FEDERAL RESERVE BANK OF PHILADELPHIA

Ten Independence Mall
Philadelphia, Pennsylvania 19106-1574
(215) 574-6428, www.phil.frb.org

Working Papers

Research Department

WORKING PAPER NO. 99-14

SOLVING AND SIMULATING A SIMPLE OPEN-ECONOMY MODEL WITH MARKOV-SWITCHING DRIVING PROCESSES AND RATIONAL LEARNING

Keith Sill
Federal Reserve Bank of Philadelphia

Jeffrey Wrase
Federal Reserve Bank of Philadelphia

October 14, 1999

Working Paper No. 99-14

Solving and Simulating a Simple Open-Economy Model with Markov-Switching Driving Processes and Rational Learning

Keith Sill
Federal Reserve Bank
of Philadelphia

Jeff Wrase
Federal Reserve Bank
of Philadelphia

October 14, 1999

1. Introduction

This paper provides detail on technical aspects of the methods used to solve and simulate the models in our Working Papers "Exchange Rates, Monetary Policy Regimes, and Beliefs" and "Exchange Rates and Monetary Policy Regimes in the U.S. and Canada." In those papers we solve and simulate a two-country, limited participation monetary model of the exchange rate. The models feature money supply growth rate processes that have regime-switching behavior as in Hamilton's (1989) Markov-switching model. We investigate the predictions of the models under the assumption of a nonlinear driving process. In addition, we look at the implications of environments where agents have full information about the monetary regime and when they must form inferences about the state of the monetary regime. In the latter case, agents form beliefs about the state of monetary policy and update beliefs rationally using a Bayesian updating procedure.

The models in our two Working Papers are distinguished by the time series processes that describe the evolution of the exogenous state variables: money growth and technology shocks. In "Exchange Rates, Monetary Policy Regimes, and Beliefs," we assume that the exogenous state variables driving the economies are independent in that there are no feedback effects between countries. In "Exchange Rates and Monetary Policy Regimes in the U.S. and Canada," we allow the two countries to have cross-correlated technology shocks and money growth rates. Consequently, the models of exogenous shocks and belief sequences differ across the two exercises.

This paper is in the nature of a technical appendix. Section 2 describes a theoretical open-economy model that features limited participation by households and firms in asset-trading markets. Section 3 gives details on how the model is solved. We use Christiano's (1991,1998) method of undetermined coefficients. Section 4 describes the simulation methods used to generate second moments and impulse response functions. The Appendix provides a listing of computer programs that solve and simulate versions of the basic model. The programs are written in Python, an object-oriented scripting language that is freely available for a wide variety of computing platforms, including Windows, Unix, and Linux. The code listing provides a reference to much of the discussion in the text. All programs used to solve and simulate the models in the two Working Papers referred to above are available from the authors on request.

2. Models

The basic model is a two-country variant of that in Christiano (1991) and is similar to the model in Schlagenhauf and Wrase (1995). The trading opportunities, objectives, and constraints of households are assumed to be isomorphic across countries. Details are provided for the representative domestic household's decisions and opportunities only: the foreign analogs are straightforward and involve only obvious notational alterations.

The domestic household begins period t with K_t^D units of capital and A_t^D units of domestic currency. At the beginning of period t, nominal wealth A_t^D is divided by sending a deposit of N_t^D currency units to the domestic financial market. The remaining $A_t^D - N_t^D$ is

allocated to trade in the foreign exchange market. In the exchange market, domestic and foreign households trade currencies to arrange balances for use in purchasing consumption goods.

Domestic currency available to the domestic household in the exchange market consists of $A_t^D - N_t^D$ along with the household worker's wages. The worker supplies $\tilde{H}_t^D$ labor units in the domestic labor market at nominal wage W_t^D. In the foreign exchange market $A_t^D - N_t^D + a_1 W_t^D \tilde{H}_t^D$ units of domestic currency are divided into a domestic currency balance, $M_{D,t}^D$, and a foreign currency balance, $M_{F,t}^D$, at nominal exchange rate e_t (expressed in domestic per foreign currency units). We allow for some of the worker's wage receipts, $a_1 W_t^D \tilde{H}_t^D$, with $0 \le a_1 \le 1$, to be used in currency trades in the foreign exchange market. The household's nominal allocation in the foreign exchange market is:

$$A_t^D - N_t^D + a_1 W_t^D \tilde{H}_t^D = M_{D,t}^D + e_t M_{F,t}^D \tag{2.1}$$

The household shopper purchases $C_{D,t}^D$ units of home-produced goods at price P_t^D, and $C_{F,t}^D$ units of foreign goods at price P_t^F, subject to the cash constraints:

$$P_t^D C_{D,t}^D \le M_{D,t}^D \tag{2.2}$$

$$P_t^F C_{F,t}^D \le M_{F,t}^D \tag{2.3}$$

When the constraints bind as equalities, the shopper and worker combine to return home at the end of the period with goods, but no cash.

The financial intermediary receives a monetary injection X_t^D in the financial market, which is deposited on behalf of the household. The intermediary can then lend $N_t^D + X_t^D$ units of cash to domestic firms. Loanable cash supplied by the intermediary is:

$$\tilde{L}_t^D = N_t^D + X_t^D \tag{2.4}$$

The firm manager hires workers, undertakes investment, and holds the household's capital stock K_t^D. Prior to producing output, the firm borrows L_t^D domestic currency units from an intermediary to finance acquisition of H_t^D units of labor at wage W_t^D per unit and to potentially finance capital accumulation, in the face of a cash constraint:

$$W_t^D H_t^D + \alpha_2 P_t^D I_t^D \leq L_t^D \tag{2.5}$$

The firm purchases $I_t^D = K_{t+1}^D - (1-\delta^D)K_t^D$ units of home-produced goods to add to the household's capital stock and finances fraction $\alpha_2 P_t^D I_t^D$, with $0 \leq \alpha_2 \leq 1$, representing the fraction of investment financed by loans. Capital and consumption goods are indistinguishable in the domestic goods market and sell at a common price P_t^D.

From (2.1)-(2.3), the shopper and the worker bring home goods but no cash when the constraints bind as equalities. The firm manager, after the close of trading in the goods markets, pays loan obligation $R_{L,t}^D L_t^D$, where $R_{L,t}^D$ is the gross domestic loan rate. The manager brings home capital and cash profits of:

$$P_t^D Y_t^D - R_{L,t}^D L_t^D - (1-\alpha_2)P_t^D I_t^D \tag{2.6}$$

where Y_t^D is real output per domestic household.

The intermediary receives loan repayments $R_{L,t}^{D}\tilde{L}_{t}^{D} = R_{L,t}^{D}(N_{t}^{D} + X_{t}^{D})$ and pays a gross deposit return $R_{D,t}^{D}(N_{t}^{D} + X_{t}^{D})$. The intermediary returns home at the end of the period with its household's own deposit return, $R_{D,t}^{D}(N_{t}^{D} + X_{t}^{D})$, plus cash derived from intermediation $R_{L,t}^{D}(N_{t}^{D} + X_{t}^{D}) - R_{D,t}^{D}(N_{t}^{D} + X_{t}^{D})$. Thus, the intermediary brings home a cash balance of:

$$R_{L,t}^{D}(N_{t}^{D} + X_{t}^{D}) \tag{2.7}$$

Combining cash brought home by the household firm manager in (2.6), the intermediary in (2.7), and cash that the household worker did not send to the foreign exchange market gives the household's end-of-period nominal wealth:

$$\begin{aligned} A_{t+1}^{D} = {} & P_{t}^{D}Y_{t}^{D} - R_{L,t}^{D}L_{t}^{D} - (1-a_{2})P_{t}^{D}I_{t}^{D} + \\ & R_{L,t}^{D}(N_{t}^{D} + X_{t}^{D}) + (1-a_{1})W_{t}^{D}\tilde{H}_{t}^{D} \end{aligned} \tag{2.8}$$

2.1 Preferences, Technology, and Shocks

The household maximizes utility measure:

$$U = E_{t}\sum_{j=0}^{\infty} b_{D}^{\,t+j}u(C_{t+j}^{D}, l_{t+j}^{D}) \tag{2.9}$$

with $0 \le b_{D} \le 1$. Domestic consumption of home-produced goods, $C_{D,t}^{D}$, and foreign-produced goods, $C_{F,t}^{D}$, is aggregated via a Cobb-Douglas function:

$$C_{t}^{D} = (C_{D,t}^{D})^{q^{D}}(C_{F,t}^{D})^{1-q^{D}} \tag{2.10}$$

and momentary utility takes the form:

$$u(C_{D,t}^D, C_{F,t}^D, 1-\tilde{H}_t^D) = \frac{1}{j^D}\left\{(C_t^D)^g\,(l_t^D)^{1-g}\right\}^{j^D} \tag{2.11}$$

Leisure is $l_t^D = 1-\tilde{H}_t^D$, with the time endowment normalized to unity. Foreign utility is the same as (2.9)-(2.10) except for the obvious notational alterations.

For output production, each domestic firm possesses the technology:

$$Y_t^D = f^D(K_t^D, H_t^D) = (K_t^D)^a\,(H_t^D)^{1-a}\exp((1-a)z_t^D) \tag{2.12}$$

with $0 < a < 1$. z_t^D is a mean-zero technology shock, so there is no long-run growth in the model. The technology for foreign firms is the same as above except for notation.

Monetary injections are given by $X_t^D = M_{s,t+1}^D - M_{s,t}^D$ and $X_t^F = M_{s,t+1}^F - M_{s,t}^F$, where $M_{s,t}^D$ and $M_{s,t}^F$ are per own-country-household stocks of domestic and foreign currencies. The exogenous money growth rates $c_t^D = X_t^D / M_{s,t}^D$ and $c_t^F = X_t^F / M_{s,t}^F$ depend on the monetary policy regime. We assume that there are two monetary policy regimes: a high-mean money growth state and a low-mean money growth state.

2.2 Monetary Policy

Money growth is exogenous and follows a Markov-switching process in the mean and, possibly, in the variance. We consider two parameterizations of the money supply process. The first is one in which money growth rates are independent across countries. For this case we estimate univariate Markov-switching models for each country. Alternatively, we allow for

money supply feedback across countries. We then model the money supply process as a Markov-switching vector autoregression.[1]

First, consider the case where money growth is independent across countries. Let s_t indicate the unobserved regime with $s_t \in \{0,1\}$. Mean money growth can take on values $m(s_t) \in \{m^L, m^H\}$ with $m^L < m^H$. The time series process for money growth follows:

$$(x_t - m(s_t)) = r\,(x_{t-1} - m(s_{t-1})) + e_t \tag{2.13}$$

The monetary control error e_t is assumed to be distributed iid normal, drawn from a state-dependent distribution with mean 0 and standard deviation that possibly depends on the state. Mean money growth switches back and forth according to a Markov transition law with known parameters:

$$p_{ij} = \Pr[m(s_t) = m^j \mid m(s_{t-1}) = m^i] \text{ for } i, j = H, L \tag{2.14}$$

For the case where money growth is not assumed independent across countries we use a Markov-switching vector autoregression framework. The time series process for money growth is assumed to be:

$$(X_t - n(s_t)) = A(X_{t-1} - n(s_{t-1})) + w_t \tag{2.15}$$

with $X_t, n(s_t), w_t$ 2x1 vectors, and A a 2x2 matrix of coefficients. Note that this parameterization assumes both countries are in the same regime at the same time. The independent money supply model allows countries to be in different regimes at the same time (though there are no monetary interactions between countries). Thus, under the independent

[1] The Markov-switching models were estimated using Hans-Martin Krolzig's MSVAR package for Ox.

money case, there are four states for monetary policy, while under the dependent case there are two states for monetary policy.

2.3 Household Decisions

The household maximizes utility measure (2.9) subject to the trading opportunities and constraints in (2.1)-(2.8) and technology (2.12).

Consider a case of full information, that is, a case in which households and firms have full knowledge of all current period shocks prior to making consumption and investment decisions. Let $V^D(A_t^D, K_t^D, S_t)$ be the value function corresponding to the domestic household's problem. $V(\bullet)$ satisfies the functional equation:

$$V^D(A_t^D, K_t^D, S_t) = \max_{\{N_t^D, K_{t+1}^D, M_{F,t}^D, \tilde{H}_t^D, L_t^D\}} (u(C_t^D, 1-\tilde{H}_t^D) + b_D \int V^D(A_{t+1}^D, K_{t+1}^D, S_{t+1}) \Phi(S_{t+1} \mid S_t))$$

with A_{t+1}^D given by the wealth evolution equation (2.8). Binding cash constraints (2.2)-(2.3) are used to eliminate $C_{D,t}^D$, $C_{F,t}^D$, and H_t^D as separate decisions. Also, from the foreign exchange market allocation (2.1) we have $A_t^D - N_t^D + a_1 W_t^D \tilde{H}_t^D = M_{D,t}^D + e_t M_{F,t}^D$. Consequently, choice of $M_{D,t}^D$ is implied by choices of N_t^D, $\tilde{H}_t^D$, and $M_{F,t}^D$, since A_t^D is predetermined and e_t and W_t^D are taken by the household as given. Optimality conditions for $N_t^D, K_{t+1}^D, M_{F,t}^D, \tilde{H}_t^D, L_t^D$ are:

$$-u_{C_{D,t}^D} \frac{1}{P_t^D} + b_D \int u_{C_{D,t+1}^D} \frac{R_{L,t}^D}{P_{t+1}^D} \Phi(S_{t+1} \mid S_t) = 0 \qquad (2.16)$$

$$-\int\left(u_{C^D_{D,t+1}}\frac{P^D_t}{P^D_{t+1}}+b_D u_{C^D_{D,t+2}}\frac{P^D_{t+1}}{P^D_{t+2}}\{f_{K^D_{t+1}}+1-d^D\}\right)\Phi(S_{t+1}\mid S_t)=0 \tag{2.17}$$

$$-u_{C^D_{D,t}}\frac{1}{P^D_t}+u_{C^D_{F,t}}\frac{1}{e_t P^F_t}=0 \tag{2.18}$$

$$-u_{l^D_t}+u_{C^D_{D,t}}a_1\frac{W^D_t}{P^D_t}-\frac{(1-a_1)}{1+x^D_t}\int b_D\frac{u_{C^D_{D,t+1}}}{P^D_{t+1}}\Phi(S_{t+1}\mid S_t)=0 \tag{2.19}$$

$$f_{H^D_t}-\frac{W^D_t}{P^D_t}R^D_{L,t}=0 \tag{2.20}$$

where $u_{l^D_t}$ is the period t marginal utility of leisure, and the period t marginal products of domestic labor and capital are denoted, respectively, by $f_{H^D_t}$ and $f_{K^D_t}$.

