A User's Guide to Solving Dynamic Stochastic Games Using the Homotopy Method – Code Description and Instructions –

Ron N. Borkovsky* Ulrich Doraszelski† Yaroslav Kryukov‡

June 7, 2010

*Rotman School of Management, University of Toronto, Toronto, ON M5S 3E6, ron.borkovsky@rotman.utoronto.ca.

†Department of Economics, Harvard University, Cambridge, MA 02138, doraszelski@harvard.edu.

‡Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213, kryukov@cmu.edu.

Contents

1 Overview

This note accompanies the code for the two examples of dynamic stochastic games that are solved using the homotopy method in Borkovsky, Doraszelski & Kryukov (2010). This note also provides a step-by-step guide to coding up a new model. The code has several parts:

1. The code common to both models (in the *Src* directory), which is written entirely in Fortran90, includes:

   (a) HOMPACK90, developed by Watson, Sosonkina, Melville, Morgan & Walker (1997), which in turn includes the BLAS and LAPACK libraries; and

   (b) several small subroutines that we have written, which provide input and record output using binary files and compute Jacobians.

2. The code specific to the models (in the *LBD* and *QLD* directories), which includes:

   (a) Matlab code that solves for a starting point and generates a binary input file;

   (b) Model-specific Fortran90 files that read the starting point binary input file, supply HOMPACK90 with the system of equations and its Jacobian, and run HOMPACK90. We use ADIFOR2.1 (developed by Bischof, Khademi, Mauer & Carle (1996)) to compute the Jacobian.

   (c) Matlab code that reads the binary output files generated by HOMPACK90 and uses the output to compute some summary statistics for equilibria.

In order to run the code, you must have:

1. Matlab 6.0 or higher version. Matlab is a commercial product developed and distributed by Mathworks. We have used versions 6.5 and 7+.

2. Fortran90 compiler. There are several commercial compilers available on the market, and there is a free version developed under the GNU project. We have successfully compiled our code with the Compaq Visual Fortran 6.6.a compiler under Windows, and the Portland Group PGF95 compiler on a Linux system.

The choice of Matlab as the "front end" is entirely due to our familiarity with it prior to using HOMPACK, which in turn stems from its ease of use and graphical capabilities. Our Fortran90 code can be used with any application that can read and write binary files.

Similarly, it is not necessary to use ADIFOR to analytically differentiate the code. We supply the code to compute the Jacobian numerically, which offers comparable performance when using the Normal flow and Augmented Jacobian algorithms.

Following a solution path of a reasonably-sized model can take a substantial amount of time, up to several hours depending on the model, the algorithm used, and computer performance. We have found it most efficient to use dedicated computational servers, typically

	dense Jacobian	sparse Jacobian
ODE based	FIXPDF	FIXPDS
normal flow	FIXPNF	FIXPNS
augmented Jacobian	FIXPQF	FIXPQS

Table 1: Path-following algorithms and dense vs. sparse Jacobian in HOMPACK90.

running Linux. At the same time, code development is much easier on a desktop, typically a Windows machine. As such, our code is designed to work on both platforms and minimal changes are required to move from one platform to the other.

The Fortran files described below might require minor editing depending on which compiler and operating system one uses. The code contains preprocessor directives (`#ifdef`, `#else`, `#endif`, etc.) that attempt to instruct the compiler to use the correct syntax automatically, but they do not always work. If this occurs, it is safe to comment out the preprocessor directives and one of the two versions of the command that they enclose. The possible code changes that we have identified are:

1. Different compilers use different syntax for opening binary files. This affects `open` statements in `*_Main.f` and `HOMPACK90/hom_fileIO.f`. To instruct the preprocessor directive to use the 'gfortran' syntax, the user must define the "`GFORTRAN`" symbol in the compiler environment or use the `-dGFORTRAN` switch in the command line.

2. Unix uses forward slash ("/") as the directory separator, Windows uses backslash ("\"). This affects the statement "`write(filenm ... HomXpt ...`" in the file `Src/hom_fileIO.f`, which is described in Section 2. Preprocessor directives assume that a Windows compiler will have the "`_WIN32`" symbol defined in its environment, or have the `-WIN32` switch in the command line.

3. At least one compiler (Absoft) does not allow a comma in the `write` statement after parentheses that enclose the output unit and format string. This affects multiple files.

The rest of this note proceeds as follows. Section 2 describes code common to all models. Sections 3 and 4 present code specific to the learning-by-doing and organizational forgetting (LBD) and quality ladder duopoly (QLD) models, respectively. Finally, Section 5 provides a step-by-step guide to applying HOMPACK90 to a new problem.

2 Model-independent code

The code common to all models is stored in the *Src* directory and consists of the Fortran90 (*.f) files listed below. We expanded HOMPACK90 by writing several subroutines (`hom_fileIO.f`, `homjac*.f`, and `homjs*.f`) that provide input and output via binary files as well as Jacobian computation.

1. `lapack.f`, `blas*.f` – The LAPACK and BLAS packages handle linear algebra and sparse matrices. The user will never need to change any of these.

