SIAM REVIEW
Vol. 44, No. 4, pp. 661–677

Nine Ways to Implement the Binomial Method for Option Valuation in MATLAB*

Desmond J. Higham[†]

Abstract. In the context of a real-life application that is of interest to many students, we illustrate how the choices made in translating an algorithm into a high-level computer code can affect the execution time. More precisely, we give nine MATLAB programs that implement the binomial method for valuing a European put option. The first program is a straightforward translation of the pseudocode in Figure 10.4 of *The Mathematics of Financial Derivatives*, by P. Wilmott, S. Howison, and J. Dewynne, Cambridge University Press, 1995. Four variants of this program are then presented that improve the efficiency by avoiding redundant computation, vectorizing, and accessing subarrays via MATLAB's colon notation. We then consider reformulating the problem via a binomial coefficient expansion. Here, a straightforward implementation is seen to be improved by vectorizing, avoiding overflow and underflow, and exploiting sparsity. Overall, the fastest of the binomial method programs has an execution time that is within a factor 2 of direct evaluation of the Black–Scholes formula. One of the vectorized versions is then used as the basis for a program that values an American put option. The programs show how execution times in MATLAB can be dramatically reduced by using high-level operations on arrays rather than computing with individual components, a principle that applies in many scientific computing environments. The relevant files are downloadable from the World Wide Web.

Key words. algorithm, American option, Black–Scholes, European option, optimization, overflow, underflow, vectorization

AMS subject classifications. 65-01, 65Y20

PII. S0036144501393266

1. Introduction.

This paper is about the efficient implementation of algorithms. We use a case study based around the binomial method for valuing financial options. This gives a compact framework in which to illustrate that (a) computations can often be organized in a number of ways and (b) the overall execution time of a program may depend crucially on the choices made.

Our example programs use the MATLAB language. In describing these programs we assume that the reader has some familiarity with scientific computing, and we focus on explaining features specific to MATLAB. For the sake of brevity our description of MATLAB commands is highly selective, and we refer the reader to [4, 7] or MATLAB's on-line help facilities for further details. The programs can be downloaded from

http://www.maths.strath.ac.uk/~aas96106/algfiles.html

*Received by the editors August 1, 2001; accepted for publication (in revised form) May 28, 2002; published electronically October 30, 2002.
http://www.siam.org/journals/sirev/44-4/39326.html

[†]Department of Mathematics, University of Strathclyde, Glasgow, G1 1XH, UK (djh@maths.strath.ac.uk).

Timings quoted were obtained by averaging over 1000 runs with MATLAB Version 6 (R12) on an IBM H70 server. Comparable results were obtained on two other machines.

Two of the techniques that we illustrate, removing redundant computation and reformulating the mathematical problem, are generally applicable. Others are closely tied to the nature of MATLAB. Indeed, some of the MATLAB-specific tinkering that goes on in sections 3 and 4 might appear a little mysterious to those students who are not familiar with MATLAB's "interpretive" nature. For this reason, we pause here to outline some of the principles that we use.

The bulk of scientific computing is done by programs written in a high-level language and compiled into machine code. Compilers generally proceed through four stages:

1. Parsing and semantic analysis.
2. Intermediate code generation.
3. Optimization of the intermediate code.
4. Generation of machine code from the optimized intermediate code.

Although optimization is mentioned only once in this list, it actually occurs twice. The kind of optimization in item 3 consists of improvements that can be made without reference to a particular computer. For example, at this stage code like

```
for i=1 to n
   a(i) = sqrt(2)*a(i)
end for
```

becomes

```
temp = sqrt(2)
for i=1 to n
   a(i) = temp*a(i)
end for
```

But optimization also occurs at item 4. It is here that registers are allocated to minimize data movement, instructions are sequenced to fill the cpu pipeline, and other machine-dependent decisions are taken.

With an interpreted language, code never reaches machine level but is read (interpreted) and executed by another program. There are two extremes. In the simplest case, a code might be read a line at a time, parsed, and executed, without reaching the first stage of compilation. At the other extreme are interpreted languages that go through all the stages of compilation but the last. The most prominent example is Java, which "compiles" its programs into byte stream code for the Java virtual machine and uses an interpreter to execute the code on the machine in question. In Java, you don't have to worry about moving things out of loops, but you don't get the benefits of code optimized for a particular machine.

MATLAB lies somewhere between these two extremes. It generates an intermediate code, which is then interpreted, but it does no optimizing. Constant expressions are left inside loops, for example. However, the functions MATLAB calls to perform its higher level operations are fully compiled code designed to run efficiently on the machine in question. Thus, hand-optimizing MATLAB code reduces to

(i) performing compiler-type optimizations, such as taking repeated computations outside a loop, and

(ii) maximizing the use of MATLAB's built-in functions, especially those that embrace a lot of calculation by operating directly on vectors and matrices (a process in which `for` loops are typically subjected to close scrutiny).

The case study proceeds as follows. In the next section we describe the binomial method for option pricing. Sections 3 and 4 form the heart of the paper. Section 3 begins with a MATLAB implementation of the pseudocode from [9, Figure 10.4], which is then improved in four separate steps. A reformulation of the problem is used in section 4 to develop alternative implementations. This approach is ultimately seen to be superior, once overflow errors have been suppressed. For comparison, section 5 gives a code that evaluates directly the Black–Scholes formula (which the binomial method approximates). American options are briefly considered in section 6. Section 7 gives some conclusions.

