A short note on dynamic programming and pricing American options by Monte Carlo simulation

August 29, 2002

There is an increasing interest in sampling-based pricing of American-style options. In fact, lattice or finite difference methods are naturally suited to coping with early exercise features, but there are limits in the number of stochastic factors they can deal with. These limits are due to the increase in the size of the grid or the lattice which is used to discretize the state space. On the contrary, one of the major strengths of Monte Carlo simulation is just the ability to price high-dimensional derivatives. Unfortunately, dealing with early exercise requires going backward in time, as at each point in the state space you must compare the value of immediate exercise with the value of keeping the option, which is simply the price of the option at that point. Hence, it would seem that one has to chase her tail a bit, since while running a simulation forward in time, you should already know the option value. Indeed, until a few years ago, it was a common belief that simulation could not be applied to American-style options.

Actually, pricing an option with early exercise features requires solving a dynamic optimization problem by dynamic programming. Hence, in this short supplement we start with a brief account on dynamic programming (a streamlined version of Appendix D of [3]). Dynamic programming is arguably the most powerful concept in optimization, and its many potential applications are well illustrated in [1]. In section 1 we illustrate the principle behind dynamic programming with the simplest example, the shortest path problem in a network. Then, in section 2, we show the connection between this simple example and more general deterministic sequential decision processes. Actually, dynamic programming is able to cope also with *stochastic* programming problems, as those commonly encountered in finance. Despite its power, dynamic programming is limited by the so-called curse of dimensionality; in other words, the more state variables (in option pricing, the more stochastic factors), the higher the memory and time requirements of the approach.

To overcome the curse of dimensionality, a great deal of effort has been devoted to the development of approximate solution methods [1, 2], which also include simulation-based methods. This has paved the way to simulation-based pricing for high-dimensional American-style options. An example of this line of research is [9]. Actually, other strategies have been pursued and are surveyed in chapter 6 of [8]. Here we illustrate (section 3) a simple least-square based method originally proposed by Longstaff and Schwartz [6].

Finally, we should also mention a recent book on Monte Carlo methods, [5], which we recommend to anyone interested in applying this methodology in finance.

[1]This supplement should be used in conjunction with the book: P. Brandimarte, *Numerical Methods in Finance: a MATLAB-Based Introduction*, Wiley, 2001. Please refer to the web page (`www.polito.it/~brandimarte`) for further updates and supplements. Any comment is welcome. My e-mail address is: `brandimarte@polito.it`.

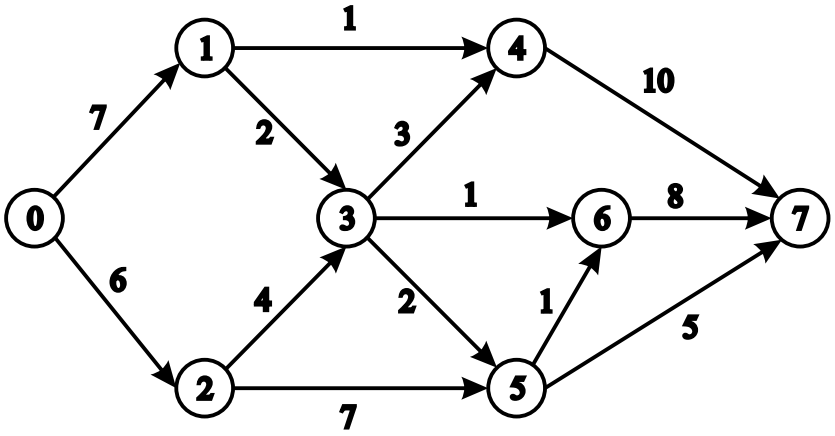

Figure 1: A shortest path problem

1 The shortest path problem

The best way of introducing dynamic programming is by considering one of its most natural applications, i.e., the problem of finding the shortest path in a network.

It is likely that a person working in finance had no previous exposure to graph and network optimization, but a quick look at figure 1 is enough to understand what we are talking about. A network consists of a set of nodes (here, numbered from 0 to 7) and arcs joining pairs of nodes. Arcs are labeled by a number which can be interpreted as the arc length. Our purpose is finding a path in the network, starting from node 0 and leading to node 7, such as the path has minimal length. For instance, path $(0, 1, 4, 7)$ has length 18, whereas path $(0, 1, 3, 5, 7)$ has length 16. Of course, you could simply enumerate all the possible paths to spot the optimal one; unlike problems in finance, here we have just a finite set of alternatives and there is no uncertainty involved. However, this approach becomes quickly infeasible, as the network size increases. So we must come up with some more clever way. Dynamic programming is one possible approach to the problem. It is worth noting more efficient algorithms are available for the shortest path problem, but the idea we illustrate can be extended to problems featuring either an infinite state space (countable or not) and uncertain data or both.

Let $\mathcal{N} = \{0, 1, 2, \ldots, N\}$ and $\mathcal{A}$ be the node and arc sets; let the start and final nodes be 0 and N respectively. For simplicity, we assume that the network is acyclic and that the arc lengths c_{ij}, $i, j \in \mathcal{N}$, are non-negative.

The starting point is to find a *characterization* of the optimal solution, that can be translated into a constructive algorithm. Let $F(j)$ be the length of the shortest path from node 0 to a node $j \in \mathcal{N}$ (denoted by $0 \xrightarrow{*} j$). Assume that, for a specific $j \in \mathcal{N}$, a node i lies on the path $0 \xrightarrow{*} j$. Then the following property holds: $0 \xrightarrow{*} i$ is a subpath of $0 \xrightarrow{*} j$. In other words, the optimal solution for a problem is obtained by assembling optimal solutions for subproblems. To understand why, consider the decomposition of $0 \xrightarrow{*} j$ into the subpaths $0 \to i$ and $i \to j$. The length of $0 \xrightarrow{*} j$ is the sum of the lengths of the two subpaths:

$$F(j) = L(0 \to i) + L(i \to j). \tag{1}$$

Note that the second subpath is not affected by *how* we go from 0 to i; this is strongly related to the concept of state in dynamic systems. Nodes can be interpreted as states of a dynamic decision process and decisions involve selecting the next state to visit. Now, assume that $0 \to i$ is not the optimal path from 0 to i. Then we could improve the first term of (1) by considering the path consisting of $0 \xrightarrow{*} i$ followed by $i \to j$. The length of this new path would be

$$L(0 \xrightarrow{*} i) + L(i \to j) < L(0 \to i) + L(i \to j) = F(j),$$

which is a contradiction, as we assumed $F(j)$ was the optimal path length.

