Alex Anas and Huibin Chang

Report on GAMS vs FORTRAN Comparison

August 18, 2012. Last part added August 30, 2012. Updated October 17, 2012

This report includes several parts. The first part describes the problem to be solved. The second part describes the algorithm used in the FORTRAN code; includes the flow chart of the code and discusses blemishes of the program. Part three introduces the GAMS code and the algorithm implemented by the solver. The fourth part compares the two and comments on both tools. The last part is an appendix where sample log files are listed.

In part B of the report the problem to be solved is modified and similar tests are conducted. Because this problem is easier to solve as compared to the original, larger problems are also solved.

PART A

1. The problem to be solved

The problem to be solved is one originally solved by Anas (1982). It is a housing assignment problem with fixed stock in each housing submarket i of type k housing and a population of tenants who are rent takers and utility maximizers and are distributed among the housing markets according to logit choice probabilities, P_{ikj} where j is workplace . The housing size in each submarket is fixed at h_{ik} . Profit maximizing landlords allocate each unit of the stock of housing to the market with probability q_{ik} .

More precisely, the problem is defined as follows:

i: zone of residence, i=1,…,50.

j: zone of employment, j=1,…,50.

k: type of housing, k=1,2. k=1 houses; k=2 apartments

N : number of consumers (input data)

h_{ik} : sq.meters of a housing unit of type k located in residential zone i (input data)

S_{ik} : aggregate sq.meters of housing of all units of type k located in zone i

M_j : income of consumer who works at zone employment zone j (data input)

T_{ij} : travel time from residence zone I to employment zone j (input data)

E_{ijk} : constant terms for each i,j,k. Assume that $E_{111} = 0.$ (input data)

K_{ik} : constants for each i,k (input data).

r_{ik} : rent per square meter of type k housing in zone i. These are the unknowns to be solved.

α : is the coefficient of disposable income in the utility between zero and one (input data)

φ : is the positive coefficient of rent in the landlord's supply function (input data)

The housing sub-markets at each i,k (note that there are 50 times 2 = 100 such housing sub-markets) are at equilibrium when the rents r_{ik} satisfy the following 100 equations simultaneously.

$$Nh_{ik}\sum_{j=1}^{50} \mathrm{P}_{ikj} = S_{ik}q_{ik}$$

Note: For a feasible solution to exist, there must be enough floor space.

Where the probability that a consumer works at employment zone j and chooses residence at zone I in housing type k is given by (note that the probabilities sum to one) :

$$\mathrm{P}_{ikj} = \frac{\exp(\lambda \tilde{U}_{ikj})}{\sum_{abc}\exp(\lambda \tilde{U}_{abc})}; \quad \sum_{ikj} \mathrm{P}_{ikj} = 1$$

Where the utility of (i,j,k) is:

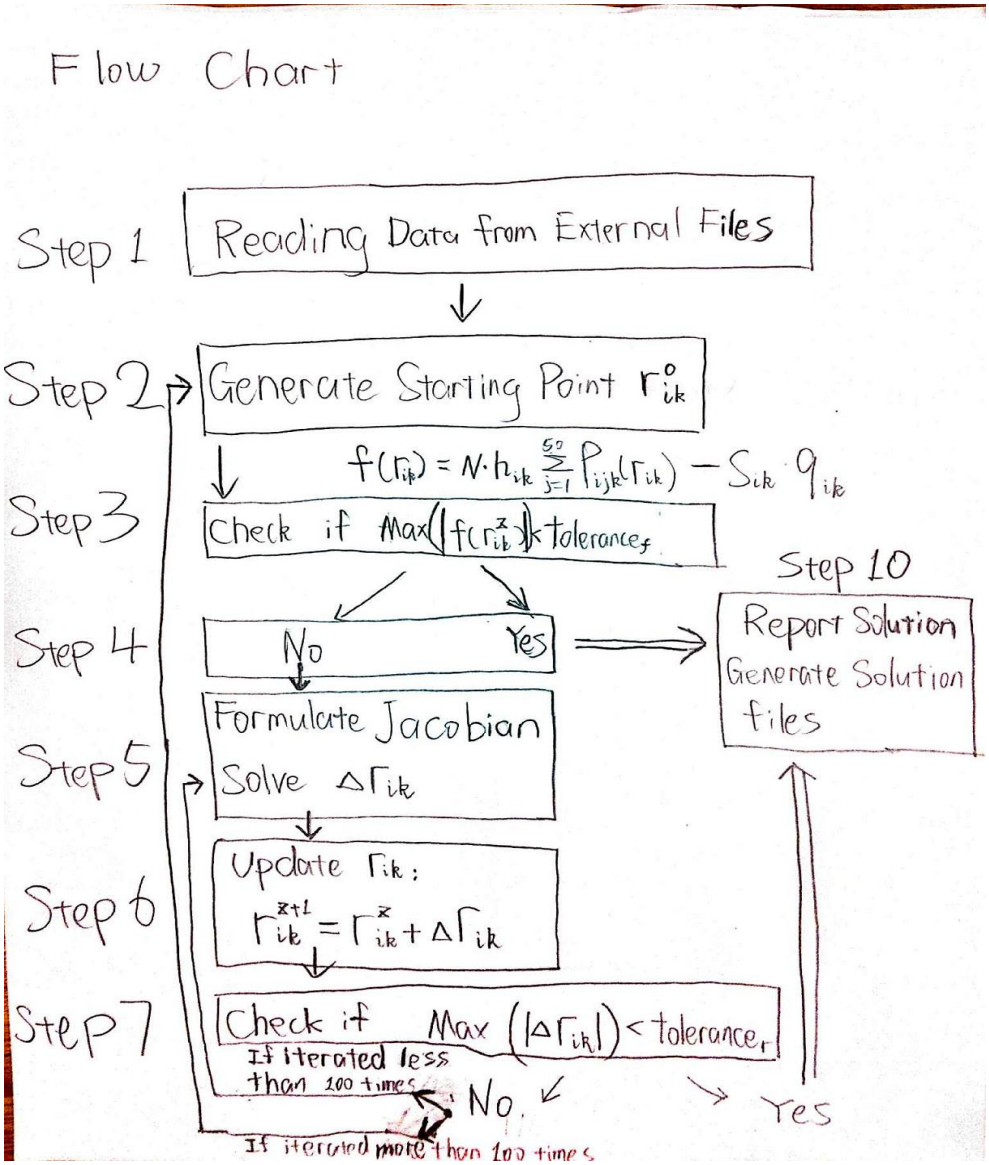
$$\tilde{U}_{ijk} = \alpha \ln\left(M_j - r_{ik}h_{ik}\right) + \left(1-\alpha\right)\ln h_{ik} + \beta \ln(T_{ij}) + E_{ijk}$$

Where the probability that a sq. meter of housing will be offered to the market by its landlord is:

$$q_{ik} = \frac{\exp\left(\phi r_{ik} + K_{ik}\right)}{1+\exp\left(\phi r_{ik} + K_{ik}\right)}$$

The problem has a unique solution because the Jacobian has the negative dominant diagonal property [Anas(1982)].

2. The FORTRAN code:



Step 1 reads data from external files. The code can be adjusted so that it can load data from other file forms.

Step 2 generates starting point for r_{ik}^{0}. If the starting point is not a "good" one, the Newton-Raphson procedure will result in one of the two following cases: (a) the gradient of r_{ik} will diverge instead of converging and this in turn will lead to the problem of the singularity of Jacobian; (b) although r_{ik} converges, but before the reduced gradient converging to close to zero, it will start "cycling" at some point and not be able to find a solution.