2. The Binomial Method.

We give only a very brief discussion of the binomial method and refer the reader to the texts [5, 6, 8, 9] or the original source [1] for further background details. We use the notation of [9, Chapter 10] with the exception that indices start from 1 rather than 0.

A European put option gives its holder the right, but not the obligation, to sell a prescribed asset for a prescribed price at some prescribed time in the future. Under certain assumptions the Black–Scholes partial differential equation (PDE) can be used to determine the value of the option, that is, a fair amount for the holder to pay for the privilege of holding it. The binomial method is a simple numerical technique for approximating the option value that may be regarded as a finite-difference approximation to the Black–Scholes PDE. Although a full justification for the binomial method requires material from financial modeling, mathematical analysis, and stochastic processes, the method itself is simple to describe.

We suppose that the option is taken out at time $t = 0$ and that the holder may exercise the option, that is, sell the asset, at a time T called the expiry time. The binomial method models the asset price only at discrete time points $t = i\delta t$, for $0 \le i \le M$, where $T = M\delta t$. A key assumption is that between successive time levels the asset price either moves up by a factor $u > 1$ or moves down by a factor $d < 1$. An upward movement occurs with probability p and a downward movement occurs with probability $1 - p$. The initial asset price, S, is known. Hence, at time $t = \delta t$ the possible asset prices are uS and dS. Similarly, at time $t = 2\delta t$ the possible asset prices are u^2S, udS, and d^2S. (The price udS may arise from an upward movement followed by a downward movement or from a downward movement followed by an upward movement.) In general, at time $t = t_i := (i-1)\delta t$ there are i possible asset prices, which we label

$$S_n^i = d^{i-n}u^{n-1}S, \quad 1 \le n \le i. \tag{2.1}$$

Hence, at the expiry time $t = t_{M+1} = T$ there are $M + 1$ possible asset prices. The values S_n^i for $1 \le n \le i$ and $1 \le i \le M + 1$ form a recombining binary tree, as illustrated in Figure 2.1.

Let E be the exercise price of the option. This means that the holder of the put may exercise the option by selling the asset at price E at time T. Clearly, the holder will exercise whenever there is a profit to be made, that is, whenever the actual price of the asset turns out to be less than E. Hence, if the asset has price S_n^{M+1} at time $t = t_{M+1} = T$, then the value of the option at that time is $\max(E - S_n^{M+1}, 0)$. Generally, we let V_n^i denote the value of the option at time $t = t_i$ corresponding to asset price S_n^i. We thus know that

$$V_n^{M+1} = \max(E - S_n^{M+1}, 0). \tag{2.2}$$

Our task is to find V_1^1, the option value at time zero. A financial modeling argument says that we can obtain the value V_n^i by taking a weighted average of the values V_n^{i+1}

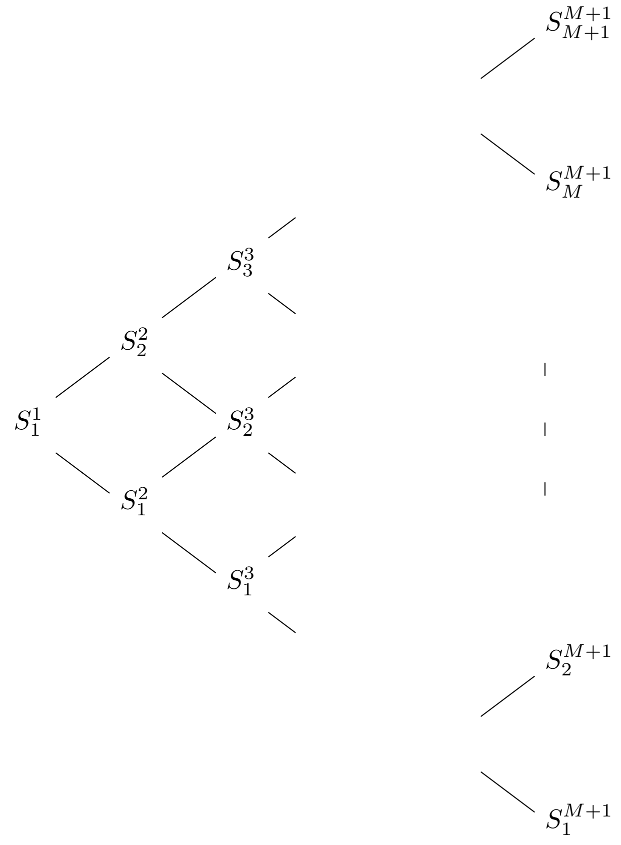

Fig. 2.1 *Recombining binary tree of asset prices.*

and V_{n+1}^{i+1} from the later time level. The formula is

$$V_n^i = e^{-r\delta t}\left(pV_{n+1}^{i+1} + (1-p)V_n^{i+1}\right), \quad 1 \le n \le i, \qquad 1 \le i \le M, \tag{2.3}$$

where the constant r represents a risk-free interest rate (arising, for example, from a deposit in a sound bank) and we recall that p is the probability of an upward movement in asset price.

Once the parameters u, d, p, and M have been chosen, the formulas (2.1)–(2.3) completely specify the binomial method. The recurrence (2.1) shows how to insert the asset prices in the binomial tree. Having obtained the asset prices at time $t = t_{M+1} = T$, (2.2) gives the corresponding option values at that time. The relation (2.3) may then be used to step backwards through the tree until V_1^1, the option value at time $t = t_1 = 0$, is computed.