This observation leads to the following recursive equation for the shortest path from 0 to a generic node j:

$$F(j) = \min_{(i,j)\in\mathcal{A}} \{F(i) + c_{ij}\} \qquad \forall j \in \mathcal{N}. \tag{2}$$

In other words, to find the optimal path from node 0 to node j, we should consider the optimal path from node 0 to each node i in the set of the immediate predecessors of j; then we compare, for each possible predecessor i, the sum of the cost of the optimal path from 0 to i and the one-step cost from i to j. This kind of recursive equation, whose exact form depends on the problem at hand, is the heart of dynamic programming and is an example of a *functional equation*. The problem requires computing $F(N)$, which is recursively defined in terms of F itself, evaluated at other nodes; the functional equation allows solving an overall problem by solving a set of smaller and simpler subproblems. In the shortest path problem, the shortest path from 0 to N is computed by finding the shortest paths from 0 to the predecessors of N; these in turn are computed considering the predecessors of the predecessors of N. By unfolding the recursion, we end up solving the trivial problem of finding the shortest paths from 0 to its immediate successors, as shown in the following example.

Example. Consider again the network depicted in Figure 1. To find the shortest path, we proceed by labeling each node $j = 1, \ldots, 7$ with $F(j)$ and its predecessor in the shortest path $0 \xrightarrow{*} j$. Note that we must label a node only after all of its predecessors have been labeled. We have numbered the nodes of the network in such a way that this is accomplished if we label the nodes following the order of their attached numbers. Such numbering can always be found in *acyclic* networks.

The initial condition is $F(0) = 0$. To label node 1 we solve

$$F(1) = \min_{(i,1)\in\mathcal{A}} \{F(i) + c_{i1}\} \qquad \forall j \in \mathcal{N}.$$

In this case this is trivial, since node 0 is the only predecessor of node 1:

$$F(1) = \min\{7 + 0\} = 7.$$

Similarly, for node 2 we have:

$$F(2) = \min\{6 + 0\} = 6.$$

For node 3, we must consider the two predecessor nodes 1 and 2:

$$F(3) = \min \left\{ \begin{array}{c} F(1) + c_{13} \\ F(2) + c_{23} \end{array} \right\} = \min \left\{ \begin{array}{c} 7 + 2 \\ 6 + 4 \end{array} \right\} = 9.$$

Going on this way yields:

$$F(4) = \min \left\{ \begin{array}{c} F(1) + c_{14} \\ F(3) + c_{34} \end{array} \right\} = \min \left\{ \begin{array}{c} 7 + 1 \\ 9 + 3 \end{array} \right\} = 8$$

$$F(5) = \min \left\{ \begin{array}{c} F(2) + c_{25} \\ F(3) + c_{35} \end{array} \right\} = \min \left\{ \begin{array}{c} 6 + 7 \\ 9 + 2 \end{array} \right\} = 11$$

$$F(6) = \min \left\{ \begin{array}{c} F(3) + c_{36} \\ F(5) + c_{56} \end{array} \right\} = \min \left\{ \begin{array}{c} 9 + 1 \\ 11 + 1 \end{array} \right\} = 10$$

$$F(7) = \min \left\{ \begin{array}{c} F(4) + c_{47} \\ F(5) + c_{57} \\ F(6) + c_{67} \end{array} \right\} = \min \left\{ \begin{array}{c} 8 + 10 \\ 11 + 5 \\ 10 + 8 \end{array} \right\} = 16.$$

Therefore, the shortest path is

$$0 \to 1 \to 3 \to 5 \to 7,$$

with a total length 16.

In the previous example we have used a *forward functional equation*, since we proceed from the start node to the final one. We can repeat the above argument the other way around, i.e., considering paths from a node i to node N, and obtaining a *backward functional equation*,

$$B(i) = \min_{(i,j)\in\mathcal{A}} \{c_{ij} + B(j)\} \qquad \forall i \in \mathcal{N}, \tag{3}$$

where $B(i)$ is the length of the shortest path from a node $i \in \mathcal{N}$ to N. The cost $B(i)$ is called, for obvious reasons, the **cost-to-go**; another term is **value function**. The value function, evaluated in a point in the state space, tells you what would be the future optimal cost whenever you reach that state and go on with an optimal policy. The backward equation above states that in order to find the optimal path from node i to the terminal node N, you must consider all the possible one-step transitions to some next state j, followed by the optimal path, whose value is $B(j)$. Hence, knowing the value function at each state allows a straightforward solution of the problem, in principle; so, solving a decision problem by dynamic programming actually requires computing the value function for each point in the state space. This might seem like a clumsy approach, but even in our simple shortest path problem this is better than an exhaustive enumeration of the alternatives. Furthermore, the same idea may be applied when uncertainty is involved and the value function is defined as an expected value, as we will see shortly. We leave as an exercise the task of finding the optimal path in the network above by backward dynamic programming. In this case you will label nodes starting from node 7 and going backward until you reach node 0.

