Discussion Paper 66

Institute for Empirical Macroeconomics
Federal Reserve Bank of Minneapolis
250 Marquette Avenue
Minneapolis, Minnesota 55480-0291

June 1992

SOLVING NONLINEAR DYNAMIC MODELS ON PARALLEL COMPUTERS

Wilbur John Coleman II*

Board of Governors of the Federal Reserve System

ABSTRACT

This paper describes an algorithm that takes advantage of parallel computing to solve discrete-time recursive systems that have an endogenous state variable.

*I wish to thank Raymond Board, Larry Christiano, Jon Faust, Christian Gilles, William Helkie, and Peter Tinsley for helpful comments, and the Institute for Empirical Macroeconomics for the use of their facilities. This paper should not be interpreted as reflecting the views of the Board of Governors of the Federal Reserve System or its staff.

Introduction

The last decade has witnessed a remarkable increase in the economic profession's interest in solving complex dynamic general equilibrium models. Of particular note is the recent interest in studying models that contain some sort of heterogeneity: even the simplest of these models can become very complex very fast (see, e.g., Lucas (1980), Scheinkman and Weiss (1986), Aiyagari and Gertler (1990), Imrohoroglu and Prescott (1991), and Coleman (1992)). Recent efforts devoted to developing the technology to solve these models has concentrated on developing algorithms (see, e.g., Taylor and Uhlig (1990)), while comparatively little effort has been devoted to implementing these algorithms on the most advanced parallel computers available. To use these machines efficiently is not as straightforward as simply compiling existing code on them, as their efficient use requires that the code be specifically structured for parallel execution.[2] This paper describes an algorithm for solving nonlinear dynamic models that can be programmed with the structure necessary for efficient execution on parallel computing systems. The effect of being able to use these systems is to enlarge significantly the class of models than can feasibly be solved.

The class of nonlinear dynamic models this algorithm is designed to solve consists of discrete-time recursive systems that may have an endogenous state variable. The first section of this paper provides a general characterization of these systems, which is used to define a particular nonlinear operator for which the system's solution is a fixed point. This operator is used to define a sequence of functions that is assumed to converge to an equilibrium. For our

[2]Indeed, a common experience at the Institute for Empirical Macroeconomics is for someone to choose to run their program on a fast serial mainframe after being disappointed with the performance of the *same* program on the CRAY X-MP supercomputer.

purpose, computing this sequence involves many *independent* and *identical* calculations and thus forms a natural set-up for parallel computing. The particular implementation of this algorithm, however, depends on what type of parallel system is used. There are essentially two types: (1) a single-instruction multiple-data (SIMD) computer, such as a CRAY X-MP supercomputer, or (2), a multiple-instruction multiple-data (MIMD) computer, such as an Intel iPSC/860 hypercube.[3] A more readily available alternative that can approximate an MIMD computer is a network of stand-alone workstations. This paper briefly describes the defining characteristics of both SIMD and MIMD systems, and then describes versions of the algorithm that are optimal for each system.

This paper is organized as follows. First, the nonlinear operator and associated sequence of functions are defined. This section is followed by an example, and then by a brief discussion on how to approximate these sequences on a digital computer. Implementations of this algorithm are then considered for both SIMD and MIMD computers, and performance results based on an example are presented. These sections are followed by some concluding remarks.

The Nonlinear Operator

Let z_t denote a vector of exogenous state variables at time t. Assume the law of motion for z_t is Markovian and can be written as

$$z_{t+1} = f(z_t, \varepsilon_{t+1}),$$

where ε_{t+1} consists of iid random variables drawn from a distribution μ. Let x_t denote a vector of endogenous state variables and let u_t denote a vector of control variables (i.e.,

[3]In this terminology, the standard serial computer is a single-instruction single-data (SISD) computer.

variables chosen by economic agents, or variables, such as prices, that are set such that markets clear), which evolve according to

$$x_{t+1} = g(x_t,z_t,u_t), \;\; u_t = h(x_t,z_t).$$

The functions f and g are given, and the task at hand is to find a control function h that satisfies a set of equilibrium conditions.

For many problems these equilibrium conditions can be written as

$$P(x_t,z_t,u_t) = \int Q(x_{t+1},z_{t+1},u_{t+1})\mu(d\varepsilon_{t+1}),$$

where x, z, and u evolve according to $g, f,$ and h respectively. Using these equilibrium conditions, and the laws of motion specified above, given a control function h define the control function Ah such that

$$(1) \qquad P[x,z,(Ah)(x,z)] = \int Q\{g[x,z,(Ah)(x,z)],f(z,\varepsilon),h[g(x,z,(Ah)(x,z)),f(z,\varepsilon)]\}\mu(d\varepsilon).$$

An equilibrium control function is a fixed point $h = A(h)$ of the nonlinear operator A. For this paper assume that A is well defined and that Gauss-Seidel iterations based on A (i.e., $h_{n+1} = A(h_n)$, $n \geq 0$, h_0 given) converge to a fixed point of A. Some examples where these properties are worked out theoretically can be found in Lucas and Stokey (1987), Coleman (1991,1992), Greenwood and Huffman (1992), and Coleman, Gilles, and Labadie (1992).

An Example

As an example, consider the Brock-Mirman (1972) Stochastic Growth Model with serially correlated productivity shocks. In this model z represents the productivity shock, x

represents the capital stock, and u represents consumption. The function $h(x,z)$ thus represents the consumption function, and for the gross production function $F(x,z)$ the function

$$g(x,z,u) = F(x,z) - u$$

represents the capital accumulation function. Also, $P(x,z,u)$ equals the marginal utility function, say $M(u)$, and $Q(x,z,u)$ equals discounted marginal utility times the marginal rate of transformation, say $\beta M(u)F_x(x,z)$, so that (1) becomes

$$M[(Ah)(x,z)] = \beta\int M\{h[g(x,z,(Ah)(x,z)),f(z,\varepsilon)]\}F_x[g(x,z,(Ah)(x,z)),f(z,\varepsilon))]\mu(d\varepsilon).$$

This condition simply states that, at the equilibrium, households are indifferent between consuming or investing an additional unit of output.

Approximating the Equilibrium on a Computer

Consider the following way to approximate a fixed point of A that is suitable for a computer. First, define a grid $D = ((x_i,z_j),\ i = 1,\dots,\ n_x;\ j = 1,\dots,\ n_z)$. Second, define the finite-dimensional set H_D in which a typical element is a function h that consists of values on the grid D along with an interpolation rule to compute the values off the grid. Third, approximate the integral in (1) by a quadrature rule with n_ε points ε_k and weights w_k: $((\varepsilon_k,w_k),\ k = 1,\ n_\varepsilon))$. Given an initial function $h_0 \in H_D$, compute h_1 on the grid D such that

$$(2)\qquad P[x_i,z_j,h_1(x_i,z_j)] = \sum_{k=1}^{n_\varepsilon} Q[g(x_i,z_j,h_1(x_i,z_j)),f(z_j,\varepsilon_k),h_0(g(x_i,z_j,h_1(x_i,z_j)),f(z_j,\varepsilon_k))]w_k$$

for $i = 1,\ldots, n_x$; $j = 1,\ldots, n_z$. Compute values of h_1 off the grid according to the interpolation rule, so that $h_1 \in H_D$. In this way a sequence $\{h_n\}$ is computed.[4] Note that computing h_1 evaluated at the grid point (x',z') is independent of computing h_1 evaluated at any other grid point (x'',z''). This feature allows the above algorithm to be structured so that it takes advantage of parallel computing systems.

