Solution Methods for Structural Models

Ronald L. Horst

Inforum, University of Maryland

Horst@inforum.umd.edu

10th INFORUM WORLD CONFERENCE 2002
July 28 to August 3, 2002
University of Maryland
USA

Solution Methods for Structural Models

I. Introduction

The speed of modern computers provides solutions for economic models in a fraction of the time required even in the recent past. Hence, a discussion of methods to speed convergence may seem unwarranted. Using *Inforum*'s *G7* program, a macroeconomic model like *Quest* can be solved within seconds. *Interdyme* provides a solution over 15 periods for *IdLift*, a large inter-industry model, in well under a minute. Why invest scarce labor resources to further reduce these modest time requirements?

Perhaps the most compelling reason to attempt to improve convergence rates is provided by the optimization routine introduced by Almon (2001). The method offers the potential to greatly improve the fit of historical data and perhaps to improve a model's ability to forecast and to simulate accurately. Unfortunately, the optimization routine requires the model to be solved many times. For optimization processes that converge slowly to an optimum, the model may need to be solved with thousands of alternative parameter vectors. Clearly, optimization within a large model is not practical without improving the performance of the optimization routine and of the root-finding mechanism.

There are many other reasons to reconsider the methods employed if only to satisfy ourselves that indeed they are most appropriate. Currently, the method of choice at *Inforum* and most affiliates is the Gauss-Seidel version of fixed-point iteration. Unfortunately, fixed-point iteration is prone to failure or may converge slowly. In following sections, we will see a simple modification of the Gauss-Seidel process that can reign in divergent processes or can speed convergence of well-behaved systems. These methods are known as *successive under-relaxation* and *successive over-relaxation* (SOR), respectively. A minor alteration of existing equations allows implementation of these techniques. In addition, a simple algorithm demonstrates that equations can be ordered so that "predetermined" variables are evaluated first, thus reducing the size of the system that must be solved and moving the initial guess closer to the solution. Finally, a routine is presented that rewrites the equations so they can be solved with Broyden's method. This variant is somewhat faster than Newton's method for large systems, and supposedly such quasi-Newton methods outperform fixed-point iteration. Neither the fixed-point routines nor Broyden's method, as presented in this paper, are found superior in all cases. Hence, this paper summarizes the construction of a program that sets up and solves both types of equations and tests of the routines with a small Keynesian model.

II. Macro Models in G7

This paper primarily is focused on finding solutions for macroeconomic models. Hence, we begin with a description of the relevant software offered by *Inforum* and with which the featured programs operate. These methods to some degree can be extended to inter-industry modeling, but such models typically are solved in parts. These techniques could be applied to such components, but for simplicity, we focus on single systems of equations that comprise a macro model.

Several programs are offered by Inforum for construction of macro models. First, *G7* is used to store and manipulate data, estimate regressions and specify equations, and to form these equations into a model. The equations and their parameters are stored in text files. Before solving a model, *Build* is executed. This program stores all data required by the model into a single databank. In addition, *Build* rewrites the equations as C++ code. After this code is compiled, the program *Run* is executed to solve the model by repeatedly evaluating the equations.

The program described in this paper operates after *Build* and before the resulting code is compiled or executed. This program reads the equations in their C++ format, changes order of the equations and the functional forms so that new methods can be implemented, and then rewrites the equations in C++ code. Next, the code is compiled and a modified version of *Run* solves the system of equations using either SOR or Broyden's method.

III. The Order

A model with a fully recursive structure could be solved with a singe evaluation per period. Few models are recursive, but a subset of the equations may be solved in a single pass. These equations can be removed from the system, placed in a recursive order, and evaluated before the remaining system is solved. Such methods offer several advantages. First, the equations need to be solved only once per period. This probably offers little advantage for most models that are solved using iterative methods on modern computers. The time needed to solve a single equation is very small; only if a large number of evaluations can be eliminated will there be a noticeable reduction in computation time. Second, both iterative methods and Broyden's method require an initial guess. Finding the solution for a subset of equations amounts to providing a perfect initial guess for those variables. If the variables also appear on the right hand side of equations in the system, then the first evaluation of those system equations likely

produces results closer to the solution. Finally, the system of equations becomes smaller. This benefit is distinct from the first for systems solved by Broyden's method. As will be seen later, quasi-Newton methods require at least an approximation of the Jacobian matrix for the system of equations. The number of elements in this matrix is n^2, where n is the number of equations. Clearly, a reduction in the number of equations can improve performance dramatically.