2. `hompack90.f` – the HOMPACK90 package. Specifically, it includes the entry-point subroutines for all six of the path-following algorithms (see Table 1), as well as all the subroutines and functions that they call. The only change that a user may need to make is to the `LIMITD` parameter that is present in each of the entry-point subroutines; it is the upper bound on the number of steps that the homotopy algorithm takes.

3. `hom_fileIO.f` – Converts output into binary files, written to a pre-existing *HomXpt* subdirectory.

4. `homjacA.f`, `homjacN.f` – These subroutines compute the dense Jacobian used in the `FIXP*F` algorithms. `homjacN.f` computes the Jacobian numerically using a two-sided finite difference scheme. `homjacA.F` computes the Jacobian analytically by calling the model-specific ADIFOR-generated file. Both files actually compute a specified column of the Jacobian; HOMPACK90 assembles the Jacobian itself.

5. `homjsA.f`, `homjsN.f` – These subroutines compute the sparse Jacobian used in the `FIXP*S` algorithms. `homjsN.f` computes the numeric Jacobian using a two-sided finite difference scheme. `homjsA.F` computes the analytic Jacobian by calling the model-specific ADIFOR-generated file.

3 Learning-by-doing code

The learning-by-doing model was our first application of HOMPACK90, so the code is not always as polished or well-commented as the code for the quality ladder model. Therefore, a user interested in applying the homotopy to his/her own problem would be better off using the quality ladder code as an example.

On the plus side, the learning-by-doing code contains the features for storing and keeping track of multiple runs of the code (for different parameterizations, that is); HOMPACK90 output for each run is converted into Matlab files, which are saved into an automatically-named subdirectory. Expected Herfindahl indices for each run are saved in a separate file for easy access and plotting.

The code is contained in the *LBD* directory and includes Matlab (*.m), Fortran90 (*.f), and C (*.c) files.

As explained in Section 1, the code can be divided into three parts, each described in a separate subsection below. In addition, we open with instructions on compiling and running the code.

3.1 Compiling and running

Here are the instructions for compiling and running the code.

1. **Compile C code** used in Matlab code: open Matlab and run `mex_PM.m`. Matlab typically has its own C compiler; use this compiler or use the `mex -setup` command to instruct Matlab to find and use compilers available on the system.

2. **Compile Fortran files**:

   (a) Compile the following Fortran files:
   `LBD/NE_Main.f`
   `LBD/g_NE_rho1.f`
   `LBD/NE_rho.f`
   `LBD/NE_QRT.f`
   `Src/hompack90.f`
   `Src/rhojsA.f`
   `Src/lapack.f`
   `Src/hom_fileIO.f`
   `Src/blas*.f`

   (b) Make sure the compiled executable (typically an `.exe` file on a Windows system, and an `a.out` file on a Unix/Linux system) is in the *LBD* directory.

   (c) Open `homNE90_Start.m`. Uncomment line 117 if running on a Windows system and line 119 if running on Unix-based system. On this line, make sure that the correct name of the compiled executable appears after the "!" character.

3. **Compute starting point**. Edit the parameter values in `masterNE.m` and run it to compute and save an equilibrium that will serve as a starting point for the homotopy algorithm. There is already an equilibrium saved for the baseline parameterization with $\delta = 0$ and $\rho = 0.85$. `masterNE.m` allows one to compute several equilibria that could later be used as starting points for different runs of the homotopy algorithm; variables `dlt_val` and `rho_val` allow the user to select one or several (δ, ρ) pairs. `masterNE.m` computes an equilibrium for each of these (δ, ρ) pairs; in particular, the code will go through them in order, using the equilibrium computed for each parameterization as the starting point for the computation of an equilibrium for the next parameterization.

4. **Run the homotopy**. Edit the starting and ending points in `homNE90_Start.m` and run it. The homotopy code will run (it can take from minutes to hours, depending on computer speed and the range of parameter values covered). It will then convert step files from binary format to Matlab's .mat format (`homNE90_read.m`). It will then compute and plot expected Herfindahl indices for each step (`hom_steps.m`).

   Finally, it will generate two files and save them in the *HomRes* directory:

   (a) `DeltaRho_*.mat` contains δ and ρ for each step in the run.

(b) `Herf_*.mat` contains several expected Herfindahl indices (limiting, maximum over $T = \{1, 2, ..., 100\}$ etc.) for each step.

3.2 Code for computing the starting point

We provide Matlab code for computing a starting point for the homotopy. While several parts of the code are written to accommodate models with entry and exit, we focus here on models without entry or exit.

1. *GridDat* – This is the directory for saved equilibrium files. It already includes one equilibrium for $\delta = 0$, $\rho = 0.85$.

2. `mex_PM.m` – Compiles all C files. Matlab typically has its own C compiler; use this compiler or use the `mex -setup` command to instruct Matlab to find and use compilers available on the system.