To make the binomial method compatible with the assumptions that underlie the Black–Scholes PDE, the first and second moments of the random walk represented by the tree in Figure 2.1 must be constrained. This leads to two equations in the three parameters u, d, and p. We will use the values

$$u = A + \sqrt{A^2 - 1}, \quad d = \frac{1}{u}, \quad p = \frac{e^{r\delta t} - d}{u - d}, \tag{2.4}$$

where

$$A = \tfrac{1}{2}\left(e^{-r\delta t} + e^{(r+\sigma^2)\delta t}\right)$$

and the constant σ represents the volatility of the asset. See [9, Chapter 10] or [6] for details of the derivation of these values. We note that some authors prefer other parameter values, where $d \neq 1/u$. Hence, we chose not to exploit the identity $d = 1/u$ in our codes.

3. MATLAB Programs that Traverse the Tree. The first program, `euro1.m`, is shown in Listing 3.1. This is essentially a translation of the pseudocode from [9, Figure 10.4]. We have specified `S = 5` for the asset starting price, `E = 10` for the exercise price, `T = 1` for the expiry time, `r = 0.06` for the risk-free interest rate, and `sigma = 0.3` for the volatility. We take `M = 256` discrete time-steps and use (2.4) to evaluate d, u, and p. Following [9] we save storage space by using a single one-dimensional array to store the asset prices, moving from left to right through the tree, and then overwrite with the option values, moving from right to left through the tree. The line `W = zeros(M+1,1)` initializes `W` to an `M+1`-by-`1` array of zeros. We then set `W(1) = S` and use a nested pair of `for` loops to traverse the tree, forming the values in (2.1). Note that the syntax `for n = i:-1:1` sets up a `for` loop in which `n` takes values from `i` down to `1` in steps of `1`. On exiting the loops, `W(n)` stores the asset price S_n^{M+1} for $1 \le n \le M+1$. The next `for` loop overwrites each S_n^{M+1} value with the corresponding option value V_n^{M+1} given by (2.2). Finally, a nested pair of `for` loops is used to proceed from right to left through the tree, using the formula (2.3).

The result, `W(1) = 4.4304`, agrees with the computation in [9, Figure 10.5]. Execution time was around 1.9 seconds, as shown in Figure 3.1.

In `euro1.m`, the asset prices at time T are computed by passing through all time levels using a nested pair of `for` loops. This requires $O(M^2)$ operations. A more efficient, $O(M)$, approach is to compute $\{S_n^{M+1}\}_{n=1}^{M+1}$ directly from (2.1). In MATLAB this can be done with

```
for n = 1:M+1
   W(n) = S*(d^(M+1-n))*(u^(n-1));
end
```

The program `euro2.m` in Listing 3.2 incorporates this change. We see from Figure 3.1 that execution time has been improved by a factor of around 0.7.

Our next enhancement involves eliminating the first two `for` loops in `euro2.m` in favor of elementwise operations on vectors. Since this facility is not common to all scientific computing languages, we will explain how it works in some detail. The syntax `[m1:m2]` in MATLAB creates a row of elements from `m1` to `m2`, so

```
>> [1:5]
ans =
     1     2     3     4     5
```

(Here the text after the prompt `>>` is the input command, and `ans` is MATLAB's answer. Blank lines have been omitted to save space.) More generally, `[m1:step:m2]` creates an array with elements from `m1` to `m2` with a spacing of `step`, so

```
>> [1:2:5]
ans =
     1     3     5
```

```
% EURO1  Binomial method for a European put.
%
% Unvectorized version.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

W = zeros(M+1,1);
W(1) = S;
% Asset prices at time T
for i = 1:M
    for n = i:-1:1
        W(n+1) = u*W(n);
    end
    W(1) = d*W(1);
end

% Option values at time T
for n = 1:M+1
    W(n) = max(E-W(n),0);
end

% Re-trace to get option value at time zero
for i = M:-1:1
    for n = 1:i
        W(n) = exp(-r*dt)*(p*W(n+1) + (1-p)*W(n));
    end
end

disp('Option value is'), disp(W(1))
```

Listing 3.1 *M-file euro1.m.*

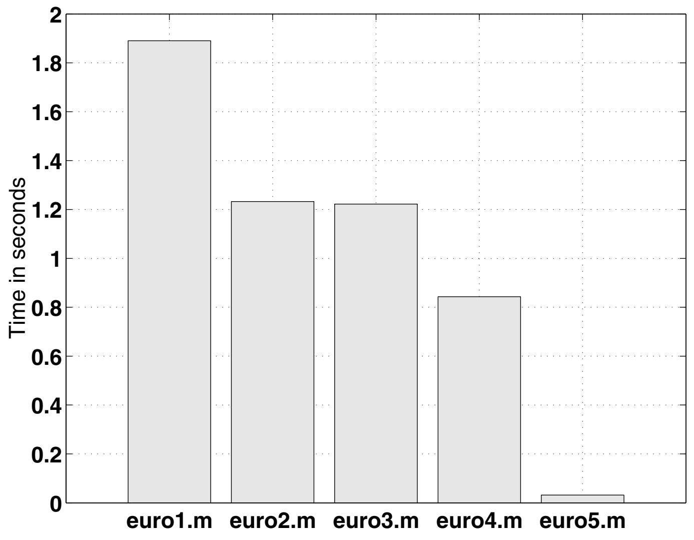

Fig. 3.1 *Execution times.*

```
% EURO2  Binomial method for a European put.
%
% Initial tree computation improved.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

W = zeros(M+1,1);
% Asset prices at time T
for n = 1:M+1
    W(n) = S*(d^(M+1-n))*(u^(n-1));
end

% Option values at time T
for n = 1:M+1
    W(n) = max(E-W(n),0);
end

% Re-trace to get option value at time zero
for i = M:-1:1
    for n = 1:i
        W(n) = exp(-r*dt)*(p*W(n+1) + (1-p)*W(n));
    end
end

disp('Option value is'), disp(W(1))
```