To make the code more efficient, something should be done: if any of the two above cases happened at any step in the program, the algorithm will branch to step 2 where a new starting point can be generated. The rule that generates a starting point is as follow:

$$r_{ik}(i\ by\ k) = \text{starting constant} \times \text{random number}\ (i\ by\ k)$$

Where the random number is an i by k matrix whose elements are random scalars between 0 and 2.

In words, each time the Newton-Raphson algorithm encounters an error or fails to converge to a point that is close enough to the true solution, the program will regenerate a new starting point r_{ik} whose elements are uniformly distributed around the mean value which equals "starting constant". What is the starting constant?

For example, if starting constant is set to 5, $r_{ik}(i\ by\ k) = 5 \times \text{random number}\ (i\ by\ k)$ where random number (i by k) is a vector whose elements are between 0 and 1. Hence, the initial guess of r_{ik} is an array whose elements are between 0 and 5. If the Newton-Raphson procedure failed to solve the problem within a preset number of iterations, the code will generate another initial guess whose elements also are between 0 and 5 but different from the last initial guess.

Step 3 will take r_{ik}^{z} and plug it into $f_{ik}(\mathbf{r}^{z})$ where $f(\mathbf{r}) = Nh_{ik}\sum_{j=1}^{50} P_{ijk}(\mathbf{r}) - S_{ik}q_{ik}(r_{ik})$ and z is a non-negative integer represents the iteration number starting from zero.

Step 4 checks if $\max_{\forall ik}\left(\left|f_{ik}(\mathbf{r}^{z})\right|\right) < tolerance_{f}$ (second stopping criterion). If yes, the program will branch to step 8 which generates solution file and solution log files; if not, the program will start the Newton-Raphson procedure.

Step 5 is the start of the Newton-Raphson procedure. According to the following system of linear equations,

$$\overbrace{\begin{pmatrix} \frac{\delta f_{1,1}}{\delta r_{1,1}^z} & \cdots & \frac{\delta f_{50,2}}{\delta r_{1,1}^z} \\ \ddots & \iddots & \\ \vdots & \frac{\delta f_{i,k}}{\delta r_{i,k}^z} & \vdots \\ & \iddots & \ddots \\ \frac{\delta f_{50,2}}{\delta r_{1,1}^z} & \cdots & \frac{\delta f_{50,2}}{\delta r_{50,2}^z} \end{pmatrix}}^{Jacobian} \overbrace{\begin{pmatrix} \Delta r_{1,1}^z \\ \Delta r_{1,2}^z \\ \vdots \\ \Delta r_{50,1}^z \\ \Delta r_{50,2}^z \end{pmatrix}}^{starting\ or\ updated\ r\ vector} = -\overbrace{\begin{pmatrix} f_{1,1}(\mathbf{r}) \\ f_{1,2}(\mathbf{r}) \\ \vdots \\ f_{50,1}(\mathbf{r}) \\ f_{50,2}(\mathbf{r}) \end{pmatrix}}^{Residuals}$$

vector Δr_{ik}^z will be solved. Within this step, the first part is to generate the Jacobian matrix, and the second part is to solve the system of linear equations using an external subroutine.

Step 6 updates the r_{ik}^z vector: $r_{ik}^{z+1} = r_{ik}^z + \Delta r_{ik}^z$.

Step 7 checks the first stopping criterion: if $\max\left(\left|\Delta r_{ik}^z\right|\right) < tolerance_r$. If "yes", exit the Newton-Raphson loop and go to Step 4 to check the second stopping criterion. If "no", carry on to the next loop. If the maximum loop number, which is set equal to 100, has reached, the program will branch to step 2 where a new randomly disturbed starting point will be generated.

Step 8 reports result and generates the solution and log files.

Comments:

Although the program is able to solve the r_{ik} provided the tolerance level is not too small, there are few problems: (a) the code is not able to get a solution when tolerance is set to lower than 7.5e-7 while GAMS is able to solve for a solution with such or higher accuracy. What happens when the tolerance level is too small is that the reduced gradient will start "cycling" at some point far from the true solution. Here reduced gradient is the terminology used in the solver manual. It refers to the solution of the Jacobian, which is the change of variables, or, Δr_{ik}. (b) For some other data sets, the FORTRAN code cannot find a solution at all regardless the starting point and the tolerance level. I believe this is because the bad scaling[1] of the model, i.e. either the coefficients is too big, say, greater than 100, or it is too small, smaller than 0.01, could cause the scaling problem which will make the convergence very unlikely. (c) The search step length in this FORTRAN code is invariant with respect to the residuals level and whether or not in some

[1] GAMS has an option which adjusts the absolute value of elements in Jacobian and Hessian. In CONOPT3, the solver will "scale" the model automatically so that the elements of Jacobian are not too much far from 1 in absolute value.

iteration the updated solution has reached it's bound. For example, $r_{ik}^{z+1} = r_{ik}^{z} + \lambda \Delta r_{ik}^{z}$ where λ is the step length and is fixed at 1 in my latest FORTRAN code. With GAMS, the CONOPT solver provides more sophisticated algorithms to adjust the step length dynamically.

3. GAMS

The GAMS code reads external data files and converts the data to GDX files which can be read directly by the GAMS system. The GAMS code and the FORTRAN code use the same data set so that they can be compared.

The solver is CONOPT3 which can be used to solve optimization problems with nonlinear objective and constraint functions as well as to solve system of nonlinear equations. To solve system of nonlinear equations, set up a dummy variable which has no relationship (implemented by using =n= relation) with the variables to be solved as the objective function will do.

The model can be directly formulated as follow:

$$f_{ik}(\mathbf{r}) = Nh_{ik} \sum_{j=1}^{50} P_{ijk}\left(U_{ijk}(r_{ik}); \forall ijk\right) - S_{ik} q_{ik}(r_{ik}) \quad i = 1,2,...,50\,; k = 1,2.$$

There are 100 equations and 100 variables. The advantage of this formulation is that the number of variables and equations are minimized (100 and 100) and it takes the smallest possible computer memory. However, this also leads to messy expressions of the equations. Moreover the time it takes to solve the problem is longer than that of using more intermediate variables in which case the memory needed is larger because there are more equations and variables.

Alternatively, the model is formulated as:

$$f(\mathbf{r}) = Nh_{ik} \sum_{j=1}^{50} P_{ijk}\left(U_{ijk}\left(mrh_{ijk}(r_{ik})\right)\right) - S_{ik} q_{ik}(r_{ik})\,; \quad i = 1,2,...,50\,;\ j = 1,2.$$

$mrh_{ijk}(r_{ik}) = M_j - h_{ik} r_{ik}$ is the argument of the log function in the utility function.

As intermediate variables, $mrh_{ijk}(r_{ik})$ serve two purposes. First, by setting their lower bound to 0.01, they can guarantee good behavior of the utility and demand functions (P_{ijk}), that is if the rent in a particular submarket (i,k) is so high as to make disposable income in that submarket zero or negative, the probability in that submarket becomes almost zero. Second, using intermediate variables, as long as there is enough computer memory to handle the variables and equations, the use of these intermediate variables will significantly reduce the execution time of solving the nonlinear system. In this case, there are 500,100 variables and 500,100 equations. If more intermediate variables were introduced, the solver will exit with a message

stating there is not enough memory. More features of the GAMS system and CONOPT solver will be discussed in the next section.

4. Comparison

Comparing the codes described above, GAMS does a better job in general solving the system of nonlinear equations.

The current FORTRAN code is a direct implementation of the Newton-Raphson algorithm, which is not best suited for nonlinear equations whose variables have upper and lower bounds. But given that the tolerance of the maximum absolute reduced gradient is set at 1.0e-5 and the process starts from a "good" starting point, the FORTRAN program will solve the system much faster. See the table below. But when the tolerance level is even smaller, or the starting point is far from the true solution, the FORTRAN code might not be able to solve the problem while GAMS could.