3. `masterNE.m` – Main control script that computes and saves equilibria for one or several sets of parameter values. Before starting computation, the script attempts to load a saved equilibrium to use as a starting point; if there is no saved equilibrium, it uses the equilibrium already in memory. Specifically, this means that if the code is used to compute equilibria for several parameterizations, then the equilibrium for each parameterization is retained in memory and used as the starting point for computation of the equilibrium for the next parameterization. If no saved file or previous equilibrium is available, computation starts from the default policy and value functions that were placed into memory by `InitParamsEE.m`.

   The file also includes functionality to save multiple equilibria for the same parameter values (the string variable `FileMod` becomes part of the filename), compute transient distributions and Herfindahl indices, and produce summary plots of equilibria.

4. `InitParamsEE.m` – Sets up and initializes the global variables used in computation.

5. `duopolyNE.m` – computes an equilibrium using the Pakes & McGuire (1994) algorithm. Note that important parts of the computation are done by `WLEc` and `SolveFOCc`, which are described below.

6. `plotEE_Iter.m` – Plots iteration progress.

7. `cost.m` – computes cost as a function of experience level.

8. `FOC.m` – FOC for the best response problem, as Matlab code. This can be used with Matlab's `fzero` command if `SolveFOCc` reports errors (uncomment line 43 in `duopolyNE.m`).

9. `FOCc.c` – FOC for the best response problem, as a C function (run `mex_PM.m` to compile). This can be used with Matlab's `fzero` command if `SolveFOCc` reports errors (uncomment line 41 in `duopolyNE.m`).

10. `SolveFOCc.c` – Solution to the FOC, as a C function (run `mex_PM.m` to compile). This code implements our own zero-finding algorithm, which combines Interpolation and Newton methods.

11. `Fsale.m` – Probability of sale going to firm 1.

12. `Fsale0.m` – Probability of sale going to the outside good.

13. `WL.m` – Conditional expectations, with no exit/entry, as Matlab code. This file can be used in place of `WLEc` if compilation fails, but it is much slower.

14. `WLEc.c` – Conditional expectations, as a C function (run mex_PM.m to compile). Recall from the paper that the conditional expectation $\overline{V}_{n1}(\mathbf{e})$ refers to the expected value of firm 1 if firm n wins the sale. For legacy reasons, the code represents $\overline{V}_{11}(\mathbf{e})$ as the `W` array (firm 1 *W*ins), $\overline{V}_{21}(\mathbf{e})$ as `L` (firm 1 *L*oses to its competitor), and $\overline{V}_{01}(\mathbf{e})$ as `Wa` (a *W*in by the outside *a*lternative). Furthermore, this code is general in that it works for the version of the model that allows exit and entry, and thus requires computation of conditional expectations for a monopolist ($e_2 = M + 1$); for this model, which does not include entry and exit, we simply set parameters to values at which exit never occurs.

3.3 Homotopy code

This set of programs initializes the homotopy path-following package HOMPACK90 and describes the model to it.

1. *HomXpt* – Directory that receives the step files generated by the homotopy.

2. `homNE90_Start.m` – Runs the homotopy:

   (a) loads a saved equilibrium that serves a starting point for the homotopy algorithm;

   (b) writes the equilibrium and parameter values into a binary file (`hom_start.dat`);

   (c) calls the executable compiled from Fortran90 files;

   (d) generates the "name" of the run (`fileMod` variable);

   (e) initiates output processing by running `homNE90_read.m` and `hom_steps.m`.

3. `NE_Main.f` – Starting point of the homotopy:

   (a) reads the binary file created by `homNE90_Start.m`;

(b) sets precision for the path-following algorithm (`ARCRE` and `ARCAE` variables) – we do not recommend making precision more strict than 10^{-12};

(c) calls the path-following algorithm.

4. `NE_rho.f` – System of equations.

5. `NE_SparseStru.f` – Defines sparsity structure of the Jacobian.

6. `g_NE_rho1.f` – ADIFOR-generated code that computes one column of the Jacobian.

ADIFOR-related files. These files serve as inputs to ADIFOR:

1. `rho1.f` – A version of `NE_rho.f` modified to fit the requirements of ADIFOR2.0:

   (a) stricter Fortran77 syntax;

   (b) single input variable: `NE_rho.f` has inputs $x \in R^N$ and $\lambda \in R$, while `rho1.f` combines them into $x \in R^{N+1}$.

2. `rho1.adf` and `rho1.cmp` provide additional input parameters to ADIFOR.

3. `ad.bat` is the Windows batch file containing the command line to the file that calls ADIFOR. Running it creates the file `output_files/g_QLD_rho1.f`. Before it can be used in compilation, it has to be modified as described in Section 5 below.

3.4 Output processing and summary statistics

HOMPACK90 output is converted into Matlab `.mat` files, which are saved into an automatically-named subdirectory; Herfindahl indices for every 10th step of the run are saved in a separate file for easy access and plotting.

1. *HomDat* – Directory that stores step files in Matlab `.mat` format, each run gets its own subdirectory.

2. *HomRes* – Directory that stores short files with homotopy results (δ and ρ values, Herfindahl indices).