Listing 3.2 *M-file* `euro2.m`.

```
>> [7:-2:3]
ans =
      7     5     3
```

The symbol `'` performs the transpose operation, so

```
>> [7:-2:3]'
ans =
      7
      5
      3
```

If a period precedes the `*`, `/`, or `^` operators, then the operation is performed in an elementwise sense. For example,

```
>> [1 2 3].*[6 8 9]
ans =
      6    16    27

>> 2.^[1 2 3]
ans =
      2     4     8
```

Using this powerful syntax, the first `for` loop in `euro2.m` can be replaced by

```
W = S*d.^([M:-1:0]').*u.^([0:M]');
```

```
% EURO3  Binomial method for a European put.
%
% Partially vectorized version.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Option values at time T
W = max(E-S*d.^([M:-1:0]').*u.^([0:M]'),0);

% Re-trace to get option value at time zero
for i = M:-1:1
    for n = 1:i
        W(n) = exp(-r*dt)*(p*W(n+1) + (1-p)*W(n));
    end
end

disp('Option value is'), disp(W(1))
```

Listing 3.3 *M-file euro3.m.*

To streamline the option valuation at time T, we note that MATLAB allows scalars and arrays to be combined linearly, so

```
>> 2 - [1 2 3]
ans =
     1     0    -1
```

and the `max` function is happy to accept arrays as arguments, so

```
>> x = [-2 5 3]'
x =
    -2
     5
     3

>> max(x,0)
ans =
     0
     5
     3
```

It follows that

```
    W = max(E-S*d.^([M:-1:0]').*u.^([0:M]'),0);
```

does the job of computing $\{V_n^{M+1}\}_{n=1}^{M+1}$. This is used in `euro3.m`; see Listing 3.3. Although trading those `for` loops for one vectorized computation has shortened the code, Figure 3.1 shows that the execution time is virtually unchanged. The $O(M^2)$ work in the final nested loop is dominant.

We can speed up that remaining nested loop by removing unnecessary computations. The $e^{-r\delta t}$ scaling on each step can be deferred until the end, where it accumulates to a single e^{-rT} factor. This is done in `euro4.m`, given in Listing 3.4, where we

```
% EURO4  Binomial method for a European put.
%
% Partially vectorized version.
% Redundant scaling removed.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Option values at time T
W = max(E-S*d.^([M:-1:0]').*u.^([0:M]'),0);

% Re-trace to get option value at time zero
q = 1-p;
for i = M:-1:1
    for n = 1:i
        W(n) = p*W(n+1) + q*W(n);
    end
end
W(1) = exp(-r*T)*W(1);

disp('Option value is'), disp(W(1))
```

Listing 3.4 *M-file euro4.m.*

```
% EURO5  Binomial method for a European put.
%
% Vectorized version, uses shifts via colon notation.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Option values at time T
W = max(E-S*d.^([M:-1:0]').*u.^([0:M]'),0);

% Re-trace to get option value at time zero
q = 1-p;
for i = M:-1:1
    W = p*W(2:i+1) + q*W(1:i);
end
W = exp(-r*T)*W;

disp('Option value is'), disp(W)
```

Listing 3.5 *M-file euro5.m.*

also precompute `1-p` as `q`. Figure 3.1 indicates that the run time has been reduced by a factor of around two-thirds.

In `euro5.m`, shown in Listing 3.5, we introduce a more significant change. Here, we remove a `for` loop from the final option valuation phase by working directly with arrays rather than scalars. In the formula (2.3), the operation of computing $\{V_n^i\}_{n=1}^{i}$

from $\{V_n^{i+1}\}_{n=1}^{i+1}$ can be written as

$$\begin{bmatrix} V_1^i \\ V_2^i \\ \vdots \\ \vdots \\ V_i^i \end{bmatrix} = e^{-r\delta t} \left(p \begin{bmatrix} V_2^{i+1} \\ V_3^{i+1} \\ \vdots \\ \vdots \\ V_{i+1}^{i+1} \end{bmatrix} + (1-p) \begin{bmatrix} V_1^{i+1} \\ V_2^{i+1} \\ \vdots \\ \vdots \\ V_i^{i+1} \end{bmatrix} \right), \; i = M, M-1, \ldots, 1. \tag{3.1}$$

This relation can be implemented using MATLAB's colon notation to access subvectors. If `x` is a `10`-by-`1` array, then `x(4:6)` is a `3`-by-`1` array consisting of `x(4)`, `x(5)`, and `x(6)`, and `x(5:10)` is a `6`-by-`1` array consisting of `x(5)`, `x(6)`, ..., `x(10)`. Using this syntax, and postponing the $e^{-r\delta t}$ scaling until the end of the loop, the formula (3.1) may be expressed as

```
for i = M:-1:1
    W = p*W(2:i+1) + (1-p)*W(1:i);
end
W = exp(-r*T)*W;
```

Note that each time around the loop the length of `W` decreases by 1; on exit `W` is 1-by-1. As indicated in Figure 3.1, vectorizing the $O(M^2)$ bottleneck has dramatically improved the execution time: the speedup factor over `euro4.m` is around 25.

