Extending YADA — A Guide to The User Interface, The Controls and The Data Structures

Anders Warne

September 15, 2017

Abstract: YADA (Yet Another Dsge Application) is a Matlab program for Bayesian estimation and evaluation of dynamic stochastic general equilibrium and vector autoregressive models. The purpose of the paper is to show how the program can be extended with new functionality. This requires that the programmer knows how to access model related information, such as the selected sample, from physical or virtual memory. The paper shows how such information can be located and used for extending the functionality of the program. Moreover, it shows how the GUI tools in YADA can be used and gives a detailed description of the programming conventions that are followed. The bulk of the paper contains an example extension and shows how that functionality can be added to YADA.

1. Introduction

YADA is a Matlab program for Bayesian estimation of and inference in Dynamic Stochastic General Equilibrium (DSGE) and Vector Autoregressive (VAR) models. YADA is developed in connection with the New Area-Wide Model (NAWM) project at the Monetary Policy Research Division (MPR) [formerly in the Econometric Modelling Division (EMO)] of the European Central Bank (ECB); see Christoffel, Coenen, and Warne (2008). The mathematical details about the algorithms used by YADA are described in the YADA Manual; see Warne (2017).

The software relies on code made available to the NAWM project by colleagues at central bank institutions, the software industry and the academic world. In particular, it uses some functions and ideas developed by Mattias Villani at Sveriges Riksbank. Moreover, it makes use of AiM (developed by Anderson and Moore at the Federal Reserve) to parse and, optionally, solve the DSGE model. It also includes csminwel and gensys, developed by Christopher Sims, as well as code from Stixbox by Anders Holtsberg, from the Lightspeed Toolbox by Tom Minka, from Dynare by Michel Juillard and Stephane Adjemian, and from the Kernel Density Estimation Toolbox by Christian Beardah.

The purpose of this document is to provide details on how YADA can be extended, while making sure that the basic principles underlying the YADA code can be followed. To achieve this I will first introduce the three main data structures that YADA are based on. Since YADA is a GUI-based (Graphical User Interface) program two of these structures concern the GUI. The first variable holds all the handles to the controls on the main dialog. Through these handles it possible to access (read) or change (write) various properties of the controls, such as if a control is enabled or disabled, the currently selected value, which control was last clicked on,

---

Remarks: Copyright © 2006–2017 Anders Warne, Forecasting and Policy Modelling Division, European Central Bank. I have received valuable comments and suggestions by past and present members of the NAWM team: Kai Christoffel, Günter Coenen, José Emilio Gumiel, Roland Straub, Michal Andrle (Česká Národní Banka, IMF), Juha Kilponen (Suomen Pankki), Igor Vetlov (Lietuvos Bankas), and Pascal Jacquinot, as well as our consultant from Sveriges Riksbank, Malin Adolfson. A special thanks goes to Mattias Villani for his patience when trying to answer all my questions on Bayesian analysis. I have also benefitted greatly from a course given by Frank Schorfheide at the ECB in November 2005. Moreover, I am grateful to Juan Carlos Martínez-Ovando (Banco de México) for suggesting the slice sampler, and to Paul McNelis (Fordham University) for sharing his dynare code and helping me out with the Fagan-Lothian-McNelis DSGE example. And last but not least, I am grateful to Magnus Jonsson, Stefan Laséen, Ingvar Strid and David Vestin at Sveriges Riksbank, to Antti Ripatti at Soumen Pankki, and to Tobias Blattner, Boris Glass, Wildo González, Tai-kuang Ho, Markus Kirchner, Mathias Trabandt, and Peter Welz for helping me track down a number of unpleasant bugs and to improve the generality of the YADA code.

etc. This structure is simply called `controls`, but like every variable in YADA it is a local rather than a global variable. That is, the name of the variable is only unique to a particular function (or section of a function) and does not interfere with selected variable names in other functions. As a rule, YADA does *not* use global variables and I would strongly advice anyone who wants to extend YADA to avoid such variables. They simply cause a terrible mess.

The second GUI-related structure contains initialization information about such things as the name, size, weight and angle of the font for the controls that display text. The initialization data is, e.g., needed by various dialog functions, such as message boxes, that can be useful for a user to have access to when extending YADA. This structure is locally called `CurrINI`.

The third structure contains data needed to work with the DSGE model. This structure, locally called `DSGEModel`, is perhaps the most important variable in YADA since all major user selected settings are stored in it. For example, the matrices with data on the observed (endogenous) variables and the exogenous variables are fields of `DSGEModel`.

The current document is structured as follows. The graphical user interface is introduced in Section 2. The focus is here on the `controls` and `CurrINI` structures. Next, Section 3 is concerned with a some useful dialog functions, such as the `About` and the `Query` message box functions, and the `TextGUI` function for displaying text from, e.g., a text-file. We will return to these dialog functions in the second half of the document. Section 4 presents the fields of the `DSGEModel` structure in detail. Together with the material discussed in the preceeding section this provides a comprehensive overview of the object on the main dialog of YADA. With this in mind we can move on to an example of how new functionality can be added to YADA. This is covered in Section 5 where, among other things, it is shown how to make use of the initial parameter values, the posterior mode values or draws from the prior or the posterior distribution such that these values can be used for solving the DSGE model. The section also presents how to display and store the results from the computations.

2. The Graphical User Interface

2.1. `YADA`

To start YADA you call the function `YADA` from the Matlab prompt. This function is located in YADA's *base directory*, i.e., the directory where you installed YADA.[1] Just to have a name for it the base directory can, for instance, be `c:\yada`.

The file `YADA.m` controls the basic startup and shutdown sequence in YADA. As a first step `YADA` checks if your Matlab version has the required features for YADA to work properly on your operating system.[2] Provided that this is the case, the function appends the YADA directories to the Matlab path and, accordingly, there is *not* any need for the user to manually add directories to the Matlab path. Once the Matlab path is setup for YADA, warning alerts are set to off to avoid unnecessary clutter in the command window.[3]

The function that prepends the YADA directories to the Matlab path is called `YADAPath`. This function is also located in the base directory. If you wish to add new directories to YADA, it is recommended that do so either through `YADAPath`, or, even better, that you create your own path prepending function and call it from `YADA.m`.

The next step in `YADA` is to check if the disclaimer and the GNU General Public License has been accepted by the user. If not, a dialog is displayed where the user is presented with the conditions for using YADA. The user can choose if he or she wants to accept the disclaimer and the license or not. If the answer is negative, YADA restores the original Matlab path, restores

[1] When running YADA the base directory is exactly the same as the working directory in Matlab. That directory is accessed through the command `pwd`.

[2] Specifically, YADA is compatible with Matlab version 5.3 and later on MS-Windows, while Matlab 7 and thereafter are needed by YADA on UNIX and Macintosh OS X.

[3] YADA has been designed to handle errors internally and display error messages when they are caught. That is, console operations are for console programs and YADA is not a console program. Nevertheless, since there may be bugs in the user m-files or, heaven forbid, in YADA, error messages may be displayed in the command window.

the original warning setting, clears the workspace, and shuts down. On the other hand, if the license and the disclaimer are accepted, the main dialog function `YADAGUI` is called. The execution control function `YADA` now waits until `YADAGUI` is done, i.e., when you finish the GUI-based session in YADA.

The `YADAGUI` function has one input variable called `selector`, a text string that supports more than 80 values. When it is first called the value `init` is given to `selector`. The initialization procedure that `YADAGUI` runs when this value is detected reads certain dialog related initialization information, checks if the file `YADA.ini` exists in the base directory and, if found, it reads the file data. This data is limited to two entries called `CurrentModelFile` and `ModelFiles`, where the latter stores at most 10 paths and names of previously opened DSGE model files. The former entry keeps the path and name of the last used DSGE model file, which will be automatically loaded if it can be located. Once this step has been completed the main dialog window is created and user interface controls (or just controls) are placed on this window.

`YADAGUI` also has an optional output variable which is locally called `YADAHandle`. Once the GUI-based session has finished, the handle to the main dialog window is given to `YADA` so that it can close this window. Before `YADA.m` finishes it also deletes all files from the directory `tmp` below the base directory,[4] restores the original Matlab path, restores the original warning setting, and clears the Matlab workspace.

2.2. `YADAGUI`

2.2.1. *Controls on the GUI*

The main dialog of YADA is displayed in Figure 1. It has three main groups of user interface controls: menus, a toolbar, and tabs for various settings and options. Most of the GUI-related functionality in YADA are hidden below the menu items and are called through the supported menu function and a unique value for the string variable `selector`. The supported menu functions are located in the `menus` directory of YADA and have names which directly link them to a particular menu. For instance, callback functions on the `file` menu are found in the `FileMenuFunctions` file. Some of the most important menu callback functions, such as opening the help file, are also located on the toolbar. The practical aspects of the GUI are documented in the help file and will not be discussed here unless it simplifies the presentation of how to extend YADA.

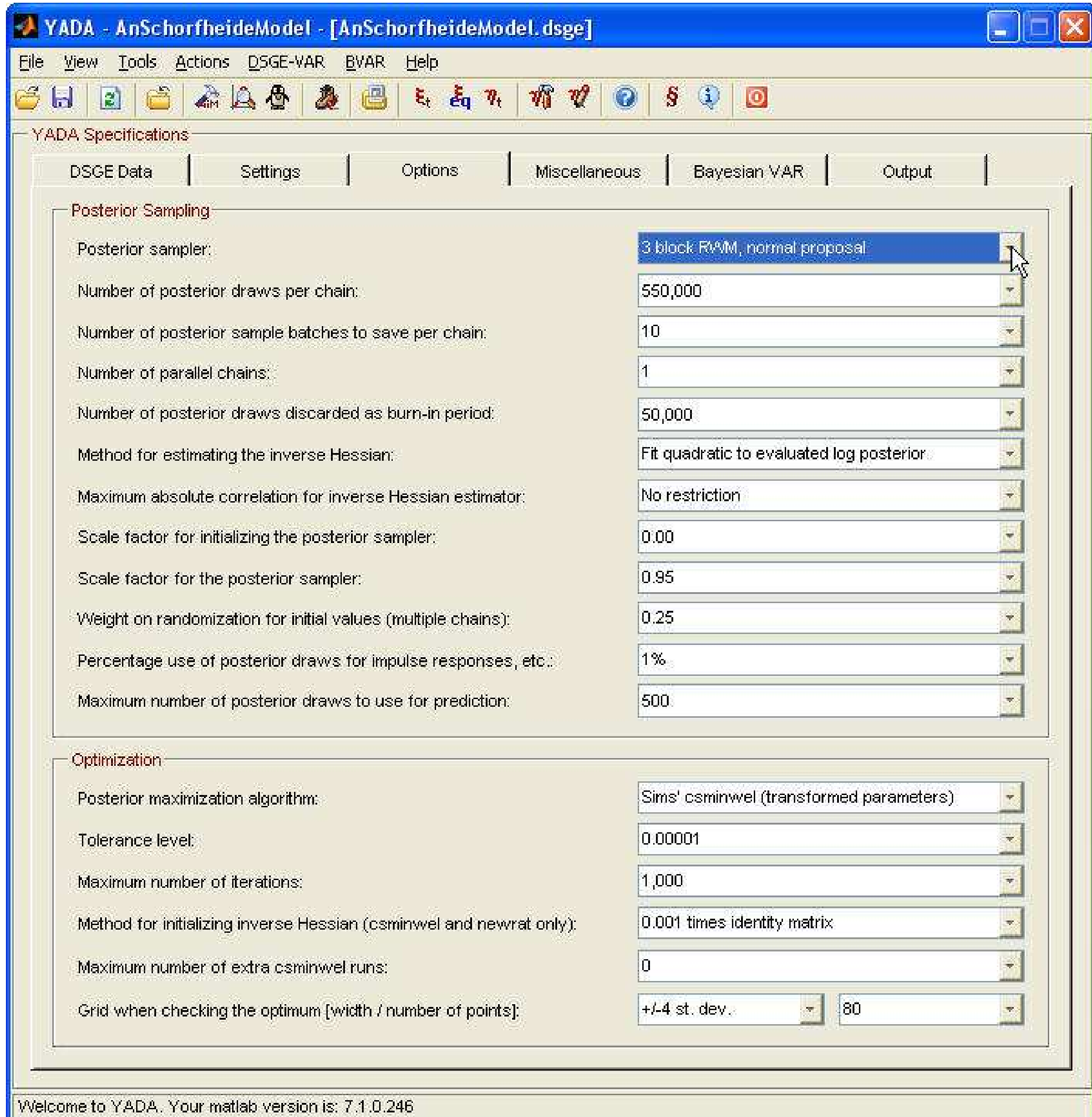
There are 6 tabs on the main dialog. All these tabs have individual frames and controls within them. For instance, the *DSGE Data* tab has 4 frames surrounding controls which hold the information that determines where the data on the endogenous and exogenous variables is located, which file contains the prior for the parameters, the location of optional functions for transforming the parameters, and the location of the AiM model file. Below the tabs there is an additional object, the status bar, where text information is displayed as the mouse is moved over the controls.

All supported values for the `selector` input variable of `YADAGUI` are cases of a `switch` statement and only a subset of the code in `YADAGUI` is therefore executed for a given value. The value `init`, mentioned above, is particularly important as it sets up the main dialog. The most important GUI-related variable that is created through this value is locally called `controls`, a structure with handles to all controls on the main dialog. For instance, there are at most 7 menus on the dialog, where at least 6 of the menus are always displayed while the Edit menu is only supported for version numbers below Matlab 7. The handle to the file menu is given by `controls.filemenu` and the parent of this control is the main dialog itself, locally called `maingui`. Items on the file menu, such as the *Open Model* function, have `controls.filemenu` as the parent. The handles to the items on the file menu are listed in Table 1. The remaining top level menu data are listed in Tables 2–8.

---

[4] YADA *copies* certain files to this directory when executing user-supplied Matlab functions; cf. Section 5.4.1.

FIGURE 1. YADA's main dialog with the Options tab.



When you want to have access to the `controls` structure from a Matlab function, such as the file with parameters to initialize (see Warne, 2017, Section 17.3), you only need to include two commands. They are:

```
maingui = findobj('Type','figure','Tag','YADA');
controls = get(maingui,'UserData');
```

The first command locates the handle to the main dialog window and lets the variable `maingui` hold its value. The second command retrieves the `controls` structure which is stored in the `UserData` property of the window. By reading the file `YADAGUI.m` you can see to which controls all handles stored in the `controls` structure are related. Notice that Matlab is case sensitive to that, e.g., `maingui` and `Maingui` are different variables.

Some of the controls on the main dialog are always enabled and the user can therefore interact with them. Other controls, however, are only enabled when certain conditions are met. For example, whenever a particular tool is based on values on the parameters of the DSGE model, these values as well as the DSGE model must be available as input to YADA. The DSGE

model is made visible to YADA through the AiM model file and the existence of the parametric solution of the model, i.e., the file `compute_aim_matrices.m`.

The four categories of parameter values that YADA supports are the following:

(1) Initial values: starting values for the parameters that can be estimated exist and calibrated values for all other parameters.
(2) Prior distribution: random draws from the prior distribution of the parameters that can be estimated exist on disk along with calibrated values for all other parameters.
(3) Posterior mode: the posterior mode of the DSGE model parameters has been estimated and exist on disk along with calibrated values for all other parameters.
(4) Posterior distribution: draws from the posterior distribution of the parameters that can be estimated exist on disk for the current selection of posterior sampling related options, as well as calibrated values for all other parameters.

Many of the tools in YADA are capable of dealing with all categories of parameter values. For instance, impulse response functions can be calculated using any one of the above choices. From the perspective of extending the set of tools in YADA an important issue is therefore how YADA learns which parameter values is has access to.

2.2.2. *Enable and Disable Controls on the GUI*

The file `YADAEnableControls` contains code that address such existential questions using the input variables `DSGEModel` and `controls`. For example, the existence of initial parameter values relies on the outcome of the test function `VerifyPosteriorModeEstimation`. This function is located in the `logic` directory, one level below the YADA base directory. If the function reports the answer `1`, then initial values are assumed to exist, while the answer `0` means that they do not.

The only input variable that `VerifyPosteriorModeEstimation` accepts is `DSGEModel`, discussed in more detail in Section 4. The outcome of the test is `1` if

- The AiM parser has been executed with a positive result, i.e., the so called AiM data file (a mat-file with the variable data from `compute_aim_data.m`) has been created in the output directory, the AiM file needed to solve the DSGE model in the output directory (`compute_aim_matrices.m`) exists, and the AiM model file exists on disk;
- The data construction file exists on disk;
- The measurement equation file exists on disk; and
- The state variables and state shocks have been selected among the candidates provided via AiM and are stored in the currently selected DSGE model file (the name of the file is displayed within brackets in the main dialog title; see Figure 1).

It is noteworthy that the existence of the prior distribution specification file is not tested here and that the parameter function files are not checked. The former file is not needed to obtain the initial parameters values if the posterior mode file exists; all such values can be inferred via the data stored is that file. The parameter function files (updating and initialization functions) are optional in YADA. Hence, we may think of the test performed by `VerifyPosteriorModeEstimation` as being a necessary but not a sufficient condition for having access to the initial parameter values. For instance, the posterior mode estimation controls are only enabled if the prior distribution specification file also exists on disk. Similarly, tools for annualized data will only be enabled for the initial parameter values when annualization information has been properly provided via the data construction file; see, e.g., Warne (2017, Section 17.5.3).

Tools that require the posterior mode values of the estimated parameters are enabled if the result from the test function `VerifyPosteriorModeEstimation` is unity and the posterior mode data file exists on disk. This file is assumed to be located in the directory `mode`, one level below the output directory. The name of the file is `PosteriorMode-`*`NameOfModel`*`.mat`, where *`NameOfModel`* is replaced with the name of the model; see, e.g., the main dialog title in Figure 1 where the name of the model is `AnSchorfheideModel`.

Similarly, tools that rely on random draws from the prior distribution of the estimated parameters are enabled if the mat-file with such draws exists on disk. This file is assumed to exist in the directory `priordraws`, one level below the output directory and its name should be `PriorDraws-`*`NameOfModel`*`.mat`. As above, *`NameOfModel`* is replaced eith the name of the model. The random draws, in turn, are created by the tool *Prior Sampling*, located on the Actions menu and on the toolbar, and this tool will only provide the draws when the prior distribution specification data can be located and verified to be valid.

Finally, some tools make use of draws from the posterior distribution. They are enabled when the result from the test function `VerifyDSGEPosteriorDraws` returns unity. This function is located in the directory `data`, one step below the YADA base directory and takes 3 input variables: `DSGEModel`, the current Markov chain (chain number 1 here), and `controls`. It simply checks if all the files with posterior draws for the user-selected posterior sampling options exist in the directory `rwm` or `slice`, depending on the choice of sampler, one level below the output directory.

2.2.3. *Initialization Information*

Information about the font properties of the text controls on the various dialog functions in YADA are stored in a structure which is locally called `CurrINI`. This structure also holds information about print settings for figures, such as the values of the `figure` properties `PaperUnits`, `PaperOrientation`, `PaperPositionMode`, `PaperType`, and `InvertHardcopy`. To access the structure from a Matlab function, you need to add the following command once the `controls` structure has been defined:

```
CurrINI = get(controls.filemenu,'UserData');
```

The variable is therefore stored in the `UserData` property of a menu item, i.e., the file menu. The default settings for most fields in `CurrINI` are set in `InitializeINIFile.m`, located in the subdirectory `\gui\initialize`. There is currently not any way the user can change these values inside YADA, but it is, of course, possible to edit this file.

3. Useful Dialog Functions in YADA

The tools in YADA typically need input by the user or require to display information to the user. For example, the same tool can have both a sample-based and a population-based metric, where the user should select which one YADA should compute. Furthermore, results may be provided both graphically and written to file. To meet these and related requirement, YADA has a supply of dialog functions that can be reused. When extending the set of tools or when attempting to interpret the code it is important to know how to make use of these functions. Moreover, other users will typically assume that the behavior of a program is consistent across the tools when possible and extensions should therefore at least try to follow the conventions used by YADA. The discussion in this section will therefore concentrate on the dialog functions that are the most likely to be useful for YADA extensions; see in Section 5.

3.1. *The* `About` *and* `Query` *Message Box Functions*

The `About` message box function is used to display information to the user. Typically, the information concerns something that cannot be performed or error information and the user needs to click on an OK button to close the dialog. The function accepts 6 input variables: `TextStr`, `ImageStr`, `NameStr`, `height`, `width`, and `CurrINI`. The variable `TextStr` holds the main text that will be displayed on the message box. The string variable `ImageStr` determines which icon will be displayed on the dialog. This variable accepts 5 different values: `information`, `question`, `warning`, `error` or `logo`. The last value is the default value so that all non-supported values will lead to the default value. The images themselves are loaded via the function `ReadImages`, located in the directory `gui` one step below the base directory. This function reads the image data stored in the mat-file `yada-images.mat`, also located in the `gui` directory.

The remaining input variables are optional. The `NameStr` string variable determines the text in the figure window title of the `About` dialog which, if missing, is given by `About`. Similarly,

FIGURE 2. The About dialog.

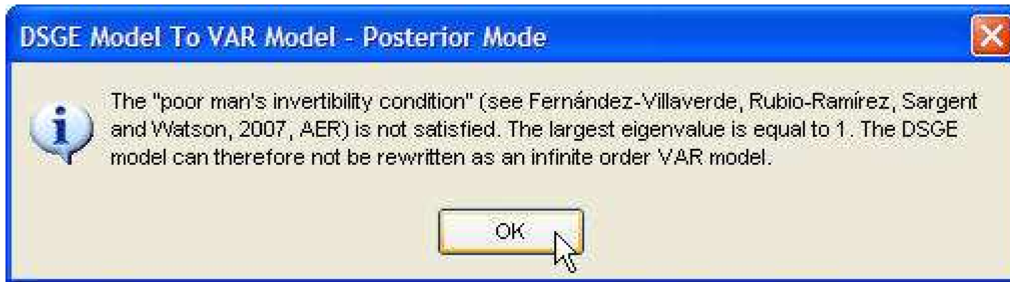
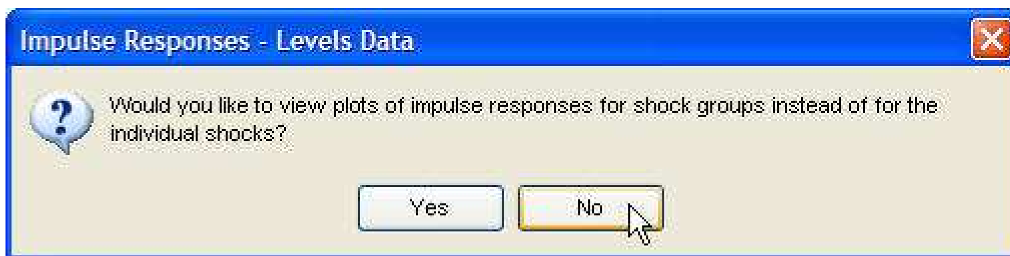


FIGURE 3. The Query dialog.



the `height` and the `width` variables provide the height and the width of the dialog, measured in pixels. The default values are 124 and 292 pixels. When the height value is provided it actually gives the maximum height of the dialog. After having set up every control on the `About` dialog, the function checks which height is needed to display the text variable `TextStr`. This is achieved through the Matlab function `textwrap`. The last input variable is `CurrINI`, discussed above in Section 2.2.3, and it contains the information needed to provide the font data for the `String` property of the controls on the dialog.

An example of how the `About` function is used in YADA is shown in Figure 2. In this case the so called "Poor man's invertibilty condition" of Fernández-Villaverde, Rubio-Ramírez, Sargent, and Watson (2007) has been tested to check if the DSGE model can be rewritten as an infinite order VAR model. For the An and Schorfheide (2007) model the invertibility condition is not satisfied; for details, see, e.g., Warne (2017), Section 11.11. The actual call to `About` is here given by:

```
About(txt,'information','DSGE Model To VAR Model - Posterior Mode',...
      120,500,CurrINI);
```

The `txt` string variable depends on the output variable `status` from the `DSGEtoVARModel` function. A unit value implies that the invertibility condition is satisfied, zero means that the matrix on lagged state variables has a unit root, while minus unity means that the number of economic shocks and measurement errors exceeds the number of observed variables.

The `Query` message box function is used to display a question to the user that only has two answers. The answer to the question is typically `Yes` or `No`, but it is possible to have other answers as well. The function accepts at most 10 variables: `TextStr`, `ImageStr`, `height`, `NameStr`, `width`, `FocusStr`, `CurrINI`, `ButtonNameStr`, `ButtonWidth`, and `ButtonTxtFile`. Only the first 3 variables are required. The first 5 and the 7th input variable serve the same purpose as the identically named variables in the `About` function. The `ImageStr` variable accepts two additional values: `exit` and `delete`, yielding an exit and a delete icon.

The `FocusStr` variable is made use of for Matlab version 7 or later and can either have the value `Yes` or `No`. The selected value determines if the left (`Yes`) button is given focus (default), or if the right (`No`) button has focus.

`ButtonNameStr` is a cell array of strings that should have 2 or 3 entries. The default values is `Yes` and `No`, but the user can provide other strings with the variable. For instance, the dialog

for displaying the YADA disclaimer uses this variable to give the strings 'I Accept' and 'I Don't Accept' for the first two entries of ButtonNameStr. A third entry is also valid, but not required, and will be placed as the text string on a third button in the bottom left corner of the Query dialog when present. This button is used to display a text-file in the TextGUI dialog when clicked on. For the YADA disclaimer example the string value is 'View License'.