Although this feature is not offered in the programs described here, it is possible to further reduce the size of the system. Consider a model with three variables: {x, y, z}. Suppose

$$x = f(x, y)$$
$$y = g(x, y)$$
$$z = h(x, y).$$

Because x and y do not depend on z, the third equation can be evaluated after the first two have been solved. Hence, any equations with only predetermined or exogenous variables on the right hand side can be ordered recursively and evaluated first. Next, the system of inter-dependent variables is solved. Finally, remaining equations for which the dependent variables do not appear earlier in the model can be ordered recursively and solved last.

A simple Keynesian model demonstrates the techniques presented in this paper. The model is presented in Appendix A. In the first equation, real consumption per capita is an affine function of per capita real GDP. Next, real investment is a function of real GDP and long-term interest rates. GDP is defined as the sum of aggregate consumption, investment, and an exogenous variable composed primarily of net exports and government expenditures. Short-term interest rates are predicted by an autoregressive equation. Finally, long-term rates are a function of a single lagged endogenous term and current short-term rates.

Appendix B shows the equations after implementation of the methods described above. Short-term rates are evaluated first because all right hand side terms are predetermined. Long-term rates depend on short-term rates and a lagged term; hence, they are evaluated next. The remaining equations depend on the dependent variables from the recursive equations, on exogenous terms, and on other endogenous variables. Hence, they cannot be removed from the system. Appendix C reproduces the model shown in Appendix A, this time in C++ form, and Appendix D similarly reproduces Appendix B.

Meade (1996) describes a method for optimally ordering a system of simultaneous linear equations. Perhaps such methods also could be used to optimally order systems of nonlinear

equations. Such a technique would be highly desirable given the possible improvements for linear systems. Even if no method exists for nonlinear systems, it is possible to separate the linear equations contained in a model and to apply the triangulation routine described by Meade. Unfortunately, the software currently does not offer this feature.

IV. SOR

Fixed-point iteration is a process of solving a set of equations repeatedly. If the proper conditions hold, such iteration will yield a solution. Judd (1998) notes that diagonal dominance ($\sum_{i \neq j} |a_{ij}| < |a_{ii}|$ for i = 1, ..., n) is a sufficient condition for convergence of linear systems. Unfortunately, convergence properties for nonlinear systems often are established by trial and error. Relaxation techniques offer a possible way to speed convergence in well-behaved linear or nonlinear systems and a way to reach convergence for some problematic systems.

Some systems can be solved by the iterating on the following equation: $y^{k+1} = F(y^k)$, where y is a vector of endogenous variables and F is a set of functions. This equation is a special case of relaxation techniques defined as $y^{k+1} = \omega F(y^k) + (1-\omega) y^k$, where $0 < \omega$. If $\omega < 1.0$, the process is called *successive under-relaxation*, and the process is called *successive over-relaxation* (SOR) for $\omega > 1.0$. (See, for example, Judd (1998), Hageman and Young (1981), or Weiss (1996).) Appendix E contains graphical descriptions of divergent and convergent linear systems; these examples hold similarly for nonlinear systems. The diagonal dominance condition described above is violated in the first system, and basic iteration ($\omega = 1.0$) diverges quickly. By forming a convex combination of each old vector (at iteration k) and the model predictions for iteration $k+1$, each step is shortened sufficiently to yield convergence. An example is shown on the same graph for iterations with $\omega < 1.0$ that allows convergence.

The second graph in Appendix E shows a system that yields convergence for fixed-point iteration. The left side of the graph demonstrates such convergence. Relatively many steps are required. This number can be reduced by increasing the size of each step. This may be accomplished by choosing $\omega > 1.0$ as shown on the right side of the graph.

Appendix F graphs the number of times that the equations must be evaluated for the test model using various values of ω. These graph represent the total number of iterations for 81 periods; the average number of iterations required per period is the value shown on the graph divided by 81. The number of iterations required for $\omega = 1.0$ is 2585. That number falls to 845

iterations for an approximately optimal value of ω = 1.20. Thus, the number of iterations can be reduced by more than 2/3 in some cases.