Another virtue of FORTRAN is that the programmer knows what is happening under the hood. The algorithm can be modified if the programmer knows a better algorithm. Due to the same reason, debugging is easier for FORTRAN in that the programmer could inspect each step of the code being executed while debugging in GAMS requires a decent understanding of the solver and possibly the algorithms employed by the solver.

The following are some virtues of GAMS and CONOPT.

1. Simple expression. In general a model can be presented to GAMS in a more concise and human-readable way which is shorter and clearer.
2. Automatic scaling. CONOPT3 can scale the system accordingly so that more coefficients of the Jacobian will fall into the region between 0.01 and 100. For example, for some data sets I created which cannot be solved by FORTRAN due to the scaling problem, CONOPT can solve the system in seconds.
3. Easier to modify models. Solvers in GAMS can generate the Jacobian and Hessian automatically according to the input model, this, in turn, can save a lot of time considering that in FORTRAN, the Jacobian and Hessian cannot be generated automatically in general. Hence, when dealing with different but similar models or models that might need to be modified in some way at any time, GAMS has a significant advantage.
4. Sophisticated algorithms and facilities of built-in solvers. In FORTRAN, this has to be done manually. That means that the programmer has to fully understand the algorithms and write the code from scratch. In contrast, CONOPT is able to use multiple built-in algorithms to improve the solution. It first uses the Newton algorithm, then it is able to improve the solution to a high degree of accuracy by adding dummy variables to the equations with large residuals, then it uses other search methods (Sequential Linear Programming or Sequential Quadratic Programming, SLP and SQP) to improve the

solution further. Moreover, CONOPT is able to adjust the search step length at each iteration according to the result of the last iteration; this can increase the speed of convergence for problems whose variables have bounds.

5. Unlike FORTRAN, CONOPT can find the solution without been given a starting point. This is useful when we have no idea what would be the benchmark level of the true solution or what are the least upper bounds and greatest lower bounds for the solution. However, providing bounds and good starting points, accompanied with intermediate variables whose initial values are also decided by the starting point can shorten the execution time, sometimes significantly.
6. Solving optimization problems. At this point, we may conclude that formulating optimization problem in GAMS is more simper than formulating the same problems in FORTRAN. And as mentioned earlier, modification of the problem and carrying out comparative static analysis is much more straightforward in GAMS.

In general, GAMS is more efficient if the problem can be solved by available solvers. Theoretically FORTRAN can do the same, but it usually needs a longer code. Even if the programmer is very fluent in FORTRAN, he or she has to fully understand the algorithms needed in order to implement it. This might be unnecessary in some cases. On the other hand, if there is no suitable solvers for the problem presented to us, FORTRAN gives an option.

Table 1 and 2 below summarize the execution times under different tolerance levels and different starting points.

In most cases, FORTRAN took shorter time finding a solution under a greater tolerance level and good starting point. However, GAMS could solve the problem even without being given an initial value and the solution is of a higher degree of accuracy. The time GAMS took is not very much longer than that taken by FORTRAN.

Table 3 compares the features of the two programs.

Table1 GAMS execution time and number of iterations w.r.t. different tolerance levels

GAMS	Max Reduced Gradient=3e-13	Max Reduced Gradient=1e-7	Max Reduced Gradient=1e-6	Max Reduced Gradient=1e-5
No initial value	Data Not Available (DNA)	3339s/49it	DNA	DNA
Initial Value r(i,k)=5	DNA	DNA	DNA	DNA
Initial Value r(i,k)=2	DNA	DNA	3332s/34it	DNA
Initial Value r(i,k)=1.5	DNA	2557s/59it	2589s/59it	DNA
Initial Value r(i,k)=1	276s/10it	288s/10it	266s/10it	DNA
Initial Value r(i,k)=0.5	DNA	DNA	302s/14it	300s/10it
Initial Value r(i,k)=0.1	DNA	DNA	374s/20it	349s/14it
Initial Value r(i,k)=0.01	DNA	DNA	419s/25it	409/20it

Table2 FORTRAN execution time and number of iterations w.r.t. different tolerance levels

FORTRAN	Max Reduced Gradient=1e-7	Max Reduced Gradient=1e-6	Max Reduced Gradient=1e-5
Initial r(i,k)=5 or r(i,k)=4	Not Solvable (NS)	NS	DNA
Initial Value r(i,k)=3	NS	NS	1.8s/23
Initial Value r(i,k)=2	NS	158s/95it	2.2s/34it
Initial Value r(i,k)=1.5	NS	2096s/61it	1.6s/23it
Initial Value r(i,k)=1	NS	363s/57it	1.5s/21it
Initial Value r(i,k)=0.5	NS	549s/40it	1.7s/23it
Initial Value r(i,k)=0.1	NS	1147s/29it	2.8s/38it
Initial Value r(i,k)=0.01	NS	663s/82it	1.3s/19it

Table3 Comparing FORTRAN and GAMS

	FORTRAN	GAMS/CONOPT3
Ability to solve without starting point	No	Yes
Automatic Scaling	No	Yes
Access to Source Code of the Solver	Yes	No
Modify Algorithm	Yes but maybe cumbersome	Yes by choosing other available solvers
Modify Model	Fast	Cumbersome
Generate Jacobian and Hessian automatically	No	Yes
Set Up Optimazation Problem	Relatively easy	Relatively cumbersome
Comination of Algorithms to Improve Solution	Need to understand Algorithms and implement them	Yes, provided by the solver
Accuracy	Current code, low: 7.5e-7.	High: 3e-13 (also default minimum)

5. Appendix

Here is a sample FORTRAN solution log (written in txt files):

execution@20120818

elapsed time= 663.0

Number of Newton-Raphson iteration= 82

maximum absolute value of reduced gradient= 9.637E-07

maximum absolute value of residual= 7.813E-02

sum of maximum absolute reduced gradient= 4.034E-05

sum of maximum absolute residuals= 7.246E-01

tolerance 1.0E-06

Here is a sample of part of the GAMS output log (generated automatically, also in txt file):

```
--- Executing CONOPT: elapsed 0:02:05.009

CONOPT 3        Jul  4, 2012 23.9.1 WIN 33924.33953 VS8 x86/MS Windows

Reading parameter(s) from
"D:\Dropbox\Dropbox\Computing\SUBMISSION\GAMS\Data1\conopt.opt"

>>  rtredg = 1.e-8

Finished reading from
"D:\Dropbox\Dropbox\Computing\SUBMISSION\GAMS\Data1\conopt.opt"

   C O N O P T 3   version 3.15F

   Copyright (C)   ARKI Consulting and Development A/S

                   Bagsvaerdvej 246 A

                   DK-2880 Bagsvaerd, Denmark

  Iter Phase Ninf   Infeasibility   RGmax    NSB   Step InItr MX OK

    0  0       9.5765239138E+05 (Input point)

                       Pre-triangular equations:       0

                       Post-triangular equations:      1

    1  0       9.5765239138E+05 (After pre-processing)

    2  0       1.7819838768E+02 (After scaling)

    3  0    0  3.5298355138E+01            9.0E-01     T  T

    4  0    1  4.2223286869E-01            9.9E-01     T  T

    5  0    1  4.2223286869E-01            0.0E+00     T  T

    6  0    1  3.2156527501E-02            1.0E+00     F  T

    7  0    1  2.6773966159E-03            1.0E+00     F  T

  Elapsed time      83.7 seconds.

  Iter Phase Ninf   Infeasibility   RGmax    NSB   Step InItr MX OK

    8  0    1  2.1402354826E-03            1.0E+00     F  T
```

```
   9   0    1  2.1289107081E-03           1.0E+00     F  T

  10   0    1  2.1284524595E-03           1.0E+00     F  T

 ** Feasible solution. Value of objective =   1.000000000000E-02

  Elapsed time      121.5 seconds.

  Iter Phase Ninf     Objective     RGmax    NSB   Step InItr MX OK

   11   3       1.0000000000E+03 1.0E+00     1 1.0E+00    1 T  T

   12   3       1.0000000000E+03 0.0E+00     0

** Optimal solution. There are no superbasic variables.

--- Restarting execution

--- Untitled_1.gms(145) 0 Mb

--- Reading solution for model tran

--- Untitled_1.gms(145) 3 Mb

*** Status: Normal completion

--- Job Untitled_1.gms Stop 01/02/07 23:39:05 elapsed 0:04:36.593
```