The 9th input variable is ButtonWidth which provides the width of all buttons. The default value is 75 pixels and values below this number are not valid. Finally, the text-file that should be displayed if the third button is shown is given by the variable ButtonTxtFile. The string should include the full path and name of this file.

The function provides one output variable, locally called answer. The value is given by the text string on the Yes and No buttons, while the third button has no effect on this.

To examplify, the YADA displaimer dialog uses the following code:

```
Answer = Query(TxtStr,'logo',500,'Appropriate Legal Notices for YADA',...
               600,'no',InitializeINIFile,...
               {'I Accept' 'I Don''t Accept' 'View License'},125,...
               [pwd '\gpl.txt']);
```

The TxtStr variable is here a string matrix that holds the disclaimer, while the YADA function InitializeINIFile provides the initialization information from CurrINI that Query needs. This function is located in the directory gui\initialize, two steps below the YADA base directory. Finally, the above code shows the only output variable that the dialog function can provide, the string vector Answer. Its value is equal to the string vector on the button the user clicked on. For the example above, the two possible values are 'I Accept' and 'I Don't Accept'.

A more typical example of how the Query dialog function is used in YADA is what happens when the user runs impulse responses for, e.g., the levels data, using draws from the posterior distribution. In this case the following code is used:

```
ShockType = 'Individual';
if length(DSGEModel.ShockGroups)>max(DSGEModel.ShockGroups);
   txt = ['Would you like to view plots of impulse responses for ' ...
          'shock groups instead of for the individual shocks?'];
   answer = Query(txt,'question',200,QueryHead,500,'no',CurrINI);
   if strcmp(lower(answer),'yes')==1;
      ShockType = 'Group';
   end;
end;
```

The resulting Query dialog is shown in Figure 3 provided that the number of shock groups (given by max(DSGEModel.ShockGroups)) is less than the number of shocks (equal to the length of this vector). The variable QueryHead is a string vector that takes the value 'Impulse Responses - Levels Data' when impulse responses for the levels has been chosen by the user.

3.2. *Displaying Text Data with the* TextGUI *Function*

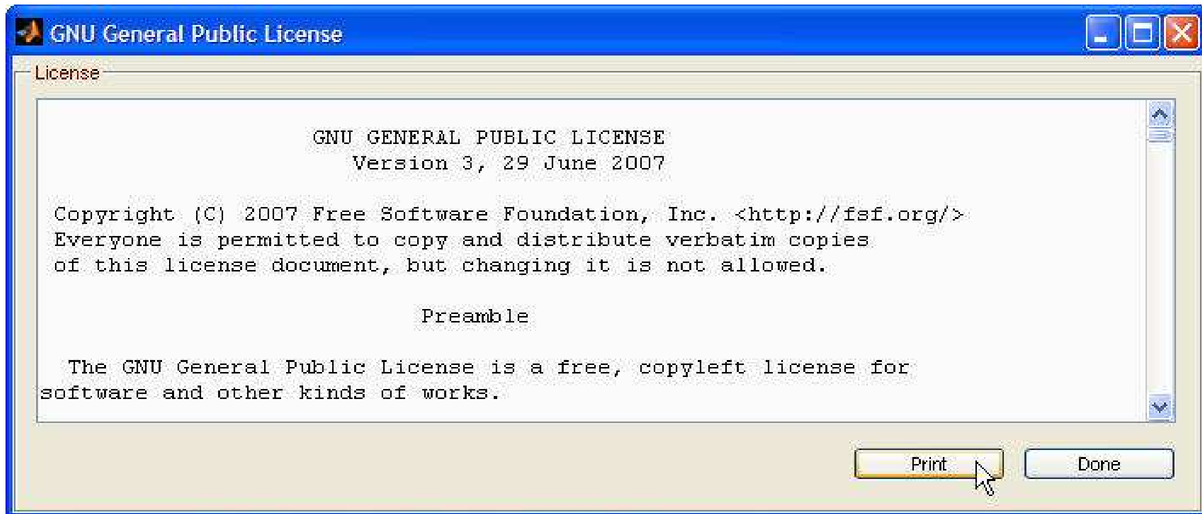
The common tool for displaying large chunks of text data in YADA is the TextGUI function. An example of a large chunk of text is the GNU General Public License as shown in Figure 4.

The TextGUI function accepts 8 input variables: TextFile, NameStr, position, AxesBoxText, AxesBoxWidth, CurrINI, DisplayCopy, and CloseSelf, where the last input is optional. The TextFile variable is either an actual text file or an string matrix. Both types of input hold the information that should be displayed in the dialog. The input NameStr holds the figure window title, e.g., 'GNU General Public License' as in Figure 4.

The position variable is a 1×4 vector with non-negative integers providing the position on screen of the dialog. The entries are the [left, bottom, width, height] positions measured in pixels.

The TextGUI dialog has one frame and the string vector for the text at the top right corner of the frame is given by AxesBoxText, while the width of the string vector, in pixels, is equal

FIGURE 4. The `TextGUI` dialog.



to `AxesBoxWidth`. In fact, the value of `AxesBoxWidth` does not matter since `AxesBox` (which creates the frame) makes sure that width matches what is required to ensure that `AxesBoxText` can be displayed. This is achieved by taking the `Extent` property of the control into account. The `AxesBox` function is located in the `gui` directory.

The next input variable is the by now well known structure `CurrINI`, while the 7th input variable is `DisplayCopy`. The latter is a boolean variable and if it is equal to unity, then `TextGUI` will display a *Copy* button provided that you are using Matlab version 5.x. In that case, the copy button makes it possible to copy the content of the dialog to the Windows clipboard. If you have a more recent version of Matlab, you are using Matlab on a unix or a mac computer, or if `DisplayCopy` is 0, then `TextGUI` will not have a copy button in the bottom right corner.

The last input variable is `CloseSelf`, a boolean variable that is 1 if `TextGUI` should deal with closing itself and therefore let the function that has called it continue directly. The alternative is that it is equal to 0, implying that the function calling it has to wait until the user manually closes the `TextGUI` dialog, and this is the default value of `CloseSelf`.

As an example, the dialog displayed in Figure 4 is created through the following code:

```
TextGUI([pwd '\gpl.txt'],'GNU General Public License',..
        [(CurrINI.scrsz(3)-700)/2 (CurrINI.scrsz(4)-400)/2 700 500],...
        'License',40,CurrINI,1);
```

The field `scrsz` of `CurrINI` is given by `get(CurrINI.GraphicsRoot,'ScreenSize')`, which retrieves the screen size property of the root window, i.e., your screen. The value for this property is

$$\begin{bmatrix} 1 & 1 & \text{screen-width} & \text{screen-height} \end{bmatrix}.$$

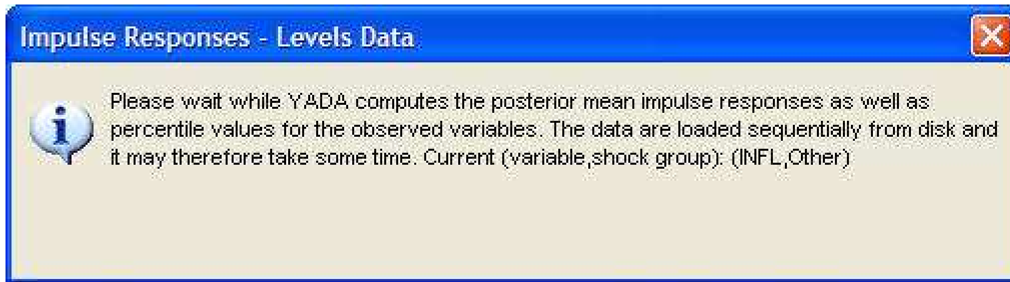
You can check this property yourself by typing `get(groot,'ScreenSize')` at the Matlab prompt, or for older Matlab versions `get(0,'ScreenSize')`. From version 2014b and later of matlab, the `groot` function should be used instead of `0` in the get call. You may also notice that the optional `CloseSelf` is not used in the above call to `TextGUI`. Hence, the default value of 0 is given to the variable.

Finally, notice also that the field `GraphicsRoot` of the structure `CurrINI` is equal to `0` if the builtin matlab function `groot` does not exist, and equal to `groot` otherwise.

3.3. *The* `WaitDLG` *and* `ProgressDLG` *Functions*

When YADA is performing time-consuming calculations it typically provides information about the status of the computations. There are two functions for providing such information called

FIGURE 5. The WaitDLG dialog.



WaitDLG and ProgressDLG. Both functions can be updated with new information and the practise used in YADA is that the latter function provides more frequent updates while the latter is updated more sparsely. For example, when running the random-walk Metropolis algorithm, the ProgressDLG is used when the option "Show dialog during optimization and posterior sampling" in the *Progress Dialog Selections* frame on the *Settings* tab is check marked. In that case it is updated for every draw from the proposal density. In contrast, if this option is not check marked, the WaitDLG dialog is shown, but is never updated during the random-walk Metropolis step for a given Markov chain.

The WaitDLG function accepts the 7 input variables TextStr, ImageStr, NameStr, width, height, CurrINI, and ShowCancel. The TextStr variable is a string matrix with the text that should be displayed on the dialog, while ImageStr is a string vector that determines which icon is displayed on the dialog. The latter variable accepts the values information, question, warning, error, and the default value logo. Furthermore, NameStr is a string vector that determines the text displayed in the figure window title, whereas width and height hold the dialog width and height values in pixels. Next, CurrINI keeps the usual font property data for the dialog and, finally, the optional boolean variable ShowCancel is unity if a Cancel button should be displayed on the dialog, and the default value zero otherwise.

The WaitDLG dialog shown in Figure 5 is created with the following code:

```
if IsPosterior==1;
   txt = ['Please wait while YADA computes the posterior mean...'];
else;
   txt = ['Please wait while YADA computes the prior mean...'];
end;
WaitHandle = WaitDLG(txt,'information',...
             ['Impulse Responses' PriorHeader ' - ' TypeStr ' Data'],...
             500,150,CurrINI,0);
WaitControls = get(WaitHandle,'UserData');
```

The boolean variable IsPosterior is unity if posterior draws have been used for the impulse responses and 0 if prior draws were used. The PriorHeader string depends on IsPosterior as well and is empty for the posterior draws. The TypeStr string vector supports the values Original, Annualized and Levels.

The handle to the dialog is the only output variable of the WaitDLG function and is here called WaitHandle. It is used to retrieve data from the dialog as well as to close it. For example, the variable WaitControls, stored as the UserData property of the dialog, is a structure that holds all the handles to the controls on the dialog. This means that it we wish to update the text on the dialog we do that via the handle to the control that holds the txt data. The field text is used for this. Hence, we may consider the following:

```
set(WaitControls.text,'String',txt);
```

where txt is here assumed to hold an updated string matrix with the text for the dialog. This code can, for example, be executed inside a for-loop. To ensure that the dialog is properly

FIGURE 6. The ProgressDLG dialog.

refreshed I would also recommend to add the calls `drawnow` and `pause(0.02)` below the `set` call.

The final step is to close the `WaitDLG` dialog. To do so we change the `UserData` property of the text control from its original value `working` to `done`. The following code will take care of this:

```
set(WaitControls.text,'UserData','done');
delete(WaitHandle);
```

The `ProgressDLG` dialog is based on the Matlab function `waitbar`. The progress dialog is initialized by calling it with 2 input variables: `x` and `whichbar`. The scalar `x` is normally set to 0 since it measures the progress between 0 and 1. The variable `whichbar` is a structure with fields `name`, `title`, `label`, `facecolor`, `startfacecolor`, `bgcolor`, `edgecolor`, `stop`, `clock`, and `CurrINI`. The `name` field is equal to a string vector with the figure window title, while the `title` field holds a string vector with the title text above the progress-bar. Similarly, the field `label` gives the string vector shown direcly below the progress meter. As an example, see Figure 6 where `name` starts with `Random Walk`, `title` begins with `Progress for`, and `label` with `Acceptance`.

The fields `facecolor` and `startfacecolor` determines the color of the progress-bar when it has performed 100 percent of the tasks and 0 percent, respectively. The `facecolor` property of the underlying `patch` object is set by these fields and the color is thus determined by a 1×3 vector with values between zero and one, representing the degree of red, green and blue. The transition from `startfacecolor` to `facecolor` is handled by `ProgressDLG`. The `bgcolor` field holds the background color of the bar. YADA always uses white, i.e., the 1×3 vector [1 1 1].

The color of the edges of the progress meter is determined by the field `edgecolor`. YADA always uses black, i.e., it sets `edgecolor` to [0 0 0]. The `stop` field is unity if a done button should be displayed next to the cancel button on the dialog, and 0 otherwise. There is currently no case where `stop` is set to unity in YADA.

The field `clock` is unity if a clock should be displayed on the dialog and 0 otherwise. The option "Show clock on dialog" in the *Progress Dialog Selections* frame on the *Settings* tab determines if a clock should be shown on the progress dialog or not.

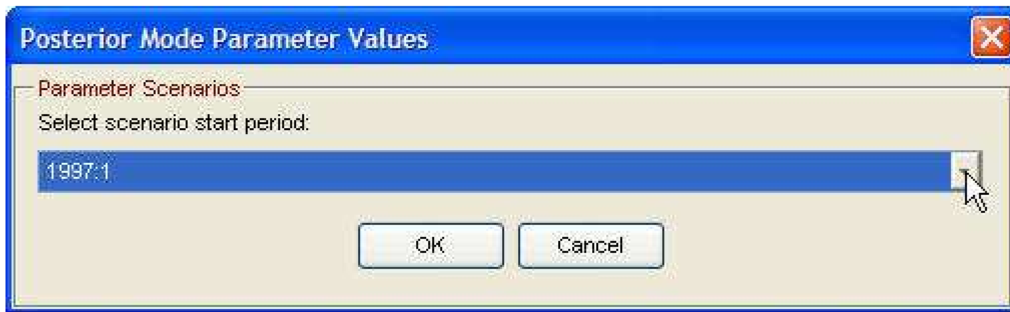
To initialize the progress dialog in YADA the following code can be used:

```
WaitHandle = ProgressDLG(0,ProgressStructure);
set(WaitHandle,'Color',get(CurrINI.GraphicsRoot,...
    'defaultuicontrolbackgroundcolor'));
```

The only output variable supported by `ProgressDLG` is `WaitHandle`, the handle to the dialog. It is assumed that the structure `ProgressStructure` has been defined as discussed above. Updating of the dialog can be performed via the following code inside a for-loop:

```
abort = get(WaitHandle,'UserData');
if strcmp(abort,'cancel')==1;
   break;
```

FIGURE 7. The `SelectionDlg` dialog.




```
else;
   ProgressDLG([NumDraws/TotalDraws AcceptanceRatio]);
end;
```

If the user has clicked on the cancel button, this changes the `UserData` property value of the progress dialog itself. Specifically, its value is set to `cancel` and the for-loop should therefore be terminated immediately. In the event that the cancel button has not been click on, the dialog receives only the input variable `x` which is now a 2-dimensional vector where the first element is a value between 0 and 1 given the degree of progress (number of draws that have been completed relative to the total number of draws to be performed), and the second element holds the updated value that should be applied to the `label` field discussed above, i.e., the string vector below the progress-bar.

Finally, to close the dialog the code

```
set(WaitHandle,'UserData','done');
close(WaitHandle);
```

should be executed. The interpretation of this is similar to the code that closes the `WaitDLG` dialog.

3.4. *Dialog for Single Popup Selection*

In some situations it is necessary that the user chooses a particular value of a single variable that can take on a large range of values. One suitable control for such a selection issue is the popup control, i.e., a `uicontrol` whose `Style` property is set to `popupmenu`. The `SelectionDlg` dialog in YADA covers such a case. Figure 7 gives an example of how this dialog can look like.

The `SelectionDlg` function accepts 8 input variables: `SelectionOptions`, `DefaultOption`, `SelectionStr`, `BoxStr`, `WindowNameStr`, `CommentsStrMat`, `CommentNameStr`, and `CurrINI`. The variable `SelectionOptions` is a string matrix with all the possible values that the user can select. The default value is given by `DefaultOption`, an integer that takes on values between 1 and the number of rows of `SelectionOptions`.

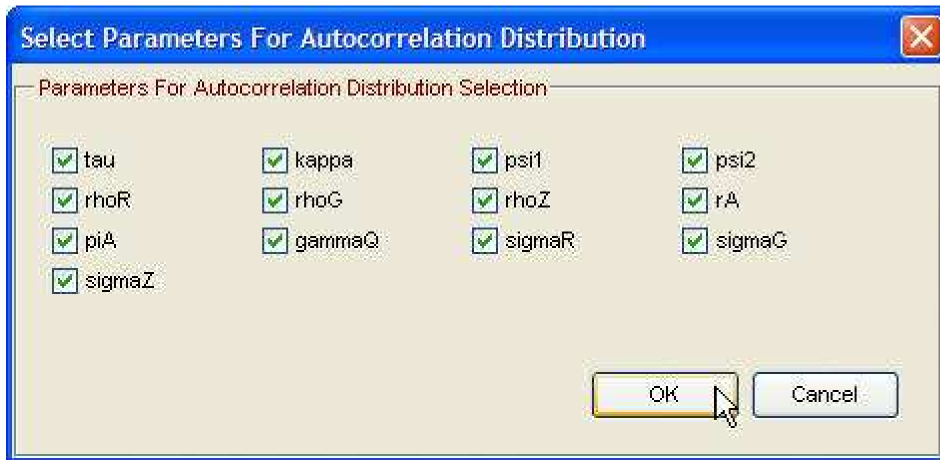
The string vector `SelectionStr` gives the text directly above the popup control with the options that the user can choose between. Next, the string vector `BoxStr` is equal to the frame text, while the string vector `WindowNameStr` is the figure window title.

The next two input variables are typically empty strings and, when not, concern an optional button that can be used to display the text in the string matrix `CommentsStrMat`. The figure window title that is displayed when the button is clicked depends on the string vector `CommentNameStr`. The final input variable is the famous `CurrINI` structure with various initialization data.

The dialog in Figure 7 is produced by the following code:

```
[action,BreakPeriod] = SelectionDlg(SampleStrMat,ParamScenarioValue,...
                'Select scenario start period:','Parameter Scenarios',...
                'Posterior Mode Parameter Values','','',CurrINI);
```

FIGURE 8. The SelectCondVarShockDLG dialog.



The function provides two output variable, locally called `action` and `SelectedOption`. The first is a string vector that can take on the values `OK` and `Cancel` based on the button the user clicked on. The second output variable is an integer value for the selected row in the input string matrix `SampleStrMat`. If the user has not changed the popup control data it is equal to `DefaultOption`.

3.5. *Dialog for Multiple Entities Selection*

It is sometimes useful to be able to select or deselect multiple entities, such as variables or parameters, from a single dialog. YADA uses such a dialog for selecting the parameters that are allowed to vary when using draws from the prior and the posterior distributions. It also uses this type of dialog when selecting which conditioning assumptions to use in a conditional forecasting exercise.

The function `SelectCondVarShockDLG` takes one required and 5 additional input variables. The first input variable is locally called `selector`, which determines which case function is executed. The other 5 input variables are required when `selector` is equal to `'init'`, i.e., when the dialog is initialized. For all other supported values of `selector`, these other input variables are not used.

The 5 additional variables are localled called `vartype`, `infotype`, `AllVars`, `positions`, and our old friend `CurrINI`. The strings `vartype` and `infotype` are used to set up the dialog window title and the frame text. The string matrix `AllVars` contains all names of variables or parameters that should be displayed in check boxes on the dialog. The vector `positions` gives the positions of the variables or parameters that are selected by default.

The first required output variable is the string vector `action` which indicates if the user has clicked on the OK or on the Cancel button of the dialog. The second required output variables is the vector `positions` which, like the input variable with the same name, indicates the positions of the variables or parameters that the user has check marked.[5]

The dialog in Figure 8 was created with the following code:

```
[Action,positions] = SelectCondVarShockDLG('init', ...
                     'Autocorrelation Distribution','Parameters For', ...
                     ParameterNames,positions,CurrINI);
```

The string matrix `ParameterNames` holds the names of all the parameters that can be estimated in its rows. When creating the dialog, the function `SelectCondVarShockDLG` chooses default

[5] YADA also has an alternative function for multiple entities selection. It is called `SetStateVarsShocksDLG` and it primarily differs from the function `SelectCondVarShockDLG` in that it also gives the names of the selected entities as an output variable and has this as in input variable. Moreover, the input variable `infotype` is not supported by that function.

number of columns and rows for the dialog based on the number of parameters. The number of columns is determined first and is first set to the minimum of the square-root of the number of parameters rounded upward and 9. Should this number be less than the minimum of 4 and the number of parameters, the number of columns is set equal to that number. Next, the number of rows for the dialog is equal to the number of parameters divided by the number of columns rounded upward.

4. The DSGEModel Structure

In this section I will focus on the fields in the DSGEModel structure. We shall first look at the fields that concern the user-written input files for the AiM model, the data construction file, the measurement equations, the prior distribution specification, and additional parameter functions. Next, we look in more detail at the fields that YADA sets up based on the specific information in the data construction file, such as the data on the endogenous and exogenous variables, names of variables, and so on. We then look at the fields of DSGEModel that are related to AiM, including the selection of state variables and state (or structural) shocks. The sample selection related fields are discussed in the following section, while the fields that are used in forecasting are discussed next. The fields concerned with optimization related features are presented thereafter, before we turn to fields that affect how posterior sampling is performed. After we give some attention to a number of fields that are connected with graphic selections we consider fields linked with analysing a Bayesian VAR model with a steady-state prior.

Before we turn to these issues, however, the structure needs to be retrieved. The following command will take care of this matter:

```
DSGEModel = get(controls.open,'UserData');
```

The open field of the controls structure holds the handle to the *Open Model* control on the toolbar. The UserData property of this control holds the DSGEModel structure whose fields are discussed next.

4.1. *Fields in* DSGEModel *Related to User Files*

Most of the fields that are related to user files in YADA are also linked to controls on the *DSGE Data* tab on the main dialog. Table 9 lists the fields of the DSGEModel structure that concern the user files, some additional settings, and the controls on the DSGE Data tab.

First of all, the data construction file, discussed in more detail below in Section 4.2, is stored as a string in the text control controls.dsge.datafile. The field name for the DSGEModel structure is here given by DataConstructionFile.

The measurement equation file location data is stored in the field MeasurementEquationFile, while the prior distribution specification file location is given by the field PriorFile. If the latter file is an Excel spreadsheet, then the field PriorFileSheet holds the sheet name string. For models with a system prior, the field SystemPriorFile holds the path and name of the system prioe file.

Two functions for transforming parameters are given by the fields InitializeParameterFile and UpdateParameterFile, where the former only computes calibrated parameters and the latter parameters that are functions of the parameters that can be estimated. The order in which these two functions are executed can be influenced via the field RunInitializeFirst. Since the file with parameters to initialize can only be executed when initial values are used for the parameters that can be estimated, or when the posterior mode estimation routine is started up, this ordering is only important in some very special cases.

Finally, the full path and the name of the AiM model file is stored in the field AIMFile, while the name of the model is given via the NameOfModel field. The latter is often used to uniquely identify a given DSGE model directly or its output. In fact, when YADA sets up a candidate output directory, another field of the DSGEModel structure, it makes use of the NameOfModel field. Furthermore, the field name of the output directory is OutputDirectory. Additional fields concerned with the AiM part of YADA are discussed in Section 4.3.

4.2. *Fields in* DSGEModel *Related to Observed Data*

All fields related to the observed data in the DSGEModel structure except two are read from the data construction file. Table 10 lists the fields alphabetically.

The most important fields concern the actual data, variable names, and a mapping to the sample of the data points. Specifically, the field Y is an $n \times T$ matrix with the T data points for the n observed (endogenous) variables in the rows. The names of the individual variables are stored in the string matrix, accessible through DSGEModel.VariableNames. The number of rows of this matrix is equal to n, while the number of columns is equal to the longest name of an individual variable. For any variable whose name has fewer characters than the longest, the row is appended with space characters. The dimensions of DSGEModel.Y are stored in two separate fields to DSGEModel, called n and T. The field YNaNs provides an indicator if Y has missing observations, represented by NaN, or not. In the former case the field has a value of unity, while it is zero for the latter case.

Data for the exogenous variables is found in the $k \times T$ matrix X, while the field k gives the number of such variables. The names of the exogenous variables are given in the rows of the string matrix XVariableNames. In order to perform out-of-sample forecasting, additional data point for the exogenous variables can be found in the $k \times T_p$ matrix DSGEModel.PredictedX. YADA assumes that the data in this matrix follows directly after the data in DSGEModel.X. If the data on observed and exogenous variables should also be used for estimating a Bayesian VAR model with a steady-state prior (see Section 4.10 below or Warne, 2017, Section 14, for more details), the vectors BVARY and BVARX with unique integer values between 1 and n and 1 and k, respectively, will determine which variables are included in such a model.

The vector structure Actuals has two subfields, data and title, which can hold alternative actuals of the observed variables. The data subfield is then an $n \times T_h$ matrix with $T_h \geq T$, while the title subfield is a string vector with a name (or title) of the data in question. Severeal such alternative datasets may be included and these are setup in the data construction file. A typical use of such alternative observed data is in a forecasting exercise using real-time data, where more than one set of actual observations can be relevant when computing, e.g., the predictive likelihood in density forecasting comparisons; see, e.g., Croushore and Stark (2001) and Croushore (2011a,b).