3. `homNE90_read.m` – Reads the step files generated by the homotopy from the *HomXpt* directory and saves them as Matlab `.mat` files in a subdirectory of *HomDat*. Also writes out list of δ and ρ values to a file in HomRes directory. The name of the run as generated by `homNE90_Start.m` is used for both these files and the subdirectory of *HomDat*.

4. `hom_steps.m` – Computes Herfindahl indices for a given run, plots them and saves them in a file in the *HomXpt* directory; this file again is named after the run. Can also plot and test step files.

5. `EE_symmetry.m` – Symmetry measures (Herfindahl indices).
6. `EE_welfExp.m` – Welfare measures.
7. `markEE_Lim.m` – Limiting (ergodic) distribution, linear algebra computation.
8. `partition.m` – Part of limiting distribution calculation.
9. `components.m` – Part of limiting distribution calculation.
10. `markEE_LimDirect.m` – Limiting (ergodic) distribution, direct computation ($T = 1024$).
11. `markEE_Trans.m` – Transient distributions.
12. `transE.m` – Experience transition probabilities, entry and exit.
13. `transLC.m` – Experience transition probabilities, no exit or entry.
14. `TransMatEE_duo.m` – Markov kernel (matrix of state-to-state transition probabilities).
15. `meanmode.m` – Computes mean and mode of distribution over states.
16. `plotNE_ResTrans.m` – Plots summary of equilibrium.
17. `subtitle.m` – Fills in "title" area in the figure (top middle).
18. `ffooter.m` – Fills in "footer" area (bottom-left corner).
19. `fpage.m` – Fills in "page #" area (bottom-right corner).

4 Quality ladder code

While the structure of the code is similar to that of the learning-by-doing model, the code for the quality ladder model was developed at a later date; therefore, it incorporates several programming improvements, better comments, and the option to use a wider range of path-following algorithms. To keep the code simple, and because this smaller model computes faster, we do not include the infrastructure for storing solution paths; recall that this infrastructure is included in the code for the learning-by-doing model.

The code is contained in the *QLD* directory and includes Matlab (*.m) and Fortran90 (*.f) files.

As explained in Section 1, the code can be divided into three parts, each described in its own subsection below. In addition, we open with instructions on compiling and running the code.

4.1 Compiling and running

The instructions for compiling and running the code are as follows.

1. **Compile Fortran files**:

   (a) Compile the following Fortran files:
   `QLD/QLD_Main.f`
   `QLD/QLD_rho.f`
   `QLD/g_QLD_rho1.f`
   `QLD/QLD_SparseStru.f`
   `Src/hompack90.f`
   `Src/rhojsA.f`
   `Src/lapack.f`
   `Src/hom_fileIO.f`
   `Src/blas*.f`
   To use a numeric Jacobian instead of an analytic Jacobian, replace `Src/rhojsA.f` with `Src/rhojsN.f`, and remove `QLD/g_QLD_rho1.f` from the above list. To use dense Jacobian storage format instead of sparse Jacobian storage format, replace `QLD/QLD_Main.f` with `QLD/QLD_MainFull.f`. To switch algorithms, comment/uncomment the relevant lines in `QLD_Main.f` (see Table 1).

   (b) Make sure the compiled executable (typically an `.exe` file on a Windows system, and a `.out` file on a Unix/Linux system) is in the *QLD* directory.

   (c) Open `QLD_StartHom.m` and uncomment line 109 if running on a Windows system or line 112 if running on a Unix-based system. In either line, make sure that the correct name of the compiled executable appears after the "!" character.

2. **Compute starting point**. Edit the parameter values in `QLDmaster.m` and run it to compute and save an equilibrium. There is already an equilibrium saved for the current parameter values.

3. **Run the homotopy**. Edit the starting and ending points in `QLD_StartHom.m` and run it. Watch the homotopy run, read the output, and compute and plot expected Herfindahl indices (`QLD_HomRead.m`); see step 4 in Section 3.1 for more detail.

4.2 Code for computing the starting point

We provide Matlab code for computing a starting point for the homotopy.

1. *equilibria* – Directory for saved equilibrium files.

2. *PeriodProfit* – The directory for saved equilibria of the product market game – i.e., equilibrium prices and profits. These product market equilibria are saved separately

from the Markov perfect equilibria in the directory *equilibria* so as to avoid repeated computation, since the parameters that we allow the homotopy algorithm to vary do not affect the product market equilibrium and thus the equilibrium profit function.

3. `QLDmaster.m` – The main control script that computes and saves equilibria for specified parameter values. Before starting computation, the script attempts to load a saved equilibrium to use as a starting point. The script has the functionality to compute transient distributions and Herfindahl indices, and to produce summary plots of equilibria.

4. `Qldsetup.m` – Computes the period profit function.

5. `myfun.m`, `myprof.m` – Used in computation of the period profit function.

6. `Qldvfi.m` – Computes an equilibrium using the Pakes & McGuire (1994) algorithm.