PART B

Here is a report on some further tests of FORTRAN and GAMS. These tests includes a) reformulation of the utility function and reformulating the equilibrium accordingly; b) change of stopping rules of the FORTRAN codes so that the error terms are measured in relative rather than absolute magnitudes; c) changes of data sets so that population is greater, income and demand for lot size are more diverse, and more realistic; d) change of size of the model (that is the number of model submarkets).

The general upshot of these tests is that GAMS is more stable, more accurate, easier to program and modify, and able to solve the nonlinear systems without a starting point in my tests. However, it is slower in execution. The time it needs to solve the problem is longer compared with FORTRAN, and if the starting point is not good or there is no starting point, GAMS needs couple of hours to solve the model with size i=50, j=50, k=2. Another fatal shortcoming of GAMS is that the physical memory it needs to solve a large

model is too big which leads to fails when i>65[2]. Lastly, although the solvers I have been testing are powerful, they have fewer options for users to customize. This will be explained with an example in part b. When i=j=50, the model also solved by PATH and MILES.

FORTRAN, On the other hand, is faster during execution[3], accurate enough although not as accurate as GAMS. With a starting point which isn't even close to the correct solution, it only spent one and half minutes to solve the model with 1000 variables and 1000 equations (i=500). Also, as opposed to GAMS, there is no memory limitation problem in my tests, and it is possible to trim the algorithm as needed. But as pointed before, it is very likely that it takes significantly longer time for a fluent FORTRAN programmer to write, solve and modify a model than it does for a fluent GAMS user.

1. Modifying the problem to be solved

In the following tests, the indirect utility function is as follows:

$$v_{ijk} = \ln(M_j) - (1-\alpha)\ln(r_{ik}) - \beta \ln(T_{ij}) + c_{ijk}$$

$$P_{ijk} = \frac{\exp\left(\lambda v_{ijk}\right)}{\sum_{i'}\sum_{k'} \exp\left(\lambda v_{i'j'k}\right)}$$

Note that the utility function is $u_{ijk} = z_{ijk}^{\alpha} h_{ijk}^{1-\alpha} T_{ij}^{\beta} E_{ijk}$, where E_{ijk} are the constants and z_{ijk} is the composite good whose price is 1.

The Marshallian demand for lot size by consumer j for house type ik is:

$$h_{ijk} = \frac{(1-\alpha)M_j}{r_{ik}}$$

Equilibrium conditions for each *ik*:

$$N\sum_j h_{ijk} P_{ijk}\left(\mathbf{r}\right) = S_{ik} q_{ik}\left(r_{ik}\right)$$

Alternatively, it can be written as:

$$N(1-\alpha)\sum_j M_j P_{ijk}\left(\mathbf{r}\right) = S_{ik} q_{ik}\left(r_{ik}\right) r_{ik}$$

[2] According to www.gams.com, in Window 32bit version, GAMS limits itself and any solver to process around 1.8 GB. This limit will be reached if our model has too many variables or equations.

[3] But it is very likely that it takes longer to write codes in FORTRAN than it does in GAMS, and this advantage of user friendliness of GAMS could make up for its slow execution.

In this case, the lot size demanded, h_{ijk}, is not constant for each ik; it also depends on consumer's income M_j. In solving the problem, there is no bound need to be set for r_{ik}, and this improves the performance of the FORTRAN codes.

2. Converging on relative magnitudes

In this part I will describe the new stopping rules in the FORTRAN codes. After each Newton-Raphson iteration, the codes will check if:

i) $$\max\left\{\frac{\left|r_{ik}^{k+1}-r_{ik}^{k}\right|}{\frac{1}{2}(r_{ik}^{k+1}+r_{ik}^{k})}\right\}_{ik} < tolerance$$

ii) $$\max\left\{\frac{\left|D_{ik}^{k+1}-\hat{S}_{ik}^{k+1}\right|}{\frac{1}{2}(D_{ik}^{k+1}+\hat{S}_{ik}^{k+1})}\right\}_{ik} < ftolerance$$ where $D_{ik}=N\sum_j h_{ikj}P_{ikj}$ and $\hat{S}_{ik}\equiv S_{ik}q_{ik}$.

If both of the above are satisfied, the code will report results. In FORTRAN codes, when setting tolerance = 1e-6 and ftolerance = 1e-5, and when using different data sets including the one with i=500, the codes are able to solve the system in less than two minutes.

As of tolerance level of GAMS, it is less straightforward to customize. For example, in CONOPT, there are different tolerance options been used in solving a model. "retredg" is the optimality tolerance which set the maximum tolerance level for Δr_{ik}^{z}. CONOPT also has another parameter "rtnwma". This is maximum feasibility[4], or the maximum residual tolerance. Since the greatest rtnwma acceptable to the solver is 1e-3, in all the tests I have been done, rtnwma is always tighter even if I set rtnwma to its maximum and retredg to its minimum.

3. Different data

I have tested 3 different data sets comparing FORTRAN and GAMS, each system's performance is similar across different data sets[5].

[4] A constraint will only be considered feasible if the residual is less than rtnwma times MaxJac, independent on the dual variable. MaxJac is an overall scaling measure for the constraints computed as max(1,maximal Jacobian element/100). The default value of rtnwma is 1.e-7

[5] Each data set is generated such that there is a correct solution known to the user which if the user plug it into the system, it is in equilibrium.

For the first data set I changed population to 100,000 (originally 10,000). The second data set I changed income upper and lower bounds so that the income disparity across regions is larger. In the last data set I have changed population to 1,000,000 and kept the income disparity in the second data set. The following tables present results using previews data set and the third data set, solved in the updated model.

Tolerance, ftolerance and rtnwma are the same as defined in part b.

Table 4: Same data set as before. i=j=50, k=2. n=10,000.

FORTRAN	Initial r(i,k)=100	Initial Value r(i,k)=1	Initial Value r(i,k)=1e-5
tolerance=1e-6 ftolerance=1e-5	0.41s/6iterations	0.34s/5iterations	0.39s/7iterations

GAMS	No initial value	Initial Value r(i,k)=1	Initial Value r(i,k)=1e-5
rtnwma=1e-3	DNA	264s/2iterations	Longer than 1h
rtnwma=1e-6	1h48m/59iterations	319s/10iterations	DNA

Table 5: Using the third data set. i=j=50, k=2. N=1,000,000. Income more dispersed.

FORTRAN	Initial r(i,k)=100	Initial Value r(i,k)=1	Initial Value r(i,k)=1e-5
tolerance=1e-2	0.18s/5iterations	0.1s/4iterations	0.23s/5iterations
tolerance=1e-3	0.2s/5iterations	DNA	0.21s/6iterations
tolerance=1e-6	0.41s/6iterations	0.34s/5iterations	0.39s/7iterations
GAMS	No initial value	Initial Value r(i,k)=1	Initial Value r(i,k)=1e-5
rtnwma=1e-3	DNA	301s/20iterations	Longer than 30m
rtnwma=1e-6	1h48m/59iterations	420s/40iterations	DNA

4. Larger problem size

This part describes the tests with larger models.