Information about which variables can be annualized and how it is achieved is located in the vectors annual and annualscale. Elements of the former that are unity indicate the corresponding observed variable is already annualized, 4 means that it is annualized by adding the current and past 3 quarters for an annualized difference, while 12 means that annualization is achieved by adding the current and previous 11 months of the variable. Similarly, the vector annualscale holds constants that should be multiplied by the resulting value from the additions indicated by the elements of annual. For some tools (like impulse responses) it is also useful to be able to calculate the effect on the levels of the observed variables. The vector levels indicates if a variable is in levels (1) or first differences (0). Individual transformation functions for the observed variables can be found in the field YTransformation, while joint transformations may be applied after the individual through the matrix YTransMatrix. Such data is provided by the user in the data construction file and more details can be found in the YADA Manual; see Warne (2017, Section 17.5.1).

The original sample date information is stored in 5 string fields: BeginYear and BeginPeriod for the first data point in the matrix DSGEModel.Y, and EndYear and EndPeriod for the T:th and last data point, while the field DataFrequency is a string with values a (annual), q (quarterly), or m (monthly). The BeginYear and EndYear string can be converted to integers via the Matlab function str2num, with the value of the converted EndYear field being greater than (or equal to) that of the converted BeginYear field. Similary, the fields BeginPeriod and EndPeriod can be converted to integers via the same Matlab function. The period entries measure the quarter number or the month number, depending on the frequency of the data.

When a user wishes to undertake conditional forecasts it is necessary to provide information on the conditioning assumptions. The general setup for conditional forecasts is discussed in

the YADA Manual (Section 12.2) where a mapping from current and past observed variables along with initial conditions to the current conditioning assumptions is shown (see also Section 17.5.6 of the YADA Manual). The data points for the conditioning assumptions are given in the $m \times T_z$ matrix `DSGEModel.Z`, where the last observation for the selected sample on the the observed data appears in a time period prior to period T_z. The names of these variables are available in the field `ZVariablesNames`, a string matrix with m rows. The relation between current observed variables and current conditioning assumptions is located in the field `K1`, an $n \times m$ matrix, while the link between lags of the observed variables and the current conditioning assumptions is given by `K2`. The initial conditions for the relationship between the observed variables and the conditioning assumptions are found in `U`, an $m \times T_z$ matrix. It is important to note that the first data point of `Z` and `U` is assumed to be from the same calendar date as the first data point in `Y` and `X`. The current choice of conditioning assumptions from `Z` is given by the vector `ConditionalVariables`, while the shocks that are manipulated to ensure that these assumptions are satisfied is given by the vector `ConditionalShocks`.

It is also possible to compute conditional forecasts based on assumptions for the (unobserved) state variables. Moreover, these conditioning assumptions can be combined with the assumptions for observed variables. The general setup for such conditioning assumptions is dicussed in the YADA Manual; cf. Section 13.4. The data for the state variable assumptions are given by the $q_z \times T_z$ matrix `DSGEModel.Zeta`, where the last observation for the selected sample on the the observed data appears in a time period prior to period T_z. The names of the state variable assumptions are given by the field `ZetaVariableNames`, a string matrix with q_z rows. The relation between the current value of the state variables and the current value of the state variable assumptions is given by `K3`. The current choice of conditioning assumptions from `Zeta` is given by the vector `ZetaConditionalVariables`, while the shocks that are manipulated to ensure that these assumptions are satisfied is given by the vector `ZetaConditionalShocks`.

YADA supports having a zero lower bound on the monetary policy rate; cf. Warne (2017, Section 3.4). Two of the `DSGEModel` fields are generated from data contained in the data construction file. They are called `ZLBData` and `ZLBPosition` in the `DSGEModel` structure and are collected from the subfields `Y.ZLBdata` and `Y.policyrate` of the output structure from the data construction file. The former gives a vector with values of the "zero" lower bound, where having a vector makes it possible for this lower bound to be time-varying. The latter subfield gives the position of the monetary policy rate among the observed variables.

The choice of percentiles for plotting various distributions is also given via the data construction file. In terms of the `DSGEModel` structure, the field `Percentiles` is a vector that holds an even number of elements with values greater than 0 and less than 100, sorted from the smallest to the largest. These values are used to construct confidende bands when plotting distributional features of the DSGE model, such a the predictive distributions. If the vector is, e.g., given by [5 20 70 90], then YADA creates an 85 percent confidence band using the first and last elements, and a 50 percent band using the middle two elements.

The two fields related to the observed data that are *not* read from the data construction file are `ObsVarGroupNames` and `ObsVarGroups`. These are instead stored in the DSGE model file. The number of observed variable groups is less than or equal to the number of observed variables, and each variable must belong to exactly one group. The names of the observed variable groups are located in the field `ObsVarGroupNames`, a string matrix, while `ObsVarGroups` is a vector of length equal to the number of observed variables and whose elements are integer values between 1 and the number of groups. By default the `ObsVarGroupNames` is equal to the field `VariableNames`, while `ObsVarGroups` is a vector from 1 to the number of observed variables.

4.3. *Fields in* `DSGEModel` *Related to AiM*

There are a few fields in `DSGEModel` that are linked with certain AiM data. Most of them are based on output from parsing of the AiM model file. For example, once the AiM parser has finished its work and YADA has concluded that parsing may have been successful, it attempts to execute the function `compute_aim_data` that the AiM parser writes to the output directory. If YADA can also run this function successfully it stores the output variables from this function

in a mat-file whose full path and name is given by the field AIMDataFile. Among other things, the output holds then names of all parameters and all variables that AiM has located in the AiM model file. Since AiM does not make any distinction between what YADA views as state variables or as state (structural) shocks, it is necessary for YADA to learn directly from the user which variables among those located by AiM should be interpreted as state variables, which are state shocks, and which equations are state equations. The latter equations are those that when the model is solved forward expresses the state variables as functions of past state variables and current shocks.

The selection of such variables and equations is handled via the toolbar or the Actions menu functions "Set State Variables", "Set State Shocks" and "Set State Equations"; see, e.g., Table 5. The names of the selected state variables are stored in the field StateVariableNames, while those of the state shock are kept in StateShockNames. Both fields hold string matrices where each name is given in one row. In addition, the field ShockAliases holds alternative names for the shocks such that, e.g., if etaR is the name of the shock in StateShockNames, then its alias may be interest rate shock. Moreover, the selected state variables and shocks have a given position in the list of all the variables in the AiM model. These positions are stored by YADA in the vectors StateVariablePositions and StateShockPositions, respectively. Similarly, the positions of the state equations among all the equations in the AiM model file are stored in a vector called StateEquationPositions. These three vectors with positions of state variables, equations, and shocks are used when selecting which elements of the solution matrices (or the structural form matrices) provided by AiM that YADA should use.

Each state (structural) shock can be linked to a certain group of shocks in YADA. For example, some shocks in the DSGE model may be demand related, while others concern supply. Shock groups can be used in YADA with tools such as variance decompositions, impulse responses, and observed variable decompositions. The structure DSGEModel has two fields where shock group data are stored. The number of shock groups is less than or equal to the number of state shocks, and each shock must belong to exactly one shock group. The names of the shock groups are located in the field ShockGroupNames, a string matrix, while ShockGroups is a vector of length equal to the number of state shocks and whose elements are integer values between 1 and the number of shock groups.

The field ModelSolver is an integer that determines which DSGE model solution algorithm is used. Apart from AiM (see Anderson and Moore, 1985, or Anderson, 2008, 2010), YADA also supports the QZ decompositions based approaches of Klein (2000) and Sims (2002). The default is to use gensys by Christopher Sims, implying that ModelSolver is 3.

The final field in DSGEModel that concerns AiM is AIMTolerance. This integer can be mapped into the tolerance level that not only AiM but also the other model solving algorithms should use. The choice of tolerance level affects both the upper bound for the modulus of the roots of the reduced form solution, and a conditioning number with respect to non-singularity (inversion) and to compute the numerical part of the left invariant subspace of H via the orthogonal-triangular (qr) decomposition.[6] The numerical tolerance is 0.000001 by default. Normally, there is no need to change this value, but alternative values are supported by YADA through controls in the *Kalman Filter Selections* frame on the *Settings* tab.

4.4. *Fields in* DSGEModel *Related to Sample Selection and the Kalman Filter*

Data about the sample to use for estimation of the DSGE model as well as how to initialize the Kalman filter are also stored in the DSGEModel structure; see Table 12. In YADA these two features are interlinked since a training sample can be used for the Kalman filter and during that part of the sample, the value of the log-likelihood function is not affected. The so called selected sample is made up of both the training sample and the sample used for evaluating the

[6] See, e.g., the AiMReducedForm and AiMNumericShift functions for details, or write help cond, help rcond or help qr at the Matlab prompt. The matrix H is constructed from the H_i matrices presented in, e.g., equation (4.2) of the YADA Manual; cf. Warne (2017).

log-likelihood. By default, the latter sample is equal to the selected sample, while the training sample is empty.

The selected sample is stored in DSGEModel via 8 variables, 4 strings and 4 integers. The start year is given by the fields SubBeginYear and SubBeginYearValue. The former is a string giving that actual year, while the latter is an integer giving the row number in YearStrMatrix that is equal that year. The field YearStrMatrix is a string matrix with all years between the BeginYear and the EndYear fields that were discussed in Section 4.2. If, for example, BeginYear is '1980' and EndYear is '2008', then YearStrMatrix has 29 rows. When SubBeginYear is equal to '1985' this means that SubBeginYearValue is equal to 6.

Similarly, the fields SubBeginPeriod and SubBeginPeriodValue stores the first period used in the sample as a string and as an integer, respectively. For this case, all possible period values are available in the string matrix PeriodStrMatrix. Moreover, the end of the selected sample is covered via the fields SubEndYear and SubEndYearValue for the year, and by SubEndPeriod and SubEndPeriodValue for the period of that year. The function that is used for setting up these variables appropriately is called CreateEstimationSampleValues and is located in the directory data, one level below the base directory.

The first observation of the sample used for evaluating the log-likelihood function, i.e., the first period after the training sample, is given by the integer KalmanFirstObservation. This variable gives the absolute position of that observation in the sample with integer dates 1 to T_e, where 1 is equal to the data point represented by (SubBeginYear,SubBeginPeriod) and T_e is equal to the data point (SubEndYear,SubEndPeriod). When the selected sample is equal to the full sample of observations, then $T_e = T$.

The Kalman Filter is initialized through the location vector and the 1-step ahead forecast covariance matrix of the state variables for period 1 of the selected sample. The location value is by default equal to zero, but variable specific values can be selected. The vector given by the field InitialStateValues stores the choice of such values, while UseOwnInitialState, a boolean variable, is unity if the user-selected values should be used, and zero if the initial location of the state variables is equal to the default zero (steady-state) values.

YADA supports three ways of specifying the choice of initial 1-step ahead forecast covariance matrix for the state variables. The integer variable in the field UseDoublingAlgorithm is equal to 1 or 2 if the covariance matrix should be given the unconditional covariance matrix,[7] 3 if it should be given by a large constant times the identity matrix, and 4 if exact diffuse initialization should be performed. The first choice means that the long-run matrix is solved via the state equation using the (slow) analytical vectorization form, while the second choice leads to using the (fast) numerical doubling algorithm. For the latter case, the fields DAToleranceValue and DAMaximumIterationsValue determine the tolerance value and the maximum number of iterations, respectively, that are applied with the doubling algorithm. The third choice for parameterizing the 1-step ahead forecast covariance matrix of the state variables requires setting a constant. The field StateCovConst determines the choice of that constant, where the constant can take on values from 100 to 10,000 thereby providing a diffuse initialization of the Kalman filter computations.

YADA supports four approaches to computing the value of the log-likelihood function with the Kalman filter. The first method is the standard or original Kalman filter (Kalman, 1960) which is discussed in many textbooks on time series analysis; see, e.g., Anderson and Moore (1979), Durbin and Koopman (2012), Hamilton (1994), and Harvey (1989). The standard filter, where the 1-step ahead state covariance matrix is updated directly, will in most situations suffice. However, it is possible that rounding errors and matrices being near to singularity can produce state covariance matrices that are not positive semidefinite. As a technique for coping with this difficulty YADA supports *square root filtering*; see Anderson and Moore (1979, Chapter 6.5) or Durbin and Koopman (2012, Chapter 6.3). This filter guarantees that the state covariance matrices are positive semidefinite, but is therefore also slower than the standard filter. Next, YADA supports the univariate approach of Kalman filtering and smoothing, discussed by, e.g.,

[7] The matrix is not really unconditional since it depends on the parameter values of the DSGE model.

Koopman and Durbin (2000) and Durbin and Koopman (2012, Chapter 6.4), and finally the so called Chandrasekhar recursions (see Morf, Sidhu, and Kailath, 1974) can be selected for covariance stationary models with a constant mapping between the state variables and the observed variables; see also Anderson and Moore (1979, Chapter 6.7). As a consequence, missing observations are not supported by the Chandrasekhar recursions. The value of the field `KalmanAlgorithm` determines which of these four approaches is used when computing the log-likelihood, where the default is to use the standard filter.

The `DSGEModel` structure also has two fields that make it possible to allow for unit roots. The first such field is the boolean variable `AllowUnitRoot`. If this variable is unity, then YADA assumes that the DSGE model has unit roots and that any state variable can be subject to such roots. As a consequence, the 1-step ahead forecast state variance matrix is initialized by a constant times the identity, i.e., the `UseDoublingAlgorithm` field is assumed to be 3. The value of the field `AllowUnitRoot` is, by default, 0, but its value can be controlled by the user via the *Allow for undefined unit roots* option in the *Kalman Filter Selections* frame on the *Settings* tab.

A second method for allowing for unit roots is handled via the field `UnitRootStates`. This vector holds the positions of the state variables that are unit root processes. To determine those positions, the user needs to run the *Specify Unit Root State Variables* tool on the *Actions* menu; see, e.g., Table 5. When this vector is not empty, the Kalman filter need not be initialized by setting the 1-step ahead forecast state variance matrix equal to a constant times the idenity (unless this has already been selected by the user). Intead, this matrix is initialized by the selected method for all state variables that are stationary, while only those that have a unit root are assigned a constant variance and 0 covariance to all other state variables.

Whenever the DSGE model is specified as having unit roots, YADA disables all functions that requires the "unconditional" covariance matrix of the state variables to exist. Accordingly, tools such as the observed variable correlations can no longer be executed.

4.5. *Forecasting Related Fields in* `DSGEModel`

Some of the fields in `DSGEModel` concern forecasting and simulation related settings. The relevant fields are listed alphabetically in Table 13.

The forecasting methods are discussed in detailt in Section 12 of the YADA manual; see Warne (2017). Whenever you run an out-of-sample forecast tool in YADA it simulates paths for the observed variables. The number of such paths per value of the vector of estimated parameters is identifiable via the field `NumPredPathsValue`. The minimum *non-unit* number of paths is equal to 100, corresponding to `NumPredPathsValue` being unity, while the maximum number of paths is 1,000,000 so that `NumPredPathsValue` is 185. It is also possible to select just one path per parameter value which amounts to having `NumPredPathsValue` being 186!

The selected number of paths is also used when computing sample related statistics, such as sample-based moments of the observed variables from the perspective of the model. Whenever data is simulated and more than one draw from the posterior distribution is used, the number of parameter draws is determined by the field `PostDrawsUsageValue`. If the selected value for this field implies, say, 500 draws from the posterior and the selected number of paths per parameter value is 500, it follows directly that the total number of paths is 250,000.

If the sample mean of the prediction paths should be exactly equal to the population mean of the forecasts, the boolean variable `AdjustPredictionPaths` is unity. If the number of paths is large enough, the sample and population mean will be very close. Hence, the main interest of this option is when one wishes to estimate the population mean of the predictive distribution.

The maximum number of out-of-sample forecast per path that the user wants to study is stored in the field `MaxForecastHorizon`. The value of this field is binding provided that either the out-of-sample data on the exogenous variables covers the whole forecast sample, or the model does not have any exogenous variables. When both conditions are violated, the length of the forecast sample is determined by the number of consecutive data points in the forecast sample for which data on the exogenous variables exist.

Before attempting to estimate the predictive distribution in YADA, it is possible to define prediction events. Such events concern the observed variables over the forecast sample and are

defined by a lower and an upper bound for each variable as well as a length of the event. For instance, a prediction event can be defined for GDP growth such that the number of times it is less than or equal to zero for at least two consecutive periods is recorded. This number is compared with the number of times the event can occur. For example, if the forecast sample is 8 periods and the total number of prediction paths is 100, then the number of times this event can take place is 700. While running a predictive distribution estimation tool, the field `RunPredictionEvent` is a boolean variable that is unity when the user has selected to run prediction events, and zero otherwise.

Conditional forecasting in YADA can be performed through three related methods. The first is to compute values for specific shocks such that the conditioning assumptions are satisfied, the second is to draws shocks from a normal distribution that has a mean and a covariance matrix which guarantee that the conditioning assumptions are met, while the relies on a subset of shocks being drawn from a normal distribution conditional on the remaining shocks that has a mean and a covariance matrix which ensure that the conditioning assumptions are satisfied. The field `ShockControlMethod` indicates which one of these methods the user has opted to use.

Moreover, the user can also choose how the distribution of the state variables in the last historical time period should be parameterized. The first option is to use a smooth estimate of the state for that period along with a smooth estimate of the state covariance matrix when only data on the observed and exogenous variables up until the same time period is used. That is, the conditioning assumptions are not included in the information set for the smooth estimator. The second possibility is to include that information for the smoother. The latter method is preferred from a theoretical perspective, but is more costly from a computation perspective. Moreover, these two estimators of the first and second conditional moments of the state variables need not be all that different. The field `KsiUseCondData` is zero if the first method is chosen and unity if the second should be used. For both methods, the simulation of paths are initialized by drawing a value for the state variable in the last period before the forecast sample from a normal distribution with mean and variance given by the selected method.

4.6. *Optimization Related Fields in* `DSGEModel`

The most important optimization related settings in YADA can be changed via controls in the *Optimization* frame on the *Options* tab; see, e.g., Figure 1. The fields in `DSGEModel` that record these settings as well as some other posterior mode estimation related settings are shown in Table 14.

The choice of optimization routine and the selected parameters to target (transformed or original) is stored in the field `MaximizeAlgorithmValue`. YADA supports four optimization routines: `csminwel` by Christopher Sims, `newrat` by Marco Ratto and which is included in Dynare, Dynare's `gmhmaxlik`, and `fminunc` from the *Optimization Toolbox*. The former ships with YADA, while the latter can be used provided that the user (i) has the toolbox, and (ii) has made use of the diff-files that can be downloaded from YADA's homepage. The selection of optimization routine can only be changed from the default value `csminwel` when YADA is able to locate the necessary files for `fminunc`.[8]

The maximum number of iterations for the selected optimizer is determined via the field `OptMaxIterationsValue`, while the tolerance level is given through `OptToleranceValue`.

The inverse Hessian needs to be initialized when `csminwel` is the selected optimization routine. The default in YADA is the constant 0.001 times the identity, but other constants can also be chosen. In addition, it is also possible to use a diagonal matrix where the diagonal elemenst are given by the sample variance for 5,000 draws from the prior distribution of the parameters that can be estimated. The field `InitializeHessian` stores the selected option.

An additional field that concerns `csminwel` is `CsminwelExtraRuns`. This integer determines the maximum number of times that `csminwel` may be executed after the original optimization run. Provided that this maximum number of at least equal to 1, the original optimization run has converged but the gradient is larger than the tolerance level, then YADA can optionally rerun

[8] See the *Frequently Asked Questions* section in YADA's help file, or have a look at YADA's Online Help.

csminwel with the latest values for the parameters as input, while the inverse Hessian is re-initialized accoridng to the InitializeHessian field. The default value of CsminwelExtraRuns is 0, but values up to 20 are also supported.

YADA can check the surface of the posterior distribution around the mode; see Warne (2017, Section 7.2). The field CheckOptimum determines if this tool should be executed in connection with posterior mode estimation. The user can also decide if the optimum should be checked for both the original and the transformed parameters. YADA will *only* check the optimum for the transformed parameters when the field CheckTransformedOptimum is unity, and both sets of parameters when it is zero. The controls for making the selection are located in the *Tools* frame on the *Miscellaneous* tab.

The check optimum function uses a grid to select values for the parameters. The width of the grid is given by the field GridWidth, while the number of points around the mode is determined via the field NumberOfGridPoints.

During optimization, a wait dialog (see Figure 5) or a progress dialog (see Figure 6) will be shown. The preferred alternative can be selected in the *Progress Dialog Selections* frame on the *Settings* tab. Since the progress dialog needs to be updated at the beginning of each iteration, the optimization run is slower under this alternative. The field ShowProgress is unity when the progress dialog should be shown, while it is zero if the wait dialog should be displayed instead. An additional item can be shown on the progress dialog, namely, a timer. The field ShowProgressClock is unity when the user wants to add this clock, while this field does not affect the behavior of the wait dialog.

The progress dialog can be shown also when running posterior draws, when estimating the predictive distribution, when using draws from the prior and posterior distributions, etc. The value for ShowProgress is used for all such situations.

Finally, in addition to the estimate of the inverse Hessian at the posterior mode that the optimization routine provides, the inverse Hessian can be estimated using finite differences after the posterior mode has been located. The field FiniteDifferenceHessian is unity if YADA should pursue this objective and zero otherwise. This matrix can also be estimated at the beginning of posterior sampling provided that it should be the basis for the covariance matrix of the proposal density; see Section 4.7 for details. The step length that the finite difference estimator should use is determined via the field StepLengthHessian, where a step length between 0.001 and 1 can be selected.

4.7. *Fields in* DSGEModel *Related to the Posterior*

Many of the posterior sampling related settings in YADA can be changed via controls in the *Posterior Sampling* frame on the *Options* tab; see, e.g., Figure 1. The first choice on this dialog is, not too surprisingly, which posterior sampler to use. YADA currently supports seven MCMC samplers, such as the random walk Metropolis algorithm, which dates back to the Manhattan project, as well as the more recently developed sequential Monte Carlo (SMC) sampler. The field PosteriorSampler is an integer which is 1 for the random walk Metropolis algorithm with a normal proposal density; 2 for the slice sampler; 3 for the random walk Metropolis sampler with a Student-*t* proposal density; 4 for the fixed blocking RWM posterior sampler with a normal proposal density; 5 for the fixed blocking RWM posterior sampler with a Student-*t* proposal density; 6 for the random blocking RWM posterior sampler with a normal proposal density; 7 for the random blocking RWM posterior sampler with a Student-*t* proposal density; and 8 for SMC with likelihood tempering. If the third, fifth or seventh option has been selected, then the field StudenttDegFree is also used. As expected it contains the number of degrees of freedom of the Student-*t* density. The fourth and fifth options make use of the number of fixed parameter blocks. The field FixedNumParamBlocks keeps track of this integer. The field BlockSize contains the lower and upper integer values for the number of random parameter blocks.

The options shown in Figure 1 are used by the seven MCMC samplers, while they change when the SMC with likelihood tempering sampler is used; see Figure 9 below. The presentation below begins with the MCMC related options, while the turn to the SMC case thereafter.

The total number of posterior draws per Markov chain is determined via the value of the field `PosteriorDrawsValue`, while the number of draws per chain that are discarded as a burn-in sample is given via the value for `BurnInValue`. These burn-in draws are always taken from the beginning of the Markov chain so that the number of posterior draws that can be used by various tools in YADA is equal to the total number of draws minus the burn-in draws. In addition to the prespecified number of draws underlying the actual values of `PosteriorDrawsValue` and `BurnInValue`, the last entry is called `Other` and, when selected makes it possible for the user to select some other value for the number of posterior and burn-in draws. The fields `PosteriorDraws` and `BurnIn` reflect such user selected values. When prespecified number of draws have been selected, these fields are empty.

Running a long Markov chain is often very time consuming. Since there is a risk that the computations may unexpectedly stop during a Markov chain due to external factors, YADA can store results for each chain in batches. The number of sample batches per chain is determined by the field `SampleBatchValue`. For instance, if the total number of posterior draws for a chain is 550,000 and the number of sample batches is 10, then YADA will save the draws to disk sequentially after each additional 55,000 draws. Furthermore, YADA can run more than one chain during posterior sampling. The number of parallel Markov chains is given via the field `ParallelChainsValue`.

The value of the field `OverwriteDraws` determines if YADA will overwrite previously computed posterior draws or not. To be overwritten the user must (i) have selected to allow the draws to be overwritten, and (ii) the draws are based on identical settings. The first case is handled via the option “Overwrite old draws” in the *DSGE Posterior sampling* frame of the *Settings* tab. When it comes to the MCMC samplers, the draws are considered to be based on identical settings if the following six fields are the same:

- `PosteriorSampler`: The chosen posterior sampler (random walk Metropolis with a normal proposal density, slice sampler, random walk Metropolis with a Student-t proposal[9]), fixed blocking RWM sampler with a normal or a Student-t proposal[10], random blocking RWM sampler with a normal or a Student-t proposal[11];
- `NameOfModel`: The name of the DSGE model;
- `InverseHessianEstimator`: The chosen estimator of the inverse Hessian that is used for the covariance matrix of the proposal density;
- `ParallelChainsValue`: The number of parallel Markov chains;
- `PosteriorDrawsValue`: The number of draws from the posterior distribution; and
- `SampleBatchValue`: The number of sample batches to save to disk per Markov chain.