7. `QLDfigures.m` - Plots figures for an equilibrium (can be run directly or called from `QLDmaster.m`).

8. `subtitle.m` – Service function used in plotting.

9. `QLDherf.m` – Computes the Herfindahl index for a single equilibrium (can be called from `QLDmaster.m`).

4.3 Homotopy code

This set of programs initializes the homotopy path-following package HOMPACK90 and describes the model to it.

1. *HomXpt* - Directory that receives the step files generated by the homotopy algorithm.

2. `QLD_StartHom.m` – Runs the homotopy:

   (a) loads a saved equilibrium that will serve as the starting point for the homotopy algorithm;

   (b) writes the equilibrium and parameter values into a binary file (`hom_start.dat`);

   (c) calls the executable compiled from Fortran90 files;

   (d) initiates output processing and Herfindahl calculation by running `QLD_ReadHom.m`.

3. `QLD_Main.f` – Starting point of the homotopy:

   (a) reads the binary file created by `homNE90_Start.m`

   (b) sets precision for the path-following algorithm (`ARCRE` and `ARCAE` variables)– we do not recommend making precision more strict than 10^{-12};

(c) calls a sparse-Jacobian path-following algorithm (to choose an algorithm, uncomment the corresponding `INCLUDE` and `CALL` statements in lines 168-177);

4. `QLD_MainFull.f` – a version of `QLD_Main.f` for dense ("full") Jacobian algorithms. Again, algorithm-specific `INCLUDE` statements and `CALL` statements are on lines 39-41 and 142-151, respectively.

5. `QLD_rho.f` – System of equations.

6. `QLD_hom1.m` - Matlab "prototype" of `QLD_rho.f`.

7. `QLD_SparseStru.f` – Defines sparsity structure of the Jacobian.

8. `g_QLD_rho1.f` – ADIFOR-generated code that computes one column of the Jacobian.

ADIFOR-related files. These files serve as inputs to ADIFOR:

1. `rho1.f` – A version of `NE_rho.f` modified to fit the requirements of ADIFOR2.0:

   (a) stricter Fortran77 syntax;

   (b) single input variable: `NE_rho.f` has inputs $x \in R^N$ and $\lambda \in R$, while `rho1.f` stacks them into $x \in R^{N+1}$.

2. `rho1.adf` and `rho1.cmp` provide additional input parameters to ADIFOR.

3. `ad.bat` is the Windows batch file containing the command line to the file that calls ADIFOR. Upon running, it will create file `output_files/g_QLD_rho1.f`. Before it can be used in compilation, it has to be modified as described in Section 5 below.

4.4 Output processing and summary statistics

HOMPACK90 output is converted into Matlab `.mat` files. The code then computes and plots various summary statistics.

1. `QLD_ReadHom.m` – Reads the step files generated by the homotopy algorithm from the *HomXpt* directory, computes the expected Herfindahl indices and other summary statistics, and plots them.

2. `QLD_TransHerfs.m` – Computes transient distributions and expected Herfindahl indices.

3. `QLD_transMat.m` – Computes transition matrix.

5 Coding up a new model

This section suggests a sequence of steps that one could take in applying HOMPACK90 to an arbitrary model. It uses the quality ladder code described above as a starting point and builds up the code in small testable steps. These steps are aimed at Fortran90 beginners (which we were when we started); experienced Fortran users might find the level of detail to be excessive.

5.1 Preliminaries

The process described below assumes that you:

1. Already have Matlab code that computes a solution for each starting point (i.e, starting parameterization) that will be chosen for the homotopy algorithm. One can obtain these solutions either analytically or numerically; for the latter, one could use a Gaussian algorith – such as the Pakes & McGuire (1994) algorithm – or an equations solver.

2. Know the basics of the Matlab scripting language and Fortran90.

3. Have access to Matlab and Fortran90 compilers. While everything can be done on a server, a desktop environment might be more convenient.

4. Have successfully compiled and run the code for the quality ladder model.

5. Have made a copy of the quality ladder code, as it will be the starting point of your own code. Feel free to rename files as long as you do not change names in the `SUBROUTINE` statements within the files.

As mentioned above, use of Matlab as the front end is not necessary; one could use any application that can read and write binary files as described below.

5.2 Write out the binary file in Matlab (`StartHom.m`)

In the quality ladder code, this is done by `QLD_StartHom.m`. The binary file should be named `hom_start.dat` and should contain:

1. Parameter values, including starting and ending values. Since Matlab does not have explicit support for integer variables, we write them out as reals, and convert them to integer variables in Fortran.

2. The starting vector x - comprising stacked policies and values (i.e., the policy and value functions evaluated at each state). If you replace the policies with the Zangwill & Garcia (1981) variables, you will need to compute the corresponding policies; see `QLD_StartHom.m` , lines 53-82 .

3. The period profit function – if the model has a period profit function and if it is not a function of the parameters that will be varied by the homotopy algorithm. If changes in these parameters do change the equilibrium period profit function, then you will need to add the period game policies (e.g., prices) to the x vector, and the period game first-order conditions (or other optimality conditions) to the system of equations.