FORTRAN codes are able to solve a system with 1000 variables and 1000 equations. It only takes less than two minutes.

On my computer, GAMS cannot solve the problem when i>65. GAMS will exit the execution during generation of the model (before solver has been invoked). Even if I formulate the model so that the number of variables and equations are minimized, the memory it takes when i>65 is still greater than 1.8 GB. Supposedly, a machine with larger physical memory would not have this problem and may be able to solve the model. The following tables summarize the information of larger models solved by FORTRAN.

Table 6: i=100. n=1,000,000. There are 200 variables and 200 equations.

FORTRAN	Initial r(i,k)=100	Initial Value r(i,k)=1	Initial Value r(i,k)=1e-5
tolerance=1e-6 ftolerance=1e-5	1.7s/6iterations	1.7s/6iterations	2s/7iterations

Table 7: i=500. n=1,000,000. There are 1000 variables and 1000 equations.

FORTRAN	Initial r(i,k)=50	Initial Value r(i,k)=1	Initial Value r(i,k)=1e-5
tolerance=1e-5	65s/6iterations	87s/8iterations	83s/7iterations

PART C

This part summarizes the result of another comparison in which the GAMS code is rewritten so that it uses the exact same algorithm and relative tolerance level as FORTRAN code does, and then the times needed for solving the problem with different starting points are recorded.

In this comparison, a different data set is used where there are 15 zones of residence and 15 zones of employment, and 2 types of buildings.

FORTRAN code is the same as before.

GAMS code mimics the procedure of the FORTRAN code. The steps of the GAMS code are as follow:

Step 1

Read data from excel files.

Step 2

Using data calculate elements of Jacobian $\frac{\delta f_{i,k}}{\delta r^z_{l,m}}$ and function values $f_{i,k}(\mathbf{r})$.

Step 3

Define “Equations” described as follow, where Jacobian elements and function values are taken as constants (referred as parameters in GAMS). $\Delta r^z_{i,k}$ are variables to be solved.

$$\overbrace{\begin{pmatrix} \frac{\delta f_{1,1}}{\delta r^z_{1,1}} & \cdots & \frac{\delta f_{50,2}}{\delta r^z_{1,1}} \\ \ddots & & \iddots \\ \vdots & \frac{\delta f_{i,k}}{\delta r^z_{i,k}} & \vdots \\ \iddots & & \ddots \\ \frac{\delta f_{50,2}}{\delta r^z_{1,1}} & \cdots & \frac{\delta f_{50,2}}{\delta r^z_{50,2}} \end{pmatrix}}^{Jacobian} \overbrace{\begin{pmatrix} \Delta r^z_{1,1} \\ \Delta r^z_{1,2} \\ \vdots \\ \Delta r^z_{50,1} \\ \Delta r^z_{50,2} \end{pmatrix}}^{starting\ or\ updated\ r\ vector} = -\overbrace{\begin{pmatrix} f_{1,1}(\mathbf{r}) \\ f_{1,2}(\mathbf{r}) \\ \vdots \\ f_{50,1}(\mathbf{r}) \\ f_{50,2}(\mathbf{r}) \end{pmatrix}}^{Residuals}$$

Step 3

Invoke solver to solve the linearized system described above. In our comparison, different solvers are used and times they take are recorded.

Step 4

Update the Jacobian elements and function values using the solution from step 3.

Step 5

Check convergence using relative tolerance which is the same level as in FORTRAN code.

$$\max_{ik}\left\{\frac{\left|r_{ik}^{k+1}-r_{ik}^{k}\right|}{\frac{1}{2}(r_{ik}^{k+1}+r_{ik}^{k})}\right\} < tolerance$$

If the above condition is satisfied, then the procedure will report result and exit; otherwise it will go to step 3, solve the linearized system again using updated Jacobian elements and function values.

We could have used tolerances for function values too, but since the relative tolerance for variable is 0.1%, and this is a tighter tolerance than function value tolerance to be set at 0.1%, we decided to drop that tolerance for simplicity.

Results:

Table 8

Starting Point	User Specified Jacobians				Direct Equilibrium Equations		FORTRAN
solver	NLP/CONOPT	NLP/MILES	MCP/PATH	LP/Cplex	NLP/CONOPT	NLP/MILES	
Starting Pt r(I,k)=0.001	114s	112s	112s	114s	14s	13s	0.03s
Starting Pt r(I,k)=1	99s	109s	119s	93s	14s	12s	0.03s
Starting Pt r(I,k)=3	106s	99s	101s	98s	15s	16s	0.01s
Starting Pt r(I,k)=5	95s	100s	112s	98s	38s	41s	0.01s
Starting Pt r(I,k)=10	114s	109s	133s	125s	12s	16s	0.01s
Starting Pt r(I,k)=100	165s	176s	166s	192s	10s	25s	0.03s

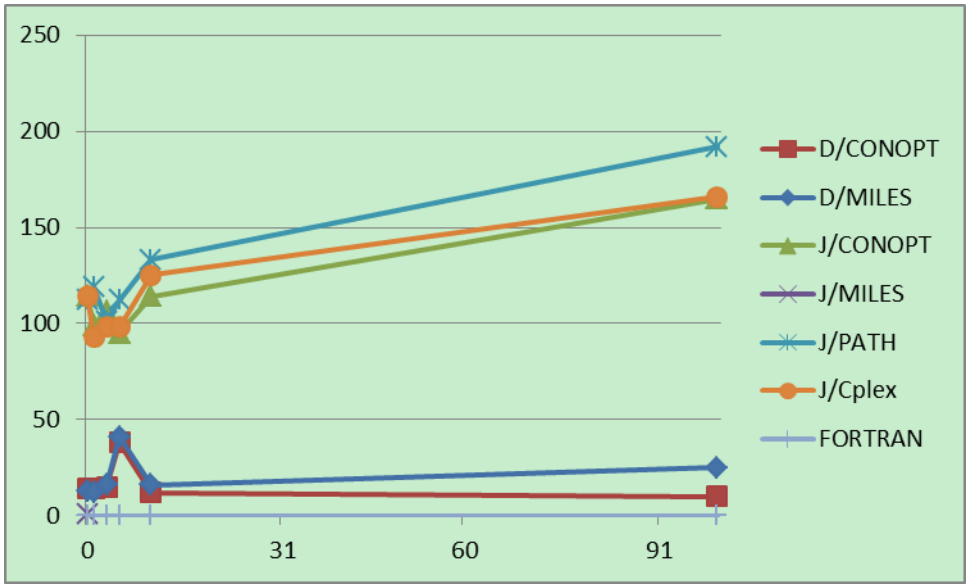
Table 8 includes the result from the above procedure as well as the result from solving the problem by only give GAMS the equations of equilibrium and the result from solving the problem using FORTRAN code.

The purple colored part or the “User Specified Jacobians” part of the table is the result from implementing the procedure just described. The dark pink-yellow part is the result from solving the problem by only giving GAMS the equations of equilibrium then leaving everything to the solvers. The brighter pink part of the table is the result from solving the problem using FORTRAN code.

As we can see from the table, by testing different solvers, given the same relative tolerance level, FORTRAN program is faster than GAMS when both implement Newton-Raphson procedure directly. But notice that even when ***relative tolerance in Newton-Raphson procedure*** is the same, the default tolerances are used by GAMS when solving the linearized system. This could be one of the reasons why GAMS is slower than FORTRAN even was been given the Jacobian.