A number of fields in the `DSGEModel` structure are used for setting up the covariance matrix of the proposal density. The first such field is `InverseHessianEstimator` which determines the user’s choice of estimator of the inverse Hessian. There are 4 generic options: (i) the inverse Hessian given by the optimization routine (`csminwel` or `fminunc`); (ii) a modified inverse Hessian based on fitting a quadratic function to the log posterior; (iii) the finite difference estimate of the inverse Hessian; and (iv) a user-determined estimator that can, for instance, be the estimated covariance matrix of a previously computed posterior sample. If the 4th option is selected, then the location of the inverse Hessian is given by the field `ParameterCovMatrix`, a string with the full path and name of the mat-file where this matrix is located (the name of the variable that holds the matrix is fixed at `ParameterCovarianceMatrix`).

---

[9] In the case of the Student-t proposal density, the value of the field `StudenttDegFree` also matters as it contains the number of degrees of freedom of the density.

[10] For these two posterior samplers, the value of the field `FixedNumParamBlocks` with the number of parameter blocks matters.

[11] The field `BlockSize` holds a 1-by-2 vector with integer values that represent the minimum and maximum number of parameter blocks and is used by the random blocking samplers.

The second generic estimator of the inverse Hessian can be computed in two ways. When fitting a quadratic to the log posterior around the posterior mode, YADA estimates the standard deviation for a normal distribution such that the normal density is as close as possible in a mean-square sense to the log posterior values taken from the grid discussed in Section 4.6 when checking the optimum. The log posterior is in this case computed via a grid for each estimated parameter around the posterior mode, but where only one parameter at a time is allowed to vary over the grid while the other parameters are fixed at their posterior mode values. This means that the log posterior is a conditional distribution and, hence, that the estimated standard deviation is matched to a conditional density. If the field `ModifiedHessian` is unity then YADA computes marginal standard deviations using the correlation structure from the inverse Hessian that is supplied by the optimization routine. If this field is zero, the YADA disregards the fact that the estimated standard deviations for the modified Hessian are conditional rather than marginal.

Given an inverse Hessian, the correlation structure for this matrix can be influenced via the field `MaxCorrelationValue`. The value for this field determines the maximum absolute correlation that the user allows for. This can range from no restriction on the correlations to 0 correlation. To ensure that the inverse Hessian remains positive definite, YADA computes the ratio between the highest absolute correlation it should allow for and the highest absolute correlation that it can find in the matrix. The correlation structure is only affected if this ratio is below unity. In that case, all off-diagonal elements are multiplied by this ratio, while the diagonal elements are constant.

The covariance matrix of the proposal density for the random walk Metropolis algorithm is also influenced by a squared scale factor. The value of the scale factor itself is given via the fields `MHInitialScaleFactor` and `MHScaleFactor` when drawing the initial value for a single chain and for the remainder of a chain, respectively. If the scale factor for initializing a single Markov chain is 0, then YADA sets the initial parameter value to the posterior mode estimate. When the number of Markov chains is greater than one, the initial value of the parameters is initialized via another field, namely, `RandomWeightValue`. This field determines the weight on random draws relative to the posterior mode values. The field `MHScaleFactor` is used by both single and multiple Markov chains once they have been initialized.

The random number generators used by YADA are initialized via the `'state'` property that the functions `rand` and `randn` accept. If the user wants to have a fixed value (equal to 0) for the state, then the field `RandomNumberValue` is unity, while a zero value means that the state is set equal to the integer value `sum(100*clock)`. It should be kept in mind that this value for `RandomNumberValue` is used whenever random draws are computed in YADA.

The marginal likelihood can be estimated sequentially directly after posterior sampling or, once posterior draws exist on disk, at the user's request. For the latter case, sequential estimation can be performed via tools on the *View* menu; see Table 3. The field `SequentialML` determines if sequential estimation of the marginal likelihood should be undertaken in direct connection with posterior sampling. Its value is given via the "Compute the marginal likelihood sequentially" option in the *DSGE Posterior Sampling* frame of the *Settings* tab.

Next, the field `MarginalLikelihoodValue` gives the choice of estimator of the log marginal likelihood. The user can here choose between the modified harmonic mean estimator (Geweke, 1999), the Chib and Jelizkov estimator (Chib and Jeliazkov, 2001), both, and none.

Sequential estimation based on draws from the posterior distribution requires a suitable sample. The field `SequentialStartIteration` determines the starting period and is measured relative to the draws after the burn-in sample. If this field implies 100 draws and the burn-in sample is 100, it means that the first sequential estimate uses draws 101 until 200 when it computes a certain statistic, such as the log marginal likelihood. The increment can be computed from the field `SequentialStepLength`. If the corresponding increment is 100, then the second sequential estimate uses draws 101 until 300, and so on, until all post burn-in sample draws have been included in the estimate.

The modified harmonic mean estimator of the marginal likelihood requires a sequence of coverage probabilities. There are three fields in the `DSGEModel` structure that determine how this

sequence is defined. The field CoverageStart determines the first coverage probability, while CoverageEnd gives the last coverage probability. The values in between these are determined through the field CoverageIncrement. For instance, if these fields imply the values 0.1 and 0.9 as the first and the last, while the increment is 0.2, then the coverage probabilities used for marginal likelihood estimator are $(0.1, 0.3, 0.5, 0.7, 0.9)$.

YADA supports 2 methods for selecting draws from the posterior distribution whenever a subset of all post burn-in sample draws is asked for. If the field RandomizeDraws is unity, then a subsample of posterior draws are selected randomly from all post burn-in draws. On the other hand, if this field is zero, then YADA picks a subsample of values from the posterior draws by letting the draws have equal distance. For example, suppose that the post burn-in sample has 10,000 parameter values and a subsample of 200 values should be selected. Given the first method, YADA picks the posterior draws determines by the integer values generated by the call ceil(10000*rand(200,1)). Under the second method, it takes draws with the integer values $1, 51, 101, \ldots, 9951$.

The length of a subsample of posterior draws when data is simulated was discussed in Section 4.5; see the PostDrawsUsageValue. For many other tools, such as impulse responses, it is also possible to only use a subset of the posterior draws. The field PostDrawsPercentValue indicates the percentage of the post burn-in sample that should be used by these tools.

For certain tools it is also possible to allow a subset of the parameters to vary from one posterior or prior draw to the next. The field ScenarioParameters is a vector which indicates with 1 that a parameter can vary, while 0 means that it is fixed. When posterior draws are used, the fixed parameters are given by the posterior mode value, while for prior draws the fixed parameters are equal to the initial values.

4.7.2. *SMC Samplers*

For the SMC with likelihood tempering, the number of posterior draws is also called the number of particles; see Figure 9. The underlying DSGEModel field PosteriorDrawsValue is, of course, the same as for the MCMC samplers. At the same time, the SMC with likelihood tempering sampler does *not* use burn-in draws and therefore sets such values to zero despite the possibility that BurnInValue indicates something else.

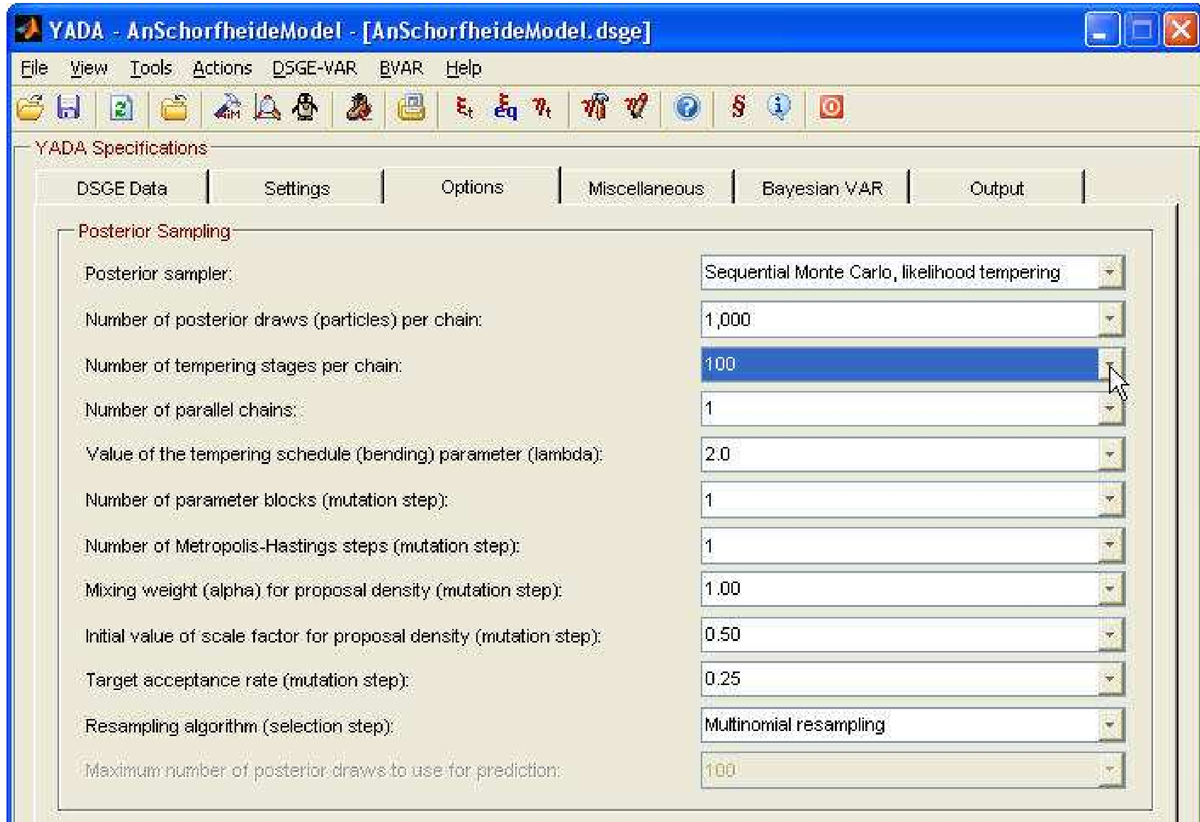
The number of tempering stages is dealt with through the field TemperingStagesValue. Its default value is unity which corresponds to $N_\tau = 100$; see Warne (2017, Section 8.4) for details. The SMC sampler does not consider sample batches like the MCMC samplers. Instead, YADA stores the results from each tempering stage to disk thus making it possible to recover older results provided that OverwriteDraws is not activate. This occurs when the user has selected to overwrite old draws and the following seven fields are the same:

- PosteriorSampler: The chosen posterior sampler;
- NameOfModel: The name of the DSGE model;
- TemperingStagesValue: The number of tempering stages;
- NumParamBlocks: The number of fixed parameter blocks;
- ResamplingAlgorithm: The resampling algorithm for the selection step and takes the values 1 (multinomial resampling), 2 (stratified resampling), 3 (systematic resampling), or 4 (residual resampling);
- ParallelChainsValue: The number of parallel Markov chains; and
- PosteriorDrawsValue: The number of draws from the posterior distribution.

It may also be noted that the bending parameter of the tempering schedule (λ) is recalled via the field TemperingLambdaValue. Furthermore, the field ResamplingThresholdValue determines the threshold value of the effective sample size below which resampling of the posterior draws (particles) takes place. The default value of this field corresponds to 50 percent of the number of posterior draws.

Turning to the mutation step of the SMC algorithm, the number of Metropolis-Hastings steps per particle is given from the field SMCNumMHSteps. Its default value is unity, but values up to 100 are supported. The mixing weight for the proposal density (α) is determined from the field

FIGURE 9. The Posterior Sampling frame on the Options tab in YADA for SMC sampling algorithms.



MixedDistWeightValue, while the initial value of the scale factor for the proposal density (c^*) is recovered from the field SMCInitialScaleFactor. Finally, the target acceptance rate (p_α) is directly linked to the field TargetAcceptanceRateValue.

4.8. *Graphic Selection Related and Miscellaneous Fields in* DSGEModel

Table 16 lists the fields that can best be linked with graphic selections. In Section 4.3 the fields ShockGroups and ShockGroupNames linking individual state (structural) shocks to groups of shocks and the names of the shock groups, respective, were introduced. These shock groups are, for instance, used by the *observed variable decomposition* tool; see Warne (2017, Section 11.8). When displaying the outcome of such an exercise, YADA used different colors to distinguish the groups from each other and individual shocks from other shocks that belong to the same group.

The field ShockGroupColors gives a matrix with 3 columns and G rows, where G is equal to the number of shock groups. Each row of this matrix is an RGB-triple (red-green-blue) with values between 0 and 1 that determines the color of a certain shock group.[12] For example, if the second row is equal to [1 0 0], the color for the second shock group is red. The field ObsVarGroupColors is defined in a similar manner based on the number of observed variable groups.

When plotting individual shocks that belong to the same shock group, YADA uses a spectrum of colors which is constructed via the field ShockColors. This field holds a matrix with as many rows as shocks and 3 columns, representing the RGB values.

The length of the impulse responses and variance decompositions is determined from the value of the field IRHorizon. Integer values between 1 and 50 years are supported in YADA.

[12] These values translate to the integer scale 0-255 ($= 2^8$), such that $x = 200/255 \approx 0.7843$ on the 0-1 scale is identical to 200 on the 0-255 integer scale.

When distributional features of a tool are plotted, the colors for the confidence bands are computed from a base color that is stored in the field `ConfidenceBandBaseColor`. Depending on the number of bands, YADA computes a constant between 0 and 1 that are multiplied by the base color, where each constant is greater than 0 and less than 1. Wider confidence bands typically have a darker shade of the base color than narrower bands. The exact values of these weight are computed from the number of bands. For example, if 3 confidence bands should be computed, the weights are 1/4, 2/4, and 3/4 for the bands. If the bands are computed from the value [5 15 25 75 85 95] for the `Percentiles` field (see Table 10), the 90 percent confidence band has color equal to 1/4 times the `ConfidenceBandBaseColor` value, while 2/4 is the constant used for the 70 percent band, and 3/4 for the 50 percent band. The method for computing the confidence bands is determined by the field `ConfidenceRegionMethod`. By default, YADA uses the equal tails method, but it is also possible to make use of highest probability density bands; see Warne (2017, Section 8.4) for details.

Densities of the marginal prior distributions for the parameters that can be estimated can be computed via two methods. The field `PriorKernel` is unity (zero) if a kernel (grid) density estimator should be used. For the kernel density case, the field `KernelDensityValue` indicates through an integer which estimator the user prefers, while `KernelDensityEstimator` is a string that identifies the selected estimator. If the grid estimator is used, the grid for the prior density is constructed from the `GridWidth` and `NumberOfGridPoints` fields when possible (see Table 14).[13]

Densities of the marginal posterior distribution are estimated based on the method implied by the value of the field `PosteriorDensityValue`. YADA supports 4 different methods:

(1) Gaussian;
(2) Silverman-type with Sköld-Roberts correction (see Silverman, 1986, and Sköld and Roberts, 2003);
(3) Sheather-Jones bandwidth (Sheather and Jones, 1991); and
(4) Bump killing bandwidth.

All methods are based on a Gaussian kernel; see Sköld and Roberts (2003) for details. The first two methods are reasonably fast, whereas the remaining are much slower.

A number of fields in `DSGEModel` that do not belong to any easily identifiable group are listed in Table 18. Forecast error variance decompositions in YADA require that a Riccati equation is solved for the 1-step ahead steady-state covariance matrix of the state variables. Since the approach in YADA is based on iterations when attempting to solve this equation, two fields deal with the maximum number of iterations and the tolerance level. The former is determined via the `RiccatiMaxIteration` and the latter through `RiccatiToleranceValue`; see Warne (2017, Section 11.5) for details.

Analyses that take the zero lower bound on the nominal monetary policy rate into account make use of four fields in the `DSGEModel` structure, beyond the two fields covered in Section 4.2. First of all, the field `AdditionalDraws` determines the percentage of extra draws of structural shocks, measurement errors, and initial states that are made available to the forward-back shooting algorithm when solving the model subject to the zero lower bound; see Warne (2017, Section 3.4). These extra draws are used when an original draw implies that either these does not exist a solution of the anticipated shocks or the algorithm is run too many times. The field `REqPosition` is needed to extract the monetary policy rule from the set of DSGE model equations, while the field `RtildePosition` determines the location of the monetary policy rate among the set of state variables. Finally, the field `ZLBSampleT` determines the length of the prediction sample over which the zero lower bound may be binding.

---

[13] The kernel density estimators of the prior densities that YADA uses are obtained from the *Kernel Density Estimation Toolbox* by Christian Beardah. The files in YADA come from version 1.3 of the toolbox, but unfortunately only an older version (1.0) of the kernel density estimation toolbox is available for download. Additional documentation on how to use the toolbox is found at the website of the electronic journal Internet Archaeolgy. In particular, see the article The Archaeological Application of Kernel Density Estimates by Christian Beardah and Michael Baxter in issue 1 of this journal.

Parameter scenarios is a tool that can be used to analyse how the data or the estimated shocks are influenced by changes in parameters over a certain sample. The field `ParameterScenario` is a vector which indicates if parameters should be changed (unity) or not (zero), while the alternative values of the parameters that the user has selected are stored in a mat-file on disk. The field `ParameterScenarioValue` is an integer that translates into the start period for the scenario.

Finally, the field `MonteCarloFilterDraws` holds an integer that determines the default number of prior draws to use when applying Monte Carlo filtering; see Ratto (2008) and Section 11.14 in Warne (2017) for details on the topic. Monte Carlo filtering is used to examine which parameters are important for obtaining a unique and convergent solution of the DSGE model. Since the analysis makes use of an asymptotic test (Kolmogorov-Smirnov), the number of draws from the prior should to be sufficiently large to allow for reliable estimates of the cumulative distribution functions of the case of a unique and convergent solution as well as of the case of either indeterminacy or the lack of a stable solution.

4.9. *DSGE-VAR Related Fields in* `DSGEModel`

The fields in the `DSGEModel` structure that are specifically used for estimating and analysing DSGE-VAR models are presented in Table 17. The DSGE-VAR models that YADA support are described in Warne (2017, Section 15); see also Del Negro and Schorfheide (2004, 2006, 2009) and Del Negro, Schorfheide, Smets, and Wouters (2007).

The λ hyperparameter measures the degree of misspecification of the DSGE model. When $\lambda = \infty$, then a VAR approximation of the DSGE model is used, where the VAR parameter values are fully determined by the first and second population moments of the observed variables according to the DSGE model. At the other extreme, $\lambda = 0$, lies the unrestricted VAR model. The `DSGEModel` field `Lambda` holds a vector with the values of λ that should be considered by YADA. It may be noted that YADA requires λT, where T is the sample size of the data, to be an integer. When this product is not an integer, YADA rounds it to an integer using the `ceil` function, i.e, upward rounding.

YADA supports both marginal and joint posterior mode estimation of DSGE-VARs. The former case means that posterior mode estimation is based on the likelihood of the data conditional on the DSGE model parameters, i.e., the likelihood one obtains once the VAR parameters have been integrated out of the product between the likelihood function of the VAR and the prior of the VAR parameters conditional on the DSGE model parameters (see Del Negro and Schorfheide, 2004, equation A.2). The marginal posterior of the DSGE model parameters seen through the VAR is propertional to the product of this marginal likelihood and the prior of the DSGE model parameters. The latter type of posterior mode estimation uses the product between the the likelihood function of the VAR and the joint prior for the VAR and DSGE model parameters (see Warne, 2017, equation 15.35). To allow the user to handle subsets of the λ hyperparameter during posterior mode estimation of DSGE-VARs, the `DSGEModel` field `MarginalLambda` is a vector that holds the positions in `Lambda` of the λ values that the user wishes to consider for marginal posterior mode estimation. The vector `JointLambda` serves the same purpose for the case of joint posterior mode estimation.

The field `DSGEVARShocks` is a vector that determines which of the structural shocks in the DSGE model that should also be used as structural shocks for the DSGE-VAR. First of all, if the number of DSGE model shocks is equal to the number of observed variables, YADA will automatically pick these. Measurement errors are not considered as candidate structural shocks. Second, if the number of structural shocks exceeds that number of observed variables, then YADA will by default pick the n first structural shocks (with n being the number of observed variables). The user can change the selected subset of shocks to use for the DSGE-VAR by running the *Set DSGE-VAR Shocks* callback function on the *DSGE-VAR* menu; see Table 6. Third, if the number of structural shocks is less than the number of observed variables, then YADA will *disable* all DSGE-VAR tools that require structural shocks.

Finally, fields determining the lag order of the DSGE-VAR and possible stationarity requirements for posterior draws are shared with the Bayesian VAR and are therefore discussed in the next section.

4.10. *Bayesian VAR Related Fields in* `DSGEModel`

A number of fields in the `DSGEModel` structure are related to settings for estimating a Bayesian VAR model with a steady-state prior; see Table 19. The BVAR model that YADA supports (see Warne, 2017, Section 14) can be thought of as a generalization of the BVAR developed by Villani (2009).

The mean and the standard deviation of the marginal prior distribution for the steady-state parameters of the VAR are given by an m-file whose full path and name are provided by the field `SteadyStatePriorFile`. The lag order of VAR and DSGE-VAR model can be computed directly from the field `BVARLags`, while the field `StationaryVAR` determines if the posterior draws of these models are required to be consistent with stationarity or not. YADA supports 3 general prior distributions for the parameters on the lagged endogenous variables of the VAR model. The field `PriorType` determines which type of prior should be applied: (1) Minnesota-style prior; (2) normal conditional on covariance matrix; and (3) diffuse.

If `PriorType` is 1, the marginal prior for the parameters on lagged endogenous variables is proper. The mean of the prior distribution is determined by the values implied by the fields `PriorDiffMeanValue` and `PriorLevelMeanValue`. The former can be used to find the mean of parameters on the first own lag for variables that are in first differences, while the latter serves the same purpose for variables that are in levels. The covariance matrix matrix is a standard prior covariance matrix for a Minnesota prior, where the field `OverallTightnessValue` makes it possible to determine the overall tightness hyperparameter. Similarly, `CrossEqTightnessValue` is the field used to find the desired cross equation tightness hyperparameter, while the harmonic lag decay hyperparameter is given through the field `HarmonicLagDecayValue`.

When the field `PriorType` takes the value 2, then the marginal prior for the parameters on lagged endogenous variables is proper if the marginal prior of the residual covariance matrix is proper. This is governed through the field `OmegaPriorType`, which is 1 when the residual covariance matrix has a diffuse marginal prior, and 2 when this matrix has an inverse Wishart marginal prior. In YADA the diffuse prior is given by the determinant of the residual covariance matrix to the power of a constant and is therefore improper, while the inverse Wishart prior is proper.

The normal conditional on the covariance matrix prior has a conditional mean which is determined via the fields `PriorDiffMeanValue` and `PriorLevelMeanValue`. The conditional covariance matrix depends on the fields `OverallTightnessValue` and `HarmonicLagDecayValue`.

If the residual covariance matrix has an inverse Wishart prior (`OmegaPriorType` is 2), YADA needs values for the matrix with location parameter (called A), as well as an integer value for the degrees of freedom parameter. The latter is determined by the value taken by the field `WishartDFValue`, while the former depends on the `WishartType` field. The A matrix is given by the maximum likelihood estimate of the residual covariance matrix when `WishartType` is 1, and by a constant times the identity when `WishartType` is 2. The constant, in turn, is determined by the field `VarianceTightnessValue`.

5. Adding Tools to YADA

In this section we shall discuss how you can add your own tools to YADA. To this end it is assumed that the tools will be available on a new menu and with your own callback functions. To illustrate the callback functions we consider the hypothetical case of spatial distortions. For this tool, we consider the 4 types of parameter values that can be used: initial values, posterior mode values, draws from the prior as well as from the posterior distribution. The final issue that will be discussed is the presentation of the results, both written to a text file and displayed in a graph. All the example functions discussed in this section are available in the `project` directory in YADA.

Before we turn to these matters, we need to consider where all new code will be stored. I recommend that you use a directory directly below the YADA base directory. The name of this directory can, for example, be `project`. To ensure that this directory is found by Matlab while YADA is running, one approach is to prepend it to the Matlab path in the `YADAPath` function. The following line of code in that function will take care of this matter:

```
path([pwd '\project'],path);
```

This line may, e.g., appear directly above the line where the `aim` directory is prepended (temporarily) to the Matlab path. In what follows, I assume that all files related to your own tools are stored in this directory, or in some other directory that appears on the path used by YADA.[14]

5.1. *Menu Controls*

The first step is to decide which label to use for the menu. The label should be just one word and below it will be *Project*. To allow for direct keyboard access to the menu via the `Alt` key the "&" sign is used before the letter `P` (since this letter is free).[15]

Moreover, the location of the menu needs to be selected. A natural place for a new menu is between the BVAR menu and the Help menu. To find this place, open `YADAGUI.m` with a text editor and search for:

```
%
% 7. BVAR menu
%
controls = BVARMenuControls(maingui,controls);
%
% 8. Help Menu
%
controls = HelpMenuControls(maingui,controls);
```

Tt is recommended that you add all your menu related code in a separate m-file, like the menu created through the code in `HelpMenuControls.m`. Let us call this file `ProjectMenu.m` and let us also assume that it takes 2 input variables and provides 1 output. The input variables can be called `maingui` and `controls`, while the output variable is `controls`. The following code is therefore added in `YADAGUI.m` directly *below* the call to the BVAR menu and *above* the call to the Help menu function:

```
if FileExist([pwd '\project\ProjectMenu.m'])==1;
   controls = ProjectMenu(maingui,controls);
end;
```

The code first checks if the file `ProjectMenu.m` exists in the directory `project` directly below the YADA base directory. If the file exists, then the `FileExist` function gives its output variable `status` the value 1; otherwise `status` is set to 0 and the `ProjectMenu` function is not executed.[16]

The input variable `maingui` is the handle to YADA's main dialog window and this window is the so called `Parent` of the Project menu. Since the structure `controls` includes handles to all controls on the YADA window via its fields, it is here assumed that you will extend this structure with handles to your own menu controls.

