed, $\mathbf{S}' \in \mathcal{S}_{(S,i)}$, set

$$\ell(x) = \overline{U}_{i+1}(\mathbf{S}', Z, \mathcal{M}) - u_i(S'_{i+1} - x),$$

i.e. the line in the plane, through the point $(S'_{i+1}, \overline{U}_{i+1}(\mathbf{S}', Z, \mathcal{M}))$ with slope u_i. Thus,

$$\overline{U}_{i+1}(\mathbf{S}', Z, \mathcal{M}) - u_i(S'_{i+1} - S_i)$$

is the intersection of ℓ with the vertical straight line $x = S_i$. Therefore, for each fixed $u_i \in I^i_{\mathbf{S}}$:

$$\sup_{\mathbf{S}' \in \mathcal{S}_{(\mathbf{S},i)}} \left\{ \overline{U}_{i+1}(\mathbf{S}', Z, \mathcal{M}) - u_i(S'_{i+1} - S_i) \right\}$$

is the largest of the ordinates of the points of intersection between the straight lines ℓ and $x = S_i$. Then $\overline{U}_i(\mathbf{S}, Z, \mathcal{M})$ becomes the lowest value of these largest intersections.

To complete the geometric procedure, assume both of the following sets are non-empty,

$$\mathcal{S}^{\text{do}}_{(\mathbf{S},i)} = \{\mathbf{S}' \in \mathcal{S}_{(\mathbf{S},i)} : S'_{i+1} \leq S_i\}, \text{ and } \mathcal{S}^{\text{up}}_{(\mathbf{S},i)} = \{\mathbf{S}' \in \mathcal{S}_{(\mathbf{S},i)} : S'_{i+1} > S_i\}.$$

For $\mathbf{S}^{\text{up}} \in \mathcal{S}^{\text{up}}_{(\mathbf{S},i)}$ and $\mathbf{S}^{\text{do}} \in \mathcal{S}^{\text{do}}_{(\mathbf{S},i)}$ denote by $u_{(\mathbf{S}^{\text{up}}, \mathbf{S}^{\text{do}})}$ the slope of the straight line in the plane through the points $(S^{\text{up}}_{i+1}, \overline{U}_{i+1}(\mathbf{S}^{\text{up}}, Z, \mathcal{M}))$ and $(S^{\text{do}}_{i+1}, \overline{U}_{i+1}(\mathbf{S}^{\text{do}}, Z, \mathcal{M}))$:

$$u_{(\mathbf{S}^{\text{up}}, \mathbf{S}^{\text{do}})} = \frac{\overline{U}_{i+1}(\mathbf{S}^{\text{up}}, Z, \mathcal{M}) - \overline{U}_{i+1}(\mathbf{S}^{\text{do}}, Z, \mathcal{M})}{S^{\text{up}}_{i+1} - S^{\text{do}}_{i+1}}.$$

It can be shown (Degano et al. (2018)) that

$$L_i(\mathbf{S}, Z, \mathcal{M}) \equiv \sup_{\mathbf{S}^{\text{up}} \in \mathcal{S}^{\text{up}}_{(\mathbf{S},i)} \mathbf{S}^{\text{do}} \in \mathcal{S}^{\text{do}}_{(\mathbf{S},i)}} [\overline{U}_{i+1}(\mathbf{S}^{\text{up}}, Z, \mathcal{M}) - u_{(\mathbf{S}^{\text{up}}, \mathbf{S}^{\text{do}})} \Delta_i S^{\text{up}}] =$$

$$\overline{U}_i(\mathbf{S}, Z, \mathcal{M}).$$

Therefore, $\overline{U}_i(\mathbf{S}, Z, \mathcal{M})$, is the largest intersection of the referred lines with the line $x = S_i$.

The next result requires extra assumptions and presents a way to solve the optimization problem for the case $I^i_{\mathbf{S}} = \mathbb{R}$. The assumption $I^i_{\mathbf{S}} = \mathbb{R}$ is a convenient way of guaranteeing $u_{(\mathbf{S}^{\text{up}}, \mathbf{S}^{\text{do}})} \in I^i_{\mathbf{S}}$.