Essentially, by this formulation in GAMS, it is solving a linear system by certain solver over and over again until the tolerance is satisfied. Previously, when we only give the equations of equilibrium, what GAMS does is to solve the non-linear system for just one time.

Figure 1



In figure 1, horizontal axis represents the starting point; vertical axis represents the time needed to solve the problem. Each line in the figure represents the time needed by certain solver/algorithm given different starting points.

J/CONOPT means the problem is solved by CONOPT and the Jacobian is specified by us, as described above.

D/CONOPT means that the problem is solved by giving only the equations of the equilibrium, and the solver will take care of Jacobian etc.

Line labeled as FORTRAN, which almost overlapped with horizontal axis, represents the time needed by FORTRAN program to solve the problem given starting point.

In words, we could duplicate the algorithm used by FORTRAN code in GAMS, but GAMS is slower. The only main difference between the two is that GAMS uses a solver to solve the linearized Jacobian system while FORTRAN uses a user provided subroutine to do that. We believe that the significant difference in time comes from here.

Also, by using user specified Jacobian system and implement Newton-Raphson procedure, GAMS is slower than that if only the original equilibrium equations were given. Our observation is that, By specifying Jacobian system, GAMS needs to solve a linear system, which is faster than solving a non-linear system, but this solving process has to be done many times until the tolerance is satisfied. On the other hand, if we only give the original equations to GAMS, it needs to solve a more complex non-linear system, but for just one time.

PART D

This part includes three codes. The first one is the GAMS code solving the problem by only giving the original equilibrium equations. The second code is the GAMS code solving the problem by directly specifying the Jacobian system. The last one is the FORTRAN code that solves the same problem.

1. GAMS with original equilibrium equations

```
 *Huibin Chang Test. Last updated 2012 Aug 14th.
$title Direct Equilibrium Equations using MILES
$onsymlist onsymxref


SETS

      I   zone of residence    / 1*15 /

      J   zone of employment   / 1*15 /

      K   type of housing      / 1*2 / ;


*alias is used when formulating the equilibrium equations
alias(i,ii)
alias(j,jj)
alias(k,kk);


*The following part which includes $call, gdxin, load is the part import


*data from excel files.
$CALL GDXXRW.EXE m.xls par=m rng=a1:b15 rdim=1
PARAMETERS

      m(j)  income of each employment zone
$GDXIN m.gdx
```

```
$LOAD  m
$GDXIN


$CALL GDXXRW.EXE s.xls par=s rng=a1:c16 Cdim=1 rdim=1
PARAMETERS

      s(i,k)  total sq. meters of each residence zone
$GDXIN s.gdx
$LOAD  s
$GDXIN


$CALL GDXXRW.EXE h.xls par=h rng=a1:c16 Cdim=1 rdim=1
PARAMETERS

      h(i,k)  unit size in sq. meters of each residence zone
$GDXIN h.gdx
$LOAD  h
$GDXIN

$CALL GDXXRW.EXE kc.xls par=kc rng=a1:c16 Cdim=1 rdim=1
PARAMETERS

      kc(i,k)  constants for each ik combo
$GDXIN kc.gdx
$LOAD  kc
$GDXIN


$CALL GDXXRW.EXE t.xls par=t rng=a1:p16 Cdim=1 rdim=1

PARAMETERS

      t(i,j)  travel time from zone i to zone j

$GDXIN t.gdx

$LOAD  t

$GDXIN


$CALL GDXXRW.EXE e.xls par=e rng=a1:ae17 rdim=1 cdim=2

PARAMETERS

      e(i,j,k)  constant terms for each ijk combo. e111=0
```

```
$GDXIN e.gdx
$LOAD  e
$GDXIN

scalar alpha /0.6/;
scalar beta  /-0.3/;
scalar phi   /0.895343/;
scalar lamda /1/;
scalar n     /1000/;

VARIABLES

      r(i,k)  rent per sq.meter of each ik. To be solved.;
r.lo(i,k) = 0.0001;
*r.up(i,k) = 1000
r.l(i,k)  = 10;
*Initial value has been given
variable
    z    this is a dummy vairable to be maximized or minimized so that nlp can be used.;
    z.lo = 0.01 ;
    z.up = 1000 ;

equations
equilibrium(i,k)       all i*k submarkets are in equilibrium.;
equilibrium(i,k)..   n*(1-alpha)*sum(j,(m(j)*(system.exp (lamda*
           (system.log(m(j))-(1-alpha)*system.log(r(i,k))+beta*system.log(t(i,j))+e(i,j,k))))
           /
(sum((ii,jj,kk), system.exp(lamda* (system.log(m(jj))-
```

```
          (1-alpha)*system.log(r(ii,kk))+beta*system.log(t(ii,jj))+e(ii,jj,kk)))))))
            =e=
            s(i,k)*r(i,k)*system.exp(phi*r(i,k)+kc(i,k))/(1+system.exp(phi*r(i,k)+kc(i,k)));
equation
zz   z is a dummy vairable formulated so that nlp can be used;
zz.. z    =n=
    sum((i,k),r(i,k))  ;


OPTION RESLIM =66666;
 option mcp=miles;

model tran  /all/;
tran.optfile = 1;
SOLVE TRAN USING nlp maximizing z;
```