2.4 The Economy's State and Equilibrium

The state of the world economy in period t is characterized by values for $M^D_{s,t}, M^F_{s,t}, k^D_t, k^F_t, A^D_t, A^F_t, K^D_t, K^F_t, b_t$, and S_t. $M^D_{s,t}(M^F_{s,t})$ and $k^D_t(k^F_t)$ are per domestic (foreign) household money and capital stocks. $A^D_t(A^F_t)$ and $K^D_t(K^F_t)$ are the domestic (foreign) representative household's beginning currency and capital stocks. S_t denotes the vector of innovations to money growth in the home and foreign country while b_t denotes belief probabilities over the states of global monetary policy (the belief process is described below).

An equilibrium involves state-contingent prices, wages, interest rates, exchange rates, and optimal household decision rules satisfying market clearing and aggregate consistency conditions. Market clearing conditions are: $\tilde{H}^D_t = H^D_t$, $\tilde{H}^F_t = H^F_t$ for labor; $Y^D_t = C^D_{D,t} + C^F_{D,t} + I^D_t$ and $Y^F_t = C^F_{F,t} + C^D_{F,t} + I^F_t$ for goods; $\tilde{L}^D_t = L^D_t$ and $\tilde{L}^F_t = L^F_t$ for loans;

and $A_t^D + X_t^D = M_{D,t}^D + M_{D,t}^F$ and $A_t^F + X_t^F = M_{F,t}^F + M_{F,t}^D$ for foreign exchange. Aggregate consistency requires that $A_t^D = M_{s,t}^D$, $A_t^F = M_{s,t}^F$ for money stocks, and $K_t^D = k_t^D$, $K_t^F = k_t^F$ for capital stocks.

3. Solving the Model

We solve the model using Christiano's (1991,1998) method of undetermined coefficients. This method is used because it can easily handle models in which different time t endogenous variables are functions of different information sets. Thus, the method can handle cases where the time t deposit decision and/or capital decision is made prior to the realization of the time t money shock. The method involves linearizing equilibrium conditions around steady state, substituting in hypothesized decision rules for endogenous variables, and using the time series representation for the exogenous driving processes to derive a system of nonlinear equations in decision rule coefficients.

We begin by combining the model's first order conditions and substituting for the utility and production functions to get a system of six Euler equations (three for each country) that completely describe the evolution of the economy. Though there is no long-run real growth in the model, the money stock is growing, so wages and prices are nonstationary variables. To render them stationary they are deflated by the level of the money stock. Thus, define $p_t^D = P_t^D / M_t^D$ and $w_t^D = W_t^D / M_t^D$. Finally, define $x_t^D = M_{t+1}^D / M_t^D - 1$. Then, the equilibrium conditions characterizing the domestic economy are given by:

$$\frac{u_{C_{D,t}^D}}{p_t^D} = E_t\left(b_D \frac{u_{C_{D,t+1}^D}}{p_{t+1}^D} \frac{R_{t+1}^D}{1+x_{t+1}^D} \right) \tag{3.1}$$

$$-u_{l_t^D} + \frac{u_{C_{D,t}^D}}{p_t^D} a_1 w_t^D = -E_t\left(b_D \frac{u_{C_{D,t+1}^D}}{p_{t+1}^D} (1-a_1) \frac{w_t^D}{1+x_t^D} \right) \tag{3.2}$$

$$\begin{aligned} &\frac{u_{C_{D,t}^D}}{p_t^D} \frac{w_t^D}{f_{H_t^D}} \left(a_2 \frac{p_t^D}{w_t^D} f_{H_t^D} + (1-a_2) \right) = \\ &E_t\left(b_D \frac{u_{C_{D,t+1}^D}}{p_{t+1}^D} \frac{w_{t+1}^D}{f_{H_{t+1}^D}} \left(f_{K_{t+1}^D} + a_2 \frac{p_{t+1}^D}{w_{t+1}^D} f_{H_{t+1}^D} (1-d^D) + (1-a_2)(1-d^D) \right) \right) \end{aligned} \tag{3.3}$$

The following conditions also hold on the equilibrium path:

$$Y_t^D = (K_t^D)^a (H_t^D)^{(1-a)} \exp((1-a) z_t^D) \tag{3.4}$$

$$I_t^D = K_{t+1}^D - (1-d^D) K_t^D \tag{3.5}$$

$$C_{D,t}^D = q^D (Y_t^D - I_t^D) \tag{3.6}$$

$$C_{F,t}^D = (1-q^F)(Y_t^F - I_t^F) \tag{3.7}$$

$$p_t^D = \frac{1-(1-a_1) N_t^D + a_1 x_t^D}{Y_t^D - (1-a_1 a_2) I_t^D} \tag{3.8}$$

$$w_t^D = \frac{N_t^D + x_t^D - a_2 p_t^D I_t^D}{H_t^D} \tag{3.9}$$

$$R_t^D = (1-a) \frac{p_t^D}{w_t^D} \left(\frac{K_t^D}{H_t^D} \right)^a \exp((1-a) z_t^D) \tag{3.10}$$

An analogous set of equations characterizes the foreign country.

3.1 Exogenous State Variable Evolution

The exogenous state variables for the economy are the shocks to technology and the money growth processes. By deriving an autoregressive representation of the exogenous shock process, we can easily backdate to time t system variables dated t+1 and higher.

Shocks to technology are assumed to follow a first order VAR:

$$\begin{bmatrix} z_t^D \\ z_t^F \end{bmatrix} = \Pi_z \begin{bmatrix} z_{t-1}^D \\ z_{t-1}^F \end{bmatrix} + \begin{bmatrix} u_t^D \\ u_t^F \end{bmatrix} \tag{3.11}$$

with the vector of innovations iid normal with mean zero, and covariance matrix Σ_z.

To derive the AR(1) representation of the money growth process, consider first the evolution of the monetary regime state vector $S_t = [\, s_t, 1 - s_t \,]^T$ where $s_t \in \{0,1\}$. Let $S_t^H = [1,0]^T$ denote the high money growth regime and $S_t^L = [0,1]^T$ denote the low money growth regime state. The regime state vector evolves according to:

$$S_t = PS_{t-1} + v_t \tag{3.12}$$

where the transition matrix P has elements:

$$P = \begin{bmatrix} p_{hh} & p_{lh} \\ p_{hl} & p_{ll} \end{bmatrix}$$

and p_{ij} is as defined in (2.14). Since S_t is discrete, the innovation vector can take on four values:

$$\begin{bmatrix} p_{hl} \\ -p_{hl} \end{bmatrix} = S_t^H - PS_{t-1}^H$$

$$\begin{bmatrix} p_{ll} \\ -p_{ll} \end{bmatrix} = S_t^H - PS_{t-1}^L$$

$$\begin{bmatrix} -p_{lh} \\ p_{lh} \end{bmatrix} = S_t^L - PS_{t-1}^L$$

$$\begin{bmatrix} -p_{hh} \\ p_{hh} \end{bmatrix} = S_t^L - PS_{t-1}^H$$

Define the vector of mean long-run money growth rates $z = [m^H, m^L]$. For the univariate case, we can write the money growth rate process as:

$$x_t - z\,S_t = r\,(x_{t-1} - zS_{t-1}) + e_t \tag{3.13}$$

which in AR(1) form is:

$$\begin{bmatrix} x_t \\ S_t \end{bmatrix} = \begin{bmatrix} r & z\,P - r\,z \\ 0 & P \end{bmatrix} \begin{bmatrix} x_{t-1} \\ S_{t-1} \end{bmatrix} + \begin{bmatrix} e_t + z\,v_t \\ v_t \end{bmatrix} \tag{3.14}$$

The extension to the case where x_t is a 2x1 vector is straightforward.

3.2 Steady State

To implement the method of undetermined coefficients, we need to linearize equilibrium conditions around steady state values. The steady state for the model can be difficult to calculate numerically, so we provide a method to calculate steady state capital and hours analytically. Once steady state capital and hours are known, steady state values for the other model variables can easily be found.

Let the capital/labor ratio be denoted by V = K/H. Use equations (2.20), (3.1), and (3.3) to solve for V as a function of parameters and the exogenous mean money growth rate $\overline{x}$:

$$V = \left[\frac{1}{a} \left(\frac{1}{b} - (1-d) \right) \left(\frac{a_2(1+\bar{x})}{b} + (1-a_2) \right) \right]^{\frac{1}{a-1}} \tag{3.15}$$

From (3.2) we can derive an expression for C_D^D, and H as a function of parameters, exogenous variables, and V:

$$\frac{(1-g)C_D^D}{q^D g(1-H)} = \frac{b}{1+\bar{x}} f_H \left(a_1 + \frac{b(1-a_1)}{1+\bar{x}} \right) \tag{3.16}$$

Note that (3.6) gives $C_D^D = H(V^a - dV)$. Substitute this into (3.16) and solve for H as a function of V. Finally, solve for K as H*V. Given steady state K and H, we can then solve for steady state values of all other model variables using (3.4)-(3.10).

The steady state value of money growth used to calculate steady state capital and hours is the mean value of money growth calculated using the ergodic probabilities of the Markov chain for the regime evolution process. The ergodic probabilities are given by the eigenvector of P associated with the unit eigenvalue.

3.3 System Solution

We solve the model by substituting equations (3.4)-(3.10) into the Euler equations (3.1)-(3.3) to get a system of equations in capital, hours, and deposits. Expectations are dropped and Euler equations are linearized around steady state values to get a prototypical domestic economy equation of the form:

$$l_1\tilde{K}^D_{t+2} + l_2\tilde{K}^F_{t+2} + l_3\tilde{K}^D_{t+1} + l_4\tilde{K}^F_{t+1} + l_5\tilde{H}^D_{t+1} + l_6\tilde{H}^F_{t+1} + l_7\tilde{N}^D_{t+1} + l_8\tilde{N}^F_{t+1} +$$
$$l_9\tilde{K}^D_t + l_{10}\tilde{K}^F_t + l_{11}\tilde{H}^D_t + l_{12}\tilde{H}^F_t + l_{13}\tilde{N}^D_t + l_{14}\tilde{N}^F_t + l_{15}\tilde{z}^D_{t+1} + l_{16}\tilde{z}^F_{t+1} + \quad (3.17)$$
$$l_{17}\tilde{x}^D_{t+1} + l_{18}\tilde{x}^F_{t+1} + l_{19}\tilde{z}^D_t + l_{20}\tilde{z}^F_t + l_{21}\tilde{x}^D_t + l_{22}\tilde{x}^F_t = 0$$

where $\tilde{X}^D_t \equiv X^D_t - X^D_{ss}$ and l_i is a linearization coefficient. To ease the implementation, we calculate the linearization coefficients numerically, though they can be computed analytically. Next, we postulate decision rules for $K_t, H_t,$ and N_t. Let t_t denote the vector of exogenous state variables, including S_t, and let $\tilde{t}_t$ denote the state vector deviations from mean values. We write the hypothesized decision rules as:

$$\tilde{K}^i_{t+1} = h^i_{K1}\tilde{K}^D_t + h^i_{K2}\tilde{K}^F_t + h^i_{K3}\tilde{t}_t + h^i_{K4}\tilde{t}_{t-1} \quad (3.18)$$

$$\tilde{H}^i_t = h^i_{H1}\tilde{K}^D_t + h^i_{H2}\tilde{K}^F_t + h^i_{H3}\tilde{t}_t + h^i_{H4}\tilde{t}_{t-1} \quad (3.19)$$

$$\tilde{N}^i_t = h^i_{N1}\tilde{K}^D_t + h^i_{N2}\tilde{K}^F_t + h^i_{N3}\tilde{t}_{t-1} \quad (3.20)$$

Note that time t decisions on deposits are made prior to the realization of time t shocks. By setting h^i_{K3} to zero we can also examine a case where both capital and deposit decisions are made prior to observing current period shocks. We experimented with both timing conventions in our analysis of the model and generally found that nominal interest rates were too variable in the case where only deposit decisions are made prior to current period shock realizations. Thus, in our simulations we focus on the case where both capital and deposits are sluggish (we set h^i_{K3} to zero).