While the curve mapped over ω appears to be fairly smooth and well behaved, this may be misleading. Highly nonlinear curves for which a small change in ω yields a large change in required iterations may be typical. For problems that require solutions for many similar models, such as employment of Almon's optimization routine, it would be helpful for the routine to attempt to improve an initial value of ω. Hageman and Young (1981) provide derivation and computer code for finding theoretically optimal values of ω for linear systems. They also provide a derivation and describe an algorithm for finding an optimal ω for nonlinear systems. Unfortunately, these routines are complex and computationally intensive. It seems use of these algorithms would yield no net reduction in computation time for most models. Given the possibility of large improvements in speed with small changes to ω, even a naive algorithm may provide significant improvements. In pseudo-code, such an algorithm may be written as:

```
If   ω^k − ω^(k−1) ≥ 0
   If (iteration count_i − iteration count_(i-1) ) > 0  { ω^(i+1) = ω^i − stepsize }
   Else  { ω^(i+1) = ω^i + stepsize }
Else
   If (iteration count_i − iteration count_(i-1) ) > 0  { ω^(i+1) = ω^i + stepsize }
   Else  { ω^(i+1) = ω^i − stepsize }
```

where *i* indexes the steps in an optimization routine or another sequence of similar models. Such an algorithm converges very slowly to an "optimal" ω if such a value exists, and this algorithm is trapped easily by local minima. The routine does offer an improvement over the initial value of ω if it is not already at a minimum. The computational cost of this algorithm is very small. Hence, the advantage of such methods over sophisticated but computationally intensive techniques may be significant.

Relaxation methods will not work always. Still, they offer significant advantages over standard iterative methods at the cost of slight revisions to functional forms. Implementation is especially easy for linear systems. A short series of experiments were performed with the (linear) input-output identities in *IdLift*. While the tests were not conclusive, only modest improvements of 1%-2% were realized for various values of ω. The solution mechanism employed triangulation (Meade 1996) to evaluate the equations in the optimal order. Perhaps little additional improvement is possible. This hypothesis could be supported by additional tests

that order the equations randomly. A preliminary conclusion is that optimal ordering and SOR both offer improvements, but in some cases, employment of one may be sufficient.

A portion of the code for the test model, modified for use with SOR, is shown in Appendix G. Note, in particular, the treatment of "fixes" (Almon 1994), which allow exogenous control of the model. Fixes are treated as part of each regression equation even though the fix is applied after the regression equation is evaluated. SOR forms a linear combination of the last model prediction and the "fixed" new model prediction.

V. Broyden's Method

Newton devised a method for solving simultaneous equations (and the similar problem of optimization) by assuming that the system was approximately quadratic. First, define R(y) = y – F(y), where R is a vector of residuals, y is a vector of dependent variables, and F denotes the set of equations. A Taylor approximation is given by

$$R(y^{k+1}) = R(y^k) - J(y^k) * (y^{k+1} - y^k) \tag{1.1}$$

where J is the Jacobian at vector y and *k* is the current iteration number. By setting the equations equal to zero, the following equation emerges which can yield a solution by iteration:

$$y^{k+1} = y^k - (J^k)^{-1} * F(y^k) \tag{1.2}$$

where J^{-1} is the inverse of the Jacobian. This method can yield rapid convergence for starting values that are sufficiently close to the solution[1]. Broyden observed that calculation of the Jacobian is computationally expensive. He thus devised an updating algorithm so that the Jacobian seldom (or possibly never) needs to be estimated.

Consider the following equation, which rearranges Equation 1.1 and introduces new notation:

$$B^k dy^{k+1} = -R^k \Rightarrow y^{k+1} \Rightarrow R^{k+1} \tag{1.3}$$

where B^k is the *k*-th approximation of the Jacobian and $dy^{k+1} = y^{k+1}-y^k$. At iteration *k*, the matrix B^k and the vector R^k are known. This system can be solved for dy^{k+1}, which yields y^{k+1} and R^{k+1}. Before computing the next iteration of Equation 1.2 or 1.3, the Jacobian approximation must be updated. Secant conditions dictate that

$$B^{k+1} dy^{k+1} = dR^{k+1} \tag{1.4}$$

[1] The routine described above is prone to failure if certain conditions are violated. The method actually employed is more robust (see Press, et. al. 1992). Before accepting y^{k+1}, $R(y^{k+1})$ is evaluated. If $R^{k+1}>R^k$, then a new value of y is chosen such that $y = y^k + \lambda y^{k+1}$, where $0<\lambda<1$. λ may be decreased until the new y yields lower values for R. Hence, the method is globally convergent.