Let $\mathcal{M} = \mathcal{S} \times \mathcal{H}$ a n-bounded discrete market. If for any $\mathbf{S} \in \mathcal{S}$, $I^i_{\mathbf{S}} = \mathbb{R}$, and either one the two following conditions for $\mathbf{S} \in \mathcal{S}$ below hold,

1. $L_i(\mathbf{S}, Z, \mathcal{M}) = \overline{U}_{i+1}(\mathbf{S}^\bullet, Z, \mathcal{M}) - u_{(\mathbf{S}^\bullet, \mathbf{S}^\circ)}\Delta_i S^\bullet$ for some $\mathbf{S}^\bullet \in \mathcal{S}^{\text{up}}_{(\mathbf{S},i)}$ and $\mathbf{S}^\circ \in \mathcal{S}^{\text{do}}_{(\mathbf{S},i)}$.
2. For any $\mathbf{S}' \in \mathcal{S}_{(\mathbf{S},i)}$, $0 < a \leq |S'_{i+1} - S_i| \leq b$ (a and b may depend on $\mathbf{S}$).

Then,

$$\overline{U}_i(\mathbf{S}, Z, \mathcal{M}) = L_i(\mathbf{S}, Z, \mathcal{M}).$$

for proof details see Degano et al. (2018).

1.4. Software Functionality

Implementing the algorithmic constructions from the paper is a three step process illustrated by `Driver.m`. Before starting, financial stock chart data and some user inputs are required. The software then first estimates the necessary parameters used to construct trajectory sets. Next a directed acyclic graph structure is constructed. Intuitively this can be thought of as a recombining tree not unlike the binomial tree in the Cox-Rubinstein model albeit adapted to historical data, the type of investor and hence more flexible and complex. This will be the data structure for our trajectory space and will be discussed in detail in Section 1.5. Finally, the algorithm provides this graph with financial meaning by assigning nodes monetary and temporal values corresponding to the possible movements of a stock chart through time. The algorithm can then be thought to "move backwards in time" by evaluating and storing the required superhedging investments required to superhedge payoff values at expiration nodes. The algorithm moves along the edges of the graph until it reaches the root node and terminates with an initial superhedging amount. This is done using a dynamic programming approach, the implementation of this algorithm is covered in detail in this appendix.

What follows is a general overview of each of the steps alluded above and the required associated functions. Examples of the data used, estimation and calibration of parameters as well as output displays can be found in the displays of the paper.

1. Estimation and user calibration:

   (a) The intended application of the software is to obtain super and sub hedging prices for path independent European options with expiration time $T = M_T\Delta$. Where M_T is an integer and Δ the time resolution. These are imputed along with some stock chart data (see `Driver.m`, lines `3-11`).

   (b) Given a user selected resolution on the stock chart data $\delta_0 \geq 0.01$ and $\delta = z\delta_0$ for $z \in \mathbb{Z}^+$ (`Driver.m` lines `14 and 17`), the software estimates the extrema of sets $\{V_\rho\}_{0\leq\rho\leq T}$ and parameters τ and Γ. Additionally the empirical conditional set $\mathcal{N}_E$ used in type II models are obtained. (`Driver.m` line `22` - note: with the exception of the extrema of V_ρ all parameters are stored globally). This is mainly achieved using the `parameter_est` function on line `8` of `Estimation.m`:

   ```
   [Gamma_array,tau_array,N_array,n1, n0, ,V]=
       paramter_est(M_T,data,delta_0,delta)
   ```

   The array `N_array` contains all the rebalancing times, over a lifetime T and a given δ, observed in the data. The extrema of this array will give us an idea of roughly how many

rebalances to expect for a given user δ based on the past behaviour of the stock. This is important when considering transaction costs.