The decision rules (3.18)-(3.20) are substituted into the linearized Euler equations (3.17) and variables dated higher than period t are backdated using the AR(1) model for the exogenous state variables. For the capital Euler equation (3.3) and the deposits Euler equation

(3.1), we collect terms in $\tilde{K}_t^D, \tilde{K}_t^F,$ and $\tilde{t}_{t-1}$, since households make decisions on these variables prior to observing the current period shock. For the hours Euler equation (3.2), we collect terms in $\tilde{K}_t^D, \tilde{K}_t^F, \tilde{t}_t,$ and $\tilde{t}_{t-1}$, since households make hours decisions after observing current period shocks. For the linearized Euler equations to hold for all values of $\tilde{K}_t^D, \tilde{K}_t^D, \tilde{t}_{t-1}$ and $\tilde{t}_t$, the coefficients on these terms must be equal to zero. Thus, setting the coefficient expressions on $\tilde{K}_t^D, \tilde{K}_t^F, \tilde{t}_t,$ and $\tilde{t}_{t-1}$ to zero gives a system of nonlinear equations in the coefficients of (3.18)-(3.20). This equation system can be solved using any one of several standard nonlinear equation solvers. The system of equations is not too nonlinear, so typically the problem is well behaved and solves quickly. See the computer code in the Appendix and Christiano (1991) for details. Note that the recursive substitution of the decision rules into the linearized Euler equations can quickly become complicated and tedious. Christiano (1998) has a matrix representation of the system that is easier to work with. Alternatively, one can use a symbolic algebra package such as MuPad or Maple to generate the substitutions and derive the nonlinear equation system. That is the route we followed. The computer code in the Appendix has the explicit nonlinear system for the model worked out.

4. Simulating the Model

To simulate the model, one must generate time-series realizations for the exogenous variables, use the decision rules to generate time series on capital, hours, and deposits, and then use equilibrium conditions to generate time series on the other model variables. This is completely straightforward except, perhaps, for generating a Markov-switching money growth rate

process. Note also that up until now we haven't worried about beliefs agents may hold over the state of monetary policy. Beliefs come into play at the model simulation stage. Basically, we generate a sequence of beliefs conditioned on realizations of money growth rates and substitute these beliefs about the state of monetary policy for values of S_t in the decision rules.

Generating the technology shocks for the model is completely straightforward and will not be discussed further. We first generate a time series on money growth with Markov-switching means and variances for the case where money growth is independent across countries. The extension to the case where there is monetary feedback between countries is a straightforward extension.

4.1 Generating a Markov-Switching Money Growth Sequence

Our method of generating money growth sequences is as follows. We set up two gaussian random number generators to generate monetary control errors. Both generators return mean zero variables; however, one sequence is high standard deviation and one is low standard deviation. We allow for both state-dependent means and variances in the model. Let *eta* be a 2x1 vector that holds the regime indicators. For a high-growth regime *eta* = [1, 0] and for a low growth regime *eta* = [0,1]. Suppose we want to generate a sequence of N money growth rate realizations.

Let x be an Nx1 vector of uniform random numbers between zero and one. Let p11 denote the probability of a high-growth regime to a high-growth regime transition, p21 the probability of a low-growth regime to high-growth regime transition, and p22 the probability of

a low-growth regime to low-growth regime transition. Suppose we start in a high money growth regime. The probability of staying in a high-growth regime is p11. We draw x[0] and see if it is less than p11. If so, we set the state indicator to high growth, set mean money growth to high growth, and draw a monetary control error from the high-growth regime distribution. If x[0] is greater than p11, we set the state indicator to low growth, set mean money growth to low growth, and draw a monetary control error from the low-growth regime distribution. Given the innovations and means, we can use (3.14) to generate a money growth rate observation. This sequence of steps then continues N times. An algorithm to do this follows. It is a modified extract from the computer code listed in the Appendix. Note that array indicies are zero offset.

Define v11 = [p12, -p12], v12 = [-p11, p11], v21 = [p22, -p22], and v22 = [-p21, p21]. The variable *eta* is a 2x1 array, and x is an Nx1 array. Recall that the form of the money growth rate process is given by:

$$x_t = m(s_t) + \Psi x_{t-1} - \Psi m(s_{t-1}) + e_t$$

```
#  This pseudo-code generates a Markov-switching money growth rate sequence.
#  The money growth rate is the variable mu_hat

for i in range(N):                    # Do the following steps N times
   if eta[0] == 1 and x[i] <= p11 :   # A high-to-high transition
     v = v11
     e = gauss(0.0,std_high)
     mean = mu_high
   elif eta[0] == 1 and x[i] > p11 :  # A high-to-low transition
     v = v12
     e = gauss(0.0,std_low)
     mean = mu_low
   elif eta[0] == 0 and x[i] <= p21 : # A low-to-high transition
     v = v21
     e = gauss(0.0,std_high)
     mean = mu_high
   elif eta[0] == 0 and x[i] > p21 :  # A low-to-low transition
     v = v22
     e = gauss(0.0,std_low)
     mean = mu_low
```

```
   eta = P*eta + v    # update the state indicator vector

# generate the money growth rate. Meanl is initially set prior to the start of the for loop
#
   mu_hat = mean + Psi*mu_hat - Psi*meanl + e
   meanl = mean                                  # save the mean, since next period it becomes m(s_{t-1})

   muVec[i,0:2] = eta       # save the state indicator matrix
   muVec[i,2] = mu_hat      # save the money growth rate observation

end for
end program
```

For the case in which there is monetary feedback between countries, the algorithm listed above is essentially the same. We draw two innovations from a bivariate distribution with low or high variance-covariance matrix. In addition, the means are 2x1 vectors of high or low values. Finally, *Psi* is replaced by a 2x2 matrix of coefficients.

4.2 Rational Learning

Given realizations on money growth rates, agents who are not fully informed about the monetary regime will have to form inferences about the state of monetary policy. Our introduction of learning into the model framework follows Andolfatto and Gomme (1997). If monetary policymakers are not credible, agents will not know with certainty the monetary policy regime. Instead, they use all relevant information at their disposal to learn about the long-run money growth rate. We model monetary policy as an exogenous process, so the only information relevant for determining the policy regime is current and past values of money growth. Let $\Omega_t = \{x_t, x_{t-1}, x_{t-2}, \ldots\}$ denote the information set consisting of current and past values of money growth. Agents assign a probability to the current-period regime being low growth, which we

denote $b_t = \Pr[m(s_t) = m^L \mid \Omega_t]$. Agents enter period t with beliefs b_{t-1}, observe Ω_t, and update their current-period beliefs b_t. Agents are assumed to share the same beliefs and b_0 is given. Belief updating is rational, following a Bayes' rule recursion:

$$b_t = \frac{g_L(b_{t-1}, x_t)}{g_L(b_{t-1}, x_t) + g_H(b_{t-1}, x_t)} \tag{4.1}$$

with:

$$\begin{aligned} g_L &= b_{t-1} p_{ll} f_{ll}(e_t^{ll}) + (1 - b_{t-1}) p_{hl} f_{hl}(e_t^{hl}) \\ g_H &= b_{t-1} p_{lh} f_{lh}(e_t^{lh}) + (1 - b_{t-1}) p_{hh} f_{hh}(e_t^{hh}) \end{aligned} \tag{4.2}$$

and:

$$e_t^{ij} = (x_t - m^j) - r(x_{t-1} - m^i) \tag{4.3}$$

with $f_{ij}(e_t^{ij})$ the normal pdf for e_t^{ij}. As written, we have a recursion that generates a belief sequence for a two-state economy given an initial value for beliefs b_0 and a time series on money growth rates $\{x_t\}$. For the case where monetary policy is independent across countries, there are four states for monetary policy: two states in each of two countries. Assuming independence, the transition probabilities for the global state are products of individual country transition probabilities. Denote the product of the transition probability and density function for a transition from state i to state j for country k by:

$$j_{ij}^{k} = p_{ij}^{k} f_{ij}^{k}(e_t^{k,ij}) \tag{4.4}$$

Then:

$$g_{HH} = b_{t-1}^{HH} j_{HH}^{D} j_{HH}^{F} + b_{t-1}^{HL} j_{HH}^{D} j_{LH}^{F} + b_{t-1}^{LH} j_{LH}^{D} j_{HH}^{F} + b_{t-1}^{LL} j_{LH}^{D} j_{LH}^{F} \tag{4.5}$$

$$g_{HL} = b_{t-1}^{HH} j_{HH}^{D} j_{HL}^{F} + b_{t-1}^{HL} j_{HH}^{D} j_{LL}^{F} + b_{t-1}^{LH} j_{LH}^{D} j_{HL}^{F} + b_{t-1}^{LL} j_{LH}^{D} j_{LL}^{F} \tag{4.6}$$

$$g_{LH} = b_{t-1}^{HH} j_{HL}^{D} j_{HH}^{F} + b_{t-1}^{HL} j_{HL}^{D} j_{LH}^{F} + b_{t-1}^{LH} j_{LL}^{D} j_{HH}^{F} + b_{t-1}^{LL} j_{LL}^{D} j_{LH}^{F} \tag{4.7}$$

$$g_{LL} = b_{t-1}^{HH} j_{HL}^{D} j_{HL}^{F} + b_{t-1}^{HL} j_{HL}^{D} j_{LL}^{F} + b_{t-1}^{LH} j_{LL}^{D} j_{HL}^{F} + b_{t-1}^{LL} j_{LL}^{D} j_{LL}^{F} \tag{4.8}$$

and:

$$b_t^{ij} = \frac{g_{ij}}{g_{HH} + g_{HL} + g_{LH} + g_{LL}} \tag{4.9}$$

with b_t^{ij} denoting the belief that the home country is in regime i and the foreign country is in regime j.

Given belief sequences, we can then simulate the model under two different assumptions. One case is full information in which there is no uncertainty about the state of monetary policy and the model is simulated using the process in (3.14). The other is a case where there is uncertainty about the state of monetary policy. In that case households do not know S_t. Instead, the model is simulated with S_t being replaced by the belief sequence, b_t, in the decision rules, though the money growth sequence itself is generated using (3.14).

The parameterization of the belief process potentially has implications for how model variables respond to different types of monetary shocks. If beliefs adjust slowly, households may adjust slowly to monetary shocks. We can examine how rapidly beliefs adjust as a function of the process parameterization using an argument in Moran (1997). Suppose the economy has been in a high money growth state for some time, and the monetary authority switches policy to low growth. Let $f(m(s_t) \mid m^i)$ denote the pdf of $m(s_t)$ if households believe the regime at time t is m^i. The odds ratio is given by:

$$r_t = \frac{f(m(s_t) \mid m^L)}{f(m(s_t) \mid m^H)}$$

and the belief that the regime is low can be expressed in terms of the odds ratio:

$$b_t = \frac{b_{t-1} r_t}{b_{t-1} r_t + (1 - b_{t-1})} \tag{4.10}$$

with $\partial b_t / \partial r_t > 0$. The distribution of the odds ratio is:

$$\frac{s_H}{s_L} \exp\left(0.5 \left(\frac{-e_t^2}{s_L^2} + \frac{(e_t + m^L - m^H)^2}{s_H^2} \right) \right) \tag{4.11}$$

Setting the disturbances e_t to zero:

$$= \frac{s_H}{s_L} \left(0.5 \left(\frac{m^L - m^H}{s_H} \right)^2 \right) \tag{4.12}$$

We see then that:

- If the spread between means in the high and low state is larger, beliefs converge more quickly: it is easier for agents to distinguish a regime shift after it has occurred.
- As s_L increases, the convergence is slower. With a wide distribution about the new mean, realized growth rates around the low-growth mean will be given less weight. The effect of an increase in s_H depends on its size. If $s_H > (m^L - m^H)$ convergence is faster; otherwise, it is slower.

References

Andolfatto, D. and P. Gomme (1997), "Monetary Policy Regimes and Beliefs," Center for Research on Economic Fluctuations and Employment Working Paper No. 48.

Christiano, L.C. (1991), "Modeling the Liquidity Effect of a Money Shock," Federal Reserve Bank of Minneapolis *Quarterly Review*, Winter: 3-34.

Christiano, L.C. (1998), "Solving Dynamic Equilibrium Models by a Method of Undetermined Coefficients," Manuscript, Northwestern University.

Hamilton, J.D. (1989), "A New Approach to the Economic Analysis of Nonstationary Time Series and the Business Cycle," *Econometrica*, 57(2): 357-384.

Moran, K. (1997), "Bayesian Updating and Welfare Costs of the Transition to a Lower Inflation Rate," Manuscript, University of Rochester.

Schlagenhauf, D. and J. Wrase (1995), "Liquidity and Real Activity in a Simple Open Economy Model," *Journal of Monetary Economics*, 35: 432-462.

Sill, K. and J. Wrase (1999), "Exchange Rates, Monetary Policy Regimes, and Beliefs," Philadelphia Federal Reserve Bank Working Paper No. 99-6.

Sill, K. and J. Wrase (1999), "Exchange Rates and Monetary Policy Regimes in the U.S. and Canada," Philadelphia Federal Reserve Bank Working Paper No. 99-13.