MIMD Computers

An MIMD computer can be thought of as a collection of *processing elements* (PEs) and a network over which they can communicate (and thereby cooperate) with each other. Some examples of a network are a *ring* in which each PE communicates with its two closest neighbors, a *completely connected network* in which each PE communicates with all other PEs, and a *star* with one central PE to which all other PEs are connected. Each PE has access to private memory and may also have access to memory shared by each of the PEs. It is important to note that in developing an MIMD computer the chief technological problem is inventing an efficient network. Substantial communication between the PEs, their competition for shared memory and the attention of a controlling PE can all contribute to a network bottleneck. Almasi and Gottlieb (1989) present a thorough and authoritative treatment of these issues.

The star configuration seems optimal for solving the models described here. With this configuration, refer to the controlling PE and associated memory as the master PE and the remaining PEs with associated memory as the slave PEs. In the context of this paper, each slave PE loads the section of the program that computes $h_1(x_i,z_j)$ for a given function h_0 and grid point (x_i,z_j). The program running on the master PE initializes h_0, sends it to each of the slave PEs (or places it into shared memory), and then sends each slave PE a different

[4]For the Stochastic Growth Model, Coleman (1990) describes this procedure in more detail.

value (or group of values) of the state vector (x_i,z_j). A particular slave PE receives (x_i,z_j), computes $h_1(x_i,z_j)$, and sends the answer back to the master PE, which immediately sends another value of the state vector to this slave PE. In this way all the slave computers are always active in computing h_1. Once an answer is received for all the values of the state vector on the grid, h_1 is completely computed. The master PE stops if convergence between h_0 and h_1 is achieved; otherwise it replaces h_0 by h_1, sends the new h_0 to all the slave PEs, and starts anew in computing h_1.

The Example, Continued

Consider solving the Stochastic Growth Model on a master-slave system. In particular, in the notation above, suppose $M(u) = u^{-\tau}$, $\tau = .5$; $\beta = .95$; $F(x,z) = e^z x^\alpha + (1-\delta)x$, $\alpha = .33$, $\delta = .1$; and suppose z evolves according to $z' = \rho z + \varepsilon$, $\rho = .95$, ε is drawn from a Normal$(0,\sigma^2)$ distribution, $\sigma = .1$.[5] Choose 50 evenly spaced grid points for the *log* of the capital stock *log* x in the interval [-1,4]. Choose 20 evenly spaced grid points for z in the interval [-1.5,1.5]. Choose 5 points in the Hermite-Gauss quadrature rule that approximates integration with respect to the Normal distribution. Use a bilinear interpolation routine on the *log* of h_0. Given h_0, use a secant-based routine to solve for h_1 on the grid and suppose convergence between h_0 and h_1 is obtained when the *log* difference between the two on the grid is less than 10^{-6}. Finally, initialize h_0 to $h_0(x,z) = \gamma F(x,z)$, $\gamma = 1 - \alpha\beta/(1 - (1-\alpha)\delta\beta)$, which initializes h so that it gives the correct value at the steady states (one for each z) in the corresponding deterministic model.

The configuration used to solve this model consists of three SUN Sparc-2 workstations linked via Ethernet.[6] The software program PVM (parallel virtual machine; see Beguelin, et.

[5]These parameter values (except for δ, which is set to 0) are used in Taylor and Uhlig (1990).
[6]Ethernet communicates at a speed of 10MB per second.

al., 1991) is used to configure this network into a master-slave system (although the physical set-up looks nothing like a star). There is no shared memory in this configuration; all communication is achieved through messages passed between PEs (i.e., workstations). Table 1 reports the performance of this configuration when the slave PEs compute h_1 for 100 values of the state vector at a time.

Table 1

Time to Solve the Stochastic Growth Model on a Master-Slave System

no. of SUN Sparc-2s[7]	elapsed time (in seconds)[8]
1	41
2	20
3	15

As Table 1 shows, for relatively few workstations the speed of the algorithm is roughly linear in the number of workstations. That is, twice the workstations results in twice the speed (half the time). Since each workstation computes one-tenth of h_1 at a time (100 grid points out of 1000), we should expect this linear dependence to continue up to ten workstations. Beyond that each workstation must receive a smaller fraction of the total number of grid points. This leads to an increase in overall communication time (due to overhead in sending a message), and thus we should expect some degradation in performance when more than ten workstations are used.[9] Many problems, however, require a grid consisting of 100,000 points

[7]When more than one workstation is used the host workstation acts as both the master PE and one slave PE. When one workstation is used the PVM program is not used.

[8]Elapsed time refers to the difference in the time on a clock from when the program was submitted to when it finished. During this time there were no other users on the system.

[9]On the other hand, if the workstations ran at different speeds or had varying degrees of excess capacity, then reducing the number of grid points sent to each workstation would allow the master workstation to allocate more of the load to the faster workstation (you would like all the workstations to be done at the same time). This may even reduce the elapsed time to run the program. In this case there is some tradeoff between communication costs and load balancing.

(or more); based on this example we should then expect the speed to be linear in the number of workstations for even rather large numbers of workstations.[10] Also, if the time to compute h_1 for a single grid point is fairly long then the communication time becomes relatively less important, and here again we should expect the speed to be linear in the number of workstations for large numbers of workstations.

SIMD Computers

An SIMD computer should be thought of in a much different way than an MIMD computer. An SIMD computer consists of an array of processing elements that operate on different data, but at any moment in time they all must execute the same instruction. In this way all the processing elements are always in perfect synchronization, thus avoiding the networking difficulties encountered in building an MIMD computer. This feature is achieved, however, at a cost of demanding much more structure on the program to exploit this capability.

For the purposes of this paper it helps to simply think of an SIMD computer as executing, subject to some restrictions, the innermost DO loops in a FORTRAN program a certain number of blocks at a time. By number of blocks I mean that a 64-processor computer executes these loops 64 blocks at a time, so that 640 iterations through the loop only takes 10 executions. This feature is referred to as *vectorization*. For vectorization to occur there are a variety of restrictions (although not without exception) that an innermost DO loop must satisfy,

[10]Think of solving a version of Lucas' (1980) continuum-household model with a serially correlated monetary growth rate. Suppose we approximate the distribution of money across households by a four parameter family, then a typical household's state vector consists of its money holdings, the monetary growth rate, and the four parameters of the distribution. If the grid consists of twenty values for money, and five values for each of the remaining five state variables, then the total grid size is $20*5^5 = 62{,}500$. If this problem took fifty hours to solve on one workstation (probably not feasible), it would take five hours to solve on ten workstations (which can be solved overnight). It seems reasonable to say that many academic departments have access to at least ten workstations.

and I will only list four of them here.[11] First, there must exist a least one vector on the left side of an equal sign. Second, the index for this vector must increment by a constant amount after each iteration through the loop. Third, there can be no calls to a subroutine within the loop. And fourth, there can be no conditional jump out of the loop. For this paper the last two restrictions pose the most difficulty.

As the largest loop usually consists of the one which loops over the grid D, to vectorize this algorithm efficiently requires placing this loop inside all others. In this context the loop is executed $n_x n_z$ times, and h_1 (or any other function evaluated at each grid point) is the vector on the left side of an equal sign. The effect of being able to vectorize these loops is to compute $h_1 = A(h_0)$ n_v blocks at a time, where n_v is the number of processors an SIMD computer has. Understanding how this is achieved requires going into a bit more detail concerning how h_1 is computed.