2. GAMS with direct user specified Jacobians

```
SETS

     I   zone of residence   / 1*15 /
     J   zone of employment  / 1*15 /
     K   type of housing     / 1*2 / ;
alias(i,ii,iii,iiii,iiiii,iiiiii)
alias(j,jj,jjj,jjjj,jjjjj,jjjjjj)
alias(k,kk,kkk,kkkk,kkkkk,kkkkkk);

$CALL GDXXRW.EXE m.xls par=m rng=a1:b15 rdim=1
PARAMETERS
m(j)  income of each employment zone
$GDXIN m.gdx
$LOAD  m
$GDXIN

$CALL GDXXRW.EXE s.xls par=s rng=a1:c16 Cdim=1 rdim=1
PARAMETERS
     s(i,k)  total sq. meters of each residence zone
$GDXIN s.gdx
$LOAD  s
```

```
$GDXIN

$CALL GDXXRW.EXE h.xls par=h rng=a1:c16 Cdim=1 rdim=1
PARAMETERS
h(i,k)  unit size in sq. meters of each residence zone
$GDXIN h.gdx
$LOAD  h
$GDXIN

$CALL GDXXRW.EXE kc.xls par=kc rng=a1:c16 Cdim=1 rdim=1
PARAMETERS
kc(i,k)  constants for each ik combo
$GDXIN kc.gdx
$LOAD  kc
$GDXIN

$CALL GDXXRW.EXE t.xls par=t rng=a1:p16 Cdim=1 rdim=1
PARAMETERS
t(i,j)  travel time from zone i to zone j
$GDXIN t.gdx
$LOAD  t
$GDXIN

$CALL GDXXRW.EXE e.xls par=e rng=a1:ae17 rdim=1 cdim=2
PARAMETERS
     e(i,j,k)  constant terms for each ijk combo. e111=0
$GDXIN e.gdx
$LOAD  e
$GDXIN

scalar alpha /0.6/;
scalar beta  /-0.3/;
scalar phi   /0.895343/;
scalar lamda /1/;
scalar n     /1000/;

parameters
r(i,k) initial level of rents;
```

```
r(i,k) = 100;

variables

     delta(i,k) change of r(ik) after each iteration;
parameters
dfdr(i,k,ii,kk);

dfdr(i,k,ii,kk)=((n
*(1-alpha)*sum(j,m(j)*( ((system.exp(lamda*(log(m(j)) - (1-alpha)*log(r(i,k)) +
beta*log(t(i,j))+ e(i,j,k))))*lamda*(((-(1-alpha)/r(i, k))$(ord(i)=ord(ii) and
ord(k)=ord(kk))*sum((iii,jj,kkk), (system.exp(lamda*(log(m(jj)) - (1-
alpha)*log(r(iii,kkk)) + beta*log(t(iii,jj))+ e(iii,jj,kkk)))))



          -sum(jjj, (-(1-alpha)/r(ii, kk))*(system.exp(lamda*(log(m(jjj)) - (1-
alpha)*log(r(ii,kk)) + beta*log(t(ii,jjj))+ e(ii,jjj,kk))))) )))
          /
          ((sum((iiii,jjjj,kkkk),(system.exp(lamda*(log(m(jjjj)) - (1-
alpha)*log(r(iiii,kkkk)) + beta*log(t(iiii,jjjj))+ e(iiii,jjjj,kkkk))))
           ))**2))))
          -
          (s(i, k) *((system.exp(phi*r(i, k)+kc(i, k))*phi)/( (1+system.exp(phi*r(i,
k)+kc(i, k)))**2)) *r(i,k)+s(i, k)
          *(system.exp(phi*r(i, k)+kc(i, k))/(1+system.exp(phi*r(i, k)+kc(i,
k)))))$(ord(i)=ord(ii) and ord(k)=ord(kk)));

display dfdr;

parameters
f(i,k);
f(i,k) = -(n*(1-alpha)*sum(jjjjj,(m(jjjjj)*(
        (system.exp(lamda*(system.log(m(jjjjj))-(1-alpha)*system.log(r(i,k))+

beta*system.log(t(i,jjjjj))+e(i,jjjjj,k))))
          /
          (sum((iiiii,jjjjjj,kkkkk), system.exp(lamda*(system.log(m(jjjjjj))-
```

```
(1-
alpha)*system.log(r(iiiii,kkkkk))+beta*system.log(t(iiiii,jjjjjj))+e(iiiii,jjjjjj,kkkkk))))))))
          -
(s(i,k)*r(i,k)*system.exp(phi*r(i,k)+kc(i,k))/(1+system.exp(phi*r(i,k)+kc(i,k))));

display f;

equations
jacobian(i,k);
jacobian(i,k)..
        sum((ii,kk),(dfdr(i,k,ii,kk)*delta(ii,kk)))=e= f(i,k);
variables
z;
equations
zz;
zz..   z =n= sum((i,k),delta(i,k)) ;

model floorspace /all/;
parameter
value(i,k);
parameter error(i,k);
parameter reltol(i,k);
       reltol(i,k)=1.e-4 ;
parameter threshold(i,k);

repeat( solve floorspace using lp minimizing z;
     value(i,k)=r(i,k)+delta.l(i,k);
     r(i,k)=value(i,k);

     dfdr(i,k,ii,kk)=((n*(1-alpha)*sum(j,m(j)*(((system.exp(lamda*(log(m(j)) - (1-
alpha)*log(r(i,k)) + beta*log(t(i,j))+ e(i,j,k))))*lamda*(((-(1-alpha)/r(i,
k))$(ord(i)=ord(ii) and ord(k)=ord(kk))
*sum((iii,jj,kkk), (system.exp(lamda*(log(m(jj)) - (1-alpha)*log(r(iii,kkk)) +
beta*log(t(iii,jj))+ e(iii,jj,kkk)))))
-sum(jjj, (-(1-alpha)/r(ii, kk))*(system.exp(lamda*(log(m(jjj)) - (1-
alpha)*log(r(ii,kk)) + beta*log(t(ii,jjj))+ e(ii,jjj,kk))))) )))

          /
```

```
            ((sum((iiii,jjjj,kkkk),(system.exp(lamda*(log(m(jjjj)) - (1-
alpha)*log(r(iiii,kkkk)) + beta*log(t(iiii,jjjj))+ e(iiii,jjjj,kkkk))))))**2))))
            -
 (s(i, k)*((system.exp(phi*r(i, k)+kc(i, k))*phi)/( (1+system.exp(phi*r(i, k)+kc(i,
k)))**2))
*r(i,k)+s(i, k)*(system.exp(phi*r(i, k)+kc(i, k))/(1+system.exp(phi*r(i, k)+kc(i,
k)))))$(ord(i)=ord(ii) and ord(k)=ord(kk)));

f(i,k) = -(n*(1-alpha)
*sum(jjjjj,(m(jjjjj)*((system.exp(lamda*(system.log(m(jjjjj))-(1-
alpha)*system.log(r(i,k))+beta*system.log(t(i,jjjjj))+e(i,jjjjj,k))))

            /

            (sum((iiiii,jjjjjj,kkkkk),system.exp(lamda*(system.log(m(jjjjjj))-
(1-
alpha)*system.log(r(iiiii,kkkkk))+beta*system.log(t(iiiii,jjjjjj))+e(iiiii,jjjjjj,kkkkk))))))))
            -
(s(i,k)*r(i,k)*system.exp(phi*r(i,k)+kc(i,k))/(1+system.exp(phi*r(i,k)+kc(i,k)))));
      error(i,k)=abs(abs(delta.l(i,k))/(r(i,k)+0.5*delta.l(i,k))) ;
loop((i,k),threshold(i,k)=reltol(i,k)-error(i,k));
      until(smin((i,k),threshold(i,k))>0)
      ) ;
display r;
```