Appendix

The following programs were used to solve and simulate the model in our paper "Exchange Rates and Monetary Policy Regimes in the U.S. and Canada." The programs are written in Python, which has a syntax that is close to pseudo-code. Thus, the programs should be fairly easy to follow and, to that extent, should clear up ambiguities in the text discussion. There is some duplication across programs in the functions called—but each program is self-contained. The program pvalues.py sets parameter values and estimates for the model driving processes. The program solvemodel.py solves two versions of the model: one with sluggish capital and sluggish deposits and one with sluggish deposits only. The program msvcap.py simulates the model. Details on obtaining and installing Python can be found at: http://www.python.org.

```
#


# Program pvalues.py
#
# Set model parameters for 2-country model that assumes independent

# Markov switching models for each country: US and Canada.

#  Setting a1 and a3 automatically sets the output file names for the
#  appropriate solution vector

p_a1 = 1.0             # how much wage bill gets sent to FX market
p_a3 = 0.0              # how much investment is financed by borrowing

p_alpha  = 0.36         # capital share
p_beta   = 0.99         # discount factor
p_delta  = 0.025        # depreciation rate
p_gamma  = 0.24          # consumption utility share
p_Psi    = -1.0         # crra
p_thd    = 0.5         # home share in consumption aggregator
p_thf    = 0.5         # foreign share in consumption aggregator

p_zrhod = 0.987         # AR coefficient on home technology AR(1) process
p_zrhof = 0.976         # AR coefficient on foreign technology AR(1) process

p_zsigd = 0.0134
p_zsigf = 0.0127

#  The regime switching process parameters for money growth are estimated
#  using the msvar ox package by Krolzig (1998).  The parameters are estimated
#  using quarterly data on US and Canada reserve money over the sample period
#  19xx:x to 19xx:x

import Numeric

p_plow  = .98927           # prob of low to low transition
p_phigh = .96872           # prob of high to high transition

p_mMeans = Numeric.array( [ [ 1.7656e-2, 1.1365e-2 ],
                            [ 2.6279e-2, 0.7918e-2] ])

p_mSigLow = Numeric.array([ [ 0.23394e-4,   -0.005687e-4],
                            [ -0.005687e-4,  1.0188e-4 ] ] )
p_mSigHigh = p_mSigLow

p_mCoeffs = Numeric.array( [ [ 0.73987, 0.024559 ],
                             [ -0.02685, 0.23105 ] ] )


# Program solvemodel.py
#

# Solve a two-country monetary model with rational learning.

# The information structure is that of a Lucas-Fuerst model.

# Two solution cases are programmed: a model with sluggish deposits
# only, and a model with both capital and deposits sluggish.
#
# Money driving process is a two state markov switching VAR.
# with a constant covariance matrix.
# This program is written in Python, available for download at
#
#                   http://www.python.org
#
# The routine makes use of the following extension modules:
#
#                   Numeric Python (NumPy)
#                   Multipack-0.6 (http://oliphant.netpedia.net)
#
# These two modules are linked to the python web site.  Both are
# availabel for a wide variety of platforms, including NT and Linux.
# This program runs under Linux and NT (provided you have the modules!).
# Make sure the module files are in your Python path.
#
#
# Keith Sill
# FRB Philadelphia
# keith.sill@phil.frb.org
#
# Last updated 8/23/99
#

import Numeric
from pvalues import *

# Define a bunch of functions that will be called by the main program

def savem(name,m):
  #
  # Save a matrix in ascii format
  #
  import sys

  fp = open(name,'w')

  for i in range(m.shape[0]):
    for j in range(m.shape[1]):
      fp.write(str(m[i,j]) + '\t')
    fp.write('\n')

  fp.close()



def steadyState( mud, muf ):

  # takes as input the money growth rates in the home and foreign
  # countries.  Returns a 6-tuple of calculated steady states for
  # capital, hours, and deposits.

  xx = Numeric.zeros(6,'d')

  for i in range(2):
      vt1 = ( p_a3 * (1+mud) / p_beta ) * ( 1.0 / p_beta - (1-p_delta) )
      vt2 =  (1-p_a3) * ( 1/p_beta - (1-p_delta) )
      vd =  ( 1/p_alpha * ( vt1 + vt2 ) )**( 1.0/(p_alpha-1) )
      tempd = ( p_gamma/(1-p_gamma) )*( p_beta/(1+mud) ) * ( (1-
p_alpha)*vd**p_alpha /
            (vd**p_alpha - p_delta*vd)  )* ( p_a1 + (1-p_a1) * p_beta / ( 1+mud )
)

      hd = tempd / ( 1+tempd )
      kd = hd * vd

      yd = kd**p_alpha * hd**(1-p_alpha)
      zd = ( p_a1 * (1-p_alpha) * yd * p_beta / ( 1+mud ) - ( yd- p_delta * kd )
) / (
            (1-p_alpha) * p_beta * yd / ( 1+mud ) + p_a3 * p_delta * kd )
      nd = ( 1 + mud * zd ) / ( 1 - zd )

      xx[i] = kd
      xx[i+2] = hd
      xx[i+4] = nd

      mud = muf

  return xx
```

```
def transition_probs( p_plow,p_phigh ):
  #
  # The ordering of the input sequence is important. It should be
  # home high-to-high, home low-to-low, foreign high-to-high, and
  # foreign low-to-low
  #

  P = Numeric.zeros(4,'d')

  P[0]  = p_plow
  P[1]  = 1-p_phigh
  P[2]  = 1-p_plow
  P[3]  = p_phigh

  return Numeric.reshape( P, (2,2) )


def euler1d( x ):
  #
  # Euler equation for consumption evolution (home country):
  #
  #         Uc / p = p_beta * Uc' / p' * (1+R) / (1+x)
  #

  ([kdpp,kfpp,kdp,kfp,hdp,hfp,ndp,nfp,kd,kf,hd,hf,nd,nf,
   zdp,zfp,mudp,mufp,zd,zf,mud,muf]) = x

  yd = kd**p_alpha * hd**(1-p_alpha) * Numeric.exp((1-p_alpha)*zd)
  yf = kf**p_alpha * hf**(1-p_alpha) * Numeric.exp((1-p_alpha)*zf)

  ydp = kdp**p_alpha * hdp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zdp)
  yfp = kfp**p_alpha * hfp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zfp)

  Id = kdp - (1-p_delta) * kd
  If = kfp - (1-p_delta) * kf

  Idp = kdpp - (1-p_delta) * kdp
  Ifp = kfpp - (1-p_delta) * kfp

  cdd = p_thd * ( yd - Id )
  cdf = (1-p_thf) * ( yf - If )

  cddp = p_thd * ( ydp - Idp )
  cdfp = (1-p_thf) * ( yfp - Ifp )

  pd  = ( 1 - nd * (1-p_a1) + p_a1 * mud ) / ( yd - (1-p_a1*p_a3) * Id )
  pdp = ( 1 - ndp * (1-p_a1) + p_a1 * mudp ) / ( ydp - (1-p_a1*p_a3) * Idp )

  wd  = ( nd + mud - p_a3 * pd * Id ) / hd

  R = ( pd / wd ) * (1-p_alpha) * ( kd / hd )**p_alpha * Numeric.exp((1-
p_alpha)*zd)


  muc = ( p_thd * p_gamma * cdd**(p_thd*p_gamma*p_Psi-1) * cdf**((1-
p_thd)*p_gamma*p_Psi) *
          (1-hd)**((1-p_gamma)*p_Psi) ) / pd

  mucp = ( p_thd * p_gamma * cddp**(p_thd*p_gamma*p_Psi-1) * cdfp**((1-
p_thd)*p_gamma*p_Psi) *
          (1-hdp)**((1-p_gamma)*p_Psi) ) / pdp


  return   -muc + p_beta * R * mucp / (1+mud)

def euler2d( x ):
  #
  # Euler equation for hours evolution (home country):
  #
  #     -U_l + Uc * p_a1 * w/p + p_beta * Uc'/p' * (1-p_a1) * w / (1+x)
  #

  ([kdpp,kfpp,kdp,kfp,hdp,hfp,ndp,nfp,kd,kf,hd,hf,nd,nf,
   zdp,zfp,mudp,mufp,zd,zf,mud,muf]) = x

  yd = kd**p_alpha * hd**(1-p_alpha) * Numeric.exp((1-p_alpha)*zd)
  yf = kf**p_alpha * hf**(1-p_alpha) * Numeric.exp((1-p_alpha)*zf)

  ydp = kdp**p_alpha * hdp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zdp)
  yfp = kfp**p_alpha * hfp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zfp)

  Id = kdp - (1-p_delta) * kd
  If = kfp - (1-p_delta) * kf

  Idp = kdpp - (1-p_delta) * kdp
  Ifp = kfpp - (1-p_delta) * kfp

  cdd = p_thd * ( yd - Id )
  cdf = (1-p_thf) * ( yf - If )

  cddp = p_thd * ( ydp - Idp )
  cdfp = (1-p_thf) * ( yfp - Ifp )

  pd  = ( 1 - nd * (1-p_a1) + p_a1 * mud ) / ( yd - (1-p_a1*p_a3) * Id )
  pdp = ( 1 - ndp * (1-p_a1) + p_a1 * mudp ) / ( ydp - (1-p_a1*p_a3) * Idp )

  wd  = ( nd + mud - p_a3 * pd * Id ) / hd

  mul = (1-p_gamma) * ( cdd**(p_thd*p_gamma*p_Psi) * cdf**((1-
p_thd)*p_gamma*p_Psi) *
                        (1-hd)**((1-p_gamma)*p_Psi-1) )

  muc = ( p_thd * p_gamma * cdd**(p_thd*p_gamma*p_Psi-1) * cdf**((1-
p_thd)*p_gamma*p_Psi) *
          (1-hd)**((1-p_gamma)*p_Psi) ) / pd

  mucp = ( p_thd * p_gamma * cddp**(p_thd*p_gamma*p_Psi-1) * cdfp**((1-
p_thd)*p_gamma*p_Psi) *
          (1-hdp)**((1-p_gamma)*p_Psi) ) / pdp

  return -mul + muc * p_a1 * wd + p_beta * mucp * (1-p_a1) * wd / (1+mud)

def euler3d( x ):
  #
  # Euler equation for capital evolution (home country):
  #
  #    -Uc/p * (w/p) * (p/fh) + p_beta* Uc'/p' * (w'/p') * (p'/fh') *
  #     ( fk' + p_a3 * (p'/w') * fh' * (1-p_delta) + (1-p_a3)*(1-p_delta) )
  #

  ([kdpp,kfpp,kdp,kfp,hdp,hfp,ndp,nfp,kd,kf,hd,hf,nd,nf,
   zdp,zfp,mudp,mufp,zd,zf,mud,muf]) = x

  yd = kd**p_alpha * hd**(1-p_alpha) * Numeric.exp((1-p_alpha)*zd)
  yf = kf**p_alpha * hf**(1-p_alpha) * Numeric.exp((1-p_alpha)*zf)

  ydp = kdp**p_alpha * hdp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zdp)
  yfp = kfp**p_alpha * hfp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zfp)

  Id = kdp - (1-p_delta) * kd
  If = kfp - (1-p_delta) * kf

  Idp = kdpp - (1-p_delta) * kdp
  Ifp = kfpp - (1-p_delta) * kfp

  cdd = p_thd * ( yd - Id )
  cdf = (1-p_thf) * ( yf - If )

  cddp = p_thd * ( ydp - Idp )
  cdfp = (1-p_thf) * ( yfp - Ifp )

  pd  = ( 1 - nd * (1-p_a1) + p_a1 * mud ) / ( yd - (1-p_a1*p_a3) * Id )
  pdp = ( 1 - ndp * (1-p_a1) + p_a1 * mudp ) / ( ydp - (1-p_a1*p_a3) * Idp )
```

```
  wd  = ( nd + mud - p_a3 * pd * Id ) / hd
  wdp = ( ndp + mudp - p_a3 * pdp * Idp ) / hdp

  fh  = (1-p_alpha) * ( kd / hd )**p_alpha * Numeric.exp((1-p_alpha)*zd)
  fhp = (1-p_alpha) * ( kdp / hdp )**p_alpha * Numeric.exp((1-p_alpha)*zdp)

  fkp = p_alpha * ( kdp / hdp )**(p_alpha-1) * Numeric.exp((1-p_alpha)*zdp)