N.B. Before moving to the next stage we note the considerations required to best calibrate the value of the parameter δ (criteria to set δ are described in the paper). There are essentially two competing interests under consideration. The smaller the value of δ, the larger the conditional set. The investor will rebalance more often her portfolio during the life of the option. As remarked repeatedly, the implemented models reflect characteristics of realistic traders hence it is crucial to calibrate the parameter δ to reflect this fact. The selection of δ and δ_0 also impact directly on the computational time to create the trajectory data structure and on the evaluation of the price bounds.

Computational resources dictate a smaller size for $\mathcal{N}_E$. Section 1.8 provides a summary of the computational complexity of the implementation. Suffice to say that the smaller the value of δ and so a larger set $\mathcal{N}_E$, the more resources will be required to construct the graph and price across a variety of strikes. As a general guideline, most output included in the paper was done with $M_T = O(100)$ and δ selected to result in 2-3 rebalances a day. On a 40 core server with 256 GB of RAM, full Type I and II models could take on the order of days. To alleviate this situation we have subsampled $\mathcal{N}_E$ and chosen relatively larger values of δ_0 and δ (see comments in the paper as well as `Model_2_Observable.m` and `Model_2_Direct_Product.m`). These choices do not affect the quantitative and qualitative conclusions one can reach and provide fast execution times. In a 4 CPU's at 2.60GHz and 7.5 GiB of RAM plus 4 GB of Swap memory (in a Linux system) and for the SPY data set (the paper provides details on values of parameters used) we obtained the following execution times.

- Model `MIIOBS` data structure construction 45.63 minutes pricing 36.28 minutes.
- Model `MIIDP` data structure construction 132.72 minutes pricing 183.35 minutes.

2. Constructing the trajectory set:

   (a) As indicated in Section 1.1, $\mathcal{N}_A(S_i, T_i, W_i) = \{(S_{i+1} = s_0 + k_{i+1}\delta_0, t_0 + T_{i+1} = t_0 + n_{i+1}\Delta, W_{i+1} = j_{i+1}\delta_0)$ with $(k_{i+1}, n_{i+1}, j_{i+1})$ *admissible*. While constructing the trajectory set, we will only consider the dependence structure (i.e., additional quantities will not be evaluated at this stage). In other words, only the relationships between the triples (k_i, n_i, j_i) and elements of $\mathcal{N}_A(S_i, T_i, W_i)$ will be taken into account. The financial interpretation (that is the relationship between each (k_i, n_i, j_i) and (S_i, T_i, W_i)) comes after in the third and final step. Such organization of computations is desirable if one plans on using models across a variety of initial stock values and variation.

   (b) The just mentioned dependency structure is realized as a directed graph and it is encoded using an adjacency matrix `A` and a separate array called the index `I`. The matrix and the index are created by means of the following function:

```
[A,I]=creatematrix([0,0],[Kmax,Nmax],'type I model')
```

or

```
[A,I]=creatematrix([0,0,0],[Kmax,Jmax,Nmax],'type II model')
```

`Kmax, Nmax` and `Jmax` represent the maximum possible absolute values of (k_i, n_i, j_i) respectively. This will be discussed in more detail in the following section. As mentioned `Model_1_Observable.m`, `Model_1_Direct_Product.m`, `Model_2_Observable.m` and `Model_2_Direct_Product.m` provide different possible implementations of this function for different models.

(c) The nodes on the graph correspond to the values $(k_{i+1}, n_{i+1}, j_{i+1})$ and the edges correspond to the nested relationships between nodes. That is, the paths in our graph will form the *trajectories*. The relationship between $(k_{i+1}, n_{i+1}, j_{i+1})$ and their unique integer representation corresponding to the list of nodes in the graph is stored using the index `I`.

3. Pricing a payoff

(a) Since δ_0, δ and s_0 are known, the values (S_i, T_i, W_i) can be obtained for any (k_i, n_i, j_i). Recall this is the "financial interpretation" mentioned in 2(a). Therefore, given the adjacency matrix, it is straightforward to access the information required to implement the pricing algorithm.