4. MATLAB Programs that Use a Binomial Expansion.

Returning to the recurrence (2.3) we see that for $M = 1$

$$V_1^1 = e^{-r\delta t}\left(pV_2^2 + (1-p)V_1^2\right),$$

and for $M = 2$

$$\begin{aligned} V_1^1 &= e^{-r\delta t}\left(pV_2^2 + (1-p)V_1^2\right) \\ &= e^{-r\delta t}\left(pe^{-r\delta t}(pV_3^3 + (1-p)V_2^3) + (1-p)e^{-r\delta t}(pV_2^3 + (1-p)V_1^3)\right) \\ &= e^{-2r\delta t}\left(p^2V_3^3 + 2p(1-p)V_2^3 + (1-p)^2V_1^3\right). \end{aligned}$$

Similarly, for $M = 3$ we find that

$$V_1^1 = e^{-3r\delta t}\left(p^3V_4^4 + 3p^2(1-p)V_3^4 + 3p(1-p)^2V_2^4 + (1-p)^3V_1^4\right).$$

The coefficients $\{1,1\}$, $\{1,2,1\}$, $\{1,3,3,1\}$ are familiar from Pascal's triangle, and having spotted this connection it is straightforward to deal with the general case.

LEMMA 4.1. *In the recurrence* (2.3)

$$V_1^1 = e^{-rT} \sum_{k=1}^{M+1} \mathrm{C}_{k-1}^{M} p^{k-1}(1-p)^{M+1-k} V_k^{M+1}, \tag{4.1}$$

where C_k^N *denotes the binomial coefficient,* $\mathrm{C}_k^N := (N!)/(k!(N-k)!)$.

Proof. This result was presented in the original binomial method article [1]; see also [6]. It can be shown by the same inductive argument that is traditionally used to establish the binomial theorem: $(a+b)^N = \sum_{r=0}^{N} \mathrm{C}_r^N a^r b^{N-r}$. ☐

We see from (4.1) that computing V_1^1 from $\{V_n^{M+1}\}_{n=1}^{M+1}$ can be reduced to the formation of a single sum. This can be done with $O(M)$ floating point operations, in

```
% EURO6  Binomial method for a European put.
%
% Uses explicit solution based on binomial expansion.
% Unvectorized, subject to overflow.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Option values at time T
W = max(E-S*d.^([M:-1:0]').*u.^([0:M]'),0);

% Binomial coefficients
v = zeros(M+1,1);
for k = 1:M+1
    v(k) = nchoosek(M,k-1);
end

value  = exp(-r*T)*sum(v.*(p.^([0:M]')).*((1-p).^([M:-1:0]')).*W)';
disp('Option value is'), disp(value)
```

Listing 4.1 *M-file euro6.m.*

contrast to the $O(M^2)$ algorithms of section 3. We now investigate whether (4.1) can be exploited in practice.

The binomial coefficient C_k^N can be evaluated in MATLAB using `nchoosek(N,k)`. The program `euro6.m` in Listing 4.1 uses this function inside a `for` loop to set up a vector `v` with ith component C_{i-1}^M and then forms the appropriate sum. However, there are two senses in which `euro6.m` is unsatisfactory. First, the `nchoosek(N,k)` function is called inside a `for` loop, so that the vector `v` is constructed element by element rather than in one fell swoop. Second, although the V_1^1 in (4.1) that we compute is of size `4.4304`, the individual binomial coefficients C_k^N take on some extremely large values: the biggest element of `v` is

```
>>  max(v)
ans = 5.7687e+75
```

It follows that rounding errors may arise as a result of significant digits being lost; see [4, Chapter 4] for details of MATLAB's floating point arithmetic. Indeed, running `euro6.m` prompts the function `nchoosek(N,k)` to warn repeatedly that "`Result may not be exact.`" In this example, the results from `euro5.m` and `euro6.m` agree to within 10^{-15}, but if `M` were increased, rounding errors would contaminate the answer and eventually the binomial coefficients would become too big to store. Execution time for `euro6.m` is roughly 0.9 seconds—around 20 times that for `euro5.m` (although MATLAB may be using a large chunk of that 0.9 seconds to write warning messages to the screen).

We can vectorize the computation of `v` in `euro6.m` via the cumulative product function, `cumprod`. If `x` is an `n`-by-`1` array then `cumprod(x)` is an `n`-by-`1` array whose ith entry is the product of the first i elements of `x`. It follows that the vector of binomial coefficients, `v`, may be constructed as

```
>> cp = cumprod([1;[1:M]']);
>> v = cp(M+1)./(cp.*cp(M+1:-1:1));
```

However, this approach generates intermediate numbers as large as $M!$. For the value $M = 256$ used in our examples, `v` becomes a vector of `NaN`s, where `NaN` is short for `Not a Number`. This happens because the factorial of M is too big for MATLAB to store. An alternative is to note that the binomial coefficients satisfy the recurrence

$$C^M_{k-1} = \left(\frac{M-k+2}{k-1}\right) C^M_{k-2}, \qquad k \geq 3, \tag{4.2}$$

and hence `v` may be generated from

```
>> v = cumprod([1;[M:-1:1]'./[1:M]']);
```

Vectorizing `euro6.m` in this way produces a fast code—more than twice as quick as `euro5.m`. However, this does not overcome the problem of large C^N_k that was inherent in `euro6.m`, and tests show that `value` degenerates to `NaN` at $M = 1030$.