  muc = ( p_thd * p_gamma * cdd**(p_thd*p_gamma*p_Psi-1) * cdf**((1-
p_thd)*p_gamma*p_Psi) *
          (1-hd)**((1-p_gamma)*p_Psi) ) / pd

  mucp = ( p_thd * p_gamma * cddp**(p_thd*p_gamma*p_Psi-1) * cdfp**((1-
p_thd)*p_gamma*p_Psi) *
          (1-hdp)**((1-p_gamma)*p_Psi) ) / pdp

  term1 = -muc * (wd / fh) * ( p_a3 * (pd/wd) * fh + (1-p_a3) )
  term2 = p_beta * mucp * (wdp / fhp) * ( fkp + p_a3 * (pdp / wdp) * fhp *
                                          (1-p_delta) + (1-p_a3) * (1-p_delta) )

  return term1 + term2


def euler1f( x ):
  #
  # Euler equation for consumption evolution (foreign country):
  #
  #         Uc / p = p_beta * Uc' / p' * (1+R) / (1+x)
  #

  ([kdpp,kfpp,kdp,kfp,hdp,hfp,ndp,nfp,kd,kf,hd,hf,nd,nf,
   zdp,zfp,mudp,mufp,zd,zf,mud,muf]) = x

  yd = kd**p_alpha * hd**(1-p_alpha) * Numeric.exp((1-p_alpha)*zd)
  yf = kf**p_alpha * hf**(1-p_alpha) * Numeric.exp((1-p_alpha)*zf)

  ydp = kdp**p_alpha * hdp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zdp)
  yfp = kfp**p_alpha * hfp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zfp)

  Id = kdp - (1-p_delta) * kd
  If = kfp - (1-p_delta) * kf

  Idp = kdpp - (1-p_delta) * kdp
  Ifp = kfpp - (1-p_delta) * kfp

  cff = p_thf * ( yf - If )
  cfd = (1-p_thd) * ( yd - Id )

  cffp = p_thf * ( yfp - Ifp )
  cfdp = (1-p_thd) * ( ydp - Idp )

  pf  = ( 1 - nf * (1-p_a1) + p_a1 * muf ) / ( yf - (1-p_a1*p_a3) * If )
  pfp = ( 1 - nfp * (1-p_a1) + p_a1 * mufp ) / ( yfp - (1-p_a1*p_a3) * Ifp )

  wf  = ( nf + muf - p_a3 * pf * If ) / hf

  R = ( pf / wf ) * (1-p_alpha) * ( kf / hf )**p_alpha *Numeric.exp((1-
p_alpha)*zf)


  muc = ( p_thf * p_gamma * cff**(p_thf*p_gamma*p_Psi-1) * cfd**((1-
p_thf)*p_gamma*p_Psi) *
          (1-hf)**((1-p_gamma)*p_Psi) ) / pf

  mucp = ( p_thf * p_gamma * cffp**(p_thf*p_gamma*p_Psi-1) * cfdp**((1-
p_thf)*p_gamma*p_Psi) *
          (1-hfp)**((1-p_gamma)*p_Psi) ) / pfp


  return   -muc + p_beta * R * mucp / (1+muf)
```

```
def euler2f( x ):
  #
  # Euler equation for hours evolution (foreign country):
  #
  #     -U_l + Uc * p_a1 * w/p + p_beta * Uc'/p' * (1-p_a1) * w / (1+x)
  #

  ([kdpp,kfpp,kdp,kfp,hdp,hfp,ndp,nfp,kd,kf,hd,hf,nd,nf,
   zdp,zfp,mudp,mufp,zd,zf,mud,muf]) = x

  yd = kd**p_alpha * hd**(1-p_alpha) * Numeric.exp((1-p_alpha)*zd)
  yf = kf**p_alpha * hf**(1-p_alpha) * Numeric.exp((1-p_alpha)*zf)

  ydp = kdp**p_alpha * hdp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zdp)
  yfp = kfp**p_alpha * hfp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zfp)

  Id = kdp - (1-p_delta) * kd
  If = kfp - (1-p_delta) * kf

  Idp = kdpp - (1-p_delta) * kdp
  Ifp = kfpp - (1-p_delta) * kfp

  cff = p_thf * ( yf - If )
  cfd = (1-p_thd) * ( yd - Id )

  cffp = p_thf * ( yfp - Ifp )
  cfdp = (1-p_thd) * ( ydp - Idp )

  pf  = ( 1 - nf * (1-p_a1) + p_a1 * muf ) / ( yf - (1-p_a1*p_a3) * If )
  pfp = ( 1 - nfp * (1-p_a1) + p_a1 * mufp ) / ( yfp - (1-p_a1*p_a3) * Ifp )

  wf  = ( nf + muf - p_a3 * pf * If ) / hf

  mul = (1-p_gamma) * ( cff**(p_thf*p_gamma*p_Psi) * cfd**((1-
p_thf)*p_gamma*p_Psi) *
                        (1-hf)**((1-p_gamma)*p_Psi-1) )

  muc = ( p_thf * p_gamma * cff**(p_thf*p_gamma*p_Psi-1) * cfd**((1-
p_thf)*p_gamma*p_Psi) *
          (1-hf)**((1-p_gamma)*p_Psi) ) / pf

  mucp = ( p_thf * p_gamma * cffp**(p_thf*p_gamma*p_Psi-1) * cfdp**((1-
p_thf)*p_gamma*p_Psi) *
           (1-hfp)**((1-p_gamma)*p_Psi) ) / pfp

  return -mul + muc * p_a1 * wf + p_beta * mucp * (1-p_a1) * wf / (1+muf)

def euler3f( x ):
  #
  # Euler equation for capital evolution (foreign country):
  #
  #    -Uc/p * (w/p) * (p/fh) + p_beta* Uc'/p' * (w'/p') * (p'/fh') *
  #     ( fk' + p_a3 * (p'/w') * fh' * (1-p_delta) + (1-p_a3)*(1-p_delta) )
  #

  ([kdpp,kfpp,kdp,kfp,hdp,hfp,ndp,nfp,kd,kf,hd,hf,nd,nf,
   zdp,zfp,mudp,mufp,zd,zf,mud,muf]) = x

  yd = kd**p_alpha * hd**(1-p_alpha) * Numeric.exp((1-p_alpha)*zd)
  yf = kf**p_alpha * hf**(1-p_alpha) * Numeric.exp((1-p_alpha)*zf)

  ydp = kdp**p_alpha * hdp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zdp)
  yfp = kfp**p_alpha * hfp**(1-p_alpha) * Numeric.exp((1-p_alpha)*zfp)

  Id = kdp - (1-p_delta) * kd
  If = kfp - (1-p_delta) * kf

  Idp = kdpp - (1-p_delta) * kdp
  Ifp = kfpp - (1-p_delta) * kfp

  cff = p_thf * ( yf - If )
  cfd = (1-p_thd) * ( yd - Id )
```

```
  cffp = p_thf * ( yfp - Ifp )
  cfdp = (1-p_thd) * ( ydp - Idp )

  pf  = ( 1 - nf * (1-p_a1) + p_a1 * muf ) / ( yf - (1-p_a1*p_a3) * If )
  pfp = ( 1 - nfp * (1-p_a1) + p_a1 * mufp ) / ( yfp - (1-p_a1*p_a3) * Ifp )

  wf  = ( nf + muf - p_a3 * pf * If ) / hf
  wfp = ( nfp + mufp - p_a3 * pfp * Ifp ) / hfp

  fh  = (1-p_alpha) * ( kf / hf )**p_alpha * Numeric.exp((1-p_alpha)*zf)
  fhp = (1-p_alpha) * ( kfp / hfp )**p_alpha  * Numeric.exp((1-p_alpha)*zfp)

  fkp = p_alpha * ( kfp / hfp )**(p_alpha-1) * Numeric.exp((1-p_alpha)*zfp)

  muc = ( p_thf * p_gamma * cff**(p_thf*p_gamma*p_Psi-1) * cfd**((1-
p_thf)*p_gamma*p_Psi) *
          (1-hf)**((1-p_gamma)*p_Psi) ) / pf

  mucp = ( p_thf * p_gamma * cffp**(p_thf*p_gamma*p_Psi-1) * cfdp**((1-
p_thf)*p_gamma*p_Psi) *
          (1-hfp)**((1-p_gamma)*p_Psi) ) / pfp

  term1 = -muc * (wf / fh) * ( p_a3 * (pf/wf) * fh + (1-p_a3) )
  term2 = p_beta * mucp * (wfp / fhp) * ( fkp + p_a3 * (pfp / wfp) * fhp *
                                    (1-p_delta) + (1-p_a3) * (1-p_delta) )

  return term1 + term2


def gradient( fcn,xbar,typx ):

# Find the gradient of fcn evaluated at xbar. typx[i] is the typical
# size of xbar[i]

  eps = 4.441e-16
  cuberteta = eps**(.332)
  m = len(xbar)

  y = Numeric.zeros( m ,'d' )

  for i in range( m ):

    h = cuberteta * Numeric.maximum( Numeric.absolute(xbar[i]), typx[i] )
    h = h * sign(xbar[i])

    if h == 0:
      h = Numeric.sqrt(eps)

    temp = xbar[i]
    xbar[i] = xbar[i] + h
    fp = fcn( xbar )
    xbar[i] = temp - h
    fm = fcn( xbar )

    y[i] = 0.5*( fp - fm ) / h

    xbar[i] = temp

  return y


def sign( x ):

  if x < 0:
    return -1
  else:
    return 1

def sysmodel1( x ):
  #
  #  Find the solution to the nonlinear system of equations
  #  This set of equations was generated using the symbolic
  #  algebra package MuPAD and is for the case where deposits
  #  only are sluggish.
  #

  neqs = len(x)

  Kd1 = x[0]
  Kd2 = x[1]
  Kdv1 = array([ x[2], x[3], x[4], x[5], x[6], x[7] ])
  Kdv2 = array([ x[8], x[9], x[10], x[11], x[12], x[13] ])

  Kf1 = x[14]
  Kf2 = x[15]
  Kfv1 = array([ x[16], x[17], x[18], x[19], x[20], x[21] ])
  Kfv2 = array([ x[22], x[23], x[24], x[25], x[26], x[27] ])

  Hd1 = x[28]
  Hd2 = x[29]
  Hdv1 = array([ x[30], x[31], x[32], x[33], x[34], x[35] ])
  Hdv2 = array([ x[36], x[37], x[38], x[39], x[40], x[41] ])

  Hf1 = x[42]
  Hf2 = x[43]
  Hfv1 = array([ x[44], x[45], x[46], x[47], x[48], x[49] ])
  Hfv2 = array([ x[50], x[51], x[52], x[53], x[54], x[55] ])

  Nd1 = x[56]
  Nd2 = x[57]
  Ndv2 = array([ x[58], x[59], x[60], x[61], x[62], x[63] ])

  Nf1 = x[64]
  Nf2 = x[65]
  Nfv2 = array([ x[66], x[67], x[68], x[69], x[70], x[71] ])

  work1 = zeros((4,14),'d')
  work2 = zeros((2,8),'d')

  for i in range(4):

    e = jacobian[i,:]
    ev1 = array([ e[14], e[15], e[16], e[17], 0.0, 0.0 ])
    ev2 = array([ e[18], e[19], e[20], e[21], 0.0, 0.0 ])

    y1 = ( ev2 + dot(ev1,Rmat) + Kdv1*e[2] + Kfv1*e[3] + Hdv1*e[10] +
           Hfv1*e[11] + (Ndv2 + Nd1*Kdv1 + Nd2*Kfv1)*e[6] +
                  (Nfv2 + Nf1*Kdv1 + Nf2*Kfv1)*e[7] +
         (Hdv2 + dot(Hdv1,Rmat) + Hd1*Kdv1 + Hd2*Kfv1)*e[4] +
         (Kdv2 + dot(Kdv1,Rmat) + Kd1*Kdv1 + Kd2*Kfv1)*e[0] +
         (Hfv2 + dot(Hfv1,Rmat) + Hf1*Kdv1 + Hf2*Kfv1)*e[5] +
         (Kfv2 + dot(Kfv1,Rmat) + Kf1*Kdv1 + Kf2*Kfv1)*e[1]   )


    y2 = ( Kdv2*e[2] + Kfv2*e[3] + Hdv2*e[10] + Hfv2*e[11] +
           Ndv2*e[12] + Nfv2*e[13] + (Hd1*Kdv2 + Hd2*Kfv2)*e[4] +
                                (Kd1*Kdv2 + Kd2*Kfv2)*e[0] +
                                (Hf1*Kdv2 + Hf2*Kfv2)*e[5] +
                                (Kf1*Kdv2 + Kf2*Kfv2)*e[1] +
                                (Nd1*Kdv2 + Nd2*Kfv2)*e[6] +
                                (Nf1*Kdv2 + Nf2*Kfv2)*e[7]   )

    y3 = ( e[8] + Kd1*e[2] + Kf1*e[3] + Hd1*e[10] +
           Hf1*e[11] + Nd1*e[12] + Nf1*e[13] +
          (Hd1*Kd1 + Hd2*Kf1)*e[4] + (Hf1*Kd1 + Hf2*Kf1)*e[5] +
          (Kd1*Kf1 + Kf1*Kf2)*e[1] + (Kd1*Nd1 + Kf1*Nd2)*e[6] +
          (Kd1*Nf1 + Kf1*Nf2)*e[7] + e[0]*(Kd2*Kf1 + Kd1*Kd1 )  )

    y4 = ( e[9] + Kd2*e[2] + Kf2*e[3] + Hd2*e[10] +
           Hf2*e[11] + Nd2*e[12] + Nf2*e[13] +
         (Hd1*Kd2 + Hd2*Kf2)*e[4] + (Kd1*Kd2 + Kd2*Kf2)*e[0] +
         (Hf1*Kd2 + Hf2*Kf2)*e[5] + (Kd2*Nd1 + Kf2*Nd2)*e[6] +
         (Kd2*Nf1 + Kf2*Nf2)*e[7] + e[1]*(Kd2*Kf1 + Kf2*Kf2 )  )
```

```
    temp1 = array([y3,y4])
    work1[i,:] = concatenate((y1,y2,temp1),1)

  R2 = dot(Rmat,Rmat)

  for i in range(2):

    e = jacobian[4+i,:]
    ev1 = array([ e[14], e[15], e[16], e[17], 0.0, 0.0 ])
    ev2 = array([ e[18], e[19], e[20], e[21], 0.0, 0.0 ])

    y5 = ( e[8] + Kd1*e[2] + Kf1* e[3] + Hd1*e[10] + Hf1*e[11] +
           Nd1*e[12] + Nf1*e[13] + (Hd1*Kd1 + Hd2*Kf1)*e[4] +
         (Hf1*Kd1 + Hf2*Kf1)*e[5] + (Kd1*Kf1 + Kf1*Kf2)*e[1] +
         (Kd1*Nd1 + Kf1*Nd2)*e[6] + (Kd1*Nf1 + Kf1*Nf2)*e[7] +
          e[0]*( Kd2*Kf1 + Kd1*Kd1 )  )

    y6 = ( e[9] + Kd2*e[2] + Kf2*e[3] + Hd2*e[10] + Hf2*e[11] +
           Nd2*e[12] + Nf2*e[13] + ( Hd1*Kd2 + Hd2*Kf2 )*e[4] +
         ( Kd1*Kd2 + Kd2*Kf2 )*e[0] + ( Hf1*Kd2 + Hf2*Kf2 )*e[5] +
         ( Kd2*Nd1 + Kf2*Nd2 )*e[6] + ( Kd2*Nf1 + Kf2*Nf2 )*e[7] +