(b) The pricing algorithm takes as input the adjacency matrix created in the previous step, δ_0, the initial stock value s_0, strike value K and payoff function. This is done using the following function (`Pricing.m` lines `6 and 12`) :

$$[\mathtt{V}] = \mathtt{valueindex}(\mathtt{A}, \mathtt{I}, \mathtt{S0}, \mathtt{K}, \mathtt{delta\_0}, '\mathtt{payofffunction}').$$

(c) The pricing algorithm makes use of the graph structure to move from terminal nodes backwards in time from the expiration time of the option $t_0 + T$ to the present time t_0. At each node $\mathbf{S}_i$ a portfolio value quantity $\overline{U}_i(\mathbf{S}, Z, \mathcal{M})$ representing the superhedging value (that is, the minimum amount of cash needed to cover the option obligations by trading with the underlying stock) is computed; see Section 1.6. A dynamic programming argument shows that $\overline{U}_0(\mathbf{S}, Z, \mathcal{M}) = \overline{V}_0(\mathbf{S}, Z, \mathcal{M})$.

1.5. Data Structure for Trajectory Space

Implementing the pricing algorithm that evaluates the conditional bounds requires an efficient representation of the conditional trajectory sets. One can conceivably list a set of trajectories and formulate the minmax optimization as a linear programming problem or some other standard convex optimization problem. But it is not hard to see how this would quickly become unwieldy for large sets of trajectories and would be practically useless in gaining informative information about prices. Also notice that a simple minded 3-dimensional grid encoding of the coordinates (S_i, T_i, W_i) will require unnecessary memory resources and initialization. The obvious solution is to bunch trajectories together on the basis of a common history and store this as some kind of combinatorial object. This is made easy by the fact that our models inherit a property common to many stochastic models in finance. Namely the type of models we develop in this paper have a Markovian property in that $\mathcal{S}_{(\mathbf{S},k)} = \mathcal{S}_{(\mathbf{S}',k)}$ if $\mathbf{S}_k = \mathbf{S}'_k$.

Given the dependency of $\mathcal{S}_{(\mathbf{S},k)}$ on $\mathbf{S}_0, \ldots, \mathbf{S}_k$, *general trajectory sets* could imply that any data structure taking advantage of the trajectories common history would behave like a tree. To leverage further

savings, in addition to being Markovian our models frequently feature recombination i.e. different initial trajectory segments $(\mathbf{S}_1, \ldots, \mathbf{S}_k) \neq (\mathbf{S}'_1, \ldots, \mathbf{S}'_k)$ may share nodes $(\hat{\mathbf{S}}, k+1) = (\tilde{\mathbf{S}}, k+1)$ with $\hat{\mathbf{S}} \in \mathcal{S}_{(\mathbf{S},k)}$ and $\tilde{\mathbf{S}} \in \mathcal{S}_{(\mathbf{S}',k)}$. The upshot being that the algorithm's space and time requirements are polynomial in the time parameter M_T (as opposed to exponential in a tree-like structure). The polynomial property allows us to execute our program for M_T of the order of hundreds (without recourse to parallelization).

Most importantly the Markovian and recombining properties guarantee that the pricing problem has optimal substructure and thus a dynamic programming approach is possible. Constructing our trajectory sets such that they satisfy these properties (Markov, recombining and therefore optimal substructure) is natural as the graph data structure allows to capture any recombination present in the conditional trajectory sets. Algorithm 1, see below, details the construction of the graph data structure through the construction of an adjacency matrix encoding the trajectory set. The algorithm can be summarized as follows:

1. Seed the algorithm with the first node.
2. Apply a model-specific function to find the connected nodes.
3. Store the new children nodes.
4. Repeat with children.

While conceptually simple, the construction is somewhat awkward. The reason being that, to reiterate, we are considering the dependence structure between the triples (k_i, n_i, j_i). Additionally, we need a way of moving from the graph where the nodes are triples to an adjacency matrix where nodes are identified by integers listed along the rows and columns. Towards that end we introduce the concept of index. In the `creatematrix.m` file there is the function `index` that creates an array of triples whose linear array indexing corresponds to the integer value of their node in the adjacency matrix.