Depending on the problem at hand, it may be better to adopt a backward or a forward approach. In option pricing, it is usual to go backward in time, since we know the option value at expiration; the same applies when you use dynamic programming to solve optimal portfolio problems. Indeed, when you use a binomial lattice to price a simple American option, you are actually building a graph, whose nodes are labeled by a backward process. The difference with general network optimization is that you have a structured graph, representing a discretization of a state space involving the price of the underlying and time. Since time is a characteristic of dynamic decision processes, we consider sequential decision processes in the next section. Another difference between the shortest path problem and option pricing is due to uncertainty: in the shortest path problem you control the next node you are going to, whereas in pricing an American option the up or down movement in the graph is random and beyond your control; the only decision you can take is exercising the option or not.

2 Sequential decision processes

In this section we generalize the functional equation approach developed in the previous section for the shortest path problem. Consider a discrete-time dynamic system modeled by the state equation

$$\mathbf{x}_t = \mathbf{h}_t(\mathbf{x}_{t-1}, \mathbf{u}_t) \qquad t = 1, 2, \ldots, T \tag{4}$$

where $\mathbf{x}_t$ is the vector of the state variables *at the end* of time interval t and $\mathbf{u}_t$ is the vector of the control variables applied *during* time interval t. The initial state $\mathbf{x}_0$ is given; the final state $\mathbf{x}_T$ may be constrained or not. No uncertainty is considered here: given the current value of the state variable $\mathbf{x}_{t-1}$, after selecting the control variable $\mathbf{u}_t$ we know exactly what the future state will be, according to the time-varying dynamics described by the $\mathbf{h}_t$ functions. This is usually not the case in finance: if the state variable is current wealth and we control

the allocation of wealth between a risky and a non-risky asset, then the future wealth will not be known for sure. Actually, by selecting a control variable in a stochastic problem, you just select a *probability distribution* for the next state.

A sequential decision process consists of selecting a sequence of controls $\mathbf{u}_t$ over a finite (or infinite) time horizon in order to minimize a certain objective function, possibly subject to constraints on the state and the control variables. We consider here only finite horizon problems. The objective function depends, in general, on the **state trajectory** $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ and the control sequence $(\mathbf{u}_1, \ldots, \mathbf{u}_T)$. Our optimization problem is

$$\min_{\mathbf{u}_1,\ldots,\mathbf{u}_T} f(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_T; \mathbf{u}_1, \ldots, \mathbf{u}_T) \tag{5}$$

subject to dynamic constraints (4). Actually, we can write the objective function in a more compact form, since, given (4):

$$\begin{aligned}
&\mathbf{x}_1 = \mathbf{h}_1(\mathbf{x}_0, \mathbf{u}_1) \\
&\mathbf{x}_2 = \mathbf{h}_2(\mathbf{x}_1, \mathbf{u}_2) = \mathbf{h}_2\{\mathbf{h}_1(\mathbf{x}_0, \mathbf{u}_1), \mathbf{u}_2\} \\
&\ldots \qquad .
\end{aligned}$$

The optimal value depends only on $\mathbf{x}_0$ and $(\mathbf{u}_1, \ldots, \mathbf{u}_T)$. Since the optimal value of the objective function depends essentially on the initial state $\mathbf{x}_0$ and the horizon length, i.e., the number T of steps of the decision process, we can denote its value as the cost-to-go $F_T(\mathbf{x}_0)$.

The optimal control sequence

$$(\mathbf{u}_1^*, \ldots, \mathbf{u}_T^*),$$

and the optimal trajectory

$$(\mathbf{x}_0, \mathbf{x}_1^*, \ldots, \mathbf{x}_T^*)$$

must comply with different types of constraints. Eq. (4) is only one of them; we could have constraints on the initial and/or the final state and on the admissible controls. A typical discrete-time optimal control problem is:

$$\begin{aligned}
\min \quad & \sum_{t=1}^{T} f_t(\mathbf{x}_{t-1}, \mathbf{u}_t) && \\
\text{s.t.} \quad & \mathbf{x}_t = \mathbf{h}_t(\mathbf{x}_{t-1}, \mathbf{u}_t) \qquad && t = 1, 2, \ldots, T \\
& \mathbf{g}_t(\mathbf{x}_{t-1}, \mathbf{u}_t) \leq 0 \qquad && t = 1, 2, \ldots, T,
\end{aligned} \tag{6}$$

where usually $\mathbf{x}_0$ is given. We could also add conditions or cost penalties for the final state $\mathbf{x}_T$. The functions $\mathbf{g}_t$ express constraints on the state and control variables. Note that we have assumed an *additive* objective function; this is fundamental in order to derive the functional equation, which is a form of decomposition. Note, however, that decomposition is possible in more general settings.

2.1 The optimality principle and solving the functional equation

The objective function (6) is *separable*, in the sense that, for a given number r, the contribution of the last r steps depends only on the current state $\mathbf{x}_{T-r}$ and the r controls $\mathbf{u}_{T-r+1}, \ldots, \mathbf{u}_T$. Furthermore, a similar separation property (known as *Markovian state property*) holds for the trajectory, in the sense that the state $\mathbf{x}_{t+1}$ reached from $\mathbf{x}_t$ by applying the control $\mathbf{u}_{t+1}$ depends only on $\mathbf{x}_t$ and $\mathbf{u}_{t+1}$, and not on the past history $\mathbf{x}_0, \ldots, \mathbf{x}_{t-1}$. A consequence of such separation properties is the **optimality principle**.

An optimal policy $(\mathbf{u}_1^*, \mathbf{u}_2^*, \ldots, \mathbf{u}_T^*)$ *is such that, whatever the initial state* $\mathbf{x}_0$ *and the first control* $\mathbf{u}_1^*$*, the next controls* $(\mathbf{u}_2^*, \ldots, \mathbf{u}_T^*)$ *are an optimal policy for the* $(T-1)$*-stage problem with initial state* $\mathbf{x}_1$*, obtained by applying the first control* $\mathbf{u}_1^*$*.*

Therefore, we may write a recursive functional equation to obtain the optimal policy:

$$F_T(\mathbf{x}_0) = \min_{\mathbf{u}_1} \{ f_1(\mathbf{x}_0, \mathbf{u}_1) + F_{T-1}(\mathbf{h}_1(\mathbf{x}_0, \mathbf{u}_1)) \}, \tag{7}$$

where the minimization is carried out taking into account constraints on the control variable. This is a *backward* functional equation.