The Fortran code assumes that the input file stores a sequence of 8-byte double precision real numbers. Binary files operate like magnetic tape; Fortran will read the numbers in the same order as they were written out by Matlab.

5.3 Read the binary file in Fortran (`Main.f`)

Here you have to decide whether you want to use a dense or sparse Jacobian. Using a dense Jacobian requires a little less coding; however, a sparse Jacobian algorithm will likely run faster. We strongly recommend using a sparse Jacobian (`Main.f`), and we do so in the example below. If you want to use a dense Jacobian, use `MainFull.f`; that line numbers are different but statements are generally quite similar.

Edit your copy of `Main.f` as follows:

1. Comment out the calls to code components that are not there (yet):
   `CALL QR_STRU(N,LENQR)` – line 122
   `CALL FIXP...` – lines 168-177
   `USE HOMPACK90, ONLY : FIXP...` – lines 39-41
   `CALL hom_TestRun(H,A,X,N)` – line 128
   `CALL RHOJS(A,LAMBDA,X)`, and `PRINT` commands around it – lines 343-345

2. Uncomment `SUBROUTINE RHOJS` (lines 381-383) and `SUBROUTINE RHO` (lines 385-387). We do not have those functions (yet), so we need to replace them with these "dummies"İn `MainFull.f`, uncomment `RHOJAC`, which replaces `RHOJS` for the time being.

3. In subroutine `hom_Initialize1Run` (lines 203-291):

   (a) Replace declarations of model parameters and `COMMON` blocks (used as "global" variables);

   (b) Replace the code that reads in model parameters, which is below the declarations;

   (c) Compute the number of equations N based on these parameters.

4. If you have a vector containing the stacked version of the period profit matrix, replace `N/2` with the length of that vector in line 112:
   `ALLOCATE(Profit(N/2))`
   Do the same in line 118:
   `read(fid) (Profit(i),i=1,N/2)`

5. If your model does not have a period profit function, remove: (a) lines 112 and 118; (b) declaration of module `mProfit` (lines 23-26); (c) `USE mProfit` (line 37).

6. If you have more than one "profit function" or other vector inputs, add them to the declaration of module `mProfit`, allocate them, and read them in.

Then compile and run the following files:

`Main.f`
`Src/hompack90.f`
`Src/lapack.f`
`Src/hom_fileIO.f`
`Src/blas*.f`

Use print statements or debugging tools to make sure that all inputs have been read correctly.

5.4 System of equations (`rho.f`)

The inputs to the system of equations are the homotopy parameter λ and the vector x that contains values and policies for all states; moreover, if one uses the Zangwill & Garcia (1981) reformulation of the complementary slackness conditions, then the vector x will include the additional variables that this reformulation introduces. The output is a vector of residuals of the system of equations $H = \mathbf{H}(x)$. (Vector a is not used because it is a part of the HOMPACK90 *artificial-parameter* homotopy algorithms; we use the HOMPACK90 *natural-parameter* homotopy algorithms.)

If you are not familiar with the process through which you recast your problem as system of equation, it might be useful to make a Matlab prototype first. In any case, it is very helpful to have the system of equations written out before you start coding.

You can edit the existing function by replacing the code below the declaration of the subroutine and its input/output arguments and above the `END` statement. Alternatively, you can create your own function using function and argument declarations.

It is good practice to copy (rather than re-type) declarations of model parameters and `COMMON` statements from the `Main.f` and `hom_Initialize1Run` subroutines.

You will need to extract policies and values for specific states from the x vector, substitute them into an equation from the system of equations in order to compute the equation residual, and then insert this residual into the corresponding element of the H vector. Specifically, this means mapping each policy and value for each state into a scalar position ("an index") within x. In `QLD_rho.f,` we do this using index "offsets" (`offV` for value function, etc.), a pre-computed index (`ndx1`), and function calls (like `ndx(i,j,L)`). Similarly, each equation for each state is mapped to a position within H vector.

Pay very close attention to syntax; Fortran is much more rigorous than Matlab. For example, one must ensure all variables are declared, either as `real*8` or `integer`; some

compilers assume undeclared variables are `real*4`, which limits precision to 7-8 digits. Consider another example: we once found that a missing comma in a variable declaration did not cause a compiler error, but instead led to a crash at runtime.

Once the function is complete, it is time to test it. In `Main.f`, do the following.

1. Comment out `SUBROUTINE RHO` (3 lines near the end of file); the function you just wrote replaces it.
2. Uncomment the line that reads:
   `CALL hom_TestRun(H,A,X,N)`.
3. Compile the following files:
   `Main.f`
   `Src/hompack90.f`
   `Src/lapack.f`
   `Src/hom_fileIO.f`
   `Src/blas*.f`
   `rho.f`

The executable should return the maximum absolute residual. Needless to say, it should be reasonably small for the starting point; specifically, it should be below the path following precision set in the `ARCRE` and `ARCAE` variables in `Main.f`. If it is not, make sure the starting value is computed to similar or higher precision. If the starting value is computed to similar of higher precision, note the number of the equation that results in the largest residual, and then use debug features or print statements to explore the computation of this element and search for the problem.