          e[1]*( Kd2*Kf1 + Kf2*Kf2 )  )

    t1 = ( dot(ev2,Rmat) + Ndv2*e[12] + Nfv2*e[13] +
           dot(ev1,R2) + (Kdv2 + dot(Kdv1,Rmat))*e[2] +
         (Hdv2 + dot(Hdv1,Rmat) )*e[10] +
        ( Kfv2 + dot(Kfv1,Rmat) )*e[3] +
        ( Hfv2 + dot(Hfv1,Rmat) )*e[11]   )

    t2 = (  ( dot(Ndv2,Rmat) + Nd1*( Kdv2 + dot(Kdv1,Rmat) ) +
                          Nd2*( Kfv2 + dot(Kfv1,Rmat) ) )*e[6] +
           ( dot(Nfv2,Rmat) + Nf1*( Kdv2 + dot(Kdv1,Rmat) ) +
                          Nf2*( Kfv2 + dot(Kfv1,Rmat) )  )*e[7]  )

    t3 = ( e[4]*( dot(Hdv2,Rmat) + Hd1*( Kdv2 + dot(Kdv1,Rmat) ) +
                   Hd2*( Kfv2 + dot(Kfv1,Rmat) ) + dot(Hdv1,R2) ) +
           e[0]*( dot(Kdv2,Rmat) + Kd1*( Kdv2 + dot(Kdv1,Rmat) ) +
                   Kd2*( Kfv2 + dot(Kfv1,Rmat) ) + dot(Kdv1,R2) ) )

    t4 = ( e[5]*( dot(Hfv2,Rmat) + Hf1*( Kdv2 + dot(Kdv1,Rmat) ) +
                  Hf2*( Kfv2 + dot(Kfv1,Rmat) ) + dot(Hfv1,R2) ) +
           e[1]*( dot(Kfv2,Rmat) + Kf1*( Kdv2 + dot(Kdv1,Rmat) ) +
                   Kf2*( Kfv2 + dot(Kfv1,Rmat) ) + dot(Kfv1,R2) ) )

    y7 = t1 + t2 + t3 + t4

    temp2 = array([y5,y6])
    work2[i,:] = concatenate((y7,temp2),1)

  work3 = reshape( work1, (1,56) )
  work4 = reshape( work2, (1,16) )

  fvec = reshape(concatenate((work3, work4),1),(neqs,))

  return fvec


def sysmodel2( x ):
  #
  #  Find the solution to the nonlinear system of equations
  #  This set of equations was generated using the symbolic
  #  algebra package MuPAD and is for the case where deposits

  #  and capital are sluggish.
  #
  neqs = len(x)

  Kd1 = x[0]
  Kd2 = x[1]
  Kdv2 = Numeric.array([ x[2], x[3], x[4], x[5], x[6], x[7] ])
  Kf1 = x[8]
  Kf2 = x[9]
  Kfv2 = Numeric.array([ x[10], x[11], x[12], x[13], x[14], x[15] ])

  Hd1 = x[16]
  Hd2 = x[17]
  Hdv1 = Numeric.array([ x[18], x[19], x[20], x[21], x[22], x[23] ])
  Hdv2 = Numeric.array([ x[24], x[25], x[26], x[27], x[28], x[29] ])

  Hf1 = x[30]
  Hf2 = x[31]
  Hfv1 = Numeric.array([ x[32], x[33], x[34], x[35], x[36], x[37] ])
  Hfv2 = Numeric.array([ x[38], x[39], x[40], x[41], x[42], x[43] ])

  Nd1 = x[44]
  Nd2 = x[45]
  Ndv2 = Numeric.array([ x[46], x[47], x[48], x[49], x[50], x[51] ])

  Nf1 = x[52]
  Nf2 = x[53]
  Nfv2 = Numeric.array([ x[54], x[55], x[56], x[57], x[58], x[59] ])

  work1 = Numeric.zeros((2,14),'d')
  work2 = Numeric.zeros((4,8),'d')

  for i in range(2):

    e = jacobian[i,:]
    ev1 = Numeric.array([ e[14], e[15], e[16], e[17], 0.0, 0.0 ])
    ev2 = Numeric.array([ e[18], e[19], e[20], e[21], 0.0, 0.0 ])



    y1 = ( ev2 + Numeric.dot(ev1,Rmat) + Kdv2 * e[0] + Kfv2 * e[1] + Hdv1 * e[10]
+
         Hfv1 * e[11] + Ndv2 * e[6] + Nfv2 * e[7] + (Hdv2 +
Numeric.dot(Hdv1,Rmat)) *
         e[4] + (Hfv2 + Numeric.dot(Hfv1,Rmat)) * e[5] )

    y2 = ( Kdv2 * e[2] + Kfv2 * e[3] + Hdv2 * e[10] + Hfv2 * e[11] +
           Ndv2 * e[12] + Nfv2 * e[13] + (Hd1 * Kdv2 + Hd2 * Kfv2) * e[4] +
         (Kd1 * Kdv2 + Kd2 * Kfv2) * e[0] + (Hf1 * Kdv2 + Hf2 * Kfv2) * e[5] +
         (Kf1 * Kdv2 + Kf2 * Kfv2) * e[1] + (Nd1 * Kdv2 + Nd2 * Kfv2) * e[6] +
         (Nf1 * Kdv2 + Nf2 * Kfv2) * e[7] )

    y3 = ( e[8] + Kd1 * e[2] + Kf1 * e[3] + Hd1 * e[10] + Hf1 * e[11] +
         Nd1 * e[12] + Nf1 * e[13] + (Hd1 * Kd1 + Hd2 * Kf1) * e[4] +
         (Hf1 * Kd1 + Hf2 * Kf1) * e[5] + (Kd1 * Kf1 + Kf1 * Kf2) * e[1] +
         (Kd1 * Nd1 + Kf1 * Nd2) * e[6] + (Kd1 * Nf1 + Kf1 * Nf2) * e[7] +
         e[0] * (Kd2 * Kf1 + Kd1*Kd1 )  )

    y4 = ( e[9] + Kd2 * e[2] + Kf2 * e[3] + Hd2 * e[10] + Hf2 * e[11] +
          Nd2 * e[12] + Nf2 * e[13] + (Hd1 * Kd2 + Hd2 * Kf2) * e[4] +
          (Kd1 * Kd2 + Kd2 * Kf2) * e[0] + (Hf1 * Kd2 + Hf2 * Kf2) * e[5] +
          (Kd2 * Nd1 + Kf2 * Nd2) * e[6] + (Kd2 * Nf1 + Kf2 * Nf2) * e[7] +
           e[1]* (Kd2 * Kf1 + Kf2*Kf2 ) )

    temp1 = Numeric.array([y3,y4])
    work1[i,:] = Numeric.concatenate((y1,y2,temp1),1)

  R2 = Numeric.dot(Rmat,Rmat)

  for i in range(4):

    e = jacobian[2+i,:]
    ev1 = Numeric.array([ e[14], e[15], e[16], e[17], 0.0, 0.0 ])
    ev2 = Numeric.array([ e[18], e[19], e[20], e[21], 0.0, 0.0 ])
```

```
    y5 =  ( e[8] + Kd1 * e[2] + Kf1 * e[3] + Hd1 * e[10] + Hf1 * e[11] +

             Nd1 * e[12] + Nf1 * e[13] + (Hd1 * Kd1 + Hd2 * Kf1) * e[4] +

            (Hf1 * Kd1 + Hf2 * Kf1) * e[5] + (Kd1 * Kf1 + Kf1 * Kf2) * e[1] +
            (Kd1 * Nd1 + Kf1 * Nd2) * e[6] + (Kd1 * Nf1 + Kf1 * Nf2) * e[7] +
             e[0] * (Kd2 * Kf1 + Kd1*Kd1 ) )

    y6 =  ( e[9] + Kd2 * e[2] + Kf2 * e[3] + Hd2 * e[10] + Hf2 * e[11] +
             Nd2 * e[12] + Nf2 * e[13] + (Hd1 * Kd2 + Hd2 * Kf2) * e[4] +
            (Kd1 * Kd2 + Kd2 * Kf2) * e[0] + (Hf1 * Kd2 + Hf2 * Kf2) * e[5]+
            (Kd2 * Nd1 + Kf2 * Nd2) * e[6] + (Kd2 * Nf1 + Kf2 * Nf2) * e[7] +
             e[1] * (Kd2 * Kf1 + Kf2*Kf2 ) )

    y7 =  ( Numeric.dot(ev2,Rmat)  + Kdv2 * e[2] + Kfv2 * e[3] + Ndv2 * e[12] +
             Nfv2 * e[13] + (Hdv2 + Numeric.dot(Hdv1,Rmat)) * e[10] +
            (Hfv2 + Numeric.dot(Hfv1,Rmat)) * e[11] + (Numeric.dot(Kdv2,Rmat) +
Kd1 * Kdv2 +
             Kd2 * Kfv2) * e[0] + (Numeric.dot(Kfv2,Rmat) + Kf1 * Kdv2 + Kf2 *
Kfv2) *
             e[1] + (Numeric.dot(Ndv2,Rmat) + Nd1 * Kdv2 + Nd2 * Kfv2) * e[6] +
            (Numeric.dot(Nfv2,Rmat) + Nf1 * Kdv2 + Nf2 * Kfv2) * e[7] +
             Numeric.dot(ev1,R2) + e[4] * (Numeric.dot(Hdv2,Rmat) +
             Hd1 * Kdv2 + Hd2 * Kfv2 + Numeric.dot(Hdv1,R2)) +
             e[5] * (Numeric.dot(Hfv2,Rmat) + Hf1 * Kdv2 + Hf2 * Kfv2 +
             Numeric.dot(Hfv1,R2)) )

    temp2 = Numeric.array([y5,y6])
    work2[i,:] = Numeric.concatenate((y7,temp2),1)

  work3 = Numeric.reshape( work1, (1,28) )
  work4 = Numeric.reshape( work2, (1,32) )

  fvec = Numeric.reshape(Numeric.concatenate((work3, work4),1),(neqs,))

  return fvec


def make_Rmat():

  #

  # Construct the exogenous shock AR matrix

  #



  # Markov transition probability matrix



  Pmat = transition_probs( p_plow,p_phigh )

  # Money growth AR process coefficient matrix is set
  # as matrix Rmat
  #
  temp = Numeric.zeros((2,2),'d')
  temp1 = Numeric.dot(p_mMeans,Pmat) - Numeric.dot(p_mCoeffs,p_mMeans)
  temp2 = Numeric.concatenate((temp,p_mCoeffs,temp1),1 )

  temp3 = Numeric.concatenate( (Numeric.zeros((2,4),'d'),Pmat),1 )
  temp4 = Numeric.concatenate( (temp2,temp3) )

  temp1 = Numeric.zeros((1,6),'d')
  temp1[0,0] = p_zrhod
  temp2 = Numeric.zeros((1,6),'d')
  temp2[0,1] = p_zrhof
  temp = Numeric.concatenate( (temp1,temp2) )

  Rmat = Numeric.concatenate((temp,temp4))

  return Rmat


def ergodic_state( ):

  #

  #  Calculate steady state at ergodic means

  #



  pih = (1-p_plow)/(2-p_plow-p_phigh)

  pil = (1-p_phigh)/(2-p_plow-p_phigh)

  emean = pil * p_mMeans[:,0] + pih * p_mMeans[:,1]

  [kds,kfs,hds,hfs,nds,nfs] = steadyState( emean[0],emean[1] )



  return emean,kds,kfs,hds,hfs,nds,nfs




if __name__== '__main__':

  #

  # Driver program for nonlinear equation solver
  #

  import Multipack
  import time

  a = time.clock()

  # Which model, sysmodel1 (sluggish deposits) or sysmodel2 (sluggish capital
  # and deposits).

  model = 'sysmodel2'

  [emean,kds,kfs,hds,hfs,nds,nfs] = ergodic_state()
  zds = 0.0
  zfs = 0.0

  ssvec = [ kds,kfs,kds,kfs,hds,hfs,nds,nfs,kds,kfs,hds,hfs,nds,
            nfs,zds,zfs,emean[0],emean[1],zds,zfs,emean[0],emean[1] ]

  # Construct Euler equation jacobian matrix

  g1 = gradient( euler2d,ssvec,ssvec )
  g2 = gradient( euler2f,ssvec,ssvec )
  g3 = gradient( euler3d,ssvec,ssvec )
  g4 = gradient( euler3f,ssvec,ssvec )
  g5 = gradient( euler1d,ssvec,ssvec )
  g6 = gradient( euler1f,ssvec,ssvec )

  global jacobian
  jacobian = Numeric.reshape(Numeric.concatenate((g1,g2,g3,g4,g5,g6)),(6,22))

  Rmat = make_Rmat()

  if model == 'sysmodel1':

    print 'Solving sluggish deposits case.'
```

```
    outfile = 'solvec1.out'

    xguess = Numeric.ones(72,'d')*.01

    g = Multipack.fsolve( sysmodel1,xguess )

    if g[0] > 1 or g[15] > 1:
      print ''
      print '************************************'
      print '*****    unstable solution    *******'
      print '***** try new starting values *******'
      print '************************************'
      print 'Home capital coefficient: ', g[0]
      print 'Foreign capital coefficient: ', g[15]
      print ''

    savem(outfile,g[:,Numeric.NewAxis])
    b = time.clock()

    print 'elapsed execution time: ', b-a, 'secs'
    svec = sysmodel1(g)
    print ''
    print 'norm is: ', Numeric.sqrt(Numeric.add.reduce(svec*svec))
    print ''
    print 'solution written to: ' + outfile
    print ''


  if model == 'sysmodel2':

    print 'Solving sluggish capital and deposits case. '
    outfile = 'solvec2.out'

    xguess = Numeric.ones(60,'d')*.1

    g = Multipack.fsolve( sysmodel2,xguess )

    if g[0] > 1 or g[9] > 1:
      print ''
      print '************************************'
      print '*****    unstable solution    *******'
      print '***** try new starting values *******'
      print '************************************'
      print 'Home capital coefficient: ', g[0]
      print 'Foreign capital coefficient: ', g[9]
      print ''

    savem(outfile,g[:,Numeric.NewAxis])
    b = time.clock()

    print 'elapsed execution time: ', b-a, 'secs'
    svec = sysmodel2(g)
    print ''
    print 'norm is: ', Numeric.sqrt(Numeric.add.reduce(svec*svec))
    print ''
    print 'solution written to: ' + outfile
    print ''

    if model == 'sysmodel2':
      execfile( 'msvcap.py' )
    elif model == 'sysmodel1':
      execfile( '' )