5.1.1. *Controls and Handles on the Project Menu*

Assuming that you have created the file `ProjectMenu.m` in the `project` directory, the top of this file may include the following lines:

---

[14] YADA does not change the Matlab path permanently. Before it closes the original Matlab path is restorded. I recommend that this principle is also followed when you add your own tools to YADA.

[15] YADA makes use of the letters `F` (File menu), `E` (Edit menu for version 6 or earlier), `V` (View menu), `T` (Tools menu), `A` (Actions menu), `D` (DSGE-VAR menu) `B` (BVAR menu), and `H` (Help menu). To access a menu via the keyboard the `Alt` key should be pressed before the letter.

[16] The `FileExist` function is included in the YADA distribution. You will find it in the `filesystem` directory, directly below YADA's base directory.

```
function controls = ProjectMenu(maingui,controls)
%
% handle to the Project menu
%
controls.projectmenu = uimenu('Parent',maingui, ...
   'Label','&Project', ...
   'Tag','ProjectMenu');
%
% end of ProjectMenu.m
%
```

The first line defines the file as a Matlab function whose name is ProjectMenu. The input variables are specified within parentheses after the function name, while the output variable is given before the equality sign.[17]

The field projectmenu is not used by YADA for the controls structure and may therefore be taken by your own menu. With these changes you will see that the Project label has been added to the menubar in YADA; see, e.g., Figure 10 below.

The next step is to add items to the Project menu. In what follows we shall consider a project tool that we call "Spatial Distortions". How these entities are related to the data or the DSGE model need not concern us here. For convenience, however, we shall simply think of them as a function of the observed variables and the parameters of the DSGE model.

To add the spatial distortion tool to the Project menu we include the following code below the handle to the menu in ProjectMenu.m:

```
%
% handle to the spatial distortions tool on the project menu
%
controls.project.spatdist.parent = ...
   uimenu('Parent',controls.projectmenu, ...
      'Label','&Spatial Distortions', ...
      'Tag','SpatialDistortionsMenuParent');
```

The label of the controls is Spatial Distortions, where the line under the S indicates that this is the key for accessing the menu item through the Alt+P combination, i.e., Alt+P+S selects the Spatial Distortions tool from the Project menu. Since the Parent property is given by the handle to the Project menu, the control appears on this menu.

Let us assume that we can compute this tool for the 4 sets of parameter values that YADA supports. For each set of parameter values we add a handle to the Spatial Distortions menu item. The following code takes care of this:

```
%
% handle to initial parameter values for spatial distortions tool
%
controls.project.spatdist.initialvalues = ...
   uimenu('Parent',controls.project.spatdist.parent, ...
      'Label','&Initial Values...', ...
      'Enable','off', ...
      'Callback','ProjectFunctions spatdist_init_values', ...
      'Tag','SpatialDistortionsMenuInitialValue');
%
% handle to prior distribution for spatial distortions tool
%
controls.project.spatdist.prior = ...
   uimenu('Parent',controls.project.spatdist.parent, ...
```

[17] If more than one output variable is supplied by a function, they should appear within brackets before the equality sign and each output variable is separated from the next by a comma, e.g., [a,b,c] for the 3 output variables a, b, and c.

FIGURE 10. The Project menu.

```
         'Label','P&rior Distribution...', ...
         'Enable','off', ...
         'Callback','ProjectFunctions spatdist_prior_dist', ...
         'Tag','SpatialDistortionsMenuPrior');
%
% handle to posterior mode values for spatial distortions tool
%
controls.project.spatdist.postmode = ...
   uimenu('Parent',controls.project.spatdist.parent, ...
         'Label','&Posterior Mode...', ...
         'Enable','off', ...
         'Separator','on', ...
         'Callback','ProjectFunctions spatdist_post_mode', ...
         'Tag','SpatialDistortionsMenuPosteriorMode');
%
% handle to posterior distribution for spatial distortions tool
%
controls.project.spatdist.posterior = ...
   uimenu('Parent',controls.project.spatdist.parent, ...
         'Label','Posterior &Distribution...', ...
         'Enable','off', ...
         'Callback','ProjectFunctions spatdist_post_dist', ...
         'Tag','SpatialDistortionsMenuPosterior');
```

The 4 controls above are all children of the spatial distribution control through the `Parent` property. The controls will appear in the same order as they are read by Matlab, i.e., the Initial Values control first, followed by the Prior Distribution control, the Posterior Mode control and, finally, the Posterior Distribution control. Notice that the posterior mode control has the property `Separator` set to `'on'`. This means that a separator line will be displayed on the menu above the posterior mode control.

The 4 controls all have the `Enable` property set to `'off'`. On the spatial distortions submenu they will therefore be disabled or, in common geek, "greyed out". YADA uses the convention that menu items that can execute a `Callback` routine are disabled unless that function can be computed. For example, the Spatial Distortions tool that uses the posterior mode of the estimated parameters is disabled unless the posterior mode has been computed and can be located by YADA. This issue is covered in Section 5.1.2.

Notice that the above controls on the Spatial Distortions submenu all have a specified value for the `Callback` property. The routine executed when the user clicks on a menu item is a Matlab function or command expressed as a string. For example, the string

```
ProjectFunctions spatdist_post_mode
```

calls the `ProjectFunctions` function with its only required input variable set to the value `spatdist_post_mode`. This function and its input variable are discussed in Section 5.2. For

now we notice that a unique value is used for each set of parameter values. The values of the input variable have been selected so that one can identify the tool and the selected parameter value. For all we know, the function `ProjectFunctions` can cover other tools beside spatial distortions. For instance, it may also deal with complex tools such as temporal or interphasic rifts, or even spatial flexures.

5.1.2. *Determining the* `Enable` *Property*

Setting the `Enable` property of a control to `'on'` (enabled) or `'off'` (disabled) is achieved by applying the `set` command for the handle to the control. Hence, the problem is really when a control should be enabled or disabled.

First, however, we need to decide where the code that takes care of this issue should be executed. YADA has a large chunk of code at the end of `YADAGUI.m` that tests which controls to enable and to disable. While it is possible to make use of these features, I would recommend to add the tests to a separate Matlab function. The main reason for this recommendation is that YADA is developing with new features being added (and perhaps some old features being removed).

Let us therefore assume that the function `VerifyProjectMenu.m` exists in the `project` directory. Its sole purpose is to test if controls (menu items) on this menu should be enabled or disabled. This function can be assumed to take the structures `DSGEModel` and `controls` for input and returns no output variables. This means that the following line needs to be added at a suitable place in `YADAGUI.m`:

```
VerifyProjectMenu(DSGEModel,controls);
```

A suitable place is, for instance, directly below the call to `YADAEnableControls` (Section 2.2.2), i.e., below the code:

```
YADAEnableControls(DSGEModel,controls);
```

The call to `VerifyProjectMenu` should be made every time a new DSGE model is loaded into YADA or new parameter values are available. This means that `VerifyProjectMenu` needs to be executed by the `FileMenuFunctions` file and one suitable place is directly below the call to `YADAEnableControls`. In addition, the `ActionsMenuFunctions` file in the `menus` directory can also affect the parameter types that are available and it is therefore recommended that `VerifyProjectMenu` is called when callback functions from the *Actions* menu are executed. Furthermore, it is adviced that the file containing all callback functions from the project function, `ProjectFunctions` also calls the `VerifyProjectMenu` function; see Section 5.2. As an alternative to adding `VerifyProjectMenu` calls to all these files, it is possible to simple call the function at the end of `YADAEnableControls`, located in the `gui` directory. This ensures that the controls on the *Project* menu are enabled and disabled just like the controls on YADA's own menus and is therefore the recommended approach.

For the Initial Values menu item to be enabled it is required that the mode of the posterior distribution can (potentially) be estimated. The function `VerifyPosteriorModeEstimation`, located in the `logic` subdirectory, performs this task. It takes the `DSGEModel` structure for input and returns a value of 0 or 1 for its output variable `status`.[18] A value of 1 is returned if posterior mode estimation may be attempted and 0 otherwise. This means that we can write the following code in the `VerifyProjectMenu` function:

---

[18] This function checks that (i) the AiM parser has been run successfully; (ii) the data construction file exists; (iii) the measurement equation file exists; and (iv) the state shocks and state variables have been selected from the set of potential variables through the AiM data. To test if the AiM parser was successfully executed, the function looks for the existence of the AiM data file through the string `DSGEModel.AIMDataFile`; see Table 11. In addition, the function checks if the AiM model file (`DSGEModel.AIMFile`; see Table 9) and the file `compute_aim_matrices.m` exist. Notice that `VerifyPosteriorModeEstimation` does not check if the prior distribution specification file exists or not. The explanation is that YADA can read all the prior distribution data directly from that file when the posterior mode has been estimated. On the other hand, if neither the prior distribution specification file nor the posterior mode results file exists, then the initial values are unknown and, hence, the function will not be able to deliver the results for the spatial distortions.

```
if VerifyPosteriorModeEstimation(DSGEModel)==1;
   set(controls.project.spatdist.initialvalues,'Enable','on');
else;
   set(controls.project.spatdist.initialvalues,'Enable','off');
end;
```

to determine if the Initial Values menu item for Spatial Distortions should be enabled or disabled.

Next, the Prior Distribution menu item can be enabled when the user has already computed draws from the prior distribution via the *Prior Sampling* tool on the toolbar or on the Actions menu (see Table 5). The following code in `VerifyProjectMenu` tests if these data exist and performs the correct action based on its finding:

```
PriorDrawsFile = FixFilePath([DSGEModel.OutputDirectory ...
   '\priordraws\PriorDraws-' DSGEModel.NameOfModel '.mat']);
if FileExist(PriorDrawsFile)==1;
   set(controls.project.spatdist.prior,'Enable','on');
else;
   set(controls.project.spatdist.prior,'Enable','off');
end;
```

The output directory for the file with the draws from the prior distribution is given by the subdirectory `priordraws` of the output directory for the current model; see Table 9. The name of the file then depends on the `NameOfModel` string. The default behavior in YADA is to let the output directory be a subdirectory to the directory where the AiM model file is located. The name of that subdirectory is equal to the string `DSGEModel.NameOfModel`. Finally, the function `FixFilePath` creates an operating consistent file path.

To test if the posterior mode has already been estimated, YADA checks if the file with the posterior mode results exists on disk or not. This file is located in the subdirectory `mode` of the output directory for the current model. This means that the code can be formulated as:

```
ModeFile = FixFilePath([DSGEModel.OutputDirectory ...
   '\mode\PosteriorMode-' DSGEModel.NameOfModel '.mat']);
if FileExist(ModeFile)==1;
   set(controls.project.spatdist.postmode,'Enable','on');
else;
   set(controls.project.spatdist.postmode,'Enable','off');
end;
```

It may be noted that the existence of the file with posterior mode results is a necessary condition for being able to use these parameter values. But whenever the callback routine for posterior mode parameters values is executed, it needs to verify that the relevant information is actually availble from that file. YADA does not perform this task when it decides if a control should be enabled or disabled, but leaves it for later.

The final control on the Project menu for the Spatial Distortions tool that we should test is the one concerning draws from the posterior distribution. For this objective the YADA function `VerifyDSGEPosteriorDraws`, located in the subdirectory `data` to YADA's base directory, can be used. This function takes 3 input variables: (i) the `DSGEModel` structure, (ii) a valid value for the number of parallel Markov chains (`CurrentChain`), and optionally (iii) the structure `controls`. If the last input is not provided, the function attempts to determine the value of this variable by using the trick from Section 2.2.1. For the second input variable the value 1 should always be possible and, hence, we can express the test as:

```
if VerifyDSGEPosteriorDraws(DSGEModel,1,controls)==1;
   set(controls.project.spatdist.posterior,'Enable','on');
else;
   set(controls.project.spatdist.posterior,'Enable','off');
end;
```

These examples serve as necessary conditions when testing if a certain control that uses parameter values should be enabled or not. Some tools in YADA rely on observed variables being expressed in levels, in simple annualization form (by adding quarters or months), or through the transformation functions in the field `YTransformation` of the structure `DSGEModel`; see Table 10. The test for computing levels is to compare the sum of `DSGEModel.levels` to `DSGEModel.n`. If the formar is smaller than the latter, then some variables should be viewed as first difference variables and levels effects for some tools, such as impulse responses, may be determined through accumulation and it therefore makes sense to enable a tool that deals with levels data. The case of annualization can be tested through `DSGEModel.annual`. If the maximum of this vector is greater than unity, then at least one obsereved variable can be annualized. Finally, the transformation case can be tested by checking if the `YTransformation` field is empty or not.[19]

5.2. *A General Callback Function*

The four controls for the spatial distortion tool that were introduced in the Section 5.1.1 share a common callback routine called `ProjectFunctions`. This function accepts at least one input variable, a string, that we may locally call `selector`. Following the practise used above, let us assume that `ProjectFunctions.m` is located in the `project` directory.

Since each call routine has different value for the `selector` input variable, we need to make sure that all these values are supported by the function. Since `ProjectFunctions` should not provide any output variables, the function can at first be as follows:

```
function ProjectFunctions(selector)
%
% collect some important data structures
%
maingui = findobj('Type','figure','Tag','YADA');
controls = get(maingui,'UserData');
CurrINI = get(controls.filemenu,'UserData');
DSGEModel = get(controls.open,'UserData');
lasterr = '';
%
switch selector
   case 'spatdist_init_values'
      %
      % initial parameter values
      %
   case 'spatdist_prior_dist'
      %
      % prior distribution values
      %
   case 'spatdist_post_mode'
      %
      % posterior mode values
      %
   case 'spatdist_post_dist'
      %
      % posterior distribution values
```

[19] If an additional test of data transformations is desired, the function `VerifyDataTransformation`, located in the subdirectory `data` of YADA's base directory, can be applied. It accepts 3 input variables: (i) the `VariableNames` string matrix (which could be any matrix with names of variables that should be transformed); (ii) the `Transformation` structure (e.g., `YTransformation`); and (iii) the `CheckAll` boolean variable. The last input variable implies that the function tests if all required fiels are available when it is unity, and only a subset when it is zero; see the internal documentation of this function for details. The function gives one output variable called `status`: it is unity if the test accepts the transformation data, and zero otherwise.

```
        %
    end
    %
    set(controls.open,'UserData',DSGEModel);
    set(controls.filemenu,'UserData',CurrINI);
    %
    % make sure that controls are enabled and disabled as they should
    %
    YADAEnableControls(DSGEModel,controls);
    VerifyProjectMenu(DSGEModel,controls);
    %
    % end of ProjectFunctions.m
    %
```

The first step is to collect the YADA structures that may be useful for various computations, i.e., `DSGEModel`, `CurrINI`, and `controls`. Next, the `switch` command is used with its `case` calls that depend on the value of the input variable `selector`. Finally, if any changes to the `DSGEModel` and `CurrINI` structure have been made, we store the revised value in the `UserData` property of the controls where YADA expects to find them.

Before we move on to using different parameter values for the callback routines, it is worthwhile to mention another convention in YADA. Namely, while one tool is running YADA prevents others from being executed. The following code, for each `case` call above, makes sure that a tool can only be executed when, so to speak, YADA is not doing anything else.

```
if get(controls.about,'UserData')==0;
   set(controls.about,'UserData',1);
   drawnow;
   %
   % do many computations and then reset
   %
   set(controls.about,'UserData',0);
end;
```

The `about` field of the `controls` structure gives the handle to the *About* button on the toolbar of YADA's main dialog window. Whenever the value of its `UserData` property is zero, then YADA is not doing anything else, while a value of unity means that it is busy.

5.3. *Using Parameter Values*

To solve the DSGE model it is necessary to have some values for the model parameters. As mentioned in Section 2.2.1, YADA supports four categories of parameters values: initial values, posterior mode values, prior and posterior draws. Below I will show how such values can be accessed through Matlab code.

5.3.1. *Initial Parameter Values*

The initial parameter values are determined by the prior distribution specification file and, if present, the file with parameters to initialize. The former is given by `DSGEModel.PriorFile` and the latter by `DSGEModel.InitializeParameterFile`; see Table 9. If the posterior mode has been estimated, the mat-file with these estimates also contains the initial values of the parameters that can be estimated. In fact, it is possible to retrieve all required information about initial parameter values from the posterior mode results file. The code presented below will take this into account.

The convention in YADA is to check if the posterior mode results exists on disk also when running a tool for the initial parameter values. When these results do not exist on disk, then YADA asks the user if the tool should be run before the mode has been estimated. This is the only form of "nagging" that YADA supports and is used mainly to remind the user that the initial

values may not provide good guidance about the model's behavior. The following code provides a simple test of "to nag or not to nag" as well as taking the user's choice into account:

```
ModeFile = FixFilePath([DSGEModel.OutputDirectory ...
   '\mode\PosteriorMode-' DSGEModel.NameOfModel '.mat']);
if FileExist(ModeFile)==1;
   answer = 'Yes';
else;
   txt = ['Are you sure you want to compute spatial distortions before' ...
          ' running the posterior mode estimation routine?'];
   answer = Query(txt,'question',150, ...
            'Spatial Distortions - Initial Values',500,'no',CurrINI);
end;
if strcmp(lower(answer),'yes')==1;
   %
   % user is willing to run spatial distortions
   %
end;
```

Provided that the posterior mode file does not exist on disk, a `Query` dialog is displayed with the question in the string `txt`, with a Yes and a No button (where the No button has focus when the code is run under Matlab version 7 or later). The width of the dialog is 500 pixels, while the maximum height is 150 pixels; see Section 3.1 for details on the `Query` dialog function.

Provided that the `answer` variable is `'Yes'`, the next step is to check the status for the initial parameter values. The function `InitializeDSGEModelSimulation`, located in the directory `data` below YADA's base directory, is used for this purpose and it takes the structures `DSGEModel` and `CurrINI` for input. The following code presents an example where the minimum number of output variables is accepted:

```
[InitStatus,theta,thetaPositions,ModelParameters] = ...
               InitializeDSGEModelSimulation(DSGEModel,CurrINI);
```

The `InitStatus` variable is unity if all parameter related input files are specified correctly, and zero otherwise. The input files are specified correctly when (i) the prior distribution specification file is valid; (ii) AiM has been successfully parsed and the DSGE model has a unique convergent solution at the initial values; (iii) the measurement equation file can be executed without error; and (iv) the optional files with parameters to initialize and to update can be executed without error.

The function `VerifyPriorData` deals with the first issue and is discussed in some detail in Warne (2017, Section 7.4.1). The second part of the test is examined by trying to load the file `DSGEModel.AIMDataFile` and by either checking if either the function `AiMSolver` returns returns a proper `mcode` value and the corresponding matrices can be properly rewritten as a state equation via `AiMToStateSpace`, or if the function `KleinSolver` or `SimsSolver` provides a proper `mcode` value; see Section 3.4.2 until 3.4.5 of Warne (2017). Which DSGE model solver test is conducted depends on the value of the field `ModelSolver` in the DSGEModel structure; see Table 11.

The `InitializeDSGEModelSimulation` function provides 4 required and 6 optional output variables. Apart from the boolean variable `InitStatus` the other required outputs are `theta` (a vector with the initial values for the parameters to be estimated), `thetaPositions` (a vector structure of the same dimension as `theta` and where `thetaPositions(i).parameter` gives the name of the parameter in the `i`:th position of `theta`), and `ModelParameters` (a structure whose fields have names equal to the parameters of the DSGE model).

Provided that `InitStatus` is unity we can safely use the initial parameter values to solve the DSGE model and use it for the spatial distortions tool. However, if this boolean variable is zero simply because the prior distribution specification file is missing it is still possible to run the tool provided that we can collect all the information about the prior distribution

from the posterior mode file. Accordingly, the following code can be added below the call to InitializeDSGEModelSimulation:

```
if (InitStatus==0)&(FileExist(DSGEModel.PriorFile)==0);
   if FileExist(ModeFile)==1;
      ErrorStr = '';
      try;
         ModeData = load(ModeFile);
      catch;
         ErrorStr = ['Unable to load the file "' ModeFile ...
                     '". Message caught is: ' lasterr];
      end;
      if isempty(ErrorStr)==1;
         theta = ModeData.theta;
         thetaPositions = ModeData.thetaPositions;
         ModelParameters = ModeData.ModelParameters;
         ModelParameters = ThetaToModelParameters(ModelParameters, ...
                                                 theta,thetaPositions);
         InitStatus = 1;
      else;
         About(ErrorStr,'error', ...
               'Spatial Distortions - Initial Values', ...
               200,500,CurrINI);
      end;
   end;
end;
if InitStatus==1;
   %
   % we can now try to compute the spatial distortions
   %
end;
```

The code first checks if the reason for the zero value of InitStatus is a missing prior file, and if so tests if the posterior mode file exists on disk. Given that this file indeed exists where it should, the code attempts to load the mat-file data using a try-catch to allow it to capture an error message if it is not successful. Given that the data was loaded without error, the needed 4 variables theta until InitStatus can be determined; otherwise an error message with an error icon is displayed in an About dialog; see Section 3.1 for details.

Notice also that the function ThetaToModelParameters is called. This ensures that the ModelParameters structure uses the values in theta for the parameters that can be estimated. If the function SolveDSGEModel is eventually used to solve the DSGE model, the call to the former function is actually superfluous since it is performed by the latter.

The three variables theta, thetaPositions and ModelParameters are needed when YADA solves the DSGE model. As already mentioned, the function InitializeDSGEModelSimulation can also provide values for 6 additional variables. These variables can also be retrieved from the VerifyPriorData function and are locally called:

thetaDist: a vector of the same dimension as theta which indicates via integer values the prior distribution of a parameter.

PriorDist: a structure with fields that have names corresponding to the selected prior distributions of the parameters that can be estimated. That is, the field names are beta, gamma, normal, invgamma, truncnormal, uniform, student, cauchy, logistic, gumbel, and pareto. Each field is a matrix with the parameters of the prior. For example, PriorDist.gamma is a matrix with 3 columns with the mean, standard deviation and lower bound of the gamma prior in the columns provided that at least one parameter is assumed to have a gamma prior, and is otherwise an empty matrix.

`ParameterNames`: a structure with field names `all`, `calibrated`, `beta`, `gamma`, `normal`, `invgamma`, `truncnormal`, `uniform`, `student`, `cauchy`, `logistic`, `gumbel`, `pareto`, and `estimated`. Each one of these fields provides a string matrix where the rows give names of parameters as they have been detected by YADA. For example, the string matrix `ParameterNames.estimated` is created by YADA such that the `i`:th row gives the name of the `i`:th element of `theta`.

`thetaIndex`: a vector of the same length as `theta` that indicates the type of transformation that should be applied to a parameter to ensure that its support is the real line. The value 0 means that no transformation is necessary (normal, student-*t*, Cauchy, logistic, or Gumbel prior); the value 1 means that the natural logarithm transformation should be applied to a parameter (gamma, inverse gamma, left truncated normal, or Pareto prior); while 2 (beta prior) and 3 (uniform prior) indicate logit transformations should be applied to the same element of `theta`.

`UniformBounds`: a matrix with 2 columns and number of rows equal to the length of `theta`. The row values gives the lower and the upper bound for parameters that are transformed with a logit function.

`LowerBound`: a vector of the same length as `theta` where lower bounds are given for parameters that are transformed with the natural logarithm.

It is also possible to load the 6 variables from the file with the posterior mode results, where they have the same names, e.g., we can let `PriorDist` be given by `ModeData.PriorDist`. For details about these additional variables, see Warne (2017, Section 7.4.1) and the internal documentation of `VerifyPriorData` or `InitializeDSGEModelSimulation`.

5.3.2. *Posterior Mode Values*

The discussion in the previous section has already informed us about how data from the posterior mode file can be retrieved. Specifically, the following code attempts to retrieve the variables `theta`, `thetaPositions`, and `ModelParameters`:

```
ModeFile = FixFilePath([DSGEModel.OutputDirectory ...
   '\mode\PosteriorMode-' DSGEModel.NameOfModel '.mat']);
ErrorStr = '';
try;
   ModeData = load(ModeFile);
catch;
   ErrorStr = ['Unable to load the file "' ModeFile ...
               '". Message caught is: ' lasterr];
end;
if strcmp(ErrorStr,'')==1;
   theta = ModeData.thetaMode;
   thetaPositions = ModeData.thetaPositions;
   ModelParameters = ModeData.ModelParameters;
   %
   % we can now try to compute spatial distortions
   %
else;
   About(ErrorStr,'error', ...
         'Spatial Distortions - Posterior Mode', ...
         200,500,CurrINI);
end;
```

Notice that there is no need to call `ThetaToModelParameters` here since the `ModelParameters` structure was last updated with the posterior mode values. Moreover, if some of the 6 variables `thetaDist`, `PriorDist`, `ParameterNames`, `thetaIndex`, `UniformBounds`, or `LowerBound` are needed by some calculation of the tool, then these can be loaded via the `ModeData` structure.