The function `index` constructs a $1 \times 3 \times N$ array; where each triplet in the second entry of the array corresponds to the integers k, j and n mentioned above. The third entry of the array, contains N integers identifying each node by a positive integer. The inputs to `index` are the maximum possible values of k, j and n in 3 dimensional models or simply the maximum possible values of k and n in 2 dimensional models (keeping in mind that since k represents offset stock values relative to S_0, it can take negative values). For example `index([1,1,0])` gives the following array:

```
array(:,:,1) = [-1 0 0]
array(:,:,2) = [ 0 0 0]
array(:,:,3) = [ 1 0 0]
array(:,:,4) = [-1 1 0]
array(:,:,5) = [ 0 1 0]
array(:,:,6) = [ 1 1 0]
```

An entry of `index([Kmax,Jmax,Tmax,0])` creates and extra column for recording more information. That is, `index([1,1,0,0])` quadruples sharing the first three entries as the output above and an additional entry set to zero.

```
array(:,:,1) = [-1 0 0 0]
array(:,:,2) = [ 0 0 0 0]
array(:,:,3) = [ 1 0 0 0]
```

```
array(:,:,4) = [-1 1 0 0]
array(:,:,5) = [ 0 1 0 0]
array(:,:,6) = [ 1 1 0 0]
```

In the construction of the adjacency matrix, specific models are made concrete as follows: any model of how the stock chart evolves through time is encoded as a function (see functions `M1_OBS`, `M1_DP` and `M2` in main directory for examples) that takes a triple k, n and j as input and it gives an array of integers corresponding to the indexes of the nodes in $\mathcal{N}_A(s_0 + k_i\delta_0, t_0 + n_i\Delta, j_i\delta_0)$. This is how edges between nodes are established; the said function specifies the sets $\mathcal{N}_A(S_i, T_i, W_i)$ introduced in Section 1.1. In the case of type II models, the function also takes variation bounds inputs (given by the sets $V_{n\Delta}$, as described in the paper) as they are required to specify type II models.

The adjacency matrix construction then proceeds in two stages. First we apply the model function to any previous children or the initial node, this corresponds to the `While` loop over *array*1 in Algoritm 1. The model output is then recorded as new children and record the previous children or initial node as a parent(s). This process continues until there are no more children generated (that is, the n coordinate has reached the value M_T).

ALGORITHM 1: Trajectory Set Construction

Input: Maximum values imposed on the tuples/triples N , Initial Node $N0$ (typically $[k, n] = [0, 0]$ or $[k, j, n] = [0, 0, 0]$, a model function.

Output: An adjacency matrix A and an index I.

*array*1 =index($[N, 0]$);
*array*2 =index($[N, 0]$);
Parent_array = [];
Child_array = [];
*array*1($N0$) = 1;
while *any*(*array*1(:, 4, *node*) == 1) **do**
  for *each array*1(:, 4, *node*) == 1 *in array*1 **do**
    Children=model(*node*);
    *array*2(*Children*) = 1;
    Child_array = [*Child_array*; *Children*];
    Parent_array = [*Parent_array*; *node*·ones(1,length(*Children*))];
  end
  clear *array*1, *array*2
  *array*1 = *array*2
end
A =sparse(*Parent_array*, *Child_array*,ones(1,length(*Children*));
I =index(N);

As a final note, it is important to construct financially meaningful models by enforcing the no-arbitrage property or the 0-neutral property. As discussed in the paper, enforcing one of these properties is particularly relevant for type II models. During the trajectory set construction, if this assumption is not locally satisfied (something that could happen after imposing the constraint $W_i \in V_{T_i}$), we will enforce it by including basic binomial states (see lines 43-46 in `M2.m` included in the software package).