where $dR^{k+1} = R^{k+1}-R^k$. Unfortunately, these conditions are not sufficient to uniquely define B for n > 1. Broyden suggests choosing the B^{k+1} "closest" to B^k while meeting the secant conditions. B^{k+1} then is defined by the iterative equation

$$B^{k+1} = B^k + \frac{(dR^{k+1} - B^k dy^{k+1}) \otimes dy^{T^{k+1}}}{dy^{T^{k+1}} dy^{k+1}}. \qquad (1.5)$$

To implement this method, four stages are required. First, any predetermined variables are computed as described above. Next, y is initialized, and the system is solved using Equations 1.3 and 1.5. Finally, y is copied and stored as model variables. This process is portrayed in Appendix H, which shows the functions for the test model.

The graph in Appendix F shows that Broyden's method requires few function evaluations than does SOR for this model; Broyden's method requires 796 evaluations while SOR requires 845 for an approximately optimal value for ω. Broyden's method does not require 796 iterations of Equations 1.4 and 1.5. Instead, 796 is the number of iterations on those equations plus *n* (3) multiplied by the number of times (81) the Jacobian must be evaluated. In this case, the Jacobian is evaluated at the beginning of each period. After that, B is updated using Equation 1.5. While it may be possible to carry B across time periods, the method fails frequently and almost always requires extra iterations to establish a sufficiently accurate B matrix. It seems best to compute the Jacobian every period even if it is not strictly required. If the system fails to converge, the Jacobian is computed again. The equations in the test model are well behaved, so the Jacobian is computed only once per period.

While in this case Broyden's method requires fewer function evaluations than does SOR, SOR is likely to be faster. Fixed point iteration simply evaluates the equations, checks for convergence, and repeats as necessary. Broyden's method requires a lot of "machinery" to operate in the background. Such machinery includes the operations required in Equation 1.5 and possibly the steps described in Footnote 1. Hence, a far greater reduction in equation evaluations is required for Broyden's method to prove more efficient.

Broyden's method seems most useful for systems that converge slowly or diverge for iterative methods. For models like *AMI*, *Quest*, and *IdLift*, SOR is preferable because it is easier to implement and it probably is faster. Still, Broyden's method may be desirable for some macro models or for inner loops in some Interdyme models.

Perhaps the most typical problem when employing Broyden's method for large models is singularity of the Jacobian. The software here automatically eliminates duplicate equations and

warns the user if a variable is redefined. The user is left to fix the latter problem and any other difficulties leading to failure. None of this, except perhaps for inconsistent equations, is a problem for iterative methods. In this sense, SOR is far more reliable and is much easier to implement. Press et. al. (1992) recommend scaling y and R to be of order unity. No such scaling is done here. While the algorithm continues to work in cases where those conditions are not violated badly, appropriate scaling must be done to ensure reliable performance.

VI. Conclusion

This paper describes my experience with three methods that can aid the search for model solutions. Each method is tested on a simple model to demonstrate feasibility and performance. While experiments with a single small model cannot offer conclusive evidence, we can draw several tentative conclusions.

First, arranging equations to be solved in a single evaluation is relatively easy, at least for macro models and for portions of inter-industry models. Given the potential for significant increases in speed, such steps should be taken wherever possible. Second, relaxation methods can be implemented with a simple change of functional form. These methods can bring convergence when Seidel iteration fails or can speed convergence for well behaved systems. When solving many similar models, a learning algorithm may help the user to find approximately optimal parameters that offer superior performance. Finally, Broyden's method may offer solutions when other routines fail, but it is relatively difficult to implement and may be inefficient in terms of memory and speed.