The reader is invited to compare the functional equation above with stochastic programming problems with recourse described in the book (page 138). Actually, the recourse function is strongly related to the value function; one of the main differences between dynamic programming and stochastic programming with recourse lies in the solution approach. For instance, the L-shaped decomposition method described in section 3.7 aims at approximating the recourse function near the optimal solution, whereas, in principle, dynamic programming requires an exact evaluation of the value function for each possible state. This may help in solving complex problems which are beyond the reach of dynamic programming. On the other hand, stochastic programming with recourse is usually based on some approximation of the uncertainty by a set of scenarios; dynamic programming, in principle, allows an exact solution of the problem. You may consider stochastic programming with recourse as a specific (and possibly quite efficient) computational way to deal with certain stochastic optimization problems, whereas dynamic programming is a bit more general.

The functional equation has a boundary condition that helps to start unfolding the recursion. If the final state $\mathbf{x}_T$ is given, $F_1(\mathbf{x}_{T-1})$ must be simply computed for each state $\mathbf{x}_{T-1}$ from which we can reach $\mathbf{x}_T$ in one step. Even if the final state is not given (as it is the case with a stochastic problem), we may still obtain the function $F_1(\mathbf{x}_{T-1})$ by solving a relatively simple or trivial problem. In the case of option pricing, it is a simple matter to find the optimal exercise strategy at maturity.

The T-stage problem can be dealt with by solving the $(T-1)$-stage problem, finding the optimal controls $(\mathbf{u}_2^*(\mathbf{x}_1), \ldots, \mathbf{u}_T^*(\mathbf{x}_1))$ for each possible initial state $\mathbf{x}_1$ reachable from x_0. Note that this control sequence depends on the next state $\mathbf{x}_1$, just like the cost-to-go function $B(\cdot)$ in equation (3) depends on the next node j. The $(T-1)$-stage problem in turn can be dealt with by solving a set of $(T-2)$-stage problems, and so on, until trivial one-step problems are obtained. When the $(T-1)$-stage problems are solved, we must find the first optimal control $\mathbf{u}_1^*$ by considering, for each pair $(\mathbf{u}_1, \mathbf{x}_1)$, only the optimal sequence starting from $\mathbf{x}_1$; the alternative trajectories starting from $\mathbf{x}_1$ are discarded when solving the T-stage problem. This is why computational savings are obtained, with respect to the complete enumeration of all the possible trajectories. Nevertheless, high requirements in memory storage and computation time may hinder the practical application of dynamic programming in many cases.

It is worth noting that a sequential decision process with a finite state space and a finite time horizon is equivalent to a shortest path problem in a *layered* network (see figure 2): the arc lengths are equal to the cost of the corresponding state transition. Due to the structure of the network, we have no difficulty in guaranteeing that all the successors (or predecessors) of a node have been labeled; we have only to label nodes layer after layer. Obviously, we can go forward or backward in the network above; the best choice depends on the problem at hand.

In finance, you commonly find sequential decision processes when dealing with portfolio optimization. In this case, neither the state space nor the set of admissible control variables are finite. Time may be assumed discrete or continuous. Continuous-time models are quite useful when the model is reasonably simple and an analytical solution can be found, usually yielding

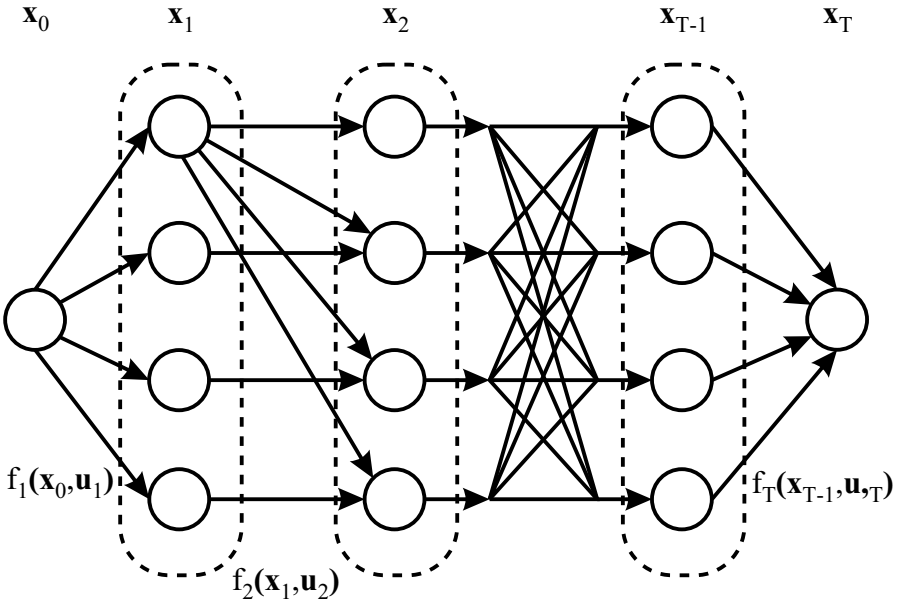

Figure 2: A shortest path representation of a finite sequential decision process (for clarity, not all transitions are shown); the final state is assumed fixed.

valuable insights in the nature of the problem (an excellent reference on continuous-time dynamic programming in finance is [7]). In other cases, even if the problem is continuous-time in nature, it is usually preferred to solve an approximate discrete-time version of the problem. By the way, when you want to price an American option, you usually discretize with respect to time, thereby effectively pricing a Bermudan one (hopefully, finer time discretization leads to a reasonably quick convergence of the approximate price to the exact one).