Returning to (4.1), we see that the "raw" binomial coefficients become scaled by powers of p and $1-p$. This suggests that working directly with the scaled values may alleviate the effect of the large C^N_k values. If we let $a_k := C^M_{k-1}p^{k-1}(1-p)^{M+1-k}$, so that the sum in (4.1) becomes $\sum_{k=1}^{M+1} a_k V_k^{M+1}$, then it is straightforward to extend the binomial coefficient recurrence (4.2) to

$$a_k = \left(\frac{p}{1-p}\right)\left(\frac{M-k+2}{k-1}\right) a_{k-1}, \qquad k \geq 2,$$

with $a_1 = (1-p)^M$. It follows that the vector of a_k values may be computed as

```
>> a = cumprod([(1-p)^M,(p/(1-p))*[M:-1:1]./[1:M]]);
```

The program `euro7.m` in Listing 4.2 uses this approach. The largest element of `a` is

```
>> max(a)
ans = 0.0498
```

However, the price we pay for this can be seen from the *smallest* element of `a`

```
>> min(a)
ans = 3.8805e-78
```

Although `euro7.m` avoids big intermediate quantities, it does so by computing very small ones. The first element of `a` is $(1-p)^M$, and if this drops below MATLAB's `realmin*eps` threshold of $\approx 4.9 \times 10^{-324}$, then it underflows to zero; see [4, p. 36]. Underflow, unlike its evil twin overflow, is often harmless, but in our computation $(1-p)^M$ is a multiplicative factor in each element of `a` and hence replacing this small but nonzero quantity by zero is disastrous. A simple expansion shows that $p = \frac{1}{2} + O(1/\sqrt{M})$ as $M \to \infty$, and hence $(1-p)^M$ looks like 2^{-M} for large M. It follows that `euro7.m` will return the spurious answer `value = 0` for M bigger than around $-\log(4.9 \times 10^{-324})/\log(2) \approx 1074$. (In practice it happens at $M = 1073$.)

To get around the overflow/underflow traps in `euro6.m` and `euro7.m`, we may compute with logarithms. We know that

$$\begin{aligned}
\log\left(\mathrm{C}^M_{k-1}p^{k-1}(1-p)^{M+1-k}\right) &= \log M! - \log(k-1)! - \log(M+1-k)! \\
&\quad + \log p^{k-1} + \log(1-p)^{M+1-k} \\
&= \sum_{i=1}^{M} \log(i) - \sum_{i=1}^{k-1} \log(i) - \sum_{i=1}^{M-k+1} \log(i) \\
&\quad + (k-1)\log p + (M+1-k)\log(1-p).
\end{aligned}$$

```
% EURO7  Binomial method for a European put.
%
% Uses explicit solution based on binomial expansion.
% Vectorized, subject to underflow.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Option values at time T
W = max(E-S*d.^([M:-1:0]').*u.^([0:M]'),0);

% Recursion for coefficients
a = cumprod([(1-p)^M,(p/(1-p))*[M:-1:1]./[1:M]]);
value = exp(-r*T)*sum(a'.*W);

disp('Option value is'), disp(value)
```

Listing 4.2 *M-file* `euro7.m`*.*

```
% EURO8  Binomial method for a European put.
%
% Uses explicit solution based on binomial expansion.
% Vectorized and based on logs to avoid overflow.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Option values at time T
W = max(E-S*d.^([M:-1:0]').*u.^([0:M]'),0);

% log/cumsum version
csl = cumsum(log([1;[1:M]']));
tmp = csl(M+1) - csl - csl(M+1:-1:1) + log(p)*([0:M]') + log(1-p)*([M:-1:0]');
value = exp(-r*T)*sum(exp(tmp).*W);

disp('Option value is'), disp(value)
```

Listing 4.3 *M-file* `euro8.m`*.*

The array `tmp` in `euro8.m`, shown in Listing 4.3, stores this quantity for $1 \le k \le M+1$. Note that `cumsum` is the summation analogue of the `cumprod` function that was used in `euro7.m`. It is perhaps worth looking back at our original code, `euro1.m`. Once the parameters have been set up, `euro1.m` computes the option value by using 16 lines of executable code with 5 `for` loops, whereas `euro8.m` does the same job in 4 lines of code without recourse to a `for` loop. Figure 4.1 shows the execution time for `euro8.m`. In the same plot we have repeated the timing for `euro5.m`, the fastest code from the previous section. We see that `euro8.m` outperforms `euro5.m` by a factor of around 12.