These methods may prove most useful when solving similar models many times, as is required by Almon's maximization algorithm. To improve performance in that process, it seems best to begin by rearranging the equations and solving the system with SOR. These methods alone offer too little improvement to make maximization feasible for Interdyme models. Currently, the maximization problem is solved by the downhill simplex method (Press, et. al. 1992). While this method does not require estimation of derivatives, it still converges slowly. Adoption of another method, for example Newton's method, may dramatically reduce the number of times that the model must be solved. By reducing the number of required solutions and also speeding convergence at each point, maximization may prove feasible even for large models.

Bibliography

Almon, Clopper. The Craft of Economic Modeling: Part 1. 3rd ed. Interindustry Economic Research Fund, Inc. 1994.

---. The Craft of Economic Modeling: Part 3. Forthcoming. 2001.

---. "Experience with Optimization in Inforum Models." Presented at the Ninth Inforum World Conference, Genzersee, Switzerland, September 2001.

IdLift User Guide: Version 1.3. Interindustry Economic Research Fund, Inc. 2001.

"*Interdyme:* A Package of Programs for Building Interindustry Dynamic Macroeconomic Models." Inforum. 2000.

Hageman, Louis A., and David M. Young. Applied Iterative Methods. New York: Academic Press, 1981.

Judd, Kenneth L. Numerical Methods in Economics. Cambridge, Massachusetts: The MIT Press, 1998.

Meade, Douglas. "Interdyme Report #4: The Triangulation Ordering." Inforum, March 1996.

Press, William, et. al. Numerical Recipes in C: The Art of Scientific Computing. 2nd edition. Cambridge: Cambridge University Press, 1992.

Weiss, Rudiger. Parameter-Free Iterative Linear Solvers. Berlin: Akademie Verlag, 1996.

Appendix A

$$C = a_1 + a_2 * Y + a_3 * \Delta Y$$
$$I = b_1 + b_2 * Y + b_3 * i^{rtb10}$$
$$Y = C + I + Other$$
$$i^{rtb} = c_1 * i_{t-1}^{rtb} + c_2 * i_{t-2}^{rtb} + c_3 * i_{t-3}^{rtb} + c_4 * i_{t-4}^{rtb}$$
$$i^{rtb10} = d_1 * i_{t-1}^{rtb10} + d_2 * i_t^{rtb}$$

Appendix B

$$i^{rtb} = c_1 * i_{t-1}^{rtb} + c_2 * i_{t-2}^{rtb} + c_3 * i_{t-3}^{rtb} + c_4 * i_{t-4}^{rtb}$$
$$i^{rtb10} = d_1 * i_{t-1}^{rtb10} + d_2 * i_t^{rtb}$$
$$----------$$
$$C = a_1 + a_2 * Y + a_3 * \Delta Y$$
$$I = b_1 + b_2 * Y$$
$$Y = C + I + Other$$

Appendix C

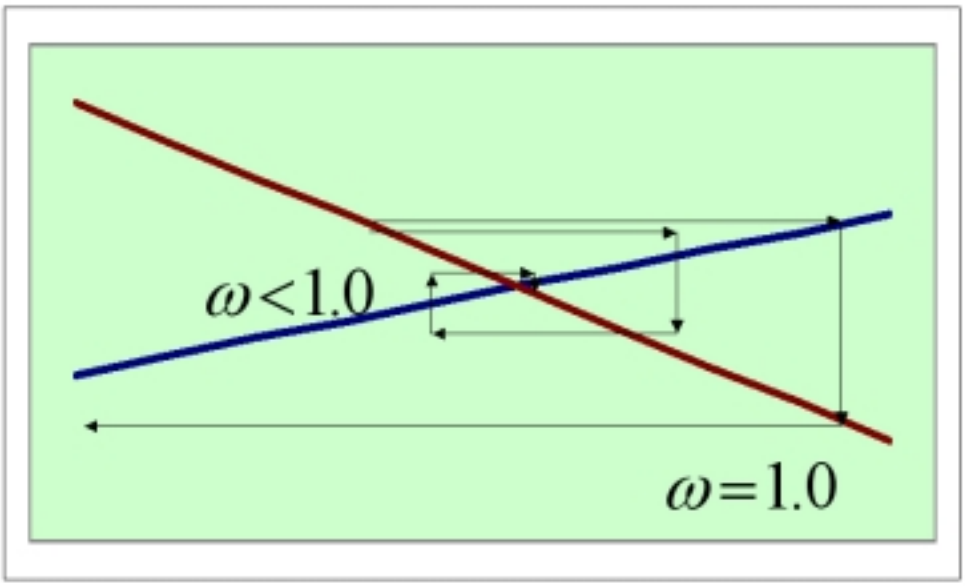
An abridged version of the functions and equations in C++ form.