Computing h_1 requires solving the system of equations (1) for each value of the state vector on the grid D. To be specific, suppose $h_1(x,z)$ is computed via Newton's algorithm. Let $T(\xi;x,z,h_0)$ represent the right side minus the left side of eq. (1) for an arbitrary guess ξ of $h_1(x,z)$:

$$T(\xi;x,z,h_0) = \int Q\{g(x,z,\xi),f(z,\varepsilon),h_0[g(x,z,\xi),f(z,\varepsilon)]\}\mu(d\varepsilon) - P(x,z,\xi).$$

The solution $h_1(x,z)$ satisfies $T[h_1(x,z);x,z,h_0] = 0$. Denote the derivative of T with respect to ξ as $T_1(\xi;x,z,h_0)$. Given a guess $h_1'(x,z)$ for the root $h_1(x,z)$ corresponding to the state (x,z), the new guess $h_1''(x,z)$ satisfies

$$T[h_1'(x,z);x,z,h_0] + T_1[h_1'(x,z);x,z,h_0][h_1''(x,z) - h_1'(x,z)] = 0.$$

[11]See CRAY-2 Computing, Introduction to (1988) for more details.

Suppose, also, that the derivative T_1 is approximated by a finite difference. The core of the program consists of the following steps: (i) begin with $h_1'(x,z)$, (ii) evaluate $T[h_1'(x,z);x,z,h_0]$ and $T_1[h_1'(x,z);x,z,h_0]$, and (iii) compute $h_1''(x,z)$. An efficient vectorization of this algorithm is then such that h_1'' is computed n_v blocks at a time. Specifically, an evaluation of $T[h_1'(x,z);x,z,h_0]$ is performed n_v blocks at a time, and given $T[h_1'(x,z);x,z,h_0]$ and $T_1[h_1'(x,z);x,z,h_0]$, h_1'' is computed n_v blocks at a time. To satisfy the restriction that there be no conditional jumps out of the innermost DO loop, solving for $h_1''(x,z)$ for n_v values of the state vector requires that convergence be obtained for *all* these values before proceeding to the next block.

For a given vector h_1', evaluating $T[h_1'(x,z);x,z,h_0]$ n_v blocks at a time is fairly straightforward. As evaluating T usually involves an additional loop over n_ε that represents the quadrature formula, one must place the loop over $n_x n_z$ inside the n_ε loop. This simply involves looping over the conditioning information set for every possible transition to a state next period. Also, the interpolation routine must be placed in-line to satisfy the restriction that there can be no calls to a subroutine within the innermost loop. This is usually no problem for simple interpolation routines such as piecewise multilinear interpolation over a uniform grid.

For a given state (x,z), denote the dimension of the control vector $h(x,z)$ by n_h. If the dimension n_h is at most two then given $T[h_1'(x,z);x,z,h_0]$ and $T_1[h_1'(x,z);x,z,h_0]$ it is fairly straightforward to compute h_1'' within the loop over $n_x n_z$ (it is easy to invert a 2x2 matrix). If the dimension is greater than two then computing h_1'' requires solving a system of linear equations

$$M_n y_n = b_n$$

for each n, where M_n and b_n vary over the loop counter $n = 1,\ldots, n_x n_z$ (y_n corresponds to $h_1''(x,z)$ - $h_1'(x,z)$, M_n corresponds to $T_1[h_1'(x,z);x,z,h_0]$, and b_n corresponds to $-T[h_1'(x,z);x,z,h_0]$). The routine based on LU factorization of M_n to solve for y_n involves a rather complicated nesting of loops and thus is not well-suited for vectorizing this problem. An alternative is to use an iterative method to compute y_n, such as the successive overrelaxation (SOR) generalization of Gauss-Seidel's method,[12] and placing the loop for this iterative procedure outside the one over $n_x n_z$. Gauss-Seidel's iterative method for solving a system of linear equations involves writing M_n as the sum of its strictly upper triangular portion, say U_n, and its lower triangular portion plus the diagonal entries, say V_n, so that $M_n = V_n + U_n$. Starting from a given y'_n, the next guess y''_n is computed according to

$$V_n y''_n = -U_n y'_n + b_n.$$

Since the upper triangular portion of V_n consists of zeros, y''_n can be easily computed recursively. The SOR method involves writing V_n as the sum of its strictly lower triangular part, L_n, and its diagonal entries, D_n, so that $V_n = L_n + D_n$. Then for a given ω, the next guess y''_n is computed according to

$$(D_n + \omega L_n)y''_n = [(1-\omega)D_n - \omega U_n]y'_n + \omega b_n.$$

For $\omega = 1$ this method reduces to the Gauss-Seidel method. The optimal choice of ω is the one which makes the largest eigenvalue of $(D_n + \omega L_n)^{-1}((1-\omega)D_n - \omega U_n)$ as small as possible, but except for certain special cases the eigenvalues and optimal choice for ω cannot be obtained explicitly. (Furthermore, expect for special cases one cannot guarantee convergence to the solution y_n.) It can be shown, however, that a necessary condition for convergence is $0 < \omega < 2$,[13] and for a large class of matrices M_n the optimal choice for ω is

[12]See Young (1971) for a discussion of this method. The SOR method was developed to handle very large linear systems, but as argued here it may work well in solving on a vector processor a large number of small linear systems.

[13]The product of the eigenvalues of the $n_h \times n_h$ matrix $(D+\omega L)^{-1}[(1-\omega)D-\omega U]$, say $\Pi\lambda_i$, is equal to its determinant. Recall that the determinant of a product is the product of the

between 1 and 2.[14] In practice, before computing a good approximation to the equilibrium one usually experiments with the algorithm while computing a fairly crude approximation, and during this experimentation phase one could try a few values of ω.

The Example, Continued

Consider again the Stochastic Growth Model along with the parameter values presented above, but now as executed on a CRAY X-MP vector processor. Although not strictly correct, think of this configuration as one in which n_v is about eight.[15] Table 2 reports performance statistics for this set-up. Serial code refers to code written for efficient execution on a serial computer, vector code refers to code written for efficient execution on a vector processor,

---

determinants, and the determinant of a triangular matrix is the product of the diagonal elements, so $\Pi\lambda_i = (1-\omega)^{n_h}$. Thus, $\max(\lambda_i) < 1$ only if $0 < \omega < 2$, and convergence of the SOR iterations requires $\max(\lambda_i) < 1$.

[14]See Young (1971, ch. 6).

[15]The CRAY X-MP approximates vectorization through pipelining. See Almasi and Gottlieb (1989) for a detailed discussion of pipelining. To briefly describe pipelining, a floating point operation takes a certain number of steps (i.e., clock cycles, which are usually between 6 and 10) to execute. Instead of executing all these steps before proceeding to the next floating point operation, pipelining executes these steps on a vector in lock-step fashion: the first step in the second floating point operation is performed at the same time as the second step in the first floating point operation, and so on. The CRAY X-MP has 64 registers of high speed memory that stores the vector which is pipelined. A true 64-processor SIMD computer would execute all 64 floating point operations at the same time. As pipelining economizes on hardware, it is evidently a cost-effective alternative to a true SIMD computer.

If a floating point operation took 8 clock cycles, then aside from start-up time a true 64-processor SIMD computer would take 8 clock cycles to process each vector of length 64. On the other hand, aside from start-up time a pipelined SIMD computer would take 64 clock cycles, which is equivalent to what a true 8-processor SIMD computer would take. This is what led me to set $n_v = 8$, but the start-up time to process vectors on a true 8-processor SIMD computer is likely to be significant, so n_v should probably be a bit higher.