It is possible to write a functional equation for continuous-time decision processes, whereby the dynamic equations are differential equations rather than discrete-time difference equations. In finance, uncertainty would be often modeled by a stochastic differential equation involving diffusion processes (just like the models discussed in chapter 1 of the book). It can be shown (see, e.g., appendix D of [3] for a heuristic justification) that this leads to a (difficult) partial differential equation. In the case of American options, this boils down to the familiar Black-Scholes partial differential equation plus some free-boundary condition accounting for the early exercise feature. However, numerical solution of this equation by finite differences requires some form of discretization, just like tackling the problem by Monte Carlo simulation.

Probably, however, the most important feature of dynamic programming models in finance is uncertainty: it is impossible to derive an optimal control sequence, as you cannot know a priori the state trajectory that you will follow. Hence, the optimal control must given in a feedback or control policy form like $\mathbf{u}_t = \boldsymbol{\phi}_t(\mathbf{x}_{t-1})$. In other words, the optimal control depends on the current state and it is selected in order to minimize the sum of immediate cost and cost-to-go; in fact, in finance a backward solution strategy is usually adopted. In this case, equation (7) could be rewritten as:

$$V_0(\mathbf{x}_0) = \min_{\mathbf{u}_1} \left\{ f_1(\mathbf{x}_0, \mathbf{u}_1) + \mathrm{E}\left[V_1(\mathbf{x}_1) | \, \mathbf{x}_0, \mathbf{u}_1 \right] \right\}. \tag{8}$$

There is a difference in the notation between (7) and (8); in the first case the notation F_T suggests that there are T decisions to go, whereas the notation V_0 or V_1 points out the time at which the decision is taken. A much more important difference, of course, is that here the definition of the value function entails a conditional expectation. In fact, the next state $\mathbf{x}_1$ is *not* known as a deterministic function $\mathbf{h}_1(\mathbf{x}_0, \mathbf{u}_1)$; rather, applying control $\mathbf{u}_1$ when we are in

state $\mathbf{x}_0$ defines a probability distribution for the next state $\mathbf{x}_1$. The expectation depends on this probability distribution, and so it is conditional on $\mathbf{x}_0$ and $\mathbf{u}_1$. Solving (8) defines a policy function $\mathbf{u}_1 = \boldsymbol{\phi}_1(\mathbf{x}_0)$; going one step ahead defines a similar policy function $\mathbf{u}_2 = \boldsymbol{\phi}_2(\mathbf{x}_1)$ depending on an uncertain state $\mathbf{x}_2$.

In practice, it is difficult to find the policy functions $\boldsymbol{\phi}_t$ explicitly, and one has to resort to numerical approximations. A typical approach is to discretize the state space in order to compute the value function on a discrete grid, using some form of interpolation[1] to obtain an approximate representation. See, e.g., [4] for numerical tricks useful in solving discrete-time models.

Unfortunately, when the number of state variables grows, the complexity of doing so increases exponentially: you should evaluate or approximate the value function for a huge collection of points in the state space (the graph of figure 2 gets unmanageable). In the next section we illustrate the relationship between dynamic programming and option pricing when early exercise is involved, and how a clever approximation enables to solve the problem.

3 Approximate dynamic programming and option pricing by Monte Carlo simulation

When adopting Monte Carlo simulation in option pricing, one usually discretizes time to generate discrete-time sample paths of the price $S(t)$ of the underlying asset. Let $t = 0$ be the current time and $t = T$ be the option maturity. Then we select a time step Δt and generate sample paths of prices, like $(S_0, S_1, \ldots, S_i, \ldots, S_N)$, where $S_i = S(i \cdot \Delta t)$ and $T = N \cdot \Delta t$. If early exercise is ruled out, and assuming a constant risk-free interest rate r, the option price is given by:

$$\mathrm{E}^Q \left[e^{-rT} f(S_0, S_1, \ldots, S_i, \ldots, S_N) \right],$$

where the function $f(\cdot)$ is the (possibly path-dependent) payoff and $\mathrm{E}^Q[\cdot]$ denotes expectation under a risk-neutral measure (see section 7.2 in the book). By running a large number of replications (i.e., generating many sample paths) we may estimate the expectation above.

In the case of European options, no decision must be taken, apart from a trivial one at maturity (you just check if the option is in-the-money or not). Since generating trajectories for multiple assets or when volatility is stochastic is relatively easy, Monte Carlo simulation turns out to be a very powerful tool for complex option pricing problems [5]. However, when early exercise is considered, we must find out an optimal exercise strategy. The point is that at each simulated time instant, the value of the contract should be evaluated as the maximum of the immediate payoff if the option is exercised, i.e., the intrinsic value $I_i(S_i)$, and the continuation value. Consider a simple American put option written on a single non-dividend paying asset; in this case, we have $I_i(S_i) = \max\{X - S_i, 0\}$, where X is the strike price. In the following we will use this simple example for illustrative purposes. The continuation value is the time-discounted expected value of the future cash flows, under a risk-neutral probability measure, assuming an optimal exercise policy is adopted in the future. When the intrinsic value is larger than the continuation value, i.e., it is more fruitful to exercise the option immediately than keeping it alive, we just do it. This implicitly defines an early exercise barrier, which is the frontier between states in which it is optimal to exercise the option and states in which it is optimal to keep the option. Assuming again a single underlying asset, the value $V_i(S_i)$ of the option at time step i, conditional on the current price S_i, is given by:

$$V_i(S_i) = \max \left\{ I_i(S_i), \mathrm{E}_i^Q \left[e^{-r \cdot \Delta t} V_{i+1}(S_{i+1}) \middle| S_i \right] \right\}. \tag{9}$$

[1] Interpolation is discussed in section 2.3. of the book.