#
# Program msvcap.py
#
# Simulate the open-economy model with Markov-switching money
# growth process and rational learning.
#
# #
# Written in Python. Uses the NumPy module.
#
# Keith Sill
# FRB Philadelphia
# 07/19/99


import Numeric
import RandomArray
import LinearAlgebra


from random import *
from pvalues import *
infile = 'solvec2.out'
def loadm(name,nr,nc):
  #
  # Load an ascii format matrix
  #
  import string
  numCols = nc
  numRows = nr

  z = Numeric.zeros([numRows,numCols],Numeric.Float)

  i=0; j= 0

  f = open(name,'r')

  for line in f.readlines():
    cols = string.split(line)
    for i in range(numCols):
      z[j,i] = string.atof(cols[i])
    j = j+1

  f.close()

  return z


def savem(name,m):
  #
  # Save a matrix in ascii format
  #
  import sys

  fp = open(name,'w')

  for i in range(m.shape[0]):
    for j in range(m.shape[1]):
      fp.write(str(m[i,j]) + '\t')
    fp.write('\n')

  fp.close()

def diag( vec,k ):
  #
  # Insert the vector k along the kth diagonal
  # and return a matrix.
```

```
  #
  n = len( vec ) + Numeric.absolute( k )
  M = Numeric.zeros((n,n),'d')

  if k > 0:

    for i in range( len(vec) ):
      M[i,k+i] = vec[i]

  elif k < 0:

    for i in range( len(vec) ):
      M[-k+i,i] = vec[i]

  else:

    for i in range( len(vec) ):
      M[i,i] = vec[i]

  return M


def invM( nr,lam ):
  #
  #  Invert the hp filter coefficient matrix
  #  We pull this op out to speed up the simulation
  #  loop
  #
  M = Numeric.zeros((nr,nr),'d')

  d1 = Numeric.ones(nr-2,'d')
  d1 = d1*lam

  d2 = Numeric.ones(nr-1,'d')
  d2 = d2*(-4.0*lam)
  d2[0] = -2*lam
  d2[nr-2] = -2*lam

  d3 = Numeric.ones(nr,'d')
  d3 = d3*(1+6*lam)
  d3[0] = 1+lam
  d3[1] = 1+5*lam
  d3[nr-2] = 1+5*lam
  d3[nr-1] = 1+lam

  M1 = diag(d1,2)
  M2 = diag(d2,1)
  M3 = diag(d3,0)
  M4 = diag(d2,-1)
  M5 = diag(d1,-2)

  M = M1 + M2 + M3 + M4 + M5

  return LinearAlgebra.inverse(M)

def hpf( Minv,y ):
  #
  #  Calculate the hp trend of y
  #
  g = Numeric.dot( Minv,y )

  return g


def steadyState( mud, muf ):

  # takes as input the money growth rates in the home and foreign
  # countries.  Returns a 6-tuple of calculated steady states for
  # capital, hours, and deposits.

  xx = Numeric.zeros(6,'d')

  for i in range(2):
      vt1 = ( p_a3 * (1+mud) / p_beta ) * ( 1.0 / p_beta - (1-p_delta) )
      vt2 =  (1-p_a3) * ( 1/p_beta - (1-p_delta) )
      vd =  ( 1/p_alpha * ( vt1 + vt2 ) )**( 1.0/(p_alpha-1) )
      tempd = ( p_gamma/(1-p_gamma) )*( p_beta/(1+mud) ) * ( (1-
p_alpha)*vd**p_alpha /
          (vd**p_alpha - p_delta*vd)  )* ( p_a1 + (1-p_a1) * p_beta / ( 1+mud )
)

      hd = tempd / ( 1+tempd )
      kd = hd * vd

      yd = kd**p_alpha * hd**(1-p_alpha)
      zd = ( p_a1 * (1-p_alpha) * yd * p_beta / ( 1+mud ) - ( yd- p_delta * kd )
) / (
          (1-p_alpha) * p_beta * yd / ( 1+mud ) + p_a3 * p_delta * kd )
      nd = ( 1 + mud * zd ) / ( 1 - zd )

      xx[i] = kd
      xx[i+2] = hd
      xx[i+4] = nd

      mud = muf

  return xx

def simulate_tfp( zrho,std,T ):
  #
  #  Generate time series for TFP shock
  #

  theta = Numeric.zeros(T,'d')

  for i in range(1,T):
    theta[i] = p_zrhod*theta[i-1] + gauss(0.0,std)

  return theta

def choleski( x ):
  #
  #  An inefficient implementation of the choleski
  #  decomposition. Returns a matrix with choleksi factors
  #  on the lower triangle, except for the diagonal, which is
  #  returned in p.
  #
  n = len(x)
  p = Numeric.zeros(n,'d')
  a = x.astype('d')
  sum = 0.0

  for i in range(1,n+1):
    for j in range( i,n+1 ):
      sum = a[i-1,j-1]
      for k in range( i-1,0,-1 ):
        sum = sum - a[i-1,k-1]*a[j-1,k-1]
      if i == j :
        if sum <= 0.0 :
          print 'choldc failed'
        p[i-1] = Numeric.sqrt(sum)
      else:
        a[j-1][i-1] = sum/p[i-1]

  return a, p

def simVAR( mPsi,mLlow,mLhigh,vMeanLow,vMeanHigh,probLow,probHigh ):
  #
  #  Simulate a markov-switching money growth process on
  #  a two-element VAR(1).
  #

  p11 = probHigh   # high-to-high transition prob
  p22 = probLow    # low-to-low transition prob
```

```
  p12 =  (1.0-p11)
  p21 =  (1.0-p22)

  vEta = Numeric.array([0,1])
  muVec = Numeric.zeros((T,3),'d')

  mP = Numeric.array([ [ p11, p21 ], [ p12, p22 ] ])

  v11 = Numeric.array([ p12, -p12 ])
  v12 = Numeric.array([ -p11, p11 ])
  v21 = Numeric.array([ p22, -p22 ])
  v22 = Numeric.array([ -p21, p21 ])

  vMu_hat = vMeanLow

  # Draw an N-vector of (0,1) random variables. We use
  # this to indicate which transition to make
  #
  x = RandomArray.random(T)
  vMeanLg = vMeanLow

  # Generate the Markov process.  The vector eta is a 2-element
  # vector of [1,0] or [0,1].  Thus, the v's are defined so that
  # eta(t) = P*eta(t-1) + v gives a 0-1 or 1-0 vector. The variable
  # e is a monetary control error, drawn from the appropriate high
  # or low distribution.
  #
  tol = 1e-8

  for i in range(T):
    if Numeric.absolute(vEta[0]-1) < tol and x[i] <= p11 :   # A high-to-high
transition
      v = v11
      ve = Numeric.array([gauss(0.0,1),gauss(0.0,1)])
      ve = Numeric.dot(ve,mLhigh)
      vMean = vMeanHigh
    elif Numeric.absolute(vEta[0]-1) < tol and x[i] > p11 :  # A high-to-low
transition
      v = v12
      ve = Numeric.array([gauss(0.0,1),gauss(0.0,1)])
      ve = Numeric.dot(ve,mLlow)
      vMean = vMeanLow
    elif Numeric.absolute(vEta[0]) < tol and x[i] <= p21 : # A low-to-high
transition
      v = v21
      ve = Numeric.array([gauss(0.0,1),gauss(0.0,1)])
      ve = Numeric.dot(ve,mLhigh)
      vMean = vMeanHigh
    elif Numeric.absolute(vEta[0]) < tol and x[i] > p21 :  # A low-to-low
transition
      v = v22
      ve = Numeric.array([gauss(0.0,1),gauss(0.0,1)])
      ve = Numeric.dot(ve,mLlow)
      vMean = vMeanLow
    else:
      print 'vEta elements are not 0 or 1 '

    vEta = Numeric.dot(mP,vEta) + v

    vMu_hat = ( vMean + Numeric.dot(mPsi,vMu_hat) -
                Numeric.dot(mPsi,vMeanLg) + ve  )

    vMeanLg = vMean

    muVec[i,0] = vEta[0]
    muVec[i,1:] = vMu_hat

  return muVec

def mshock( vMean1,vMean2,mPsi,mMgrowth ):
  #
  # Given an AR1 sequence of money growth rates, and the parameters
  # governing the process, return the implied innovations.
  #
  mShocks = Numeric.zeros( (2,T),'d' )

  for i in range(1,T):
    mShocks[:,i] = (mMgrowth[i,:]-vMean2) - Numeric.dot(mPsi,(mMgrowth[i-1,:] -
vMean1))

  return Numeric.transpose(mShocks)


def npdf( mX,mSig ):
  #
  # Multivariate normal pdf.  mX is mean zero
  # Returns a vector of length mX with npdf(mX(i))
  #
  n = len(mSig)

  c0 = LinearAlgebra.inverse( mSig )
  c1 = 1 / ( (Numeric.pi**(n/2.0)) *
       Numeric.sqrt( LinearAlgebra.determinant( c0 )))
  c2 = Numeric.dot( mX,c0 )
  c3 = Numeric.dot( c2,Numeric.transpose(mX) )

  fn = c1 * Numeric.exp( -0.5*Numeric.diagonal( c3 ) )
  return fn

def beliefs( mMgrow,mSigLow,mSigHigh,plow,phigh ):
  #
  # Calculate regime belief, returns prob regime is low
  #

  vml = p_mMeans[:,1]
  vmh = p_mMeans[:,0]

  mP = Numeric.array([ [ phigh,    (1-plow)],
                       [(1-phigh),  plow   ] ] )

  mell = mshock( vml,vml,p_mCoeffs,mMgrow )
  melh = mshock( vml,vmh,p_mCoeffs,mMgrow )
  mehl = mshock( vmh,vml,p_mCoeffs,mMgrow )
  mehh = mshock( vmh,vmh,p_mCoeffs,mMgrow )

  mfll = npdf( mell,mSigLow )
  mflh = npdf( melh,mSigHigh )
  mfhl = npdf( mehl,mSigLow )
  mfhh = npdf( mehh,mSigHigh )

  vblow = Numeric.zeros(T,'d')
  vblow[0] = 0.5

  for i in range(1,T):
    gl = ( vblow[i-1] * mP[1,1] * mfll[i] +
           (1-vblow[i-1])*mP[1,0]*mfhl[i] )

    gh = ( vblow[i-1]*mP[0,1]*mflh[i] +
           (1-vblow[i-1])*mP[0,0]*mfhh[i] )

    vblow[i] = gl / ( gl + gh )

  return vblow

def ergodic_state():
  #
  # Return ergodic probs and steady state vals
  #
  pil = (1-p_phigh)/(2-p_plow-p_phigh)
  pih = 1 - pil

  vEmean = pil * p_mMeans[:,1] + pih * p_mMeans[:,0]
```

```
  [kds,kfs,hds,hfs,nds,nfs] = steadyState(vEmean[0],vEmean[1])

  return vEmean,kds,kfs,hds,hfs,nds,nfs


def make_stateb( mState,vEmean ):
  #
  #  Calculate state matrix for active beliefs
  #
  vHtfp = simulate_tfp( p_zrhod,p_zsigd,T)
  vFtfp = simulate_tfp( p_zrhof,p_zsigf,T)

  mMgrow = simVAR( p_mCoeffs,mLl,mLl,p_mMeans[:,1],p_mMeans[:,0],
                   p_plow,p_phigh )

  vblow = beliefs( mMgrow[:,1:],p_mSigLow,p_mSigHigh,p_plow,p_phigh )
  avgb = Numeric.add.reduce( vblow )/T

  mState[0,:] = vHtfp
  mState[1,:] = vFtfp
  mState[2,:] = mMgrow[:,1] - vEmean[0]
  mState[3,:] = mMgrow[:,2] - vEmean[1]
  mState[4,:] = vblow - avgb
  mState[5,:] = avgb - vblow

  return mState,mMgrow,vHtfp,vFtfp

def make_statec( mState,vEmean ):
  #
  #  Calculate state for inactive beliefs
  #
  vHtfp = simulate_tfp( p_zrhod,p_zsigd,T)
  vFtfp = simulate_tfp( p_zrhof,p_zsigf,T)

  mMgrow = simVAR( p_mCoeffs,mLl,mLl,p_mMeans[:,1],p_mMeans[:,0],
                   p_plow,p_phigh )

  mState[0,:] = vHtfp
  mState[1,:] = vFtfp
  mState[2,:] = mMgrow[:,1] - vEmean[0]
  mState[3,:] = mMgrow[:,2] - vEmean[1]
  mState[4:6,:] = 0.0

  return mState,mMgrow,vHtfp,vFtfp

def capital_path( coeff,state,kdss,kfss ):
  #
  # Calculate the equilibrium path for the level of home and foreign
  # capital stocks.  Returns Tx1 vectors for home capital and foreign
  # capital.  The inputs are:
  #
  #     coeff:  a 28-element vector of solution coefficients
  #     state:  a 6xT matrix of states, assumed to be ordered row-wise
  #             as home money growth (dev from ss )
  #                foreign money growth (dev from ss )
  #                belief for home-high, foreign-high (ss dev)
  #                belief for high, low  (ss dev)
  #                belief for low, high  (ss dev)
  #                belief for low, low   (ss dev)
  #
  #    kds0,kfs0 :  Initial steady states for home and foreign
  #    kds1,kfs1 :  final steady states for home and foreign

  kd = Numeric.zeros( T,'d' )