5.5 Sparsity structure (`SparseStru.f`)

If you are using a dense Jacobian, skip this section. If you are using a sparse Jacobian, you need to specify the location (row and column) of all *potentially* non-zero elements of the Jacobian. In other words, for any row j, you need to specify the set of columns $I \subset \{1, ..., N+1\}$ such that each variable in the set $\{x_i\}_{i \in I}$ enters the expression for $H_j(x)$. This structure must remain unchanged throughout the entire run, so one cannot quite "read" the sparsity structure from the Jacobian as one computes it, and instead has to specify it in advance.

That might sound like a challenge, but in fact you have already done this when you computed $H_j(x)$ in `rho.f`. Now, you simply need to go though the `rho.f` code and identify the components of `X` used to compute each component of `H`.

The sparse Jacobian uses the so-called "sparse row" storage format, which is described in detail on p. 528 of Watson et al. (1997). It requires three vectors:

1. QRSPARSE – This vector comprises the potentially nonzero elements of the Jacobian, ordered by row; the elements within a row need not be in column order, however, we feel that is more straightforward to arrange them in column order. The LENQR variable is set equal to the number of potential nonzeros; therefore, the length of the QRSPARSE vector is LENQR.

2. COLPOS – This vector comprises column indices of potentially non-zero elements, in the same order as in QRSPARSE. The length of this vector is LENQR, as there is one such index for each potentially non-zero element of the Jacobian.

3. ROWPOS – ROWPOS(j) provides the location within QRSPARSE (COLPOS) of the beginning of the elements (indices) for the j^{th} row of the Jacobian. ROWPOS is an $(N+1)$-dimensional vector; element $(N+1)$ is set equal to LENQR+1.

COLPOS and ROWPOS represent the "sparsity structure" of the Jacobian and remain unchanged as HOMPACK90 runs. Unlike them, QRSPARSE changes every time the Jacobian is evaluated (by the rhojs subroutine).

One approach is to set COLPOS and ROWPOS in Fortran using SparseStru.f. Detailed instructions are included within the file; the general idea is to have a loop that goes through every row (equation) in order, and within each equation, passes column indices to a function (RECNZ) that "records" them in the sparsity structure.

An alternative approach is to create COLPOS and ROWPOS in Matlab, write them into a binary file, and then read them in Fortran.

Once the sparsity structure is defined, one can test it by modifying Main.f:

1. Comment out the SUBROUTINE RHOJS placeholder (3 lines at the end of the file).

2. Uncomment CALL RHOJS(A,LAMBDA,X) and the PRINT commands around it.

3. Uncomment CALL QR_STRU(N,LENQR).

4. Compile the following files:
   Main.f
   Src/hompack90.f
   Src/lapack.f
   Src/hom_fileIO.f
   Src/blas*.f
   rho.f
   SparseStru.f
   Src/RhojsN.f (the numeric computation of a sparse Jacobian)

5. Once you run, you will either get a "RHOJS done" success message or an "unexpected nonzero" message from Rhojs indicating that it came across a nonzero that was not specified in the sparsity structure.

5.6 HOMPACK90 with numeric Jacobian

Below we provide instructions for running HOMPACK90 with a numeric Jacobian.

1. Uncomment the following lines in `Main.f`:
   (a) `CALL FIXP...`
   (b) `USE HOMPACK90, ONLY : FIXP...`
2. Set `ARCRE` and `ARCAE` to `1.0D-7` in `Main.f`; you cannot achieve better precision with a numeric Jacobian.
3. Compile the following files:
   `Main.f` (is using a dense Jacobian, replace with `MainFull.f`)
   `rho.f`
   `SparseStru.f` (not used with dense Jacobian)
   `Src/hompack90.f`
   `Src/rhojsN.f` (if using a dense Jacobian, replace with `rhojacN.f`)
   `Src/lapack.f`
   `Src/hom_fileIO.f`
   `Src/blas*.f`
4. Create the *HomXpt* subdirectory (if it does not already exist).
5. Run the compiled executable. You should see the homotopy algorithm going through the steps, and the step files should be saved in *HomXpt*.

5.7 Analytic Jacobian with ADIFOR

This step is optional for the Normal flow and Augmented Jacobian algorithms. However, the ODE-based algorithm is very sensitive to Jacobian precision and therefore should be run with an analytic Jacobian. Moreover, with any algorithm, one cannot set precision parameters (`ARCRE` and `ARCAE` variables) below 10^{-7} when using a numeric Jacobian because limited precision of numeric Jacobian computation limits the precision of path-following.

In order to obtain the code for the analytic Jacobian, we create a copy of `rho.f`, adapt it as per the ADIFOR requirements, run ADIFOR on it, and edit the result.