This discussion actually pertains to each processor the CRAY X-MP has, and the CRAY X-MP has four processors. Strictly speaking, then, it is more accurate to say the CRAY X-MP is a 4-processor MIMD computer that approximates an SIMD computer on each processor. The MIMD capability of the CRAY X-MP is not used in this paper, although the results of the first part of this paper suggests that both the MIMD and SIMD features of the CRAY X-MP can be exploited--even at the *same* time.

serial CRAY refers to the CRAY X-MP with vectorization turned off, and vector CRAY refers to the CRAY X-MP with vectorization turned on. Code explicitly tailored to the CRAY X-MP vector processor ran five times faster than code written for a serial computer (.5 seconds versus 2.6 seconds). Although not overwhelming, this is an appreciable gain in speed. Note also that simply compiling existing code written for a serial computer did not take advantage of the vector processor.

Table 2

Time to Solve the Stochastic Growth Model on a Vector Processor

set-up	execution time (in seconds)[16]
serial code, serial CRAY	2.60
serial code, vector CRAY	2.60
vector code, serial CRAY	5.00
vector code, vector CRAY	0.47

Solving the Stochastic Growth Model did not require using the successive overrelaxation method to solve a system of linear equations (because $n_h = 1$). To get an idea of the efficiency of the SOR method, consider solving

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Table 3 compares solving this system 6400 times on the CRAY X-MP using two different methods. The first method is LU factorization as programmed in Press, et. al., (1988). The

[16]Execution time refers to the time the CRAY X-MP spent executing the program on one of its four processors (the other three processor were not used).

second method is SOR with various values of ω (starting with the guess $x = (1,1,1)$;[17] note that the solution is $x = (1.5, 2.0, 1.5)$). Convergence of the SOR method was said to obtain if successive iterations were within 10^{-6} of each other in the sup norm. As can be seen in this table, solving the system using the SOR method with a good choice of ω ($\omega = 1.2$) is thirty times faster than the LU factorization method (.017 versus .480 seconds). Since the execution time for the SOR method depends on the initial guess of the solution, it is difficult to know how much of the increase in speed to attribute to vectorization and how much to attribute to a good initial guess. Note, however, that the SOR method with vectorization turned on is seven times faster than when vectorization is turned off (.017 versus .123 seconds). Using the SOR method in this way thus can take advantage of vectorization, and it may thus significantly increase the speed in solving models for which $n_h \geq 3$.

Table 3

Time to Solve a 3×3 Linear System 6400 Times on a Vector Processor

LU factorization versus the SOR method

method	execution time (in seconds)
LU factorization, vector CRAY	.480
SOR, $\omega = 1$, vector CRAY	.036
SOR, $\omega = 1.1$, vector CRAY	.028
SOR, $\omega = 1.2$, vector CRAY	.017
SOR, $\omega = 1.3$, vector CRAY	.023
SOR, $\omega = 1.2$, serial CRAY	.123

[17]In solving a particular linear system while computing h_{n+1}, with the SOR method you can start with a guess of the solution equal to the corresponding solution after computing h_n. After the first couple of iterations of $\{h_n\}$ these guesses should be reasonably accurate, so the SOR method may even dominate the LU factorization method on a serial computer (for both small and large n_h).

Concluding Remarks

Parallel computing is an emerging technology that may make it feasible to solve models we would not have dreamed of solving a decade ago. Whether this technology delivers, say, a thousand-fold increase in speed over the next decade depends on both technological innovations and on the existence of applications that can take advantage of this technology. Indeed, given that the efficiency of an MIMD or SIMD computer *depends* on the application run on these systems, the success of a technological innovation depends every bit as much on these applications. In this light, this paper shows that a large class of models in economics can use this technology, even in the state we see this technology embodied in current hardware.

References

Aiyagari, Rao S. and Mark Gertler (1990): "Asset Returns with Transactions Costs and Uninsurable Individual Risk: A Stage III Exercise," working paper, C.V. Starr Center for Applied Economics, 90-43, New York University.

Almasi, George S. and Allan Gottlieb (1989): *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Redwood City, California.

Beguelin, Adam, Jack Dongarra, Al Geist, Bob Manchek, and Vaidy Sundaram (1991): "A Users' Guide to PVM: Parallel Virtual Machine," working paper, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

Brock, William A. and Leonard J. Mirman (1972): "Optimal Economic Growth and Uncertainty: the Discounted Case," *Journal of Economic Theory*, 4, 497-513.

Coleman, Wilbur John II (1990): "Solving the Stochastic Growth Model by Policy-Function Iteration," *Journal of Business & Economic Statistics*, 8, 27-29.

__________ (1991): "Equilibrium in a Production Economy with an Income Tax," *Econometrica*, 59, 1091-1104.

__________ (1992): "Precautionary Money Balances with Aggregate Uncertainty," working paper, Board of Governors of the Federal Reserve System.

Coleman, Wilbur John II, Christian Gilles, and Pamela Labadie (1992): "The Liquidity Premium in Average Interest Rates," working paper, Board of Governors of the Federal Reserve System.

CRAY-2 Computing, Introduction to (1988). Minnesota Supercomputer Center, Minneapolis, Minnesota.

Greenwood, Jeremy and Gregory W. Huffman (1992): "On the Existence and Uniqueness of Nonoptimal Equilibria in Dynamic Stochastic Economies," Research Department Staff Report, 151, Federal Reserve Bank of Minneapolis.

Hildebrand, F. B. (1956): *Introduction to Numerical Analysis*, New York, McGraw-Hill.

Imrohoroglu, Ayse and Edward C. Prescott (1991): "Evaluating the Welfare Effects of Alternative Monetary Arrangements," Federal Reserve Bank of Minneapolis, *Quarterly Review*, 15, 3-10.

Lucas, Robert E., Jr. (1980): "Equilibrium in a Pure Currency Economy," *Economic Inquiry*, 28, 203-220.

Lucas, Robert E., Jr. and Nancy L. Stokey (1987): "Money and Interest in a Cash-In-Advance Economy," *Econometrica*, 55, 491-513.

Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling (1988): *Numerical Recipes*, Cambridge University Press.

Scheinkman, Jose A. and Laurence Weiss (1986): "Borrowing Constraints and Aggregate Economic Activity," *Econometrica*, 54, 23-45.

Stroud, A. H. and Don Secrest (1966): *Gaussian Quadrature Formulas*, New Jersey, Prentice-Hall.

Taylor, John B. and Harold Uhlig (1990): "Solving Nonlinear Stochastic Growth Models: A Comparison of Alternative Solution Methods," *Journal of Business and Economic Statistics*, 8, 1-17.

Young, David M. (1971): *Iterative Solution of Large Linear Systems*, New York, Academic Press.

Appendix

This appendix contains some of the programs, data, and sample output that are used in the text. Some additional programs to simulate the Stochastic Growth Model are also included. These files are available on diskette from the author.

Short description of the files:

sgm.txt	- Contains a description of the program sgm.f.
sgm.f	- FORTRAN program that solves the Stochastic Optimal Growth Model.
hermit.f	- Subroutine called by sgm.f that computes Hermite-Gauss quadrature points and weights.
sgm.dat	- Data read by sgm.f.
tpath.f	- FORTRAN program that simulates time paths using the solution computed by sgm.f.
gasdev.f	- Subroutine called by tpath.f that returns Normal(0,1) random variates.
smpl.out	- Sample output from sgm.f and tpath.f using sgm.dat.
vsgm.f	- version of program sgm.f that is written for a vector processor.
vsgm.dat	- Data read by vsgm.f.