  kf = Numeric.zeros( T,'d' )

  kdv2 = coeff[2:8]
  kfv2 = coeff[10:16]

  for i in range(2,T):

    kd[i] = ( coeff[0]*kd[i-1] + coeff[1]*kf[i-1] +
              Numeric.dot( kdv2,state[:,i-2]) )

    kf[i] = ( coeff[8]*kd[i-1] + coeff[9]*kf[i-1] +
              Numeric.dot( kfv2,state[:,i-2]) )

  return kd, kf, kd+kdss, kf+kfss


def hours_path( coeff,kd,kf,state,hss ):
  #
  # Equilibrium path for hours worked. Returns the Tx1 vector for
  # hours as deviations from steady state. Inputs are:
  #
  #     coeff : a 14 element vector of solution coefficients
  #     kd, kf: Tx1 vectors of capital stock deviations from steady state
  #     state : The 6xT matrix of states, as defined in function
  #             capital_path( )
  #     hs0,hs1 : initial and final steady states for hours
  #

  h = Numeric.zeros( T,'d' )

  hv1 = coeff[2:8]
  hv2 = coeff[8:14]

  for i in range(1,T):
    h[i] = ( coeff[0]*kd[i] + coeff[1]*kf[i] +
             Numeric.dot( hv1,state[:,i] ) + Numeric.dot( hv2,state[:,i-1] ) )

  return h, h + hss


def deposits_path( coeff,kd,kf,state,nss ):
  #
  # Equilibrium path for deposits. Returns a Tx1 vector of deviations
  # of deposits from steady state.  Inputs are the a 8x1 vector of
  # solution coefficients, the Tx1 vectors of home and foreign capital
  # stocks (expressed as deviations from steady state) and the 6xT matrix
  # of state variables, as defined in function capital_path( ).
  #

  n = Numeric.zeros( T,'d' )
  nv1 = coeff[2:8]

  for i in range(1,T):
    n[i] =  coeff[0]*kd[i] + coeff[1]*kf[i] + Numeric.dot( nv1,state[:,i-1] )

  return n, n + nss


def mvars1( k,h,n,u,tfp ):
  #
  #  Calculate time paths for model variables. Inputs are levels of
  #  capital, hours, deposits, and money growth
  #

  lm = Numeric.ones(T,'d')
  ivest = Numeric.zeros( T,'d' )

  for i in range( T):
    lm[i] = Numeric.multiply.reduce( Numeric.exp(u[:i+1]) )

  for i in range(T-1):
    ivest[i] = k[i+1] - (1.0- p_delta)*k[i]
  ivest[T-1] = p_delta * k[T-1]

  y = k**p_alpha * h**(1-p_alpha) * Numeric.exp((1-p_alpha)*tfp)

  p = ( 1 - (1-p_a1)*n + p_a1*u ) / ( y + (p_a1*p_a3-1)*ivest )
  lp = p * lm
  w = ( n + u - p_a3*p*ivest ) / h
  lw = w * lm
```

```
  rate = (1-p_alpha) * ( p/w ) * ( k/h )**p_alpha * Numeric.exp((1-p_alpha)*tfp)

  return lp,lw,rate,invest,y

def mvars2( lpd,lpf,yd,yf,investd,investf ):
  #
  #  Calculate time series for nominal fx rate, real fx rate
  #  domestic consumption, foreign consumption.
  #
  fx = (lpd*(1-p_thd)*(yd - investd))/(lpf*(1-p_thf)*(yf-investf))
  rx = fx * lpf / lpd
  cd = p_thd * (yd - investd) + (1-p_thf) * ( yf - investf )
  cf = p_thf * (yf - investf) + (1-p_thd) * ( yd - investd )

  return fx,rx,cd,cf

def stdvec( matrix ):
  #
  #  Calculate standard deviation of series in the
  #  Txn matrix 'matrix'
  #
  temp = Numeric.dot( Numeric.transpose(matrix),matrix )
  temp = Numeric.sqrt( Numeric.diagonal(temp)/(len(matrix)-1) )

  return  temp

def lag( mX,k ):
  #
  #  lag the matrix mX by k periods
  #
  nr = mX.shape[0]

  try:
    nc = mX.shape[1]
  except:
    nc = 0
    mX = mX[:,Numeric.NewAxis]

  if nc > 0:
    padding = zeros( (Numeric.absolute(k),nc),'d' )

  if nc == 0:
    padding = Numeric.zeros( Numeric.absolute(k), 'd' )
    padding = padding[:,Numeric.NewAxis]

  if k > 0 :
    temp = Numeric.concatenate((padding,mX))
    rmat = temp[:nr,]
  if k < 0:
    temp = Numeric.concatenate((mX,padding))
    rmat = temp[Numeric.absolute(k):,]
  if k == 0:
    rmat = mX

  return rmat

def correlate( X ):
  #
  #  correlation between the column vectors
  #  of X
  #
  n = X.shape[0]
  mX = Numeric.sum(X)/n
  covar = ( Numeric.dot(Numeric.transpose(X),X)/n -
            Numeric.multiply.outer(mX,mX)  )

  var = Numeric.diagonal(covar)

  return covar / Numeric.sqrt(Numeric.multiply.outer(var,var))
```

```
def acf1( var ):
  #
  # First order autocorrelation of var
  #
  n = len( var )
  xss = Numeric.innerproduct( var[0:T-1],var[1:T] )
  ssq = Numeric.innerproduct( var[0:T-1],var[0:T-1] )

  return xss/ssq


########################################
#
#  Simulation Driver Program
#
########################################

if __name__ == '__main__':

  import time

  iter = 100      # Number of times to run the simulation.
  T = 500         # Length of the time series.

  a = time.clock()      # Start timer

  coeffs = loadm(infile,60,1)
  coeffs.shape=(60,)

  vK  = coeffs[0:16]
  vHH = coeffs[16:30]
  vFH = coeffs[30:44]
  vHD = coeffs[44:52]
  vFD = coeffs[52:60]

  Minv = invM( T,1600 )
  mState = Numeric.zeros( (6,T),'d' )
  [vEmean,kds,kfs,hds,hfs,nds,nfs] = ergodic_state()

  [mLl,vpl] = choleski( p_mSigLow )

  mLl[0,0] = vpl[0]
  mLl[1,1] = vpl[1]
  mLl[0,1] = 0.0

  print ''
  print 'Simulations for sluggish deposits, sluggish capital model'
  print 'Model parameterization is a1 = %3.2f, a3 = %3.2f' %(p_a1,p_a3)
  print 'Statistics for HP-Filtered series'
  print 'Running %d iterations on series of length %d' %(iter,T)
  print ''

  for count in range(1):

    temp = Numeric.zeros((T,1),'d')

    homecount = 0.0
    awaycount = 0.0
    hstdvec = 0.0
    fstdvec = 0.0
    nstdvec = 0.0
    rhohi  = 0.0
    rhofi  = 0.0
    rhofx  = 0.0
    rhorx  = 0.0

    if count == 0 :
      print 'Results for model with active beliefs'
    if count == 1 :
      print 'Results for model with inactive beliefs'

    for index in range(iter):
```

```
  if count == 0 :
    [mState,mMoney,vHtfp,vFtfp] = make_stateb( mState,vEmean )
  if count == 1 :
    [mState,mMoney,vHtfp,vFtfp] = make_statec( mState,vEmean )


  [kdhat,kfhat,lkd,lkf] = capital_path( vK,mState,kds,kfs )
  [hdhat,lhd] = hours_path( vHH,kdhat,kfhat,mState,hds )
  [hfhat,lhf] = hours_path( vFH,kdhat,kfhat,mState,hfs )
  [ndhat,lnd] = deposits_path( vHD,kdhat,kfhat,mState,nds )
  [nfhat,lnf] = deposits_path( vFD,kdhat,kfhat,mState,nfs )

  # Time paths for prices, wages, nominal interest rates,
  # investment, and output

  [lpd,lwd,nrd,ivestd,yd] = mvars1( lkd,lhd,lnd,mMoney[:,1],vHtfp )
  [lpf,lwf,nrf,ivestf,yf] = mvars1( lkf,lhf,lnf,mMoney[:,2],vFtfp )
  [fx,rx,cd,cf]           = mvars2( lpd,lpf,yd,yf,ivestd,ivestf )

  hvmat = Numeric.concatenate((lpd[:,Numeric.NewAxis],
                               lwd[:,Numeric.NewAxis],
                               nrd[:,Numeric.NewAxis],
                               ivestd[:,Numeric.NewAxis],
                               yd[:,Numeric.NewAxis]),1)

  fvmat = Numeric.concatenate((lpf[:,Numeric.NewAxis],
                               lwf[:,Numeric.NewAxis],
                               nrf[:,Numeric.NewAxis],
                               ivestf[:,Numeric.NewAxis],
                               yf[:,Numeric.NewAxis]),1)

  nvmat = Numeric.concatenate((fx[:,Numeric.NewAxis],
                               rx[:,Numeric.NewAxis],
                               cd[:,Numeric.NewAxis],
                               cf[:,Numeric.NewAxis]),1)
  for j in range(T):
    if hvmat[j,2] < 1:
      homecount = homecount + 1
    if fvmat[j,2] < 1:
      awaycount = awaycount + 1

  hvmat = Numeric.log( hvmat )
  fvmat = Numeric.log( fvmat )
  nvmat = Numeric.log( nvmat )

  temp0 = nvmat[:,0]
  temp = Numeric.concatenate((temp,temp0[:,Numeric.NewAxis]),1)

  hpf_hvmat = hvmat - hpf(Minv,hvmat)
  hpf_fvmat = fvmat - hpf(Minv,fvmat)
  hpf_nvmat = nvmat - hpf(Minv,nvmat)

  hpf_hstd = stdvec( hpf_hvmat )
  hpf_fstd = stdvec( hpf_fvmat )
  hpf_nstd = stdvec( hpf_nvmat )

  hstdvec = hstdvec + (1.0/(1+index))*(hpf_hstd - hstdvec)
  fstdvec = fstdvec + (1.0/(1+index))*(hpf_fstd - fstdvec)
  nstdvec = nstdvec + (1.0/(1+index))*(hpf_nstd - nstdvec)

  rhohi = rhohi + (1.0/(1+index))*(acf1(hpf_hvmat[:,2]) - rhohi )
  rhofi = rhofi + (1.0/(1+index))*(acf1(hpf_fvmat[:,2]) - rhofi )
  rhofx = rhofx + (1.0/(1+index))*(acf1(hpf_nvmat[:,0]) - rhofx )
  rhorx = rhorx + (1.0/(1+index))*(acf1(hpf_nvmat[:,1]) - rhorx )

  # End of iteration loop

print ''
print '               Home          Foreign'
print 'Output        %6.5f         %6.5f  ' % (hstdvec[4],fstdvec[4])
print 'Inv/y         %6.5f         %6.5f  ' % (hstdvec[3]/hstdvec[4],
                                               fstdvec[3]/fstdvec[4])
  print 'Cons/y        %6.5f         %6.5f  ' % (nstdvec[2]/hstdvec[4],
                                                 nstdvec[3]/fstdvec[4])
  print 'Int Rate      %6.5f         %6.5f  ' % (hstdvec[2],fstdvec[2])
  print 'Prices        %6.5f         %6.5f  ' % (hstdvec[0],fstdvec[0])
  print 'Nom  fx                     %6.5f  ' % (nstdvec[0])
  print 'Real fx                     %6.5f  ' % (nstdvec[1])
  print ''
  print 'Frac i < 0         %6.5f         %6.5f  ' % (homecount/(iter*T),
                                                      awaycount/(iter*T))
  print ''
  print 'First order autocorrelations: '
  print ''
  print 'Dom irate     %6.5f                ' % rhohi
  print 'For irate     %6.5f                ' % rhofi
  print 'Nfx rho  %6.5f                ' % rhofx
  print 'Rfx rho  %6.5f                ' % rhorx
  print ''

savem('temp.dat',temp)
z = time.clock()
print 'Elapsed execution time: ', z - a, ' secs'
```