1. Obtain ADIFOR from http://www-unix.mcs.anl.gov/autodiff/ADIFOR
2. Make a copy of the `rho.f` file; we usually call it `rho1.f`.
3. Edit the file so that it has a single input vector and Fortran77 syntax:
   (a) Copy subroutine and argument declarations from `QLD_rho1.f`.

(b) Use `X(N+1)` instead of `LAMBDA`.

(c) Replace "`USE mProfit`" command with "`real*8 profit(10000)`".

(d) Replace all "`DOUBLE PRECISION`" declarations with `real*8`.

(e) Replace all array size declarations with constants or parameters (see LBD code).

4. Make sure you have `ad.bat`, `rho1.adf` and `rho1.cmp` in your directory. Open `rho1.cmp` with Notepad (or any text editor) and insert the name of your `rho1.f` file; this file should include only the name of this file.

5. Run `ad.bat` or execute the command for it on a Unix machine (see the ADIFOR manual for more detail).

   (a) If you get an error message regarding a particular line and this message contains only a "smiley face" character, delete or comment out all empty lines or make sure that there are no spaces in them.

   (b) If you get an "Incompatible argument types" error message, make sure integer constants are written as "$2.0D0$" when used in the same expressions as reals.

6. If successful, ADIFOR will generate `Output_Files/g_*_rho1.f` file (g for augmented). Copy it to the main directory.

7. Open `g_*_rho1.f` file, and undo all "Replace" changes made in step 3 (i.e., steps (c), (d) and (e)). Keep `X(N+1)` and the declarations of the subroutine and the arguments.

8. Search for and comment out lines beginning with "`call eh`". These are calls to ADIFOR's "exception handler" for handling potential kinks; we eliminate these calls because we design our model to be free of kinks. Alternatively, add the ADIFOR exception handling libraries into compilation (see the ADIFOR manual).

   If your model has been designed to be free of kinks, you can comment out all calls to ADIFOR's exception handler. There is one call, "`call ehsfid`" at the beginning of every procedure that can safely be commented out. On further calls to the exception handler, you might want to add a print statement indicating that an exception has been reached. If your model has been designed to be free of kinks, this print statement should never be executed. If you see the output of the print statement in the homotopy output, a mistake has occurred; examine the code and read the ADIFOR documentation to identify the exception that ocurred and try to resolve it – either by directly altering the code or using ADIFOR's exception-handling libraries.

9. If `rho.f` includes any functions that are not affected by `LAMBDA` or the `X` vector (like `NDX()` in `QLD_rho.f`), `g_*_rho1.f` will contain exact copies of these functions and you should delete them in order to avoid "already defined" compiler errors.

10. Replace RhojsN.f in the compilation list with `g_*_rho1.f` and `rhojsA.f`. It follows that the complete list becomes:
    `Main.f` (if using a dense Jacobian, replace with `MainFull.f`)
    `rho.f`
    `SparseStru.f` (not used with dense Jacobian)
    `Src/hompack90.f`
    `Src/lapack.f`
    `Src/hom_fileIO.f`
    `Src/blas*.f`
    `g_*_rho1.f`
    `Src/rhojsA.f` (if using a dense Jacobian, replace with `rhojacA.f`)

- compile and run
- you should see the homotopy algorithm going through the steps, and step files should be saved in HomXpt

5.8 Reading the output (`HomRead.m`)

To read the output, you can create a Matlab script that will go through the files in the HomXpt directory. For each file:

1. Use `fread` to read an $(N+2)$-dimensional vector from the file; it contains (x, λ, s). $s \in R$ is the path-following parameter (or path length) described in the main paper. Changes in s from step to step show you large each step is and accordingly how quickly the algorithm moves along the solution path.

2. Recover model parameters from λ: It is a good practice to either keep parameter values in Matlab's memory, or read them from `hom_start.dat`.

3. Break up x into policy and value functions, and convert them into matrices (use `reshape`).

Do whatever you please with the equilibrium you have – save it, test it, compute economic indicators, plot it etc. See `QLD/QLD_HomRead.m` and `LBD/hom_steps.m` for some ideas.

References

Bischof, C., Khademi, P., Mauer, A. & Carle, A. (1996), 'ADIFOR 2.0: Automatic differentiation of Fortran 77 programs', *IEEE Computational Science and Engineering* **3**(3), 18–32.

Borkovsky, R., Doraszelski, U. & Kryukov, S. (2010), 'A user's guide to solving dynamic stochastic games using the homotopy method', *Operations Research* **forthcoming**.

Pakes, A. & McGuire, P. (1994), 'Computing Markov-perfect Nash equilibria: Numerical implications of a dynamic differentiated product model', *Rand Journal of Economics* **25**(4), 555–589.

Watson, L., Sosonkina, M., Melville, R., Morgan, A. & Walker, H. (1997), 'Algorithm 777: HOMPACK90: A suite of Fortran 90 codes for globally convergent homotopy algorithms', *ACM Transcations on Mathematical Software* **23**(4), 514–549.

Zangwill, W. & Garcia, C. (1981), *Pathways to solutions, fixed points, and equilibria*, Prentice Hall, Englewood Cliffs.