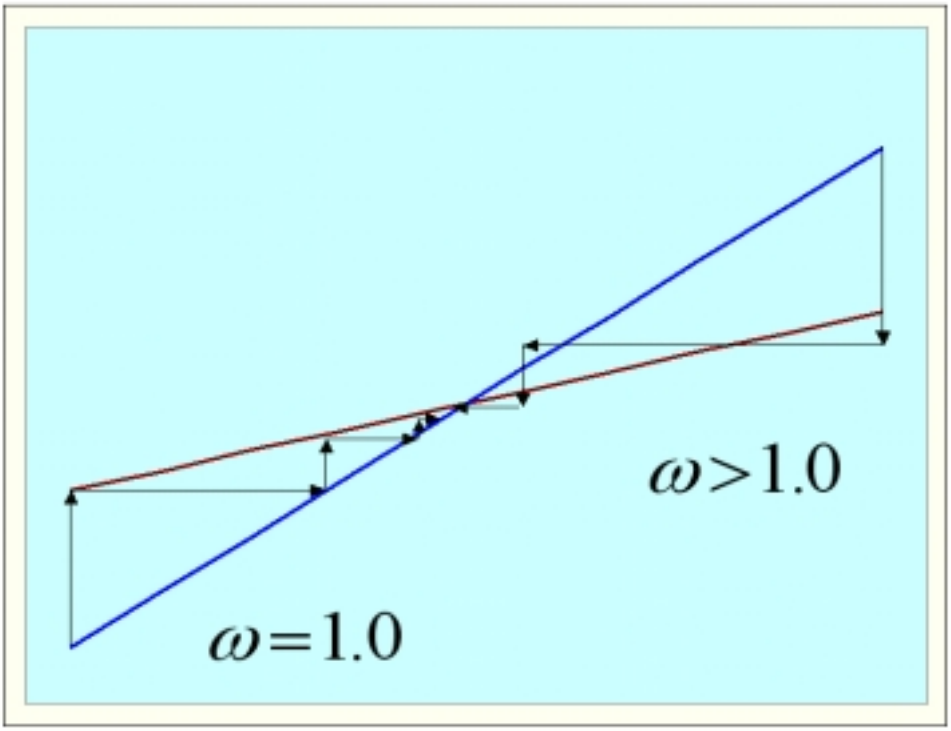
```
void heart1(void){
/*  cnR Real Consumption */
   vp[4][t] = vp[2][t]/ vp[3][t];
   vp[5][t] =  vp[4][t]- vp[4][t-1];
/* cRpc */ depend =+coef[0][0]+coef[0][1]* vp[4][t]+coef[0][2]* vp[5][t];
           modify(1,depend,0);
   vp[6][t] = vp[1][t]* vp[3][t];

/*  Gross Priv. Dom. Fixed Investment */
/* vfR */ depend =+coef[1][0]+coef[1][1]* vp[2][t]+coef[1][2]* vp[8][t];
          modify(7,depend,1);

/*  GDP */
   vp[2][t] = vp[6][t]+ vp[7][t]+( vp[9][t]+ vp[10][t]- vp[11][t]+ vp[12][t]);

/*   ti rtb Treasury Bill Rate */
/* rtb */ depend =coef[2][0]+coef[2][1]* vp[13][t-1]+coef[2][2]* vp[13][t-2]+
          coef[2][3]* vp[13][t-3]+coef[2][4]* vp[13][t-4];
          modify(13,depend,2);

/*  rtb10y 10 Year Treasury Bond Rate */
/*  rtb10y */ depend =coef[3][0]+coef[3][1]* vp[8][t-1]+coef[3][2]* vp[13][t];
              modify(8,depend,3);
}
```