Apart from loading previously computed draws from the prior distribution, it is necessary to collect certain variables that are needed when we wish to solve the DSGE model. The convention in YADA is to check from which sources the prior distribution information can be read. In the event that both the prior distribution specification file and the posterior mode results file exist on disk, YADA will ask the user which of these sources should be used. If only one exists, then YADA attempts to retrieve the prior information from that source.

Assuming that we only need the data in `theta`, `thetaPositions`, and `ModelParameters`, the following code deals with these matters:

```
ModeFile = FixFilePath([DSGEModel.OutputDirectory ...
   '\mode\PosteriorMode-' DSGEModel.NameOfModel '.mat']);
if FileExist(ModeFile)==1;
   ErrorStr = '';
   try;
      ModeData = load(ModeFile);
   catch;
      ErrorStr = ['Unable to load the file "' ModeFile ...
                  '". Message caught is: ' lasterr];
   end;
   if strcmp(ErrorStr,'')==1;
      if FileExist(DSGEModel.PriorFile)==1;
         txt = ['Would you like to use the prior information ' ...
                'stored in the posterior mode results file "' ...
                GetFilename(ModeFile) '"? If you answer ''No'', ' ...
                'YADA will reread the prior file "' ...
                GetFilename(DSGEModel.PriorFile) '".'];
         answer = Query(txt,'question',200, ...
                  'Spatial Distortions - Prior',500,'no',CurrINI);
      else;
         answer = 'Yes';
      end;
      if strcmp(lower(answer),'yes')==1;
         InitStatus = 1;
         theta = ModeData.theta;
         thetaPositions = ModeData.thetaPositions;
         ModelParameters = ModeData.ModelParameters;
      else;
         [InitStatus,theta,thetaPositions,ModelParameters] = ...
                     InitializeDSGEModelSimulation(DSGEModel,CurrINI);
      end;
   else;
      InitStatus = 0;
      About(ErrorStr,'error', ...
            'Spatial Distortions - Prior', ...
            200,500,CurrINI);
   end;
else;
   [InitStatus,theta,thetaPositions,ModelParameters] = ...
               InitializeDSGEModelSimulation(DSGEModel,CurrINI);
end;
```

It may be noticed that the code does not call `ThetaToModelParameters`. The reason is that the `ModelParameters` structure needs to be updated for each draw of the original parameters

(θ) from the prior distribution and, hence, that step should be taken care of by the function that computes the spatial distortions.

The next step is to load the draws from the prior distribution. The following code deals with this matter:

```
if InitStatus==1;
   PriorDrawsFile = FixFilePath([DSGEModel.OutputDirectory ...
      '\priordraws\PriorDraws-' DSGEModel.NameOfModel '.mat']);
   PriorData = load(PriorDrawsFile);
   thetaDraws = PriorData.thetaDraws;
   NumPriorDraws = size(thetaDraws,2);
end;
```

The matrix with draws from the prior distribution, `thetaDraws`, has as many rows as the length of `theta`, while the number of columns is equal to the number of draws from the prior. The ordering of the parameters in `thetaDraws` is the same as the ordering of the parameters in `theta`.

5.3.4. *Posterior Distribution*

When draws from the posterior distribution are used by a tool, the typical situation in YADA is that only a subset of all post burn-in sample draws are used. Not only does this speed up the computation, but the loss in numerical precision is often small when using, say, every hundreth draw of the 500,000 draws that may be available. Naturally, the loss of numerical precision depends on what we are interested in. If the key aspect of the exercise is a tail-event, then more draws may be needed.

It is often the case that YADA needs to know the names of the estimated parameters when it is computing a tool. For example, the `ModelParameters` structure can only be updated with new values of the estimated parameters if the fields of this structure that represent the estimated parameters can be properly linked to the elements of the vector of values for the estimated parameters. The following code tries to collect 4 variables from the posterior mode results file:

```
ModeFile = FixFilePath([DSGEModel.OutputDirectory ...
   '\mode\PosteriorMode-' DSGEModel.NameOfModel '.mat']);
ErrorStr = '';
try;
   ModeData = load(ModeFile);
catch;
   ErrorStr = ['Unable to load the file "' ModeFile ...
               '". Message caught is: ' lasterr];
end;
if strcmp(ErrorStr,'')==1;
   theta = ModeData.thetaMode;
   thetaPositions = ModeData.thetaPositions;
   ModelParameters = ModeData.ModelParameters;
   ParameterNames = ModeData.ParameterNames.estimated;
   %
   % prepare for using draws from posterior
   %
else;
   About(ErrorStr,'error', ...
         'Spatial Distortions - Posterior', ...
         200,500,CurrINI);
end;
```

Apart from the usual variables `theta`, `thetaPositions` and `ModelParameters`, we here also collect a string matrix with the names of all parameters that can be estimated. These names

can be used to allow the user to consider the impact of parameter uncertainty based on a subset of these parameters.

The next step is to check if the user has multiple Markov chains to choose from. The convention in YADA is to let the user select which chain to make use of when multiple chains are available. The code below checks how many parallel Markov chains the user has selected in the *Posterior sampling* frame on the *Options* tab. The DSGEModel field of interest is ParallelChainsValue (see Table 15).

```
ChainsStr = get(controls.posterior.chains,'String');
NumChains = str2num(StringTrim(ChainsStr( ...
                 DSGEModel.ParallelChainsValue,:)));
if NumChains>1;
   ChainsStr = '';
   for i=1:NumChains;
      ChainsStr = strvcat(ChainsStr,['Posterior chain number ' ...
                          int2str(i)]);
   end;
   [action,CurrChain] = SelectionDlg(ChainsStr,1, ...
        'Select the posterior draws chain to load', ...
        'MCMC Chain Selection','Spatial Distortions', ...
        '','',CurrINI);
else;
   CurrChain = 1;
   action = 'ok';
end;
```

The String property of the control controls.posterior.chains holds a string matrix with the possible number of Markov chains the user can select between. The selected number of chains is determined in the next line. If the number of chains (NumChains) is equal to 1, then the only possible chain is selected by default. When there are more than one chain, a selection dialog is presented; see, e.g., Section 3.4 for details on SelectionDlg. The selected chain is given by the variable CurrChain, whose default value is 1.

Since the selection dialog has both an OK and a cancel button, the user can actually quit the tool at this stage. This following code allows for this behavior by testing which button the user clicked on:

```
if strcmp(lower(action),'cancel')==0;
   %
   % check if the posterior draws exist
   %
   if VerifyDSGEPosteriorDraws(DSGEModel,CurrChain,controls)==1;
      %
      % load the data
      %
   end;
end;
```

The function VerifyDSGEPosteriorDraws, located in the data directory, is called with the input variables DSGEModel, CurrChain and controls when the user has clicked on the OK button of the selection dialog. This function checks if the posterior draws for the current user settings[20] exist on disk in the directory rwm, one level below the base output directory DSGEModel.OutputDirectory: the function returns unity for its boolean output variable status when all the files with the posterior draws exist on disk, and zero if at least one such file is missing.

[20] The current user settings for posterior sampling matters are discussed in Section 4.7.

Given that the files with the posterior draws indeed exist, the next step is to load the data from these file. The code below performs this action:

```
[PostSample,thetaPostSample,LogPost,AcceptedDraws,NumBurnin,Weights] = ...
              LoadDSGEPosteriorDraws(DSGEModel,CurrINI,CurrChain);
NumDraws = length(LogPost);
clear('PostSample','LogPost','AcceptedDraws');
thetaPostSample = thetaPostSample(NumBurnin+1:NumDraws,:);
NumDraws = NumDraws-NumBurnin;
drawnow;
```

The function `LoadDSGEPosteriorDraws`, located in the `data` directory, collects data for the selected Markov chain and provides this in terms of 6 output variables. The matrix `PostSample` contains the draws for the transformed parameters, while `thetaPostSample` has the draws of the original parameters. The number of rows of these matrices is equal to the total number of posterior draws, while the number of columns is equal to the length of `theta`. The vector `LogPost` has the value of the log posterior for each draw from the posterior, while `AcceptedDraws` is a vector that keeps track of the number of accepted draws at each stage in the chain. The integer `NumBurnin` gives the number of burn-in draws that should be discarded from the posterior. Finally, the vector `Weights` is either empty or contains the normalized weights from the SMC with likelihood tempering posterior sampler. It is assumed below that only the `theta` draws are needed and, hence, the other potentially big matrices are cleared from memory.

The last step is the selection of a subsample of the post burn-in sample posterior draws. This step only applies to the MCMC posterior samplers and *not* to SMC samplers, i.e., it is only used when `DSGEModel.PosteriorSampler` is less than or equal to 7.

There are two approaches for determining the size of the subsample in YADA. The first is used when data are simulated from the model and the second in all other situations. If we assume that spatial distortions should be performed for simulated data the `DSGEModel` field `PostDrawsUsageValue` is used to determine the subsample size; see Table 13. The following code can then be used:

```
PostDrawsUsageStr = get(controls.posterior.postdrawsusage,'String');
NumPostDrawsUsage = min(str2num(strrep(StringTrim(PostDrawsUsageStr( ...
                        DSGEModel.PostDrawsUsageValue,:)), ...
                        ',','')),NumDraws);
%
% get the draws to use
%
if DSGEModel.RandomizeDraws==0;
   DrawFreq = round(NumDraws/NumPostDrawsUsage);
   SelectDraws = (1:DrawFreq:NumDraws);
   if length(SelectDraws)+1==NumPostDrawsUsage;
      SelectDraws = [SelectDraws NumDraws];
   elseif length(SelectDraws)>NumPostDrawsUsage;
      SelectDraws = SelectDraws(1:NumPostDrawsUsage);
   end;
   thetaPostSample = thetaPostSample(SelectDraws,:);
else;
   if DSGEModel.RandomNumberValue==1;
      rand('state',0);
   else;
      rand('state',sum(100*clock));
   end;
   thetaPostSample = thetaPostSample(ceil(rand(NumPostDrawsUsage,1) ...
```

```
                                          *NumDraws),:);
   end;
```

The first line simply collects the string matrix with the maximum number of draws that the user can choose among. The second line extracts the value that the user has selected and sets the size of the subsample to the minimum of this value and the total number of available post burn-in sample draws.

Given the size of the subsample, the code thereafter checks if the draws in the subsample should be picked randomly or not. If the DSGEModel field RandomizeDraws is zero, the selected draws are separated by a common distance (DrawFreq) and start from the first post burn-in draw. In contrast, if the selected draws should be picked randomly, the code generates integer values between unity and the number of post burn-in draws. The uniform random number generator is here first initialized using the user preference regarding having a fixed or a variable state; see Table 15.

If the second method is used, then the DSGEModel field PostDrawsPercentValue determines the size of the subsample. In this case a percentage value of the post burn-in sample is considered. The code can now be expressed as:

```
UseDraws = KeepPosteriorDraws(DSGEModel, ...
               controls.posterior.usepostdraws,NumDraws);
if length(UseDraws)<NumDraws;
   thetaPostSample = thetaPostSample(UseDraws,:);
end;
```

The function KeepPosteriorDraws, located in the data directory, provides the positions of the draws that should be used. This function picks the draw positions in the same way as the code for the first method, except that the size of the subsample is determined by the value of PostDrawsPercentValue and if the user has opted for 100 percent, then UseDraws is simply the sequence 1, 2 until NumDraws.

5.4. *Tool Computations: Spatial Distortions*

For a tool such as spatial distortions, YADA uses the convention that separate Matlab functions are written for single and for multiple parameter values. An important reason is that it is convenient to store the results for multiple values to disk for future reuse, while results for a single parameter value can typically be computed very quickly and, thus, do not need to be stored in that manner. I will therefore first discuss the single parameter value case (initial values or posterior mode values) before moving to the multiple parameter values (prior or posterior draws).

5.4.1. *Single Parameter Value*

The function for computing spatial distortions for a single parameter value is assumed to be located in the project directory and to be called DSGESpatialDistortionsTheta. The naming convention used for tools is that the name clearly indicates the type of tool that the function computes, prepended by DSGE or BVAR depending on which type of model they can be used by, and appended by Theta when single parameter values are considered. For simplicity we shall let this function accept a minimum number of input variables, given by theta, thetaPositions, ModelParameters, DSGEModel, and CurrINI. Furthermore, the minimum number of output variables is also assumed and we shall let them be called SpatDist, a required output with the results on spatial distortions, and the optional outputs status and kalmanstatus. The optional output variables are only used when theta has the initial parameter values. In that case, status reports if the DSGE model has a unique convergent solution or not, while kalmanstatus indicates if the Kalman filter can be executed successfully.

The top of the spatial distortions function is therefore written:

```
function [SpatDist,status,kalmanstatus] = ...
   DSGESpatialDistortionsTheta(theta,thetaPositions, ...
      ModelParameters,DSGEModel,CurrINI)
```

```
% DSGESpatialDistortionsTheta: Computes spatial distortions
%                              for a single value of theta
%
% USAGE:
.
.
.
%
% initialize output
%
SpatDist = [];
if nargout==1;
   %
   % copy files to the tmp directory for posterior mode values
   %
   [stat,msg] = CopyFile([GetPath(DSGEModel.AIMDataFile) ...
                      'compute_aim_matrices.m'],[pwd '\tmp']);
   [stat,msg] = CopyFile(DSGEModel.MeasurementEquationFile, ...
                      [pwd '\tmp']);
   if FileExist(DSGEModel.UpdateParameterFile)==1;
      [stat,msg] = CopyFile(DSGEModel.UpdateParameterFile, ...
                      [pwd '\tmp']);
   end;
else;
   status = 1;
   kalmanstatus = 0;
end;
```

Particular care is taken in YADA to ensure that functions are individually documented, using a certain style, and with a great level of detail. This not only facilitates future work on the function, but also helps other users to see what a given function needs, does and provides.

The output variables are initialized with default values before the actual tool code is executed. The required output variable, `SpatDist`, is empty and therefore makes it easy for code that calls the function to determine if it has completed its task or not. Furthermore, the code assumes that the only the required output variables should be supplied for posterior mode values, while in the case of the initial values also the optional output variables need to be provided. For the initial values, the Matlab function for solving the DSGE model has already been copied to the `tmp` directory. The measurement equation file and, when used, the file with parameters to update have also been copied to this directory.[21] For posterior mode values there is no guarantee that these files exist in that directory and, hence, the code copies the files to that location.[22] The `CopyFile` function takes care of file copying, ensuring that old entries in the `tmp` directory are overwritten. Note that there is no need to call `FixFilePath` for the input variables which are used by `CopyFile`; it handles operating system consistent paths internally.

YADA has a function that can be used to solve the DSGE model, rewrite the AiM generated solution into the state equation, and provide the state-space representation by running the measurement equation. The function, which is aptly called `SolveDSGEModel` (located in the `tools` directory), needs data from AiM. The following code tries to load data from the mat-file `DSGEModel.AIMDataFile` (see Table 11) and report any problems should the data not be available:

```
%
% load the AIMData file
```

[21] The copy operations are performed inside the `InitializeDSGEModelSimulation` function; see Section 5.3.1.

[22] As already mentioned in Section 2, the `YADA` function deletes all files from the `tmp` directory before it finishes.

```
%
ErrorStr = '';
try;
   AIMData = load(DSGEModel.AIMDataFile);
catch;
   ErrorStr = ['Unable to load the file "' DSGEModel.AIMDataFile ...
               '". Message caught is: ' lasterr];
end;
if isempty(ErrorStr)==0;
   if nargout>1;
      status = 0;
      kalmanstatus = 0;
   end;
   About(ErrorStr,'information','YADA - Bad MAT File',120,500,CurrINI);
   return;
end;
```

The function SolveDSGEModel takes 6 input variables: DSGEModel, theta, thetaPositions, ModelParameters, AIMData, and OrderQZ. The first 4 input variables are also inputs for the function DSGESpatialDistortionsTheta, the fifth input variable is obtained above, while the last input is a boolean whcih is unity if ordqz is a built-in Matlab function and zero otherwise. This variable is located as a field in the CurrINI structure. As a precaution it may be useful to call the DSGE model solution routine with try-catch code. The following takes this approach and also makes use of the 6:th and only optional output variable from this function:

```
ErrorStr = '';
try;
   [A,H,R,F,B0,mcode,ErrMsg] = SolveDSGEModel(DSGEModel,theta, ...
                               thetaPositions,ModelParameters, ...
                               AIMData,CurrINI.OrderQZ);
   if isempty(ErrMsg)==0;
      ErrorStr = ['YADA caught an error when trying to solve ' ...
                  'the DSGE model. Message caught is:' ErrMsg];
   end;
catch;
   ErrorStr = ['YADA caught an error when trying to solve ' ...
               'the DSGE model. Message caught is:' lasterr];
end;
if isempty(ErrorStr)==0;
   About(ErrorStr,'error','YADA Error',CurrINI.scrsz(4)-50,500,CurrINI);
   drawnow;
   return;
end;
[B0,KeepVar] = RemoveRedundantColumns(B0);
```

The matrices A, H and R come from the measurement equation file, while F and B0 determine the state equation; see, e.g., Warne (2017, Section 3). The mcode variable is particularly important here since it holds the key information about the properties of a possible solution of the DSGE model at theta. The last line in the above code calls RemoveRedundantColumns which, to no big surprise, examines a matrix and removed columns with zeros (or with numbers arbitrarily close to zero). Using this function is important if the structural shocks need to be estimated since this requires that B0 has full column rank. Moreover, the second output variable from the function, KeepVar, gives the non-zero column numbers for the original B0.

This variable is useful whenever the model is solved for multiple parameter values since further calls to RemoveRedundantColumns can be avoided.[23]

When the call to the spatial distortions function includes the optional output variables, we should let status be equal to mcode. The following code takes care of this and, when mcode is not unity, a message is displayed with the type of AiM solution problem that has been detected:

```
if nargout>1;
   status = mcode;
   if mcode~=1;
      if mcode==2;
         mcodeStr = 'Roots are not correctly computed by real_schur.'
      elseif mcode==3;
         mcodeStr = 'Too many big roots.';
      elseif mcode==35;
         mcodeStr = 'Too many big roots, and q(:,right) is singular.';
      elseif mcode==4;
         mcodeStr = 'Too few big roots.';
      elseif mcode==45;
         mcodeStr = 'Too few big roots, and q(:,right) is singular.';
      elseif mcode==5;
         mcodeStr = 'q(:,right) is singular.';
      elseif mcode==61;
         mcodeStr = 'Too many exact shiftrights.';
      elseif mcode==62;
         mcodeStr = 'Too many numeric shiftrights.';
      elseif mcode==7;
         mcodeStr = 'Infinite or NaN values detected.';
      elseif mcode==8;
         mcodeStr = ['The function "compute_aim_matrices" returns ' ...
                     'complex numbers.'];
      elseif mcode==-1;
         mcodeStr = 'No stable solution.';
      elseif mcode==-2;
         mcodeStr = 'Too many large generalized eigenvalues.';
      elseif mcode==-3;
         mcodeStr = 'Too few large generalized eigenvalues.';
      else;
         mcodeStr = 'Return code not properly specified.';
      end;
      if DSGEModel.ModelSolver==1;
         SolverStr = 'AiM';
      elseif DSGEModel.ModelSolver==2;
         SolverStr = 'Klein';
      elseif DSGEModel.ModelSolver==3;
         SolverStr = 'Gensys';
      end;
      txt = ['The ' SolverStr ' solver provided the return code: ' ...
            int2str(mcode) ', i.e., "' mcodeStr '"'];
```

[23] The shocks $v_t = B_0\eta_t = \xi_t - F\xi_{t-1}$ can always be estimated once the state variables, ξ_t, have been estimated via either the Kalman updater $(t|t)$ or the Kalman smoother $(t|T)$. Provided that the $q \times q$ matrix $B_0'B_0$ has full rank q, the structural shocks, η_t, are uniquely determined from v_t and B_0. The function SolveDSGEModel does not test if B_0 has full column rank. A typical situation where B_0 indeed has rank less than q occurs when the user has fixed a standard deviation parameter to zero in, say, the prior distribution specification file, but has not deselected the corresponding shock via the *Set State Shocks* function on the *Actions* menu.

```
        About(txt,'information',['YADA - ' SolverStr ' Solver Error'], ...
              200,500,CurrINI);
        return;
     end;
  end;
```

Since YADA allows for a time-varying measurement matrix `H`, the time dimension of the matrix is covered by a third dimension. The length of this dimension must at least be equal to the number of time series observation in `DSGEModel.Y`, the matrix with data on the observed (endogenous) variables. A simple test of this requirement can be formulated as:

```
  if length(size(H))==3;
     if size(H,3)<DSGEModel.T;
        ErrorStr = ['The number of time periods for the time-varying ' ...
                    'measurement matrix H (' int2str(size(H,3)) ') is ' ...
                    'less than the number of observations (T = ' ...
                     int2str(DSGEModel.T) '). YADA has therefore ' ...
                    'aborted from the spatial distortions.'];
        About(ErrorStr,'error','Error - Measurement Equation', ...
              200,500,CurrINI);
        return;
     end;
  end;
```

If we assume that the spatial distortions tool is a function of the data, we need to make sure that the user selected sample settings are satisfied. The first step is to obtain the positions of the first and the last observation in the selected sample in relation to the full sample that is available in `DSGEModel.Y`. YADA has a function called `CreateSubSample` for obtaining this information. The following code runs this function and collects the positions of the first and the last observation in the integer variables `FirstPeriod` and `LastPeriod`

```
  [FirstPeriod,LastPeriod] = CreateSubSample(DSGEModel);
```

The position `FirstPeriod` marks period 1 for the Kalman filter, while `KalmanFirstObservation` (a field in `DSGEModel`) is equal to the first period after the training sample when `FirstPeriod` is period 1. This means that the effective sample for the data on the observed variables is:

```
  Y = DSGEModel.Y(:, ...
        FirstPeriod+DSGEModel.KalmanFirstObservation-1:LastPeriod);
```

The final issue to consider for a single parameter value tool is the Kalman filter. YADA supports Kalman filtering and smoothing function that allow for unit roots, a time-varying matrix with coefficients on the state variables in the measurement equations, missing observations based on the standard Kalman filter and smoother as well as the square root versions thereof; see Warne (2017, Section 5) for details.

In the event that the Kalman filter needs to be applied, it is important to note that YADA allows for user input values when initializing the filter. The `DSGEModel` field `InitialStateValues` has the values for all possible state variables, while the field `UseOwnInitialState` indicates if these values should be used by the current selection of the state variables instead of the default zero values. The following code takes care of this and can be executed before the Kalman filter is called;

```
  r = length(DSGEModel.StateVariablePositions);
  if DSGEModel.UseOwnInitialState==1;
     if length(DSGEModel.InitialStateValues)==size(AIMData.endog,1);
        InitStateVector = DSGEModel.InitialStateValues( ...
                                          DSGEModel.StateVariablePositions);
     else;
        InitStateVector = zeros(r,1);
     end;
```

```
else;
   InitStateVector = zeros(r,1);
end;
```

The vector InitStateVector now contains the initial values for the set of selected state variables.

5.4.2. *Multiple Parameter Values*

It was mentioned in Section 4.7 that it may be of interest to allow only some parameters to vary across draws from the posterior, while others are fixed at, say, the posterior mode. This is not really kosher from a sampling perspective since the posterior draws are generated from the joint posterior distribution of *all* parameters and we thereafter replace the values for some parameters with fixed values. What we instead should have had access to are draws from a conditional posterior distribution for a subset of the parameters given the parameters that are fixed.[24] In fact, this means that when we take draws from the joint posterior and replace the values for some parameters with the posterior mode values, then some of the manipulated draws of theta will probably not be compatible with a unique convergent solution of the DSGE model.

Still, provided that the number of fixed parameters is low relative to the length of theta, this caveat need not be all that important. The following code shows how to choose both the parameters to fix and the parameters that can vary from one draw to the next.

```
if (isempty(DSGEModel.ScenarioParameters)==1)| ...
       (length(DSGEModel.ScenarioParameters)~=length(theta));
   DSGEModel.ScenarioParameters = ones(1,length(theta));
end;
positions = DSGEModel.ScenarioParameters .* (1:length(theta));
positions = positions(positions>0);
%
[PosAction,positions] = SelectCondVarShockDLG('init', ...
                        'Spatial Distortions','Parameters For', ...
                        ParameterNames,positions,CurrINI);
if (strcmp(lower(PosAction),'ok')==1)&(isempty(positions)==0);
   DSGEModel.ScenarioParameters = zeros(1,length(theta));
   DSGEModel.ScenarioParameters(positions) = ones(1,length(positions));
   %
   % continue with the next step
   %
elseif isempty(positions)==1;
   txt = ['You didn''t select any parameters that should be varied ' ...
          'for the estimation of the spatial distortions distribution.'];
   About(txt,'information','Spatial Distortions',150,500,CurrINI);
end;
```

The function SelectCondVarShockDLG was already discussed in Section 3.5. Notice that the vector positions contains only integer values equal to the positions of the selected parameters. By default this involves all the parameters that can be estimated as indicated by the first test regarding the field ScenarioParameters of the DSGEModel structure. Furthermore, the code also ensures that at least one of these parameters is selected by the user.