It is possible to trim the execution time of `euro8.m` by noticing that, as mentioned in [1], certain terms in the sum (4.1) are predictably zero. To see this, observe from (2.1) that the values $\{S_n^{M+1}\}_{n=1}^{M+1}$ are ordered: $S_1^{M+1} < S_2^{M+1} < S_3^{M+1} < \cdots < S_{M+1}^{M+1}$. In general, the exercise price E will lie somewhere between S_1^{M+1} and S_{M+1}^{M+1}

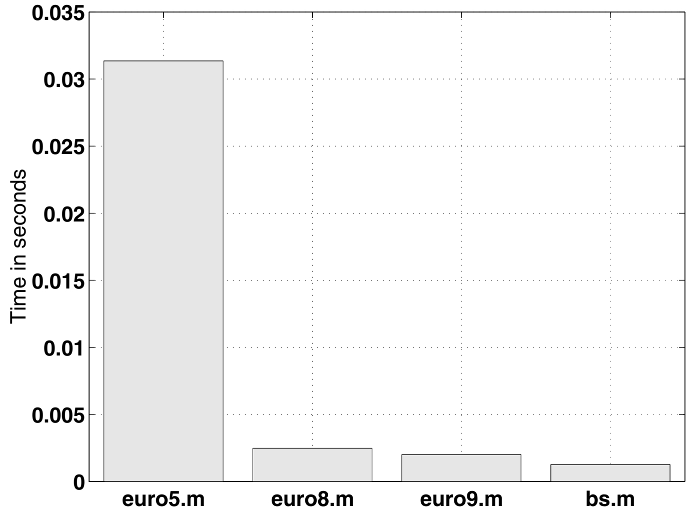

Fig. 4.1 *Execution times.*

and hence, from (2.2), there will be some index z such that $V_1^{M+1} > V_2^{M+1} > \cdots > V_z^{M+1} > V_{z+1}^{M+1} = V_{z+2}^{M+1} = \cdots = V_{M+1}^{M+1} = 0$. It follows that the terms from $k = z+1$ to $k = M+1$ in the summation (4.1) need not be computed. A little calculation shows that the cut-off index z is the largest nonnegative integer such that

$$z < \frac{\log\left(\frac{Eu}{Sd^{M+1}}\right)}{\log\left(\frac{u}{d}\right)} \quad \text{and} \quad z \le M+1.$$

The program `euro9.m` in Listing 4.4 begins by computing `z`. Here, `floor` rounds down to the nearest integer, and for convenience we force `z` to be at least as big as 1. Having found `z`, we then adapt the computations in `euro8.m` so that only vectors of length `z` are used. We see from Figure 4.1 that `euro9.m` is faster than `euro8.m`—the improvement is around 20%. In general, the speedup will depend on the percentage of zeros in $\{V_n^{M+1}\}_{n=1}^{M+1}$, which in turn depends strongly on the values of S and E. In our example `z = 147`, so around 40% of the array `W` in `euro8.m` is made up of zeros.

5. A MATLAB Program that Uses Black–Scholes.

The simple European option that we are studying can be valued via the Black–Scholes PDE, for which an analytical solution is available [5, 6, 8, 9]. The value of the put option is given by

$$Ee^{-rT}\mathrm{N}(-d_2) - S\mathrm{N}(-d_1), \tag{5.1}$$

where

$$d_1 = \frac{\log(S/E) + (r + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T},$$

and $\mathrm{N}(x)$ is the cumulative $\mathrm{N}(0,1)$ distribution function, so

$$\mathrm{N}(x) := \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-s^2/2}\,ds.$$

```
% EURO9  Binomial method for a European put.
%
% Uses explicit solution based on binomial expansion.
% Vectorized, based on logs to avoid overflow,
% and avoids computing with zeros.

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% cut-off index
z = max(1,min(M+1,floor( log((E*u)/(S*d^(M+1)))/(log(u/d)) )));

% Option values at time T
W = E - S*d.^([M:-1:M-z+1]').*u.^([0:z-1]');

% log/cumsum version using cut-off index z
tmp1 = cumsum(log([1;[M:-1:M-z+2]'])) - cumsum(log([1;[1:z-1]']));
tmp2 = tmp1 + log(p)*([0:z-1]') + log(1-p)*([M:-1:M-z+1]');

value = exp(-r*T)*sum(exp(tmp2).*W);

disp('Option value is'), disp(value)
```

Listing 4.4 *M-file* `euro9.m`.

```
% BS  Black-Scholes European put price.
%

%%%%%%%%%% Problem parameters %%%%%%%%%%%%%%%
S = 5; E = 10; T = 1; r = 0.06; sigma = 0.3;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

d1 = (log(S/E) + (r + 0.5*sigma^2)*T)/(sigma*sqrt(T));
d2 = d1 - sigma*sqrt(T);
N1 = 0.5*(1+erf(-d1/sqrt(2)));
N2 = 0.5*(1+erf(-d2/sqrt(2)));

value = E*exp(-r*T)*N2 - S*N1;

disp('Option value is'), disp(value)
```

Listing 5.1 *M-file* `bs.m`.

MATLAB has a built-in function `erf` that evaluates the error function

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-s^2}\, ds.$$

This can be used to evaluate the quantities $\mathrm{N}(-d_1)$ and $\mathrm{N}(-d_2)$ in (5.1) via the relation $\mathrm{N}(x) = \frac{1}{2}(1 + \mathrm{erf}(x/\sqrt{2}))$. The program `bs.m` in Listing 5.1 computes the value of the option considered in the previous two sections using the formula (5.1). The answer, `4.4305`, was computed in around 0.0013 seconds. Note that `euro9.m` gets within a factor 1.6 of this time.