Appendix D

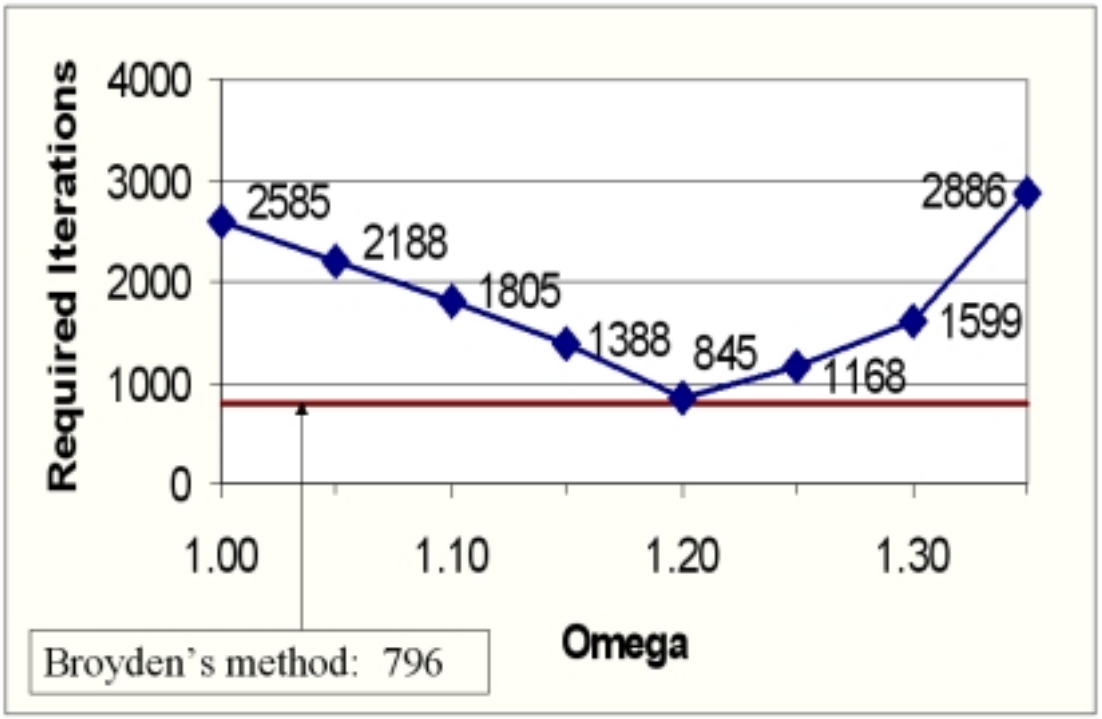
```
int hearta(void){  // The recursive equations
/*   ti rtb Treasury Bill Rate */
/* rtb */ depend =coef[2][0]+coef[2][1]* vp[13][t-1]+coef[2][2]* vp[13][t-2]+
          coef[2][3]* vp[13][t-3]+coef[2][4]* vp[13][t-4];
          modify(13,depend,2);

/*  rtb10y 10 Year Treasury Bond Rate */
/* rtb10y */ depend =coef[3][0]+coef[3][1]* vp[8][t-1]+coef[3][2]* vp[13][t];
             modify(8,depend,3);
return 1;
}

void heart1(void){  // The system of equations
/*  cnR Real Consumption */
   vp[4][t] = vp[2][t]/ vp[3][t];
   vp[5][t] =  vp[4][t]- vp[4][t-1];
/* cRpc */ depend =+coef[0][0]+coef[0][1]* vp[4][t]+coef[0][2]* vp[5][t];
           modify(1,depend,0);
   vp[6][t] = vp[1][t]* vp[3][t];

/*  Gross Priv. Dom. Fixed Investment */
/* vfR */ depend =+coef[1][0]+coef[1][1]* vp[2][t]+coef[1][2]* vp[8][t];
          modify(7,depend,1);

/*  GDP */
   vp[2][t] = vp[6][t]+ vp[7][t]+( vp[9][t]+ vp[10][t]- vp[11][t]+ vp[12][t]);

/*   ti rtb Treasury Bill Rate */
/* rtb */ depend =coef[2][0]+coef[2][1]* vp[13][t-1]+coef[2][2]* vp[13][t-2]+
          coef[2][3]* vp[13][t-3]+coef[2][4]* vp[13][t-4];
          modify(13,depend,2);
}
```