5.4.2.1. *Testing for Spatial Distortions Results on Disk*

Before a tool is executed for multiple parameter values the convention in YADA is to test if the tool has already been run, i.e., if the results from the tool can be located below the output directory. Let us therefore assume that the function SpatDistExist has been written for this

[24] That is, if $\theta = (\theta_1, \theta_2)$ and $\theta_2 = \bar{\theta}_2$ is fixed, the density we should have sampled from is the conditional density for θ_1 given θ_2.

purpose and is given by the file SpatDistExist.m in the directory project. This file may, for instance, have the following code:

```
function status = SpatDistExist(DSGEModel,SelectedParameters, ...
                       NumDraws,TotalDraws,CurrChain,IsPosterior)
% SpatDistExist: Tests if spatial distortions results exist
%                  on disk
%
% USAGE:
.
.
.
%
% initialize output
%
status = 0;
%
% fix the information needed to setup the file names
%
SelParamStr = strrep(num2str(SelectedParameters),' ','');
NumFiles = ceil(NumDraws/min(NumDraws,1000));
for ThisSave=1:NumFiles;
   if IsPosterior==1;
      file = FixFilePath([DSGEModel.OutputDirectory '\spatdist\DSGE-SD-' ...
           DSGEModel.NameOfModel '-' int2str(CurrChain) '-' ...
           SelParamStr '-' int2str(ThisSave) '-' int2str(NumFiles) ...
           '.' int2str(NumDraws) '-' int2str(TotalDraws) '.mat']);
   else;
      file = FixFilePath([DSGEModel.OutputDirectory ...
           '\spatdist\DSGE-SD-Prior-'DSGEModel.NameOfModel '-' ...
           SelParamStr '-' int2str(ThisSave) ...
           '-' int2str(NumFiles) '.' int2str(TotalDraws) '.mat']);
   end;
   if FileExist(file)==0;
      return;
   end;
end;
status = 1;
%
% end of SpatDistExist.m
%
```

This function has one output variable which is locally called status. The spatial distortion results exist on disk when this variable is unity. On the other hand, the function returns with status being zero if at least one of the files with results is missing.

In case the results already exist, the convention in YADA is to let the user decide if the old results should be reused or if new results should be computed. The following code takes care of this when posterior draws are used:

```
%
% continue with the next step
%
if SpatDistExist(DSGEModel,DSGEModel.ScenarioParameters, ...
                  size(thetaPostSample,1),NumDraws,CurrChain,1)==1;
   txt = ['YADA has located results for the spatial distortions ' ...
         'on disk. Would you like to use these results?'];
```

```
   answer = Query(txt,'question',150,'Spatial Distortions',500, ...
                  'no',CurrINI);
else;
   answer = 'No';
end;
```

When the string answer is equal to 'No', the spatial distortions should be recomputed, while the value 'Yes' means that the code should skip this operation and move directly to the step of loading the results from disk.

In the event that the multiple parameter values are draws from the prior distribution, the code can instead be expressed as:

```
%
% continue with the next step
%
if SpatDistExist(DSGEModel,DSGEModel.ScenarioParameters, ...
                 NumPriorDraws,NumPriorDraws,1,0)==1;
   txt = ['YADA has located results for the spatial distortions ' ...
          'on disk. Would you like to use these results?'];
   answer = Query(txt,'question',150,'Spatial Distortions',500, ...
                  'no',CurrINI);
else;
   answer = 'No';
end;
```

The value of CurrChain is irrelevant for prior draws, while the value of the input variable IsPosterior is zero. Furthermore, the input variables NumDraws and TotalDraws are here equal since the convention in YADA is to use all prior draws.

5.4.2.2. *Spatial Distortions Tool For Multiple Parameter Values*

The function that computes spatial distortions for multiple parameter values is assumed to be called DSGESpatialDistortions. As usual, this function is located in the project directory, below YADA's base directory. The top of this function can, e.g., be given by:

```
DoneCalc = DSGESpatialDistortions(theta,thetaSample,thetaPositions, ...
                    ModelParameters,SelectedParameters,IsPosterior, ...
                    TotalDraws,CurrChain,DSGEModel,CurrINI)
% DSGESpatialDistortions: Computes spatial distortions using multiple
%                         values of theta, and saves them to disk.
%
% USAGE:
.
.
.
%
% intialize output
%
DoneCalc = 0;
%
% determine which parameters should be updated
%
ScenarioParameters = SelectedParameters .* (1:length(theta));
ScenarioParameters = ScenarioParameters(ScenarioParameters>0);
SelParamStr = strrep(num2str(SelectedParameters),' ','');
%
% copy files to the tmp folder
%
```

```
[stat,msg] = CopyFile([GetPath(DSGEModel.AIMDataFile) ...
                      'compute_aim_matrices.m'],[pwd '\tmp']);
[stat,msg] = CopyFile(DSGEModel.MeasurementEquationFile, ...
                      [pwd '\tmp']);
if FileExist(DSGEModel.UpdateParameterFile)==1;
   [stat,msg] = CopyFile(DSGEModel.UpdateParameterFile, ...
                      [pwd '\tmp']);
end;
%
% load the AIMData file
%
ErrorStr = '';
try;
   AIMData = load(DSGEModel.AIMDataFile);
catch;
   ErrorStr = ['Unable to load the file "' DSGEModel.AIMDataFile ...
               '". Message caught is: ' lasterr];
end;
if isempty(ErrorStr)==0;
   About(ErrorStr,'information','YADA - Bad MAT File',120,500,CurrINI);
   return;
end;
%
% create the output directory
%
stat = MakeDir(DSGEModel.OutputDirectory,'spatdist');
if stat~=1;
   if ispc==1;
      txt = ['YADA was for some reason unable to create the directory "' ...
             DSGEModel.OutputDirectory '\spatdist". The computation of ' ...
             'spatial distoritions has therefore been aborted.'];
   else;
      txt = ['YADA was for some reason unable to create the directory "' ...
             DSGEModel.OutputDirectory '/spatdist". The computation of ' ...
             'spatial distoritions has therefore been aborted.'];
   end;
   About(txt,'information','YADA - Directory Creation Problem', ...
         180,500,CurrINI);
   return;
end;
```

The function only provides one output variable, the boolean entity `DoneCalc` which indicates if all the calculations were completed or not. This variables is initialized at the top of the function code. Thereafter, some standard procedures follow where the location for the parameters that are allowed to vary is determined as well as a string that is used in the name of the files with the spatial distortion results, Matlab m-files are copied to the `tmp` directory, AiM related data is collected, and a directory for storing spartial distortion results is (when necessary) created.

The next step is to check if the DSGE model can be solved at the fixed vector `theta` which is expected to hold either the initial values of the posterior mode values. The code is therefore almost identical to the single parameter value code in Section 5.4.1:

```
%
% try to solve the model at theta
%
ErrorStr = '';
```

```
try;
   [A,H,R,F,B0,mcode,ErrMsg] = SolveDSGEModel(DSGEModel,theta, ...
                                   thetaPositions,ModelParameters, ...
                                   AIMData,CurrINI.OrderQZ);
   if isempty(ErrMsg)==0;
      ErrorStr = ['YADA caught an error when trying to solve ' ...
                  'the DSGE model. Message caught is:' ErrMsg];
   end;
catch;
   ErrorStr = ['YADA caught an error when trying to solve ' ...
               'the DSGE model. Message caught is:' lasterr];
end;
if isempty(ErrorStr)==0;
   About(ErrorStr,'information','YADA - Error Message', ...
         CurrINI.scrsz(4)-50,500,CurrINI);
   return;
end;
if mcode~=1;
   if mcode==2;
      mcodeStr = 'Roots are not correctly computed by real_schur.'
   elseif mcode==3;
      mcodeStr = 'Too many big roots.';
   elseif mcode==35;
      mcodeStr = 'Too many big roots, and q(:,right) is singular.';
   elseif mcode==4;
      mcodeStr = 'Too few big roots.';
   elseif mcode==45;
      mcodeStr = 'Too few big roots, and q(:,right) is singular.';
   elseif mcode==5;
      mcodeStr = 'q(:,right) is singular.';
   elseif mcode==61;
      mcodeStr = 'Too many exact shiftrights.';
   elseif mcode==62;
      mcodeStr = 'Too many numeric shiftrights.';
   elseif mcode==7;
      mcodeStr = 'Infinite or NaN values detected.';
   elseif mcode==8;
      mcodeStr = ['The function "compute_aim_matrices" returns ' ...
                 'complex numbers.'];
   elseif mcode==-1;
      mcodeStr = 'No stable solution.';
   elseif mcode==-2;
      mcodeStr = 'Too many large generalized eigenvalues.';
   elseif mcode==-3;
      mcodeStr = 'Too few large generalized eigenvalues.';
   else;
      mcodeStr = 'Return code not properly specified.';
   end;
   if DSGEModel.ModelSolver==1;
      SolverStr = 'AiM';
   elseif DSGEModel.ModelSolver==2;
      SolverStr = 'Klein';
   elseif DSGEModel.ModelSolver==3;
      SolverStr = 'Gensys';
```

```
        end;
        txt = ['The ' SolverStr ' solver provided the return code: ' ...
               int2str(mcode) ', i.e., "' mcodeStr '"'];
        About(txt,'information',['YADA - ' SolverStr ' Solver Error'], ...
              200,500,CurrINI);
        return;
     end;
     %
     if length(size(H))==3;
        if size(H,3)<DSGEModel.T;
           ErrorStr = ['The number of time periods for the time-varying ' ...
                       'measurement matrix H (' int2str(size(H,3)) ') is ' ...
                       'less than the number of observations (T = ' ...
                        int2str(DSGEModel.T) '). YADA has therefore ' ...
                       'aborted from the spatial distortions.'];
           About(ErrorStr,'error','Error - Measurement Equation', ...
                 200,500,CurrINI);
           return;
        end;
     end;
     %
     % store the state-space values for backup
     %
     AOrig = A;
     Horig = H;
     ROrig = R;
     FOrig = F;
     B0Orig = B0;
     %
     [B0,KeepVar] = RemoveRedundantColumns(B0);
     %
     % make sure that theta is not overwritten
     %
     thetaOrig = theta;
     %
     % initialize index for saving
     %
     NumDraws = size(thetaSample,1);
     SaveAfterDraws = min(NumDraws,1000);
     NumFiles = ceil(NumDraws/SaveAfterDraws);
```

Notice that the number of values of `theta` is given by the rows of `thetaSample` and that the the code should save results to disk after at most 1000 such values.[25]

Before the code starts the real work on spatial distortions, it may be appropriate to set up the data with its sample settings, and to test run the Kalman filter and smoother functions at `theta`. Assuming that all such matters have been taken care of, the next step is to write the code for a loop over the draws in `thetaSample`. The convention in YADA is to either display a progress dialog or a wait dialog during such a loop. Which one is displayed depends on the user's settings regarding the progress dialog in the *Progress Dialog Selections* frame on the *Settings* tab; see also Table 14.

The following code initializes a few variables that are used by the loop and creates the progress or the wait dialog:

[25] The convention in YADA is to save the results after each value of `theta` when the results of a tool makes use of simulated data. For tools that do not require simulation, the results can effectively be saved less often.

```
%
% begin the loop
%
PriorHeader = '';
if IsPosterior==0;
   PriorHeader = ' - Prior';
end;
abort = '';
LastDraw = 0;
MeanEstimationTime = 0;
ThisSave = 0;
SptlDstrtns = [];
%
% check if we should setup a progress dialog
%
if DSGEModel.ShowProgress==1;
   ProgressStructure.title = ['Progress for ' int2str(NumDraws) ...
                              ' parameter draws'];
   ProgressStructure.facecolor = CurrINI.progress_facecolor;
   ProgressStructure.startfacecolor = CurrINI.progress_startfacecolor;
   ProgressStructure.edgecolor = CurrINI.progress_edgecolor;
   ProgressStructure.bgcolor = CurrINI.progress_bgcolor;
   ProgressStructure.stop = 0;
   ProgressStructure.clock = DSGEModel.ShowProgressClock;
   ProgressStructure.label = 'Mean estimation time:';
   %
   ProgressStructure.name = ['Spatial Distortions Distribution' ...
                             PriorHeader];
   ProgressStructure.CurrINI = CurrINI;
   WaitHandle = ProgressDLG(0,ProgressStructure);
   set(WaitHandle,'Color',get(CurrINI.GraphicsRoot, ...
                              'defaultuicontrolbackgroundcolor'));
   drawnow;
else;
   txt = ['Please wait while YADA computes the distribution of the ' ...
          'spatial distortions. Computations started at: ' ...
          StringTrim(datestr(now,14))];
   WaitHandle = WaitDLG(txt,'information',['Spatial Distortions ' ...
                        'Distribution' PriorHeader],500,150,CurrINI,0);
   WaitControls = get(WaitHandle,'UserData');
end;
```

The variable SptlDstrtns is, e.g., a structure that holds the results for each value of theta that is used for the spatial distortions within the current output file.

The basic bulding block of the for-loop is considered next.

```
for it=1:NumDraws;
   LastDraw = LastDraw+1;
   if DSGEModel.ShowProgress==1;
      abort = get(WaitHandle,'UserData');
      if strcmp(abort,'cancel')==1;
         break;
      else;
         ProgressDLG([it/NumDraws MeanEstimationTime]);
      end;
```

```
end;
%
% Solve the model for the current parameter vector
%
theta(ScenarioParameters) = thetaSample(it,ScenarioParameters)';
tic;
[A,H,R,F,B0,status] = SolveDSGEModel(DSGEModel,theta, ...
                        thetaPositions,ModelParameters, ...
                        AIMData,CurrINI.OrderQZ);
if status~=1;
   %
   % use backup values of state-space model
   %
   A = AOrig;
   H = HOrig;
   R = ROrig;
   F = FOrig;
   B0 = B0Orig;
end;
B0 = B0(:,KeepVar);
%
% do a bunch of complex computations
%
.
.
.
%
% store the results in the SptlDstrtns structure
%
SptlDstrtns(LastDraw).SpatDist = SD;
%
% measure average computation time
%
MeanEstimationTime = (((it-1)/it)*MeanEstimationTime)+((1/it)*toc);
%
% check if we should save to disk
%
if (LastDraw==SaveAfterDraws)|(it==NumDraws);
   ThisSave = ThisSave+1;
   LastDraw = 0;
   %
   % set up the file name and then save
   %
   if IsPosterior==1;
      file = FixFilePath([DSGEModel.OutputDirectory ...
           '\spatdist\DSGE-SD-' DSGEModel.NameOfModel '-' ...
           int2str(CurrChain) '-' SelParamStr '-' int2str(ThisSave) ...
           '-' int2str(NumFiles) '.' int2str(NumDraws) '-' ...
           int2str(TotalDraws) '.mat']);
   else;
      file = FixFilePath([DSGEModel.OutputDirectory ...
           '\spatdist\DSGE-SD-Prior-' DSGEModel.NameOfModel '-' ...
           SelParamStr '-' int2str(ThisSave) '-' ...
           int2str(NumFiles) '.' int2str(TotalDraws) '.mat']);
```

```
        end;
        save(file,'SptlDstrtns');
        %
        % restore the data structure
        %
        SptlDstrtns = [];
     end;
     drawnow;
  end;
```

The first step in the loop is to check if a progress dialog is shown. If so, the dialog is *either* updated with the current sample number for the parameter values relative to the total number of parameter values and the average computation time for the spatial distortions, *or* the execution of the for-loop is stopped. The latter occurs when the *Cancel* button on the progress dialog has been pushed during the previous for-loop iteration, resulting in the `UserData` property of the dialog being set to the string `'cancel'`.

Next, the entries of the parameter vector `theta` that can vary across the parameter values are updated with the values for the current sample number. Based on these values the code attempts to solve the DSGE model. If there does not exist a unique convergent solution, a backup solution is provided. Once the solution of the DSGE model is available, the code can perform the necessary computations for the spatial distortions. The variable `SD` is computed during this step and, for expositional reasons, it is assumed to be a matrix with `n` rows and `r` columns. In the last part of the loop, the code checks if the results should be stored to disk or not.

Before the function `DSGESpatialDistortions` exits, we need to make sure that the progress or wait dialog is closed and that `DoneCalc` is set to unity if all the computations have been performed:

```
  %
  % close the wait dialog
  %
  if DSGEModel.ShowProgress==1;
     if ishandle(WaitHandle)==1;
        set(WaitHandle,'UserData','done');
        close(WaitHandle);
        drawnow;
        pause(0.02);
     end;
  else;
     set(WaitControls.text,'UserData','done');
     delete(WaitHandle);
     drawnow;
     pause(0.02);
  end;
  %
  % check if we have computed all output
  %
  if strcmp(abort,'cancel')==0;
     DoneCalc = 1;
  end;
```

Notice that `DoneCalc` is set to unity when the `abort` variable is not equal to `'cancel'`, i.e., when the *Cancel* button on the progress dialog has not been pushed.

We are now at the stage where the code for calling the function `DSGESpatialDistortions` can be expressed. If the multiple values for `theta` are given by draws from the posterior distribution, the following code can be used:

```
if strcmp(lower(answer),'yes')==1;
   DoneCalc = 1;
else;
   DoneCalc = DSGESpatialDistortions(theta,thetaPostSample, ...
                  thetaPositions,ModelParameters, ...
                  DSGEModel.ScenarioParameters,1,NumDraws,CurrChain, ...
                  DSGEModel,CurrINI);
end;
```

Similarly, when the draws are taken from the prior distribution, the call to the spatial distortions function can be replaced with:

```
DoneCalc = DSGESpatialDistortions(theta,thetaDraws', ...
                  thetaPositions,ModelParameters, ...
                  DSGEModel.ScenarioParameters,0,NumPriorDraws,1, ...
                  DSGEModel,CurrINI);
```

Notice that the matrix with multiple parameters is here transposed to ensure that the rows hold the multiple `theta` values.

5.4.2.3. *Computing Distribution Results for the Spatial Distortions Tool*

Once the spatial distortions have been calculated for each value of parameter vector `theta`, the convention in YADA is to turn to the distributional aspects of the results. This means that the data need to be read from disk, sorted, and stored in temporary output variables. It is assumed below that this task is performed by the function `DSGESpatDistDistribution`.

It is assumed that the output variable from this function is given by the structure `SpatDist` which has three fields: `Mean`, `Quantiles`, and `ShortestConfidence`. The first field holds, as the name suggests, the mean of the spatial distortions, while the second field is a vector structure of length equal to the number of quantiles and with subfields `Mean` and `percent`. The former gives a particular quantile value of the spatial distortions, while the latter is the percentile value of the quantile in the range between 0 and 100. The third field is a structure of length equal to half the number of percentiles. It has fields `UpperBound` and `LowerBound` that hold upper and lower bound, respectively, for the shortest confidence bands. In addition, it has the field `percent` that holds the percentage point value of the band.

The following code initializes the output from the function:

```
function SpatDist = DSGESpatDistDistribution(DSGEModel,CurrINI, ...
                        SelectedParameters,NumDraws,TotalDraws, ...
                        CurrChain,IsPosterior,Weights)
% DSGESpatDistDistribution: Computes the mean, quantiles and shortest
%                           confidence bands of the spatial distortions
%
% USAGE:
.
.
.
%
% determine parameters
%
SelParamStr = strrep(num2str(SelectedParameters),' ','');
NumFiles = ceil(NumDraws/min(NumDraws,1000));
NumQuants = length(DSGEModel.Percentiles);
if isempty(Weights)==1;
   Weights = ones(NumDraws,1);
end;
%
if IsPosterior==1;
```

```
   file = FixFilePath([DSGEModel.OutputDirectory ...
        '\spatdist\DSGE-SD-' DSGEModel.NameOfModel '-' ...
        int2str(CurrChain) '-' SelParamStr '-1-' ...
        int2str(NumFiles) '.' int2str(NumDraws) '-' ...
        int2str(TotalDraws) '.mat']);
else;
   file = FixFilePath([DSGEModel.OutputDirectory ...
        '\spatdist\DSGE-SD-Prior-' DSGEModel.NameOfModel ...
        '-' SelParamStr '-1-' int2str(NumFiles) '.' ...
        int2str(TotalDraws) '.mat']);
end;
%
SDData = load(file);
[n,r] = size(SDData.SptlDstrtns(1).SpatDist);
%
% initialize output
%
SpatDist.Mean = zeros(n,r);
for i=1:NumQuants;
   SpatDist.Quantiles(i).Mean = zeros(n,r);
   SpatDist.Quantiles(i).percent = DSGEModel.Percentiles(i);
end;
for i=1:(NumQuants/2);
   SpatDist.ShortestConfidence(i).UpperBound = zeros(n,r);
   SpatDist.ShortestConfidence(i).LowerBound = zeros(n,r);
   SpatDist.ShortestConfidence(i).percent = DSGEModel.Percentiles( ...
                              NumQuants-i+1)-DSGEModel.Percentiles(i);
end;
PriorHeader = '';
if IsPosterior==0;
   PriorHeader = ' - Prior';
end;
```

The number of spatial disortions is here equal to `n` times `r`. For each such statistic there are `NumDraws` values. Let us assume that `n` is equal to the number of observed variables while `r` is equal to the number of state variables. The marginal distribution of the spatial distortions for the pair `(i,j)` with `i=1,...,n` and `j=1,...,r` is the object of interest. This distribution is estimated by sorting the spatial distortions for each pair `(i,j)`. The following code provides the outer and inner loop for the computation spatial distortions distribution and sets up a wait dialog for displaying progress:

```
%
% setup a wait dialog
%
txt = ['Please wait while YADA computes the mean spatial ' ...
      'distortions as well as percentile values. The data are ' ...
      'loaded sequentially from disk and it may therefore take ' ...
      'some time. Current pair: ' ...
      StringTrim(DSGEModel.VariableNames(1,:)) ',' ...
      StringTrim(DSGEModel.StateVariableNames(1,:)) '.'];
WaitHandle = WaitDLG(txt,'information',['Spatial Distortions' ...
                    PriorHeader],500,200,CurrINI,0);
WaitControls = get(WaitHandle,'UserData');
drawnow;
pause(0.02);
```

```
%
for i=1:n;
   for j=1:r;
      %
      % update the wait dialog
      %
      txt = ['Please wait while YADA computes the mean spatial ' ...
          'distortions as well as percentile values. The data are ' ...
          'loaded sequentially from disk and it may therefore take ' ...
          'some time. Current pair: ' ...
          StringTrim(DSGEModel.VariableNames(i,:)) ',' ...
          StringTrim(DSGEModel.StateVariableNames(j,:)) '.'];
      set(WaitControls.text,'String',txt);
      drawnow;
      %
      % load the results
      %
      .
      .
      .
   end;
end;
%
% close the wait dialog
%
set(WaitControls.text,'UserData','done');
delete(WaitHandle);
drawnow;
pause(0.02);
```

The next step within the inner loop is to load all the spatial distortions for the pair `(i,j)`. The following code takes care of this and stores the results for this pair in the variable `Temp`:

```
%
% load the results
%
Temp = zeros(NumDraws,1);
CurrSaves = 0;
for ThisSave=1:NumFiles;
   if IsPosterior==1;
      file = FixFilePath([DSGEModel.OutputDirectory ...
           '\spatdist\DSGE-SD-' DSGEModel.NameOfModel '-' ...
           int2str(CurrChain) '-' SelParamStr '-' int2str(ThisSave) ...
           '-' int2str(NumFiles) '.' int2str(NumDraws) '-' ...
           int2str(TotalDraws) '.mat']);
   else;
      file = FixFilePath([DSGEModel.OutputDirectory ...
           '\spatdist\DSGE-SD-Prior-' DSGEModel.NameOfModel '-' ...
           SelParamStr '-' int2str(ThisSave) '-' int2str(NumFiles) ...
           '.' int2str(TotalDraws) '.mat']);
   end;
   SDData = load(file);
   NumSaves = length(SDData.SptlDstrtns);
   for s=1:NumSaves;
      Temp(CurrSaves+s,1) = SDData.SptlDstrtns(s).SpatDist(i,j);
```

```
    end;
    CurrSaves = CurrSaves+NumSaves;
    drawnow;
 end;
```

The final step is to compute the mean or some other location statistic (median or mode) and the percentiles that are defined in the `DSGEModel` field `Percentiles`:

```
%
% compute the mean
%
SpatDist.Mean(i,j) = (1/NumDraws)*sum(Temp.*Weights);
%
% sort the results
%
Temp = sort(Temp);
%
% determine the quantiles
%
for l=1:NumQuants;
   if DSGEModel.Percentiles(l)<50;
      QuantVal = ceil((DSGEModel.Percentiles(l)/100)*NumDraws);
   else;
      QuantVal = floor((DSGEModel.Percentiles(l)/100)*NumDraws);
   end;
   SpatDist.Quantiles(l).Mean(i,j) = Temp(QuantVal,1);
end;
```

YADA determines confidence (or error) bands directly from the percentiles. If the vector `DSGEModel.Percentiles` has 4 elements, then YADA compute one confidence band using the outer values of the vector and a second using the inner values. The percentiles are always sorted and hence, if the first and the last values are 5 and 95, a 90 percent equal-tail confidence band is computed from these percentiles. An alternative approach is to compute, say, a 90 percent confidence bands such that the distance between the upper and the lower bound for a given coverage probability is as short as possible; see, e.g., Bernardo and Smith (2000, Appendix B, Section 3.2). Such bands are often called shortest confidence bands.[26]

The following code computes the upper and the lower bound of the shortest confidence bands:

```
for l=1:floor(NumQuants/2);
   %
   % determine number of elements in confidence band
   %
   ConfDraws = floor((SpatDist.ShortestConfidence(l).percent/100)*NumDraws);
   %
   UpperBound = Temp(ConfDraws,1);
   LowerBound = Temp(1,1);
   for k=2:NumDraws-ConfDraws+1;
      if (Temp(ConfDraws+k-1,1)-Temp(k,1))<(UpperBound-LowerBound);
         UpperBound = Temp(ConfDraws+k-1,1);
         LowerBound = Temp(k,1);
      end;
   end;
```

[26] A related approach is concerned with *highest probability density regions*. Such credible regions have the property that (i) the probability that a value is an element of the region is equal to $1 - \alpha$ (a $100(1 - \alpha)$ percent region), and (ii) the density value for any element in the region is greater than or equal to the density value for all elements that do not belong to the region.

```
    %
    % store the shortest confidence bands
    %
    SpatDist.ShortestConfidence(l).UpperBound(i,j) = UpperBound;
    SpatDist.ShortestConfidence(l).LowerBound(i,j) = LowerBound;
end;
```

The number of confidence bands is equal to half the length of the `Percentiles` field. YADA does not currently calculate shortest confidence bands, but this may change in the future.