File: sgm.txt.

The FORTRAN program sgm.f solves the Stochastic Optimal Growth Problem using the algorithm outlined in Coleman (JBES, 1990). The problem is to solve for the fixed point c = A(c), where A is defined by

```
u'[(Ac)(x,z)] = beta*E{u'[c(f(x,z)-(Ac)(x,z),z')f'(f(x,z)-(Ac)(x,z),z')},

u'(c) = c**(-tau),  f(x,z) = exp(z)*x**alpha + delta*x,

z' = rho*z + q, q distributed Normal(0,sigma**2).
```

This note defines the program's variables and outlines the program's flow.

Definition of Variables:

nx	number of grid points for the capital stock x.
xmin	smallest grid point for the log of the capital stock.
xmax	largest grid point for the log of the capital stock.
x	vector of length nx containing equally spaced grid points from xmin to xmax for the log of the capital stock.
nz	number of grid points for the productivity shock z.
zmin	smallest grid point for the productivity shock.
zmax	largest grid point for the productivity shock.
z	vecor of length nz containing equally spaced grid points from zmin to zmax for the productivity shock.
nq	number of quadrature points for q used to approximate integration with respect to the Normal(0,sigma**2) distribution.
qz	vector of length nq containing the quadrature points for q.
qw	vector of length nq containing the quadrature weights for q.
azpe	matrix of dimension nq by nz which equals alpha*exp(rho*z(i)+qz(j)).
kzp	matrix of dimension nq by nz containing values such that z(ksp(i,j)) is the nearest grid point z less than rho*z(i)+qz(j).
tz	matrix of dimension nq by nz containing the distance (between 0 and 1) to the nearest grid point greater than rho*z(i)+qz(j).
f	matrix of dimension nx by nz that equals production at the grid pnts.
cpm	matrix of dimension nx by nz containing the log of consumption c at the grid points at the beginning of each iteration. Consumption off the grid points is determined using bilinear interpolation. The consumption function is initialized such that cpm(i,j) is correct at the "deterministic steady state" for each z, and equals the same fraction of output for other capital stocks.
cm	matrix of dimension nx by nz corresponding to the values determining consumption after each iteration.
eps	vector of length 2 containing the convergence parameters.
maxit	limit imposed on the number of iterations.

Program Flow:

The main program loops over calls to the subroutine a, which succesively computes cm = a(cpm). The program stops if cm is within eps(2) of cpm. The subroutine a uses a secant version of Newton's algorithm to compute the root cm(i,j) to the function in the subroutine zfun. A root is obtained if the function in zfun is within eps(1) of zero. The function in zfun is the rhs - lhs of the Euler equation at the top of this page. The solution cm is written to the file sgm.fun.

File: sgm.f.

```
c
c***********************************************************************
c  Program to solve the Stochastic Optimal Growth Model.
c    Written by John Coleman  (updated:  April, 1992).
c***********************************************************************
c
      implicit double precision (a-h,o-z)
      dimension cpm(100,20),cm(100,20),f(100,20),x(100),z(20),
     * azpe(21,20),kzp(21,20),tz(21,20),eps(2),qz(21),qw(21)
      common/pm/cpm,cm,f,x,z,azpe,kzp,tz,eps,qz,qw,beta,tau,alpha,delta,
     * rho,sigma,xmin,dx,nx,nz,nq,maxit
c
      write(*,'(a35)') ' stochastic optimal growth model'
      open(10,file='sgm.dat')
      read(10,'(10x,f10.4)') beta,tau,rho,sigma,alpha,delta,eps,
     *  xmin,xmax,zmin,zmax
      read(10,'(10x,i10)') nx,nz,nq,maxit
c
      call hermit(nq,qz,qw)
      scalez = sigma*dsqrt(2.0d0)
      scalew = 1.0d0/dsqrt(3.141592654d0)
      do 100 i = 1, nq
        qz(i) = scalez*qz(i)
100     qw(i) = scalew*qw(i)
c
      dx = (xmax-xmin)/dfloat(nx-1)
      dz = (zmax-zmin)/dfloat(nz-1)
c
      dapc = 1.0d0 - alpha*beta/(1.0d0-(1.0d0-alpha)*delta*beta)
      do 200 kx = 1, nx
200     x(kx) = xmin + dfloat(kx-1)*dx
      do 500 kz = 1, nz
        z(kz) = zmin + dfloat(kz-1)*dz
        do 300 i = 1, nq
          zp = rho*z(kz) + qz(i)
          azpe(i,kz) = alpha*dexp(zp)
          kzp(i,kz) = 1 + dmax1(0.0d0, (zp-zmin)/dz)
          if (kzp(i,kz).ge.nz) kzp(i,kz) = nz-1
          zr = zmin + dfloat(kzp(i,kz))*dz
300       tz(i,kz) = (zr-zp)/dz
        do 400 kx = 1, nx
          f(kx,kz) = dexp(z(kz))*dexp(x(kx))**alpha + delta*dexp(x(kx))
400       cpm(kx,kz) = dlog(dapc*f(kx,kz))
500     continue
c
      sup = 1.0d10
      do 800 numit = 1, maxit
        call a
        sup = 0.0d0
        do 700 kz = 1, nz
        do 600 kx = 1, nx
          sup = dmax1(sup, dabs(cm(kx,kz)-cpm(kx,kz)))
600       cpm(kx,kz) = cm(kx,kz)
700       continue
        write(6,'(a10,i5,a10,f10.4)') 'a iter. #',numit,'norm = ',sup
        if (sup.le.eps(2)) goto 900
800     continue
```

```
      write(*,'(a40)') ' maximum no. of iterations exceeded'
900   continue
c
      open(11,file='sgm.fun')
      write(11,'(8f15.6)') ((cm(kx,kz),kx=1,nx),kz=1,nz)
      stop
      end
c
c***********************************************************************
      subroutine a
c     cm = a(cpm)
c***********************************************************************
      implicit double precision (a-h,o-z)
      dimension cpm(100,20),cm(100,20),f(100,20),x(100),z(20),
     * azpe(21,20),kzp(21,20),tz(21,20),eps(2),qz(21),qw(21)
      common/pm/cpm,cm,f,x,z,azpe,kzp,tz,eps,qz,qw,beta,tau,alpha,delta,
     * rho,sigma,xmin,dx,nx,nz,nq,maxit
      do 300 kz = 1, nz
      do 200 kx = 1, nx
        croot = dexp(cpm(kx,kz))
        croot = dlog(croot/(f(kx,kz)-croot))
        croot2 = croot + eps(2)
        call zfun(croot,kx,kz,zval)
        call zfun(croot2,kx,kz,zval2)
        if (dabs(zval2).lt.dabs(zval)) then
          swap = croot
          croot = croot2
          croot2 = swap
          swap = zval
          zval = zval2
          zval2 = swap
          endif
        do 100 numit = 1, maxit
          adj = (croot2-croot)*zval/(zval-zval2)
          if (dabs(adj).lt.eps(1)) goto 200
          croot2 = croot
          zval2 = zval
          croot = croot + adj
          call zfun(croot,kx,kz,zval)
100       if (dabs(zval).lt.eps(1)) goto 200
        write(*,'(a40)') ' warning: max no. of iter. in z exceeded'
200     cm(kx,kz) = dlog(f(kx,kz)/(1.0d0+dexp(-croot)))
300     continue
      return
      end
c
c***********************************************************************
      subroutine zfun(croot,kx,kz,zval)
c     input: state (kx,kz), guess croot. output: zval = rhs - lhs of foc
c***********************************************************************
      implicit double precision (a-h,o-z)
      dimension cpm(100,20),cm(100,20),f(100,20),x(100),z(20),
     * azpe(21,20),kzp(21,20),tz(21,20),eps(2),qz(21),qw(21)
      common/pm/cpm,cm,f,x,z,azpe,kzp,tz,eps,qz,qw,beta,tau,alpha,delta,
     * rho,sigma,xmin,dx,nx,nz,nq,maxit
      c = f(kx,kz)/(1.0d0+dexp(-croot))
      xpe = f(kx,kz) - c
      xp = dlog(xpe)
      kxp = 1 + dmax1(0.0d0, (xp-xmin)/dx)
      if (kxp.ge.nx) kxp = nx-1
      xr = xmin + dfloat(kxp)*dx
      tx = (xr-xp)/dx
      w = 0.0d0
      do 100 i = 1, nq
        cp = tx*tz(i,kz)*cpm(kxp,kzp(i,kz))
     *    + (1.0d0-tx)*tz(i,kz)*cpm(kxp+1,kzp(i,kz))
     *    + (1.0d0-tx)*(1.0d0-tz(i,kz))*cpm(kxp+1,kzp(i,kz)+1)
     *    + tx*(1.0d0-tz(i,kz))*cpm(kxp,kzp(i,kz)+1)
        cp = dexp(cp)
        dfxp = azpe(i,kz)*xpe**(alpha-1.0d0)+delta
100     w = w + cp**(-tau)*dfxp*qw(i)
      zval = beta*w - c**(-tau)
      return
      end
```