Note how the value function $V(\cdot)$ at different time steps is defined recursively, just like in the dynamic programming recursion (8). Here the value function is not additive, as you receive at most one payoff from exercising the option; furthermore, the only control decision you can take is exercise versus continuation. The next point in the state space, i.e., the next price of the underlying asset, is not influenced by your decision. These differences notwithstanding, this recursive formulation points out the dynamic programming nature of option pricing when early exercise is involved (we can see our problem as a specific case of a more general class of optimal stopping problems under uncertainty). It also makes clear that the difficulty derives from the fact the we should know the expected value of the future option value, conditional on the current underlying price, but this depends on the next exercise decisions. In other words, to know the optimal strategy now, we should know the optimal strategy for the future, which is defined implicitly by a value function. In fact, in lattice-based methods (implemented, e.g., in the Financial Toolbox function `binprice`), one generates the price lattice going forward in time, but the exercise decisions are taken going backward in time; a similar consideration applies to finite difference methods. Indeed, the binomial lattice approach (see, e.g., section 1.4.4), is nothing but an approximate way to solve a stochastic dynamic programming problem based on a very specific discretization of the state space. We emphasize again that, unlike general dynamic programming problems, here the control variable is quite simple: you exercise or not, and this makes the approach simple and powerful. However, when multiple stochastic factors are present, coming up with a compact discretization of the state space is difficult. This is an instance of the well-known "curse of dimensionality" which makes dynamic programming difficult when the number of state variables is large.

So a direct generalization and application of (9) is not feasible. However, in order to price high-dimensional options with early exercise features, one can try to approximate the value function. This applies to the general problem (8): if you are able to find a good approximation of the expected value function $\mathrm{E}[V_t(\mathbf{x}_t) \mid \mathbf{x}_{t-1}, \mathbf{u}_t]$, then solving the optimization problem yields the policy function $\mathbf{u}_t = \boldsymbol{\phi}_t(\mathbf{x}_{t-1})$. Note that we will get in general a lower bound on the option value, as we are actually approximating the optimal early exercise policy and we are considering discrete rather than continuous time (in other words, we are pricing Bermudan-style options rather than true American-style ones). A simulation-based strategy for doing so in a rather general setting is described in [9]. Typically, approximating a function entails projecting it onto a set of *basis functions*. This means that a complex (or unknown) function $f(\mathbf{x})$ may be approximated as

$$f(\mathbf{x}) \approx \sum_{i=k}^{N} a_k \psi_k(\mathbf{x}),$$

where ψ_k, $k = 1, \ldots, N$, is a collection of simpler basis functions. The problem then boils down to finding the best set of coefficients a_k, assuming a certain metric measures the approximation error. Least squares is a natural way to address this projection problem, and in the following we illustrate a function implementing an approach described by Longstaff and Schwartz [6].

One possibility to approximate the conditional expected value is to regress against a simple polynomial in the current asset price, for instance:

$$\mathrm{E}_i^Q \left[e^{-r \cdot \Delta t} V_{i+1}(S_{i+1}) \middle| S_i \right] \approx a_1 + a_2 S_i + a_3 S_i^2.$$

In this case, the basis functions are defined as $\psi_k^i(S_i) = S_i^{k-1}$. Alternative approximations may be obtained, e.g., by using sets of orthogonal polynomials. Note that the approximation is a nonlinear function, but it is actually linear in the unknown coefficients a_1, a_2, and a_3. In order to obtain the coefficients, the algorithm uses the prices S_i along the sample paths, in a sense, as the X-values.[2]. The Y-values are provided by the future payoffs on the same

[2]The terminology here is due to the fact that in common least squares problems you regress a vector of

sampled paths. While the generation of sampled paths goes forward in time, the regressions go backwards in time, starting from the last period. In the last period, the exercise policy is trivial: you exercise the option if and only if it is in-the-money. Let us denote the price at time step i in replication j as S_i^j. Then, for the price path j, the cash flow for put option at maturity would be $\max\{X - S_N^j, 0\}$, provided the option has not been exercised yet. Now consider time step $N-1$, i.e., the second-to-last period. If the option is in-the-money on a path, i.e., $S_{N-1}^j < X$, then we may consider exercising the option. To approximate the continuation value, we use a regression: here the cash flows deriving from the exercise decisions in the last period are discounted and used as Y-values. A trick suggested in the literature is to base the regression only on paths where the option is in-the-money; if the option is out-of-the-money, then you would not exercise it anyway and the knowledge of the continuation value is no use. So, considering the subset $\mathcal{J}_{N-1}$ of paths for which the option is in-the-money at time $N-1$, to approximate the continuation value we would consider a regression like:

$$e^{-r\cdot\Delta t}\max\{X - S_N^j, 0\} = y^j = a_1 + a_2 S_{N-1}^j + a_3 (S_{N-1}^j)^2 + \epsilon^j \qquad j \in \mathcal{J}_{N-1}, \tag{10}$$

where the parameters a_1, a_2, and a_3 are obtained by minimizing $\sum_{j\in\mathcal{J}}(\epsilon^j)^2$. An important point to understand is that the early exercise decision is based on the regressed polynomial, in which the coefficients are common on each sample path; the decision is *not* based on the knowledge of the future prices along each sample path. In other words, when considering price S_{N-1}^j, we do *not* use knowledge of the future price S_N^j on the same path to take the exercise decision, because this would imply a form of "clairvoyance". Rather, we compare the intrinsic value $X - S_{N-1}^j$ with an approximation of the continuation value: $a_1 + a_2 S_{N-1}^j + a_3 (S_i^{N-1})^2$. The process is repeated backward in time. To carry out the regression, you must consider the cash flows on each path, resulting from the early exercise decisions. Say we are at time step i, and consider path j. For each path j, there will be an exercise time j_e^*, which we set conventionally to $N+1$ if the option will never be exercised in the future. Then the regression equation (10) should be rewritten as

$$a_1 + a_2 S_i^j + a_3 (S_i^j)^2 + \epsilon^j = \begin{cases} e^{-r(j_e^* - i)\cdot\Delta t}\max\{X - S_{j_e^*}^j, 0\} & \text{if } j_e^* \le N \\ 0 & \text{if } j_e^* = N+1 \end{cases} \qquad j \in \mathcal{J}_i, \tag{11}$$

for the paths $j \in \mathcal{J}_i$ which are in-the-money at time step i. Note that there can be only one exercise time for each path: it may be the case that after comparing the intrinsic value with the continuation value on a path, the exercise time i^* is reset to a previous period.