Appendix E





Appendix F



8

Appendix G

```
int hearta(void){ // Recursive equations

/*   ti rtb Treasury Bill Rate */
/* rtb */ depend =coef[2][0]+coef[2][1]* vp[13][t-1]+coef[2][2]* vp[13][t-2]+
          coef[2][3]* vp[13][t-3]+coef[2][4]* vp[13][t-4];
          modify(13,depend,2);

/*  rtb10y 10 Year Treasury Bond Rate */
/* rtb10y */ depend =coef[3][0]+coef[3][1]* vp[8][t-1]+coef[3][2]* vp[13][t];
             modify(8,depend,3);
return 1;
}

void heart1(void){ // System of equations to be solved by SOR

/*  cnR Real Consumption */
 vp[4][t]=omega*(  vp[2][t]/ vp[3][t] )+(1.-omega)*vp[4][t];
 vp[5][t]=omega*(  vp[4][t]- vp[4][t-1] )+(1.-omega)*vp[5][t];
/* cRpc */ depend =+coef[0][0]+coef[0][1]* vp[4][t]+coef[0][2]* vp[5][t];

        // Note the way fixes are handled:  They are treated as part of the function.
        tempdep = vp[1][t]; // Store old function value.
        modify(1,depend,0); // Apply fix to new function value.
        vp[1][t]=omega*(vp[1][t])+(1.-omega)*tempdep;  // Apply SOR

 vp[6][t]=omega*(  vp[1][t]* vp[3][t] )+(1.-omega)*vp[6][t];
```

```
/*  Gross Priv. Dom. Fixed Investment */
/* vfR */ depend =+coef[1][0]+coef[1][1]* vp[2][t]+coef[1][2]* vp[8][t];
          tempdep = vp[7][t];  // Note same application of fix as described above.
          modify(7,depend,1);
          vp[7][t]=omega*(vp[7][t])+(1.-omega)*tempdep;
 vp[2][t]=omega*(  vp[6][t]+ vp[7][t]+( vp[9][t]+
 vp[10][t]- vp[11][t]+ vp[12][t]) )+(1.-omega)*vp[2][t];
}
```

Appendix H

```
int hearta(void){ // Recursive system

/*   ti rtb Treasury Bill Rate */
/* rtb */ depend =coef[2][0]+coef[2][1]* vp[13][t-1]+coef[2][2]* vp[13][t-2]+
          coef[2][3]* vp[13][t-3]+coef[2][4]* vp[13][t-4];
          modify(13,depend,2);

/*  rtb10y 10 Year Treasury Bond Rate */
/* rtb10y */ depend =coef[3][0]+coef[3][1]* vp[8][t-1]+coef[3][2]* vp[13][t];
             modify(8,depend,3);
return 1;
}

void getx(double x[]){  // initialize y (called x here)
    x[1] = vp[4][t];
    x[2] = vp[5][t];
    x[3] = vp[1][t];
    x[4] = vp[6][t];
    x[5] = vp[7][t];
    x[6] = vp[2][t];
    return;
}

void funcv(int n, double x[], double f[]){  // system of equations to be solved
    /*  cnR Real Consumption */             // with Broyden's method: f = y – g(y)
    f[1]=x[1]-( x[6]/ vp[3][t]);
    f[2]=x[2]-( x[1]- vp[4][t-1]);

    /*cRpc*/ depend=+coef[0][0]+coef[0][1]*x[1]+coef[0][2]*x[2];//compute consumption
               modify(1,depend,0);                              // apply fix
               f[3] = x[3] - vp[1][t];                          // compute f

    f[4]=x[4]-( x[3]* vp[3][t]);
    /*  Gross Priv. Dom. Fixed Investment */
    /* vfR */ depend =+coef[1][0]+coef[1][1]* x[6]+coef[1][2]* vp[8][t];
            modify(7,depend,1);
    f[5] = x[5] - vp[7][t];
    f[6]=x[6]-( x[4]+ x[5]+( vp[9][t]+
     vp[10][t]- vp[11][t]+ vp[12][t]));
    /*   ti rtb Treasury Bill Rate */
    /*  rtb10y 10 Year Treasury Bond Rate */
    return;
}

void setx(double x[]){ // store solution as model variables
    vp[4][t] = x[1];
    vp[5][t] = x[2];
    vp[1][t] = x[3];
    vp[6][t] = x[4];
    vp[7][t] = x[5];
    vp[2][t] = x[6];
    return;
```