File: hermit.f.

```
      subroutine hermit(nn,x,a)
c     Computes nn points x and weights a for Hermite-Gauss quadrature
c     that approximates integration with respect to exp(-x**2).
c     This program was copied from Stroud and Secrest, Gaussian
c     Quadrature Formulas, Prentice-Hall, 1966.
      implicit double precision (a-h,o-z)
      dimension x(50),a(50)
      eps = 1.0d-10
      fn = nn
      n1 = nn - 1
      n2 = (nn + 1)/2
      cc = 1.7724538509*gamm(fn)/(2.0d0**n1)
      s = (2.0d0*fn + 1.0d0)**0.16667d0
      do 10 i = 1, n2
        if (i-1) 10, 1, 2
c  largest zero
1       xt = s**3 - 1.85575/s
        goto 9
2       if (i-2) 10, 3, 4
c  second zero
3       xt = xt - 1.14d0*fn**0.426d0/xt
        goto 9
4       if (i-3) 10,5,6
c  third zero
5       xt = 1.86d0*xt - 0.86d0*x(1)
        goto 9
6       if (i-4) 10,7,8
c  fourth zero
7       xt = 1.91d0*xt - 0.91d0*x(2)
        goto 9
c  all other zeros
8       xt = 2.0d0*xt - x(i-2)
c
9       call hroot(xt,nn,dpn,pn1,eps)
        x(i) = xt
        a(i) = cc/dpn/pn1
        ni = nn - i + 1
        x(ni) = -xt
        a(ni) = a(i)
10      continue
c
      do 20 i = 1, nn
20      x(i) = -1.0d0*x(i)
c
      return
      end
c
      subroutine hroot(x,nn,dpn,pn1,eps)
      implicit double precision (a-h,o-z)
      iter = 0
1     iter = iter + 1
      call hrecur(p,dp,pn1,x,nn)
      d = p/dp
      x = x - d
      if (dabs(d) - eps) 3,3,2
2     if (iter - 10) 1,3,3
3     dpn = dp
```

```
      return
      end
c
      subroutine hrecur(pn,dpn,pn1,x,nn)
      implicit double precision (a-h,o-z)
      p1 = 1.0d0
      p = x
      dp1 = 0.0d0
      dp = 1.0d0
      do 1 j = 2, nn
        fj = j
        fj2 = (fj - 1.0d0)/2.0d0
        q = x*p - fj2*p1
        dq = x*dp + p - fj2*dp1
        p1 = p
        p = q
        dp1 = dp
        dp = dq
1       continue
      pn = p
      dpn = dp
      pn1 = p1
      return
      end
c
      double precision function gamm(x)
      implicit double precision (a-h,o-z)
      gam(y) = (((((((0.035868343d0*y - 0.193527818d0)*y
     * + 0.482199394d0)*y - 0.756704078d0)*y + 0.918206857d0)*y
     * - 0.897056937d0)*y + 0.988205891d0)*y - 0.577191652d0)*y + 1.0d0
      z = x
      if (z) 1,1,4
1     gamm = 0.0d0
      write(2,*) z
2     format(2x,19harg error for gamma ,e15.6)
      goto 14
4     if (z - 70.0d0) 6,1,1
6     if (z - 1.0d0) 8,7,9
7     gamm = 1.0d0
      goto 14
8     gamm = gam(z)/z
      goto 14
9     za = 1.0d0
10    z = z - 1.0d0
      if (z - 1.0d0) 13,11,12
11    gamm = za
      goto 14
12    za = za*z
      goto 10
13    gamm = za*gam(z)
14    continue
      return
      end
```

File: sgm.dat.

```
beta              0.95
tau               0.50
rho               0.95
sigma             0.10
alpha             0.33
delta             0.90
eps1            1.0d-8
eps2            1.0d-4
xmin           -1.0d0
xmax            4.0d0
zmin           -1.5d0
zmax            1.5d0
nx                  50
nz                  20
nq                   5
maxit             1000
nt               10000
```

File: tpath.f.

```
c
c***************************************************************************
c  Program to simulate time paths for the Stochastic Optimal Growth Model
c***************************************************************************
c
      implicit double precision (a-h,o-z)
      dimension cm(100,20),eps(2),v(4),vmin(4),vmax(4),vmean(4),stdev(4)
      character*8 vname(4)
      data vname/'  x','  z','c(x,z)','f(x,z)'/
c
      open(10,file='sgm.dat')
      read(10,'(10x,f10.4)') beta,tau,rho,sigma,alpha,delta,eps,
     * xmin,xmax,zmin,zmax
      read(10,'(10x,i10)') nx,nz,nq,maxit,nt
      open(11,file='sgm.fun')
      read(11,'(8f15.6)') ((cm(kx,kz),kx=1,nx),kz=1,nz)
      dx = (xmax-xmin)/dfloat(nx-1)
      dz = (zmax-zmin)/dfloat(nz-1)
c
c ***** initialize variables to compute summary statistics
      nvars = 4
      do 100 i = 1, nvars
        vmean(i) = 0.0d0
        vmin(i) = 1.0d10
        vmax(i) = -1.0d10
100     stdev(i) = 0.0d0
      xs = (xmin+xmax)/2.0d0
      zs = (zmin+zmax)/2.0d0
c
c **** simulate time series of length nt
      idum = -1
      do 500 n = 1, nt
c ***** compute new values of variables
        v(1) = xs
        v(2) = zs
        kx = 1 + dmax1(0.0d0, (xs-xmin)/dx)
        if (kx.ge.nx) kx = nx-1
        xr = xmin + dfloat(kx)*dx
        tx = (xr-xs)/dx
        kz = 1 + dmax1(0.0d0, (zs-zmin)/dz)
        if (kz.ge.nz) kz = nz-1
        zr = zmin + dfloat(kz)*dz
        tz = (zr-zs)/dz
        c = tx*tz*cm(kx,kz) + (1.0d0-tx)*tz*cm(kx+1,kz)
     *    + (1.0d0-tx)*(1.0d0-tz)*cm(kx+1,kz+1)
     *    + tx*(1.0d0-tz)*cm(kx,kz+1)
        v(3) = c
        c = dexp(c)
        y = dexp(zs)*dexp(xs)**alpha + delta*dexp(xs)
        v(4) = dlog(y)
c ***** compile summary statistics
        do 600 i = 1, nvars
          vmean(i) = vmean(i) + v(i)
          if (v(i) .lt. vmin(i)) vmin(i) = v(i)
          if (v(i) .gt. vmax(i)) vmax(i) = v(i)
600       stdev(i) = stdev(i) + v(i)**2
c ***** compute next period's values of the state variables
```

```
          xs = dlog(y-c)
          zs = rho*zs + sigma*gasdev(idum)
500       continue
c
c ***** compute summary statistics
        do 800 i = 1, nvars
          vmean(i) = vmean(i)/dfloat(nt)
800       stdev(i) = dsqrt(stdev(i)/dfloat(nt)-vmean(i)**2)
c
        write(*,'(/25x,a20)') 'summary statistics'
        write(*,'(/10x,4a15)') (vname(i),i=1,nvars)
        write(*,'(/a8,4f15.4)') 'mean',(vmean(i),i=1,nvars)
        write(*,'(a8,4f15.4)') 'min',(vmin(i),i=1,nvars)
        write(*,'(a8,4f15.4)') 'max',(vmax(i),i=1,nvars)
        write(*,'(a8,4f15.4)') 'sd',(stdev(i),i=1,nvars)
        return
        end
```