More details on the rationale behind the method and its convergence are given in the original reference [6], as well as more significant examples.

The reader is invited to reflect on the connection of this idea with the metamodeling approach described in section 4.6 to integrate optimization and simulation. The main difference is that when you use metamodeling as described in the book, you are looking for a *local* approximation of the response surface, i.e., you are interested in approximating an unknown function just in a neighborhood of the current point within an iterative optimization procedure. Here, we are looking for a *global* approximation to be used as a value function within an approximate dynamic programming strategy. This is why metamodeling may even exploit a simple linear approximation to approximate a gradient of a complex function, which would be of little use here.

observed values $\mathbf{y}$ against a matrix of values $\mathbf{X}$ of explanatory variables according to a relation like $\mathbf{y} = \mathbf{X}\mathbf{a} + \boldsymbol{\epsilon}$, where the optimal set of parameters $\mathbf{a}$ is obtained by minimizing the norm of the error vector $\boldsymbol{\epsilon}$.

```
function SPaths=GenPathsA(S0,mu,sigma,T,NSteps,NRepl)
dt = T/NSteps;
nudt = (mu-0.5*sigma^2)*dt;
sidt = sigma*sqrt(dt);
RandMat = randn(round(NRepl/2), NSteps);
Increments = [nudt + sidt*RandMat ; nudt - sidt*RandMat];
LogPaths = cumsum([log(S0)*ones(NRepl,1) , Increments] , 2);
SPaths = exp(LogPaths);
```

Figure 3: Generating antithetic sample paths.

3.1 A MATLAB implementation of the least squares approach

Now we illustrate a possible MATLAB implementation of this approach for an American put on a single underlying asset.

To begin with, in figure 3, we illustrate a function to generate antithetic sample paths. Since antithetic sampling is very simple to apply, it is worth a try. The function is a straightforward extension of function `AssetPaths1` described in page 319 of the book. Note here a couple of limitations. First, the function assumes that the number of replications `NRepl` is even and generates a set of `NRepl/2` antithetic pairs; using an odd value of this parameter results in an error. Second, care must be taken in the way you store the replications in the matrix `SPaths` if you want to use them to come up with a confidence interval, since you should pair antithetic replications before computing a confidence interval; this is not done in the following, and is left as an exercise for the reader. We also recall that antithetic sampling is not always guaranteed to work; so you may consider other variance reduction strategies.

The function above is just a simple example with many limitations, but it illustrates the idea showing it is rather simple to implement. After a few lines of initialization, the main `for` loop goes through the regressions backward in time. The vectors `CashFlows` and `ExerciseTime` store, for each replication, the exercise time and the cash flow from early exercise. The vector `InMoney` stores the set of paths for which the option is in-the-money at each time step (it corresponds to the set $\mathcal{J}_i$ above). Then we prepare the data for the regression:

- the matrix `RegrMat` which results from evaluating the chosen set of basis functions for the prices `XData`;
- the elements of `YData`, which are nothing but discounted future cash flows deriving from possible future early exercise.

From the least-squares regression[3] we get the set of parameters `a` which is used to approximate the continuation value, which is checked against the intrinsic value. Then in vector `k` we collect the replications for which it is optimal to exercise the option now, and we update vectors `CashFlows` and `ExerciseTime` accordingly.

Now we may check the results we obtain by a comparison with those provided by a lattice based method.

```
>> S0 = 36;
>> X = 40;
```

[3]Simple least-squares regression is obtained in MATLAB by applying the same backslash operator \ used for solving systems of linear equations; MATLAB can tell the difference by checking the dimensions of the matrix and vector involved.

```
function price = LongstaffSchwartz(S0,X,r,T,sigma,NSteps,NRepl)
dt = T/NSteps;
discount = exp(-r*dt);
% discount rates over different time intervals
discountVet = exp(-r*dt*(1:NSteps)');
a = zeros(3,1); % regression parameters
% generate sample paths
SPaths=GenPathsA(S0,r,sigma,T,NSteps,NRepl);
SPaths(:,1) = []; % get rid of starting prices
%
CashFlows = max(0, X - SPaths(:,NSteps));
% first set exercise time at expiration for convenience
ExerciseTime = NSteps*ones(NRepl,1);
for step = NSteps-1:-1:1
    InMoney = find(SPaths(:,step) < X);
    XData = SPaths(InMoney,step);
    RegrMat = [ones(length(XData),1), XData, XData.^2];
    YData = CashFlows(InMoney) .* discountVet(ExerciseTime(InMoney) - step);
    a = RegrMat \ YData;
    IntrinsicValue = X - XData;
    ContinuationValue = RegrMat * a;
    Exercise = find(IntrinsicValue > ContinuationValue);
    k = InMoney(Exercise);
    CashFlows(k) = IntrinsicValue(Exercise);
    ExerciseTime(k) = step;
end % for
price = mean(CashFlows.*discountVet(ExerciseTime));
```

Figure 4: Implementing the least squares approach by regression against a second-order polynomial.

```
>> r = 0.06;
>> T = 1;
>> sigma = 0.2;
>> NSteps = 50;
>> NRepl = 100000;
>> randn('state',0)
>> price = LongstaffSchwartz(S0,X,r,T,sigma,NSteps,NRepl)
price =
    4.4720
>> [asset, option] = binprice(S0,X,r,T,1/1000,sigma,0);
>> option(1,1)
ans =
    4.4868
```