6. Valuing an American Option. The binomial method approach can be used to value many more exotic classes of option than the basic European, including cases

```
% AMERICAN  Binomial method for an American put.
%
% Vectorized version, based on euro5.m

%%%%%%%%%% Problem and method parameters %%%%%%%%%%%
S = 9; E = 10; T = 1; r = 0.06; sigma = 0.3; M = 256;
dt = T/M; A = 0.5*(exp(-r*dt)+exp((r+sigma^2)*dt));
u = A + sqrt(A^2-1); d = 1/u; p = (exp(r*dt)-d)/(u-d);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Re-usable computations
dpowers = d.^([M:-1:0]');
upowers = u.^([0:M]');
scale1 = p*exp(-r*dt);
scale2 = (1-p)*exp(-r*dt);

% Option values at time T
W = max(E-S*dpowers.*upowers,0);

% Re-trace to get option value at time zero
for i = M:-1:1
    Si = S*dpowers(M-i+2:M+1).*upowers(1:i);
    W = max(max(E-Si,0),scale1*W(2:i+1) + scale2*W(1:i));
end

disp('Option value is'), disp(W)
```

Listing 6.1 *M-file* `american.m`*.*

where an analytical solution is not available; see, for example, [6, 9]. In particular, the method is easily modified to value American options. An American put option differs from the European version in that the holder may elect to exercise at any time up to and including the expiry time. For the binomial method, this has the effect of changing (2.3) to

$$V_n^i = \max\left(\max\left(E - S_n^i, 0\right), e^{-r\delta t}\left(pV_{n+1}^{i+1} + (1-p)V_n^{i+1}\right)\right), \quad 1 \le n \le i, \quad 1 \le i \le M. \tag{6.1}$$

In this case, the binomial expansion in Lemma 4.1 can no longer be used as a starting point, and hence the codes in section 4 are not relevant. Some of the tricks that were introduced in section 3 may still be used, though, and we will base a code around `euro5.m`.

The program `american.m` in Listing 6.1 implements an American put with `S = 9` and all other parameters as used in the previous sections. We use the colon notation to access subvectors, as in `euro5.m`. As we trace backwards through the tree, an array of S_n^i values is formed at each time level. Hence, the storage requirements remain linear in M, rather than quadratic as in the pseudocode algorithm of [9, Figure 10.7]. The answer, `W = 1.4349`, agrees with that in [9, Figure 10.8]. Execution time was around 0.1 seconds.

7. Conclusion.

We have presented a case study based on the binomial method to show how reorganizing a high-level computer code can affect its efficiency. We implemented the binomial method in the MATLAB language, which is particularly suited to the type of linear algebra–based computations that were required. (We mention that only pure MATLAB code was considered. Two other approaches to speeding up MATLAB programs are to compile them with the MATLAB compiler or to rewrite them in FORTRAN or C and link them to MATLAB as MEX files. A third option that should become available in MATLAB Version 6.5 (R13) for certain chips is to use the new Just-in-Time (JIT) compiler.)

Although it is not always easy to predict the effect of reorganizing a computation in MATLAB, a useful general rule for improving execution speed that is consistent with the examples above is:

> Eliminate redundant computations, eschew `for` loops, avoid individual array elements and instead work directly on entire arrays or subarrays. This can be done by (a) supplying array-valued arguments to MATLAB functions, (b) applying elementwise operations such as `.*` and `.^` to arrays, and (c) exploiting the colon notation to access subarrays.

Other examples where MATLAB code has been optimized under this philosophy can be found in [4, Chapter 20]. More generally, the principle of thinking about how data is stored and accessed in a computational algorithm is of value in almost any scientific computing environment. Modern, high-performance computers work best with vector and matrix-based operations; good references for this general area are [2, 3].

Acknowledgments. I thank Paul Fackler, Nick Higham, Cleve Moler, Ser-Huang Poon, and Hans van der Weide for suggesting improvements to earlier versions of this document. *SIAM Review* editor Pete Stewart provided valuable feedback and, in particular, information for the summary of compiled versus interpreted code that appears in section 1. Substantial revisions to the manuscript were made while I enjoyed the hospitality of The Fields Institute for Research in Mathematical Sciences, Toronto, a visit which was supported by a research fellowship from the Leverhulme Trust.

REFERENCES

[1] J. C. Cox, S. A. Ross, and M. Rubinstein, *Option pricing: A simplified approach*, J. Financial Economics, 7 (1979), pp. 229–263.

[2] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Numerical Linear Algebra for High-Perfomance Computers*, SIAM, Philadelphia, 1998.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, London, 1996.

[4] D. J. Higham and N. J. Higham, *MATLAB Guide*, SIAM, Philadelphia, 2000.

[5] J. C. Hull, *Options, Futures, & Other Derivatives*, 4th ed., Prentice–Hall, Englewood Cliffs, NJ, 2000.

[6] Y. K. Kwok, *Mathematical Models of Financial Derivatives*, Springer-Verlag, Berlin, 1998.

[7] *Using MATLAB*, The MathWorks, Inc., Natick, MA. Online version.

[8] S. M. Ross, *An Introduction to Mathematical Finance*, Cambridge University Press, Cambridge, UK, 1999.

[9] P. Wilmott, S. Howison, and J. Dewynne, *The Mathematics of Financial Derivatives*, Cambridge University Press, Cambridge, UK, 1995.