File: gasdev.f.

```
      double precision function gasdev(idum)
c     Returns a Normal(0,1) random variate (idum < 0 for first call).
c     This program was copied from Press, Flannery, Teukolsky, and
c     Vetterling, Numerical Recipies, Cambridge University Press, 1988.
      implicit real*8 (a-h,o-z)
      data iset/0/
      if (iset.eq.0) then
1       v1 = 2.0d0*ran1(idum) - 1.0d0
        v2 = 2.0d0*ran1(idum) - 1.0d0
        r = v1**2 + v2**2
        if (r.ge.1) goto 1
        fac = dsqrt(-2.0d0*dlog(r)/r)
        gset = v1*fac
        gasdev = v2*fac
        iset = 1
      else
        gasdev = gset
        iset = 0
      endif
      return
      end
c
      double precision function ran1(idum)
      implicit real*8 (a-h,o-z)
      dimension r(97)
      parameter (m1=259200,ia1=7141,ic1=54773,rm1=1.0d0/m1)
      parameter (m2=134456,ia2=8121,ic2=28411,rm2=1.0d0/m2)
      parameter (m3=243000,ia3=4561,ic3=51349)
      data iff /0/
      if (idum.lt.0 .or. iff.eq.0) then
        iff = 1
        ix1 = mod(ic1-idum,m1)
        ix1 = mod(ia1*ix1+ic1,m1)
        ix2 = mod(ix1,m2)
        ix1 = mod(ia1*ix1+ic1,m1)
        ix3 = mod(ix1,m3)
        do 11 j = 1, 97
          ix1 = mod(ia1*ix1+ic1,m1)
          ix2=mod(ia2*ix2+ic2,m2)
          r(j) = (dfloat(ix1) + dfloat(ix2)*rm2)*rm1
11        continue
        idum = 1
      endif
      ix1 = mod(ia1*ix1+ic1,m1)
      ix2 = mod(ia2*ix2+ic2,m2)
      ix3 = mod(ia3*ix3+ic3,m3)
      j = 1 + (97*ix3)/m3
      if (j.gt.97 .or. j.lt.1) pause
      ran1 = r(j)
      r(j) = (dfloat(ix1) + dfloat(ix2)*rm2)*rm1
      return
      end
```

File: smpl.out.

```
    stochastic optimal growth model
 a iter. #    1   norm =     .0695
 a iter. #    2   norm =     .0511
 a iter. #    3   norm =     .0378
 a iter. #    4   norm =     .0280
 a iter. #    5   norm =     .0205
 a iter. #    6   norm =     .0149
 a iter. #    7   norm =     .0106
 a iter. #    8   norm =     .0074
 a iter. #    9   norm =     .0050
 a iter. #   10   norm =     .0033
 a iter. #   11   norm =     .0021
 a iter. #   12   norm =     .0013
 a iter. #   13   norm =     .0007
 a iter. #   14   norm =     .0004
 a iter. #   15   norm =     .0002
 a iter. #   16   norm =     .0002
 a iter. #   17   norm =     .0001
 a iter. #   18   norm =     .0001

                          summary statistics

                  x              z              c(x,z)         f(x,z)

    mean        1.1528         -.0069          .1372         1.4619
     min        -.2480        -1.1442        -1.2460          .0658
     max        2.8203         1.1893         1.7877         3.1249
      sd         .4443          .3218          .4386          .4428
```