Note that we have reset the state of the random number generator to allow exact replication of the experiment; using the `'state'` string ensures that the most recent generator is used, whereas `'seed'` would tell MATLAB to use the MATLAB 4 generator. Of course, one run with no analysis of the confidence interval does not mean anything. The point here was just to show that the regression based strategy proposed in recent papers is definitely worth considering, and that the approach can be easily implemented. If you try the experiment, you will notice that the time needed by the simulation strategy is larger than that required by the lattice approach. Better variance reduction strategies, or low-discrepancy sequences, could be used to improve the code but, clearly, there is little point in using Monte Carlo simulation for an American put on a single underlying asset. Still, when the number of stochastic factors is large, a sampling based method may well be the only feasible alternative. In the case of multiple assets, say S_1 and S_2, one could consider regressing against polynomials involving cross-products such as S_1S_2, $S_1^2S_2$, $S_1S_2^2$, etc. (see [8] for some examples).

There is one limitation in the program above that we would like to address here. The program assumes a specific form of the basis functions. Clearly, it would be better to write a more generic function allowing for more flexibility and experimentation with the basis functions. This can be done using the concept of a function handle. A function handle is a sort of "pointer" to a function, which is created by using the `@` operator, and is passed to the `feval` function to be applied on some argument:

```
>> f=@sin
f =
    @sin
>> feval(f,2)
ans =
    0.9093
```

You may build a vector of function handles and apply a function handle to a vector argument:

```
>> vf = [@sin @cos]
vf =
    @sin    @cos
>> feval(vf(2),[2 3])
ans =
   -0.4161   -0.9900
```

However, you may not pass `feval` a vector of function handles. Now we need a function to create a vector of function handles corresponding to the set of basis functions. The job is

```
function vf = BuildHandles;
vf(1) = @basisf1;
vf(2) = @basisf2;
vf(3) = @basisf3;

function r = basisf1(x)
r = ones(length(x),1);
function r = basisf2(x)
r = x;
function r = basisf3(x)
r = x.^2;
```

Figure 5: A function creating a vector of function handles for the basis functions.

done by the function `BuildHandles` illustrated in figure 5. Note that some care must be taken in making the function work with vector arguments, since this is required by the function `GenericLS` illustrated in figure 6.

The new function works much like the previous one. There is one extra argument containing the vector of function handles. Since `feval` does not work with a vector of function handles, we must build the regression matrix by a `for` loop. Notice also that we preallocate the matrix `RegrMat`, and then we use only a subset of rows, corresponding to the number of paths for which the option is in-the-money at the time instant considered in each iteration. Now we may check the function with the same basis functions as before:

```
>> fvet = BuildHandles
fvet =
    @basisf1    @basisf2    @basisf3
>> randn('state',0)
>> price = GenericLS(S0,X,r,T,sigma,NSteps,NRepl,fvet)
price =
    4.4720
```

In practical cases, a suitable selection of basis functions may affect the quality of the results.

References

[1] D.P. Bertsekas. *Dynamic Programming and Optimal Control (2nd Ed., Vols 1 and 2).* Athena Scientific, Belmont (MA), 2001.

[2] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont (MA), 1996.

[3] P. Brandimarte and A. Villa. *Advanced Models for Manufacturing Systems Management.* CRC Press, Boca Raton (FL), 1995.

[4] C. Carroll. *Solving Microeconomic Dynamic Stochastic Optimization Problems.* Lecture Notes downloadable from `http://www.econ.jhu.edu/people/ccarroll/index.html`.

[5] P. Jaeckel. *Monte Carlo Methods in Finance.* Wiley, Chichester, 2002.

```
function price = GenericLS(S0,X,r,T,sigma,NSteps,NRepl,fhandles)
dt = T/NSteps;
discount = exp(-r*dt);
discountVet = exp(-r*dt*(1:NSteps)');
NBasis = length(fhandles); % number of basis functions
a = zeros(NBasis,1); % regression parameters
RegrMat = zeros(NRepl,NBasis);
% generate sample paths
SPaths=GenPathsA(S0,r,sigma,T,NSteps,NRepl);
SPaths(:,1) = []; % get rid of starting prices
%
CashFlows = max(0, X - SPaths(:,NSteps));
ExerciseTime = NSteps*ones(NRepl,1);
for step = NSteps-1:-1:1
    InMoney = find(SPaths(:,step) < X);
    XData = SPaths(InMoney,step);
    for i=1:NBasis
        RegrMat(1:length(XData), i) = feval(fhandles(i), XData);
    end
    YData = CashFlows(InMoney) .* discountVet(ExerciseTime(InMoney) - step);
    a = RegrMat(1:length(XData),:) \ YData;
    IntrinsicValue = X - XData;
    ContinuationValue = RegrMat(1:length(XData),:) * a;
    Exercise = find(IntrinsicValue > ContinuationValue);
    k = InMoney(Exercise);
    CashFlows(k) = IntrinsicValue(Exercise);
    ExerciseTime(k) = step;
end % for
price = mean(CashFlows.*discountVet(ExerciseTime));
```

Figure 6: A more general implementation of the least squares approach.

[6] F.A. Longstaff and E.S. Schwartz. Valuing American Options by Simulation: a Simple Least-Squares Approach. *The Review of Financial Studies*, 14:113-147, 2001.

[7] R.C. Merton. *Continuous-Time Finance*. Blackwell Publishers, Malden (MA), 1990.

[8] D.A. Tavella. *Quantitative Methods in Derivatives Pricing: an Introduction to Computational Finance*. Wiley, New York, 2002.

[9] J.N. Tsitsiklis and B. Van Roy. Optimal Stopping of Markov Processes: Hilbert Space Theory, Approximation Algorithms, and an Application to Pricing High-Dimensional Financial Derivatives. *IEEE Transactions on Automatic Control*, 44:1840-1851, 1999.