1.6. Pricing Algorithm

Code fragments of the pricing algorithm are displayed below in Algorithm 2. A new element that appears as an input to the algorithm is the payoff function of an option $O(S, K)$. That is $O(S, K) = (S - K)_+$ for calls and $O(S, K) = (K - S)_+$ for puts.

ALGORITHM 2: Pricing (Dynamic Programming)

Input: A,I, S_0 the initial stock level, K the strike(s) of an option contract, δ_0, $O(S, K)$

Output: A $1 \times 6 \times N$ array where each row is of the form: $[k, j, n, S_k, \overline{U}(S_k), h(S_k)]$

l=length(I);

$[r1, r2, r3]$ =size(I);

V =zeros($r1, r2 + 3, r3$);

$V(1, 1 : r2, :) = I$;

terminal_nodes=find($I(1, r2, :) == max(I(1, r2, :))$);

last=length(*terminal_nodes*);

Fill in terminal nodes with the option payoff

for i = *terminal_nodes* **do**
  if *sum*$(A(:, i)) > 0$ *i.e if the* i^{th} *node is connected/in the traj. set* **then**
    $k = I(1, 1, i)$;
    $V(1, r2 + 1, i) = s_0 + k\delta_0$;
    $V(1, r2 + 2, i) = O(s_0 + k\delta_0, K)$;
  end
end

Go through the rest of the rows, first making sure it is the child of some previous node:

for *r=l-last:-1:1* **do**
  col=find($A(r, :)$);
  if *isempty(col)=/=1* **then**
    $k = I(1, 1, r)$;
    $S_i = s_0 + k\delta_0$ i.e. the "current" stock value.
    $V(1, r2 + 1, r) = S_i$;
    n=[];
    for *k=1:length(col)* **do**
      n=[n;$V(1, r2 + 2, col(k)), V(1, r2 + 1, col(k))$];

      i.e. n=[column of $\overline{U}$'s, column of stock values] from "future" times.
    end
    $[\overline{U}(S_i), h(S_i)]$=convexhull($n, S_i$);
    $V(1, r2 + 2, r) = \overline{U}(S_i)$;
    $V(1, r2 + 3, r) = h(S_i)$;
  end
end

1.7. Convex Hull Algorithm

Algorithm 3 below describes our implementation of the convex hull algorithm.

ALGORITHM 3: Convex hull pricing

Input: Set of a nodes children values and corresponding payoffs $n = [S_i, \overline{U}(S_i)]$ and stock value at node S_i

Output: Hedging value $\overline{U}$ and hedging ratio h

$\overline{U} = 0$;

$h = 0$;

$y_{max} = -\infty$;

for *i=1:l* **do**

  for *j=1:l* **do**

    if $n(i,2) = / = n(j,2)$ *and* $n(i,2) >= S_i$ *and* $n(j,2) < S_i$ **then**

      $x = \frac{n(i,1)-n(j,1)}{n(i,2)-n(j,2)}$;

      $y = n(i,1) - x(n(i,2) - s0)$;

      if $y >= y_{max}$ **then**

        $y_{max} = y$;

        $\overline{U} = y_{max}$;

        $h = x$;

      end

    end

  end

end

It is easy to see that the time complexity of this pricing algorithm will be on the order of b^2 at every node where $|\mathcal{N}_A(S_i, T_i, W_i)| \leq b$ (where $|B|$ denotes the cardinality of a set B). Combined with Algorithm 2, it follows that the time complexity for pricing is $O(b^2 N)$ where N is the number of nodes.

1.8. Complexity of Implementation

We describe next a simplified complexity analysis but that, nonetheless, captures the main features of space and time requirements for our implementation.