File: vsgm.f.

```
c
c*********************************************************************
c  Version of the program to solve the Stochastic Optimal Growth Model
c    that is written for a vector processor computer.
c    Written by John Coleman  (updated:  April, 1992).
c*********************************************************************
c
      implicit double precision (a-h,o-z)
      dimension cpm(2000),cm(2000),f(2000),x(100),z(20),
     * azpe(21,20),kzp(21,20),tz(21,20),eps(2),qz(21),qw(21)
      common/pm/cpm,cm,f,x,z,azpe,kzp,tz,eps,qz,qw,beta,tau,alpha,delta,
     * rho,sigma,xmin,dx,nx,nz,ns,nq,maxit,nvec
c
      write(*,'(a35)') ' stochastic optimal growth model'
      open(10,file='vsgm.dat')
      read(10,'(10x,f10.4)') beta,tau,rho,sigma,alpha,delta,eps,
     *  xmin,xmax,zmin,zmax
      read(10,'(10x,i10)') nx,nz,nq,maxit,nvec
      ns = nx*nz
c
      call hermit(nq,qz,qw)
      scalez = sigma*dsqrt(2.0d0)
      scalew = 1.0d0/dsqrt(3.141592654d0)
      do 100 i = 1, nq
        qz(i) = scalez*qz(i)
100     qw(i) = scalew*qw(i)
c
      dx = (xmax-xmin)/dfloat(nx-1)
      dz = (zmax-zmin)/dfloat(nz-1)
c
      dapc = 1.0d0 - alpha*beta/(1.0d0-(1.0d0-alpha)*delta*beta)
      do 200 kx = 1, nx
200     x(kx) = xmin + dfloat(kx-1)*dx
      do 500 kz = 1, nz
        z(kz) = zmin + dfloat(kz-1)*dz
        do 300 i = 1, nq
          zp = rho*z(kz) + qz(i)
          azpe(i,kz) = alpha*dexp(zp)
          kzp(i,kz) = 1 + dmax1(0.0d0, (zp-zmin)/dz)
          if (kzp(i,kz).ge.nz) kzp(i,kz) = nz-1
          zr = zmin + dfloat(kzp(i,kz))*dz
300       tz(i,kz) = (zr-zp)/dz
        do 400 kx = 1, nx
          k = kx + nx*(kz-1)
          f(k) = dexp(z(kz))*dexp(x(kx))**alpha + delta*dexp(x(kx))
400       cpm(k) = dlog(dapc*f(k))
500     continue
c
      sup = 1.0d10
      do 800 numit = 1, maxit
        call a
        sup = 0.0d0
        do 600 k = 1, ns
          sup = dmax1(sup, dabs(cm(k)-cpm(k)))
600       cpm(k) = cm(k)
        write(6,'(a10,i5,a10,f10.4)') 'a iter. #',numit,'norm = ',sup
        if (sup.le.eps(2)) goto 900
```

```
800     continue
      write(*,'(a40)') ' maximum no. of iterations exceeded'
900   continue
c
c      open(11,file='sgm.fun')
c      write(11,'(8f15.6)') ((cm(k),k=1,ns)
      stop
      end
c
c***********************************************************************
      subroutine a
c     cm = a(cpm)
c***********************************************************************
      implicit double precision (a-h,o-z)
      dimension cpm(2000),cm(2000),f(2000),x(100),z(20),
     * azpe(21,20),kzp(21,20),tz(21,20),eps(2),qz(21),qw(21)
      dimension croot(2000),croot2(2000),zval(2000),zval2(2000)
      common/pm/cpm,cm,f,x,z,azpe,kzp,tz,eps,qz,qw,beta,tau,alpha,delta,
     * rho,sigma,xmin,dx,nx,nz,ns,nq,maxit,nvec
      do 100 k = 1, ns
        croot(k) = dexp(cpm(k))
        croot(k) = dlog(croot(k)/(f(k)-croot(k)))
100     croot2(k) = croot(k) + eps(2)
      call zfun(croot,zval)
      call zfun(croot2,zval2)
      do 200 k = 1, ns
        if (dabs(zval2(k)).gt.dabs(zval(k))) goto 200
        swap = croot(k)
        croot(k) = croot2(k)
        croot2(k) = swap
        swap = zval(k)
        zval(k) = zval2(k)
        zval2(k) = swap
200     continue
      nbk = 1 + (ns-1)/nvec
      do 500 nk = 1, nbk
        n1 = 1 + (nk-1)*nvec
        n2 = min0(n1+nvec-1,ns)
      do 400 numit = 1, maxit
        conv = 0.0d0
        znorm = 0.0d0
cdir$ shortloop
        do 300 k = n1, n2
          adj = (croot2(k)-croot(k))*zval(k)/(zval(k)-zval2(k))
          conv = dmax1(conv, dabs(adj))
          znorm = dmax1(znorm, dabs(zval(k)))
          croot2(k) = croot(k)
          zval2(k) = zval(k)
300       croot(k) = croot(k) + adj
        if (conv.lt.eps(2) .or. znorm.lt.eps(2)) goto 500
        call zfuns(croot,n1,n2,zval)
400     continue
        write(*,'(a40)') ' warning: max no. of iter. in z exceeded'
500     continue
        do 600 k = 1, ns
600     cm(k) = dlog(f(k)/(1.0d0+dexp(-croot(k))))
      return
      end
c
c***********************************************************************
```

```
      subroutine zfuns(croot,n1,n2,zval)
c     input: state (kx,kz), guess croot. output: zval = rhs - lhs of foc
c**********************************************************************
      implicit double precision (a-h,o-z)
      dimension cpm(2000),cm(2000),f(2000),x(100),z(20),
     * azpe(21,20),kzp(21,20),tz(21,20),eps(2),qz(21),qw(21)
      dimension croot(2000),zval(2000)
      dimension c(2000),xpe(2000),kxp(2000),tx(2000),w(2000)
      common/pm/cpm,cm,f,x,z,azpe,kzp,tz,eps,qz,qw,beta,tau,alpha,delta,
     * rho,sigma,xmin,dx,nx,nz,ns,nq,maxit,nvec
cdir$ shortloop
      do 100 k = n1, n2
        c(k) = f(k)/(1.0d0+dexp(-croot(k)))
        xpe(k) = f(k) - c(k)
        xp = dlog(xpe(k))
        kxp(k) = 1 + dmax1(0.0d0, (xp-xmin)/dx)
        if (kxp(k).ge.nx) kxp(k) = nx-1
        xr = xmin + dfloat(kxp(k))*dx
        tx(k) = (xr-xp)/dx
100     w(k) = 0.0d0
      do 300 i = 1, nq
cdir$ shortloop
        do 200 k = n1, n2
          kz = 1 + (k-1)/nx
          i1 = kxp(k) + nx*(kzp(i,kz)-1)
          i2 = i1 + 1
          i3 = i2 + nx
          i4 = i3 - 1
          cp = tx(k)*tz(i,kz)*cpm(i1)
     *      + (1.0d0-tx(k))*tz(i,kz)*cpm(i2)
     *      + (1.0d0-tx(k))*(1.0d0-tz(i,kz))*cpm(i3)
     *      + tx(k)*(1.0d0-tz(i,kz))*cpm(i4)
          cp = dexp(cp)
          dfxp = azpe(i,kz)*xpe(k)**(alpha-1.0d0)+delta
200       w(k) = w(k) + cp**(-tau)*dfxp*qw(i)
300     continue
cdir$ shortloop
      do 400 k = n1, n2
400     zval(k) = beta*w(k) - c(k)**(-tau)
      return
      end
c
c**********************************************************************
      subroutine zfun(croot,zval)
c     input: state (kx,kz), guess croot. output: zval = rhs - lhs of foc
c**********************************************************************
      implicit double precision (a-h,o-z)
      dimension cpm(2000),cm(2000),f(2000),x(100),z(20),
     * azpe(21,20),kzp(21,20),tz(21,20),eps(2),qz(21),qw(21)
      dimension croot(2000),zval(2000)
      dimension c(2000),xpe(2000),kxp(2000),tx(2000),w(2000)
      common/pm/cpm,cm,f,x,z,azpe,kzp,tz,eps,qz,qw,beta,tau,alpha,delta,
     * rho,sigma,xmin,dx,nx,nz,ns,nq,maxit,nvec
      do 100 k = 1, ns
        c(k) = f(k)/(1.0d0+dexp(-croot(k)))
        xpe(k) = f(k) - c(k)
        xp = dlog(xpe(k))
        kxp(k) = 1 + dmax1(0.0d0, (xp-xmin)/dx)
        if (kxp(k).ge.nx) kxp(k) = nx-1
        xr = xmin + dfloat(kxp(k))*dx
```

```
          tx(k) = (xr-xp)/dx
  100     w(k) = 0.0d0
        do 300 i = 1, nq
          do 200 k = 1, ns
            kz = 1 + (k-1)/nx
            i1 = kxp(k) + nx*(kzp(i,kz)-1)
            i2 = i1 + 1
            i3 = i2 + nx
            i4 = i3 - 1
            cp = tx(k)*tz(i,kz)*cpm(i1)
       *      + (1.0d0-tx(k))*tz(i,kz)*cpm(i2)
       *      + (1.0d0-tx(k))*(1.0d0-tz(i,kz))*cpm(i3)
       *      + tx(k)*(1.0d0-tz(i,kz))*cpm(i4)
            cp = dexp(cp)
            dfxp = azpe(i,kz)*xpe(k)**(alpha-1.0d0)+delta
  200       w(k) = w(k) + cp**(-tau)*dfxp*qw(i)
  300     continue
        do 400 k = 1, ns
  400     zval(k) = beta*w(k) - c(k)**(-tau)
        return
        end
```

File: vsgm.dat.

```
beta            0.95
tau             0.50
rho             0.95
sigma           0.10
alpha           0.33
delta           0.90
eps1          1.0d-8
eps2          1.0d-4
xmin         -1.0d0
xmax          4.0d0
zmin         -1.5d0
zmax          1.5d0
nx                50
nz                20
nq                 5
maxit           1000
nvec              64
```