3. FORTRAN code that solves the same problem

```
 !Huibin Chang's FORTRAN code for the test
program main
use globle
implicit none
!reading data part
open (501, file='n.txt')
read (501, *) n
!open (502, file='h.txt')
!read (502, *) h
open (503, file='s.txt')
read (503, *) s
open (504, file='m.txt')
```

```
read (504, *) m
open (505, file='t.txt')
read (505, *) t
open (506, file='e.txt')
read (506, *) e
open (507, file='kc.txt')
read (507, *) kc
open (508, file='alpha.txt')
read (508, *) alpha
open (509, file='beta.txt')
read (509, *) beta
open (510, file='phi.txt')
read (510, *) phi
open (511, file='lamda.txt')
read (511, *) lamda
!-----------------------------------end of data reading
!_____This following sub block can be used to check: if the rent data that used to
generate other data is read
!&into the program, the procedure will converge in one iteration
!open (511, file='checkrent.txt')
!read (511, *) r
!rewind 511
!-------------------------------------------------------------------end of reading data part
101 print *, "New attempt"
!Generate initial guess for solution
call random_number(randmtplr_rent)
!forall (i=1:imax, k=1:kmax) r(i, k)=1*(randmtplr_rent(i, k))*2
forall (i=1:imax, k=1:kmax) r(i, k)=100
!read (511, *) r
!rewind 511
!forall(i=1:imax, k=1:kmax) r(i,k) = r(i,k)+0.1
tolerance = 1.0e-2
ftolerance = 1.0e-5
!End of Generation initial guess------------------------------
!Call the subroutine for Newton Raphson Procedure
call newtraphs(r)
!_________________This block check how close to zero is f(r(i,k))
forall (i=1:imax, k=1:kmax) q(i, k)=exp(phi*r(i, k)+kc(i, k))/(1+exp(phi*r(i, k)+kc(i, k)))
forall (i=1:imax, j=1:jmax, k=1:kmax)
u(i, j, k) = log(m(j)) - (1-alpha)*log(r(i,k)) + beta*log(t(i,j))+ e(i,j,k)
end forall
forall (i=1:imax, j=1:jmax, k=1:kmax)
expu(i, j, k)=exp(lamda*u(i, j, k))
end forall
sumu=sum(expu)
forall(i=1:imax, j=1:jmax, k=1:kmax) p(i, j, k)=(expu(i, j, k))/(sumu)
```

```
forall (i=1:imax, j=1:jmax, k=1:kmax) pm (i,j,k) = p(i,j,k)*m(j)
sumpmj = sum(pm, dim=2)
forall(i=1:imax, k=1:kmax) ff(i, k)=-n*(1-alpha)*sumpmj(i, k)+s(i, k)*q(i, k)*r(i,k)
!-----------------------------------------------------------end of checking
forall(i=1:imax, k=1:kmax) ferror(i,k)=abs(ff(i,k))/s(i,k)
if (maxval(ferror)<ftolerance) then
!if (maxval(abs(dd))<\tolerance) then
print *, 'Success'
open (998, file='Solution_Result.txt')
write (998, *) r
call CPU_TIME(elapsed_time)
call DATE_AND_TIME(time)
print *, 'iteration=', iteration
print *, 'Solved'
print *, 'Elapsed time=', elapsed_time
print *, 'max abs error=', maxval(abs(dd))
print *, 'max abs f(r(i,k))=', maxval(abs(ff))
print *, 'sum of abs error=', sum(abs(dd))
print *, 'sum of abs f(r)', sum(abs(ff))
open (999, file='Solution_log.txt')
write (999, 20) time, elapsed_time, iteration, maxval(abs(dd)), maxval(abs(ff)),
sum(abs(dd)), sum(abs(ff)), tolerance
20 format ( /, 'execution@', a10 /, 'elapsed time=', f12.1, /, 'Number of Newton-Raphson
iteration=', i3 &
& /, 'maximum absolute value of reduced gradient=', Es12.3E2/, 'maximum absolute
value of residual=', Es12.3E2 &
& /, 'sum of maximum absolute reduced gradient=', Es12.3E2 /, 'sum of maximum
absolute residuals=', Es12.3E2 &
& /, 'tolerance' Es12.1e2 /)
else
goto 101
endif
!-------------------------------------------------end of checking f(r(i,k))
end !end of main program
!_____________________________________________________________________________
_____________________
!_____________________________________________________________________________
_____________________
!_____________________________________________________________________________
_____________________
!Start of the NEWTON_RAPHSON Subroutine
subroutine newtraphs(*)
use globle
implicit none
!Start of the loop
do iteration=1, 100
```

```
!_________________This block calculate f(r(i,k))
forall (i=1:imax, k=1:kmax) q(i, k)=exp(phi*r(i, k)+kc(i, k))/(1+exp(phi*r(i, k)+kc(i, k)))
forall (i=1:imax, j=1:jmax, k=1:kmax)
u(i, j, k) = log(m(j)) - (1-alpha)*log(r(i,k)) + beta*log(t(i,j))+ e(i,j,k)
end forall
forall (i=1:imax, j=1:jmax, k=1:kmax)
expu(i, j, k)=exp(lamda*u(i, j, k))
end forall
sumu=sum(expu)
forall(i=1:imax, j=1:jmax, k=1:kmax) p(i, j, k)=(expu(i, j, k))/(sumu)
forall (i=1:imax, j=1:jmax, k=1:kmax) pm (i,j,k) = p(i,j,k)*m(j)
sumpmj = sum(pm, dim=2)
forall(i=1:imax, k=1:kmax) ff(i, k)=-n*(1-alpha)*sumpmj(i, k)+s(i, k)*q(i, k)*r(i,k)
!-----------------------------------------------end of calculation of f(r(i,k))
!_____________________________________________________________________The
following block calculate Jacobian
!dqdr. In fact, dqdr is a matrix with dimension (imax*kmax, imax*kmax),
!but since all the off-diagonal elements are zero), here I will treat it as a vector with
imax*kmax elements
!also, since the expression is too long, I will use substitutions to make it shorter
forall (i=1:imax, k=1:kmax) dqdr(i, k)=(exp(phi*r(i, k)+kc(i, k))*phi)/((1+exp(phi*r(i,
k)+kc(i, k)))**2)
!end of calculation for dqdr
forall (i=1:imax, k=1:kmax)
dudr(i,k)=-(1-alpha)/r(i, k)
end forall
!end of dudr
!dpdr
forall (i=1:imax, j=1:jmax, k=1:kmax) x(i, j, k)=dudr(i, k)*expu(i, j, k)
x1=sum(x, dim=2)
!Here calculation of dpdr uses do loop might be clearer for people to read. Two cases in
the do loop are the diagonal elements case of dpdr
!&and offdiagonal elements case
do k=1, kmax
do j=1, jmax
do i=1, imax
do kk=1, kmax
do ii=1, imax
if (i.eq.ii.and.k.eq.kk) then
dpdr(i, j, k, ii, kk)=(expu(i, j, k)*lamda*(dudr(i,k)*sumu-x1(i, k)))/(sumu**2)
else
dpdr(i, j, k, ii, kk)=(-expu(i, j, k)*lamda*x1(ii, kk))/(sumu**2)
endif
enddo
enddo
enddo
```

```
enddo
enddo
!end of calculation of dpdr
!mdpdr
forall (i=1:imax, j=1:jmax, k=1:kmax, ii=1:imax, kk=1:kmax) mdpdr(i, j, k, ii, kk) =
m(j)*dpdr(i,j,k,ii,kk)
!Calculate RHS of NewtonRaphson System, which is -f
forall (i=1:imax, j=1:jmax, k=1:kmax) pm (i,j,k) = p(i,j,k)*m(j)
sumpmj = sum(pm, dim=2)
forall(i=1:imax, k=1:kmax) ff(i, k)=-n*(1-alpha)*sumpmj(i, k)+s(i, k)*q(i, k)*r(i,k)
!This block read ff as a 2 dimensional array then convert it to 1 dimension RHS of the
system to be solved, which is -f.
open (601, file='rhs.txt')
write (601, *) ff
rewind 601
open (602, file='rhs.txt')
read (602, *) f
rewind 602
!end of conversion
!This block get Jacobian
summdpdrj=sum(mdpdr, dim=2)
do k=1, kmax
do j=1, jmax
do i=1, imax
do kk=1, kmax
do ii=1, imax
if (i.eq.ii.and.k.eq.kk) then
jacjac(i, k, ii, kk)=n*(1-alpha)*summdpdrj(i, k, ii, kk)-s(i, k)*dqdr(i, k)*r(i,k)-s(i, k)*q(i,
k)
else
jacjac(i, k, ii, kk)=n*(1-alpha)*summdpdrj(i, k, ii, kk)
endif
enddo
enddo
enddo
enddo
enddo
!end of assignment for Jacobian
!This block convert the five dimensional 'jacjac' to two dimensional Jacobian, which is
the LHS of the system to be solved.
open (603, file='jacobian.txt')
write (603, *) jacjac
rewind 603
open (604, file='jacobian.txt')
read (604, *) jacobian
rewind 604
```

```
!end of conversion of Jacobian
!----------------------------------------------------------------------------------------END OF
CALCULATION OF JACOBIAN
call mainls(jacobian, f, imax*kmax, imax*kmax) !solving the linear system, and f is
returned as the solution.
!this block convert the solution 'f', which is the correction of initial guess in the NR
procedure, to
!& a two dimensional (imax, kmax) matrix so that the update of solution can be handled.
open (701, file='dr.txt')
write (701, *) f
rewind 701
open (702, file='dr.txt')
read (702, *) dd
rewind 702
!end of conversion
!update initial guess
forall (i=1:imax, k=1:kmax) r(i, k)=r(i, k)+dd(i, k)
print *, 'sum of abs error=', sum(abs(dd))
forall(i=1:imax, k=1:kmax) error(i,k)=abs(abs(dd(i,k))/(r(i,k)+0.5*dd(i,k)))
if (maxval(error)<tolerance) return
enddo !end of one iteration
end !end of subroutine NewtonRaphson Procedure
```