1.9. Space and Time Requirements for Implementation

Consider type I models first, we analyze the *direct product* version of that model (`MIDP`) as the obtained bound complexity also applies to the remaining model `MIOBS`. The number of nodes created when moving forward from the node (S_0, T_0) to stage $i = 1$ is `#NodesI` $\equiv (m_{max} - m_{min})\,(q_{max} - q_{min})$ for $i = 2$ is 4 `#NodesI` etc. Therefore, assuming the possibility that $q_{min} = 1$, an upper bound for the distinct number of nodes (S_i, T_i) will be:

$$\texttt{\#NodesI} \sum_{i=0}^{M_T} i^2 \propto \texttt{\#NodesI}\, M_T^3. \tag{3}$$

The pricing algorithm alluded above requires running the convex hull algorithm at every node, such computation is of the order `#NodesI`^{2} of operations per node. Therefore the pricing algorithm requires a number of operations of order `#NodesI`^{3} M_T^3. We note that a small value of δ_0, relative to δ, will enlarge $(m_{max} - m_{min})$.

For type II models, we restrict ourselves to the *direct product* version of that model (`MIIDPVAR`) as the obtained bound complexity also applies to the remaining model `MIIOBSVAR` (these models are defined in the paper). To simplify the analysis we assume that the variation constraint sets introduced in the paper are intervals: $V_{n\Delta} = [u_n, v_n]$. Also, set $v \equiv \max_{1\le n\le M_T}(v_n - u_n)$ which, for simplicity, will be used to obtain a complexity upperbound.

The number of nodes created when moving forward from the node (S_0, T_0) to $i = 1$ is $(m_{max} - m_{min})\ (q_{max} - q_{min})\ (\frac{v_1-u_1}{\delta_0}) \le$ `#NodesII` $\equiv (m_{max} - m_{min})\ (q_{max} - q_{min})\ \frac{v}{\delta_0}$. For $i = 2$ we have the upperbound 8 `#NodesII` etc. Therefore, assuming the possibility that $q_{min} = 1$, an upper bound for the distinct number of nodes (S_i, T_i, W_i) will be:

$$\texttt{\#NodesII} \sum_{i=0}^{M_T} i^3 \propto \texttt{\#NodesII}\ M_T^4. \tag{4}$$

The pricing algorithm alluded above requires running the convex hull algorithm at every node but it is independent of the W_i variable hence such computation is still of the order `#NodesI`^{2} of operations per node. Therefore the pricing algorithm requires a number of operations of order `#NodesI`^{2} `#NodesII` M_T^4.

Regarding the increase in computations when δ_0 is small relative to δ, as in the case of type I models, it will enlarge $(m_{max} - m_{min})$ but now it will also increase `#NodesII` through its dependence on v/δ_0. Our implementation of computations related to the variable W_i addresses this issue by providing user control over the possible number of values W_{i+1} that the program generates (such a number is denoted by K in Section 1.10 below). This topic is explained in Section 1.10.

The following analysis highlights the effect of the size of δ_0 on the number of computations. Let $\gamma_0 = C_0 0.01$ be the maximum historical jump (i.e. maximum chart change in one unit of time). It follows that $\delta \le x(t_{i+1}) - x(t_i) \le \delta + \gamma_0$ (we are assuming the chart moves up but the analysis is analogous for a down δ-move). If we discretize this range of values using 0.01 or δ_0 we obtain $R\ 0.01 = Q\ \delta_0 \equiv Q\ G\ 0.01 = \delta + \gamma_0 = F\ \delta_0 + \gamma_0$ (recall that $\delta_0 = G\ 0.01$). Therefore $Q = R/G$ will represent useful savings if G is substantially large. Also: $Q = \frac{\delta}{\delta_0} + \frac{\gamma_0}{\delta_0} = F + \frac{\gamma_0}{\delta_0}$. So $\frac{R}{G} = Q = F + \frac{C_0}{G}$.

1.10. Comments on Computation of Sums $\sum_{j=0}^{J-1} |p_j|$

Here we review some notation and then proceed to explain how to evaluate the possible admissible integers j_{i+1} leading to $\Delta_i W = j_{i+1}\ \delta_0$.

Assume (S_i, T_i, W_i) to be fixed and (m, q) given such that $S_{i+1} = S_i + m\delta_0 = S_0 + k_{i+1}\delta_0$ and $T_{i+1} = T_i + q\Delta = t_0 + n_{i+1}\Delta$. For each such pair (m, q) there will be many j_{i+1} leading to possible values of $W_{i+1} = j_{i+1}\delta_0$. As W_i is fixed, we concentrate in evaluating the possible values of $\Delta_i W = W_{i+1} - W_i = \delta_0\ \sum_{j=0}^{J-1} |p_j|$ where, according to previous explanations, the p_j and J are constrained as follows:

$$p_j \in \mathbb{Z},\ 1 \le |p_j|,\ \ 1 \le J \le q,\ J \in \mathbb{N},\ \ m = \sum_{j=0}^{J-1} p_j,\ |\sum_{j=0}^{R-1} p_j| < F, \tag{5}$$

$$\text{for all } 0 \le R \le J-1,\ \ |p_j| < 2F \ \text{ for all } 0 \le j \le J-2,$$

where $\delta \equiv F\ \delta_0$, $F \in \mathbb{N}$. The set of all J-tuples $(p_j)_{0\le j\le J-1}$ specified by (5) is denoted by $\mathcal{N}_W(m, q)$. At a given node (S_i, T_i, W_i) we want to generate future nodes $(S_{i+1}, T_{i+1}, W_{i+1}) \in \mathcal{N}_A(S_i, T_i, W_i)$. Systematically going through all the cases in (5) is cumbersome and rather unnecessary from a practical point

of view. For example, it is easier to produce J-tuples (p_j) from larger sets than $\mathcal{N}_W(m,q)$ and, for the purposes of the paper, it is enough to proceed that way. We have implemented another computational approach which consists in obtaining K, an integer that the user sets at will, values of W_{i+1} for each pair (m,q). This is achieved by an algorithm to generate $(p_j)_{j=0}^{J-1} \in \mathcal{N}_W(m,q)$. For each pair (m,q) we do the following and store the results globally before matrix construction. See lines `8-102` and `7-116` in`Model_2_Direct_Product.m` and `Model_2_Observable.m` to see the details for each type II model.

- If $q = 1$ take $p_0 = m$, $J = 1$ (only possibility).
- If $q > 1$ take also $p_0 = m$, $J = 1$ (so we move directly from (S_i, T_i) to (S_{i+1}, T_{i+1}) without any δ_0 increment in between. We force this case as it is easy and useful to include for all (m,q) and it gives the minimum possible value of variation as $m = \sum_{j=0}^{J-1} p_j$ and hence $m \leq \sum_{j=0}^{J-1} |p_j|$.
- $q > 1$, sample at random integers: $-F < p_0 < F$, $p_0 \neq 0$ and p_j, $1 \leq j \leq J-1$, in the range $-2F < p_j < 2F$, $p_j \neq 0$ for $j = 0, \ldots, J-1$ with

$$J \equiv \min(q, r^*) \tag{6}$$

  where r^* is the largest integer such that $|\sum_{j=0}^{i-1} p_j| < F$ for all $1 \leq i < r^*$ and $|\sum_{j=0}^{r^*-1} p_j| \geq F$ (or, equivalently, the smallest integer such that $|\sum_{j=0}^{r^*-1} p_j| \geq F$, notice that $r^* \geq 2$ as we are taking $-F < p_0 < F$). Now set $p_{J-1} \equiv m - \sum_{j=0}^{J-2} p_j$.
- `repeat` until there are K sequences $(p_j)_{j=0}^{J-1}$ for the given (m,q). K is typically chosen to be on the order of hundreds, usually exhausting all the possibilities.
- The above procedure gives $W_{i+1}^k - W_i \equiv \delta_0 \sum_{j=0}^{J-1} |p_j^k|$ where $(p_j^k)_{j=0}^{J-1} \in \mathcal{N}_W(m,q)$ is one of the K J-tuples described above. When at a node with W_i One then accepts W_{i+1}^k if it satisfies $W_{i+1}^k = W_i + \delta_0 \sum_{j=0}^{J-1} |p_j^k| \in V_{T_i}$ otherwise it is discarded.

AIMS Press