The marginal distribution of the spatial distortions can now be called by the functions that use posterior and prior draws of the `theta` parameters. The following code performs this task for the posterior draws:

```
if DoneCalc==1;
    %
    % compute the marginal distributions for posterior
    %
    SpatDist = DSGESpatDistDistribution(DSGEModel,CurrINI, ...
                  DSGEModel.ScenarioParameters, ...
                  size(thetaPostSample,1),NumDraws,CurrChain,...
                  1,Weights);
    %
    % check if user wants to save results to disk
    %
end;
```

Similarly, if the draws are taken from the prior distribution we may use the following example:

```
if DoneCalc==1;
    %
    % compute the marginal distributions for prior
    %
    SpatDist = DSGESpatDistDistribution(DSGEModel,CurrINI, ...
                  DSGEModel.ScenarioParameters, ...
                  NumPriorDraws,NumPriorDraws,1,0,[]);
    %
    % check if user wants to save results to disk
    %
end;
```

Once these results have been computed by the function for multiple parameter values, we can turn to the last issue. Namely, to display and save the results from the spatial distortions exercise.

5.5. *Displaying Results*

The convention is YADA is to ask the user if the results should be saved to disk. If the user answers in the positive, then the results are saved in a mat-file while access to these data is made available through a Matlab script that YADA writes in an m-file. This script gives direct access to all the variables that have been saved to the mat-file.

The following code gives an example how this can be achieved for the posterior draws:

```
%
% ask if we should save results to disk
%
txt = ['Would you like to save the mean, the quantiles, and the ' ...
       'shortest confidence band of the spatial distortions to file?'];
answer = Query(txt,'question',140,'Save - Spatial Distortions',500, ...
               'no',CurrINI);
if strcmp(lower(answer),'yes')==1;
```

```
   SelParamStr = strrep(num2str(DSGEModel.ScenarioParameters),' ','');
   YNames = DSGEModel.VariableNames;
   StateVariableNames = DSGEModel.StateVariableNames;
   %
   file = FixFilePath([DSGEModel.OutputDirectory '\SpatialDDist-' ...
           DSGEModel.NameOfModel '-' int2str(CurrChain) '-' ...
           SelParamStr '.' int2str(NumDraws) '-' ...
           int2str(TotalDraws) '.mat']);
   save(file,'YNames','StateVariableNames','SpatDist');
   %
   mfile = FixFilePath([DSGEModel.OutputDirectory '\SDDist' ...
            DSGEModel.NameOfModel int2str(CurrChain) SelParamStr ...
            int2str(NumDraws) int2str(TotalDraws) '.m']);
   fid = fopen(mfile,'wt');
   %
   % write code for m-file
   %
   fprintf(fid,['%%\n%% load the data in ' GetFilename(file) '\n%%\n']);
   fprintf(fid,'DataStruc = load(''%s'');\n',GetFilename(file));
   fprintf(fid,['%%\n%% string matrix with the names of the ' ...
             'observed variables\n%%\n']);
   fprintf(fid,'YNames = DataStruc.YNames;\n');
   fprintf(fid,['%%\n%% string matrix with the names of the ' ...
             'state variables\n%%\n']);
   fprintf(fid,'StateVariableNames = DataStruc.StateVariableNames;\n');
   fprintf(fid,'%%\n%% structure with spatial distortions results\n%%\n');
   fprintf(fid,'SpatDist = DataStruc.SpatDist;\n');
   fprintf(fid,'%%\n%% Add your own commands below\n%%\n\n\n');
   fprintf(fid,'%%\n%% Created by YADA on %s\n%%\n',datestr(now,0));
   fclose(fid);
   %
   % display a dialog with information
   %
   txt = ['The spatial distortions data have been saved to the file "' ...
          GetFilename(file) '" in the directory "' GetPath(file) ...
          '". The file contains 3 entries: YNames (string matrix ' ...
          'with the names of the observed variables), ' ...
          'StateVariableNames (string matrix with the names of the ' ...
          'state variables), and SpatDist (structure with data on ' ...
          'mean, quantiles, and shortest confidence bands). To ' ...
          'access this data you may run the Matlab script file "' ...
          GetFilename(mfile) '".'];
   About(txt,'information','Spatial Distortions',200,500,CurrINI);
end;
```

Notice that if draws from the prior distribution had been used for the computations, only the names of the two files would need to be different.

There are two basic approaches to display results from an exercise such as the spatial distortions. The evidence may be written to a text file and then displayed on screen through the `TextGUI` function; see Section 3.2. The alternative is to plot the results in graphs. The convention in YADA for the latter case is to allow the user to select which variables to plot and, when applicable, which sample to show.

5.5.1. *Writing Results to a Text File*

The custom in YADA when writing results to a text file is to have a separate function take care of this issue. Once this function has been executed, YADA checks if the text file exists on disk and if it is located the code instructs the `TextGUI` function to display the content of the file. The output from the spatial distortions exercises that have been discussed above for single and multiple values of the parameters `theta` is assumed to be available as the variable `SpatDist`. For the single value case this is simply a matrix with `n` rows and `r` columns, representing the distortions for the `n` observed (endogenous) variables as measured by the `r` state variables. For the multiple parameter values case this variable is instead a structure with fields `Mean`, `Quantiles` and `ShortestConfidence`. The function that writes the results to disk needs to either take the different cases into account, or there needs to be one such function per case. Below I will discuss the first possibility.

Let us therefore assume that the function that writes the spatial distortions to disk is called `PrintSpatialDistortions` and that it exists in the usual directory, i.e., in `project` directly below YADA's base directory. The following code is located at the top the file:

```
function PrintSpatialDistortions(DSGEModel,CurrINI,SpatDist, ...
                                 SelectedParameters,NumDraws, ...
                                 TotalDraws,CurrChain,IsPosterior)
% PrintSpatialDistortions: writes the results on spatial distortions to
%                          a text file
%
% USAGE:
.
.
.
%
% determine the file name
%
if NumDraws==1;
   if IsPosterior==1;
      file = FixFilePath([DSGEModel.OutputDirectory '\SpatDist-' ...
              DSGEModel.NameOfModel '-PosteriorMode.txt']);
   else;
      file = FixFilePath([DSGEModel.OutputDirectory '\SpatDist-' ...
              DSGEModel.NameOfModel '-InitialValues.txt']);
   end;
else;
   SelParamStr = strrep(num2str(SelectedParameters),' ','');
   if IsPosterior==1;
      file = FixFilePath([DSGEModel.OutputDirectory '\SpatDist-' ...
              DSGEModel.NameOfModel '-' int2str(CurrChain) '-' ...
              SelParamStr '-' int2str(NumDraws) '.' ...
              int2str(TotalDraws) '.txt']);
   else;
      file = FixFilePath([DSGEModel.OutputDirectory '\SpatDistPrior-' ...
              DSGEModel.NameOfModel '-' SelParamStr '-' ...
              int2str(NumDraws) '.txt']);
   end;
end;
```

Through the input variables `NumDraws` and `IsPosterior` we can determine if we have single or multiple parameter values and if the values are related to the posterior mode or the initial values, or the posterior distribution or the prior.

The basic formating of the output file in YADA is as follows:

```
%
% open the output file
%
fid = fopen(file,'wt');
%
fprintf(fid,['*******************************************************' ...
                                        '*************************\n']);
fprintf(fid,['*                                                      ' ...
                                        '                        *\n']);
fprintf(fid,['*                    S P A T I A L   D I S T O R T I O ' ...
                                        'N S                     *\n']);
fprintf(fid,['*                                                      ' ...
                                        '                        *\n']);
fprintf(fid,['*******************************************************' ...
                                      '*************************\n\n']);
%
% initializing the formatting of output numbers.
% we usually apply the 12.6f format.
%
prt_val = ['%' num2str(6+CurrINI.decimals,'%0.0f') '.' ...
                   num2str(CurrINI.decimals,'%0.0f') 'f'];
%
[n,vn] = size(DSGEModel.VariableNames);
if vn<15;
   AddNameStr = SpaceStr(15-vn);
else;
   AddNameStr = '';
end;
%
% write the output in SpatDist
%
.
.
.
%
% bottom file data
%
fprintf(fid,'Directory for file: %s\n',GetPath(file));
fprintf(fid,'Name of file:       %s\n',GetFilename(file));
fprintf(fid,'Output created on:  %s\n\n',datestr(now,0));
fclose(fid);
%
% end of PrintSpatialDistortions.m
%
```

Since the variable SpatDist is different for the single and the multiple parameter values cases we need to ensure that the code takes this into account. Like in the case of the name of the output file we may use the NumDraws variable to separate these cases from one another.

This function may now be called as follows in the case of posterior draws:

```
%
% write output fo file
%
PrintSpatialDistortions(DSGEModel,CurrINI,SpatDist, ...
        DSGEModel.ScenarioParameters,size(thetaPostSample,1), ...
```

```
            NumDraws,CurrChain,1);
    %
    SelParamStr = strrep(num2str(SelectedParameters),' ','');
    file = FixFilePath([DSGEModel.OutputDirectory '\SpatDist-' ...
                DSGEModel.NameOfModel '-' int2str(CurrChain) '-' ...
                SelParamStr '-' int2str(size(thetaPostSample,1)) '.' ...
                int2str(NumDraws) '.txt']);
    %
    if FileExist(file)==1;
       TextGUI(file,'Spatial Distortions',[(CurrINI.scrsz(3)- ...
              min(1000,CurrINI.scrsz(3)))/2 32 min(1000,CurrINI.scrsz(3)) ...
              CurrINI.scrsz(4)-100],'Posterior Distribution',100,CurrINI,1,0);
    end;
```

It is noteworthy that the results are displayed only if YADA can locate the file on disk. Moreover, the value of the last input variable for `TextGUI` (locally called `CloseSelf`) ensures that the execution of the code is halted until the dialog is closed by the user. The reason why this value for last input variable is used is simply that it is assumed that the results can also be displayed graphically.

5.5.2. *Displaying Results in a Graph*

Let us assume that it makes perfect sense to plot the spatial distortions for one variable at a time. Moreover, since the second dimension of the results is assumed to represent the state variables there is no need to plot time series data. Instead, we shall consider scatter-plots.

Let us call the file for displaying scatter-plots `SpatialDistortionsDLG.m`. The function having the same name as the file takes 5 input variables. First of all, the variable `selector` is the only required input variable. It can take on 3 values: `'init'`, `'showgraph'`, and `'done'`. The first value initializes the dialog, the second leads to displaying a graph of the spatial distoritions, while the last value results in the dialog being deleted and the function returning to the function that called it. In addition, when `selector` is equal to `'init'`, the `SpatialDistortionsDLG` accepts the familiar `DSGEModel` and `CurrINI` structures as well as the `SpatDist` variable with the results on spatial distortions as input. Finally, we shall let the function accept the string variable `TypeStr` which can take on 4 different values: `Initial Values`, `Posterior Mode`, `Prior Distribution`, and `Posterior Distribution`.

The basic building block of the dialog for displaying the scatter-plots of the spatial distortions can be expressed as follows:

```
function SpatialDistortionsDLG(selector,DSGEModel,CurrINI, ...
                               SpatDist,TypeStr)
% SpatialDistortionsDLG: Displays scatter-plots of the
%                        spatial distortions
%
% USAGE:
.
.
.
if strcmp(selector,'init')==1;
   DSGEModel.SpatDist = SpatDist;
   DSGEModel.TypeStr = TypeStr;
else;
   SDGUI = findobj('Type','figure','Tag','SpatialDistortionsDLG');
   SDControls = get(SDGUI,'UserData');
   DSGEModel = get(SDControls.show,'UserData');
   CurrINI = get(SDControls.variablex,'UserData');
end;
```

```
%
switch selector
   case 'init'
      %
      % create the dialog
      %
   case 'showgraph'
      %
      % check which variable to plot
      %
   case 'done'
      set(SDControls.done,'UserData','done');
      delete(SDGUI);
      pause(0.02);
      drawnow;
end;
%
if strcmp(selector,'init')==1;
   %
   set(SDGUI,'Visible','on');
   drawnow;
   if MatlabNumber>=7;
      uicontrol(SDControls.done);
   end;
   waitfor(SDControls.done,'UserData','done');
end;
%
% end of SpatialDistortionsDLG.m
%
```

The DSGEModel and CurrINI structures are the main variables for creating the dialog and for displaying the graphs. Hence, the initialization case makes sure that SpatDist and TypeStr are stored as fields of the DSGEModel structure. For other values of the selector variable, these two structures are assumed to be stored under the UserData property of two controls that have been created by the init value of the input variable.

At the end of the function, the dialog is first made visible, the *Done* button is given focus (provided that the user's version of Matlab supports this feature), and finally the waitfor function is called. This function blocks the execution of SpatialDistortionsDLG when selector is equal to 'init' until the UserData property of the control SDControls.done is set equal to the value 'done'. Below we will find that the value of this control is initially set to 'waiting'.

The controls on the dialog as well as the figure window have not been discussed yet. The code for the init case in the switch part of the function is:

```
case 'init'
   %
   % create the dialog
   %
   SDGUI = figure('Color',get(CurrINI.GraphicsRoot, ...
                        'defaultuicontrolbackgroundcolor'), ...
                  'FileName','SpatialDistortionsDLG.m', ...
                  'MenuBar','none', ...
                  'PaperUnits','points', ...
                  'Units','pixels', ...
                  'Position',[(CurrINI.scrsz(3)-500)/2 ...
                              (CurrINI.scrsz(4)-120)/2 500 120], ...
```

```
               'Tag','SpatialDistortionsDLG', ...
               'Visible','off', ...
               'Resize','off', ...
               'Name',['Spatial Distortions - ' TypeStr], ...
               'NumberTitle','off', ...
               'CloseRequestFcn','SpatialDistortionsDLG done', ...
               'ToolBar','none');
%
if MatlabNumber>=7.0;
   set(SDGUI,'DockControl','off');
end;
%
% Outer axis
%
AxesBox([2 2 498 110],'Graphics',45,[0.5 1],'on',CurrINI);
%
% Check where the sample of data actually begins
%
[NewStartYear,NewStartPeriod] = AdjustSampleStart( ...
  DSGEModel.SubBeginYear,DSGEModel.SubBeginPeriod, ...
  DSGEModel.DataFrequency,DSGEModel.KalmanFirstObservation-1);
%
% display the sample
%
SDontrols.sample = uicontrol('Units','pixels', ...
      'BackgroundColor',get(CurrINI.GraphicsRoot, ...
                        'defaultuicontrolbackgroundcolor'), ...
      'Position',[24 75 360 20], ...
      'FontSize',CurrINI.gui_fontsize, ...
      'FontName',CurrINI.gui_fontname, ...
      'FontWeight',CurrINI.gui_fontweight, ...
      'FontAngle',CurrINI.gui_fontangle, ...
      'Style','text', ...
      'String',['Selected Sample: ' NewStartYear ':' NewStartPeriod ...
            ' - ' DSGEModel.SubEndYear ':' DSGEModel.SubEndPeriod], ...
      'HorizontalAlignment','left', ...
      'Tag','SampleText');
%
% Select variables controls
%
SDControls.variableytext = uicontrol('Units','pixels', ...
      'BackgroundColor',get(CurrINI.GraphicsRoot, ...
                            'defaultuicontrolbackgroundcolor'), ...
      'Position',[24 45 176 20], ...
      'FontSize',CurrINI.gui_fontsize, ...
      'FontName',CurrINI.gui_fontname, ...
      'FontWeight',CurrINI.gui_fontweight, ...
      'FontAngle',CurrINI.gui_fontangle, ...
      'Style','text', ...
      'String','Select variable for Y-axis:', ...
      'HorizontalAlignment','left', ...
      'Tag','VariableYText');
%
SDControls.variabley = uicontrol('Units','pixels', ...
```

```
      'BackgroundColor',[1 1 1], ...
      'Position',[200 50 165 20], ...
      'Style','popupmenu', ...
      'FontSize',CurrINI.gui_fontsize, ...
      'FontName',CurrINI.gui_fontname, ...
      'FontWeight',CurrINI.gui_fontweight, ...
      'FontAngle',CurrINI.gui_fontangle, ...
      'UserData',CurrINI, ...
      'String',DSGEModel.VariableNames, ...
      'HorizontalAlignment','center', ...
      'Tag','VariableYPopup',...
      'Value',1);
%
SDControls.variablextext = uicontrol('Units','pixels', ...
      'BackgroundColor',get(CurrINI.GraphicsRoot, ...
                           'defaultuicontrolbackgroundcolor'), ...
      'Position',[24 15 176 20], ...
      'FontSize',CurrINI.gui_fontsize, ...
      'FontName',CurrINI.gui_fontname, ...
      'FontWeight',CurrINI.gui_fontweight, ...
      'FontAngle',CurrINI.gui_fontangle, ...
      'Style','text', ...
      'String','Select variable for X-axis:', ...
      'HorizontalAlignment','left', ...
      'Tag','VariableXText');
%
SDControls.variablex = uicontrol('Units','pixels', ...
      'BackgroundColor',[1 1 1], ...
      'Position',[200 20 165 20], ...
      'Style','popupmenu', ...
      'FontSize',CurrINI.gui_fontsize, ...
      'FontName',CurrINI.gui_fontname, ...
      'FontWeight',CurrINI.gui_fontweight, ...
      'FontAngle',CurrINI.gui_fontangle, ...
      'UserData',CurrINI, ...
      'String',DSGEModel.VariableNames, ...
      'HorizontalAlignment','center', ...
      'Tag','VariableXPopup',...
      'Value',2);
%
% show graph button
%
SDControls.show = uicontrol('Units','pixels', ...
      'BackgroundColor',get(CurrINI.GraphicsRoot, ...
                           'defaultuicontrolbackgroundcolor'), ...
      'Position',[395 50 90 20], ...
      'String','Display', ...
      'FontSize',CurrINI.gui_fontsize, ...
      'FontName',CurrINI.gui_fontname, ...
      'FontWeight',CurrINI.gui_fontweight, ...
      'FontAngle',CurrINI.gui_fontangle, ...
      'CallBack','SpatialDistortionsDLG showgraph', ...
      'UserData',DSGEModel, ...
      'Enable','on', ...
```

```
         'Tag','Done');
   %
   % done button
   %
   SDControls.done = uicontrol('Units','pixels', ...
         'BackgroundColor',get(CurrINI.GraphicsRoot, ...
                           'defaultuicontrolbackgroundcolor'), ...
         'Position',[395 20 90 20], ...
         'String','Done', ...
         'FontSize',CurrINI.gui_fontsize, ...
         'FontName',CurrINI.gui_fontname, ...
         'FontWeight',CurrINI.gui_fontweight, ...
         'FontAngle',CurrINI.gui_fontangle, ...
         'CallBack','SpatialDistortionsDLG done', ...
         'Enable','on', ...
         'UserData','waiting', ...
         'Tag','Done');
   %
   % set UserData property of the dialog
   %
   set(ScatterPostDrawsGUI,'UserData',SDControls, ...
                 'HandleVisibility','callback');
```

Excluding the size of the window frame of the dialog, the width is 500 pixels and the height is 120 pixels. This is a fairly small dialog, but sufficient for our needs. The dialog also has a frame inside it which is painted by the AxesBox function and whose text is given by 'Graphics'.

In the top left corner of the dialog there is a text control which shows the currently selected sample, taking the training sample of the Kalman filter into account. The function AdjustSampleStart (located in the data directory) performs this task. Directly below this control there are four controls. Another text control, displaying the text 'Select variable for Y-axis:', and to its right a popup control that shows the currently selected observed variable for the Y-axis. The default value is here the first variable, determined by the Value property of SDControls.variabley. Below these controls there are two additional controls that deal with the selection of the variable for the X-axis of the scatter-plot.

There are also two buttons on the right hand side of the dialog. In the bottom right corner we find the *Done* done, and above it the *Display* button. When clicked on, the former button invokes the value 'done' for the selector input variable of the function, while the latter invokes the 'showgraph' case.

The final step of the 'init' case inside the switch part of SpatialDistortionsDLG is that the figure window receives as its UserData property the structure with handles to the dialog controls.

To display the graph we take a look at the code that may occur inside the 'showgraph' case. To begin with it should collect positions and names of the currently selected observed variables for ths scatter-plot. Moreover, the controls where the variable names are displayed may be updated. The typical behavior in YADA is to move the control to the next variable unless the currently selected variable is the last. In that case, the control is shifted to the first variable. The following code performs these tasks:

```
case 'showgraph'
   %
   % check which variables to plot
   %
   VariableValueX = get(SDControls.variablex,'Value');
   CurrVariableX = StringTrim(DSGEModel.VariableNames(VariableValueX,:));
   VariableValueY = get(SDControls.variabley,'Value');
```

```
   CurrVariableY = StringTrim(DSGEModel.VariableNames(VariableValueY,:));
   if VariableValueX<size(DSGEModel.VariableNames,1);
      set(SDControls.variablex,'Value',VariableValueX+1);
   else;
      set(SDControls.variablex,'Value',1);
      if CurrVariableY<size(DSGEModel.VariableNames,1);
         set(SDControls.variabley,'Value',VariableValueY+1);
      else;
         set(SDControls.variabley,'Value',2);
      end;
   end;
```

Notice that the variable for the Y-axis is only changed when the variable for the X-axis is reset to the first variable. In that case the variable for the Y-axis is either the next variable or the second variable.

When a new graph is prepared in YADA, the `Visible` property of the figure is always set to the value `'off'` until all objects on the graph have been created. The default code for this may be represented by the following:

```
%
% create the figure
%
FigHandle = figure('Visible','off', ...
                   'Position',[(CurrINI.scrsz(3)-650)/2 ...
                     (CurrINI.scrsz(4)-500)/2 650 500], ...
                   'Units','pixels', ...
                   'Tag',['ScatterPlot-' CurrVariableX '-' ...
                                         CurrVariableY], ...
                   'Name',['Spatial Distortions Of ' CurrVariableX ...
                                  ' And ' CurrVariableY ...
                                  ' - ' DSGEModel.TypeStr]);
%
SetFigureProperties(FigHandle,CurrINI);
%
% create the scatter-plot
%
.
.
.
%
% make the figure visible
%
SetToolbarImages(FigHandle,CurrINI.images);
set(FigHandle,'Visible','on','CloseRequestFcn', ...
            'delete(gcf); drawnow; pause(0.02);');
drawnow;
pause(0.02);
```

The figure is here centered on the user's screen, having a width of 650 pixels and a height of 500 pixels. It should be recalled that the window frame will be added and, thus, the figure will not be perfectly centered on the screen. The call to the function `SetFigureProperties` ensures that certain defaults for the graphs will be met. This function is located in the directory `gui\graphics` below YADA's base directory. The call to the function `SetToolbarImages` towards the end of the selection above ensures that the certain icons on the toolbar are replaced with YADA's more "modern" icons.

The actual content on the graphs is taken from the matrix `DSGEModel.SpatDist.Mean` for multiple parameter values, and from the matrix `DSGEModel.SpatDist` for a single parameter value. In the case of multiple values of `theta` this matrix is `n` times `r` with the former being the number of observed variables and the latter the number of state variables. For the scatter-plots we have already selected the observed variable, while the spatial distortions are measured in the state variable dimension. The data on the variables are therefore located in row numbers `VariableValueX` and `VariableValueY`.

The following code creates a scatter-plot of the spatial distortions for two observed variables.

```
if isstruct(DSGEModel.SpatDist)==1;
   x = DSGEModel.SpatDist.Mean(VariableValueX,:);
   y = DSGEModel.SpatDist.Mean(VariableValueY,:);
else;
   x = DSGEModel.SpatDist(VariableValueX,:);
   y = DSGEModel.SpatDist(VariableValueY,:);
end;
if MatlabNumber>8.3;
   ScatHandle = scatter(x,y,25,'filled');
else;
   if MatlabNumber>=7;
      ScatHandle = scatter('v6',x,y,4,'filled');
   else;
      ScatHandle = scatter(x,y,4,'filled');
   end;
end;
%
% Show title and labels
%
SetAxesFonts(CurrINI,'Title',['Spatial distortions for ' ...
       lower(DSGEModel.TypeStr)],'XLabel',CurrVariableX, ...
       'Ylabel',CurrVariableY);
if strcmp(get(gca,'YGrid'),'on')==1;
   set(gca,'XGrid','on');
end;
```

The option `'v6'` is used to ensure that `ScatHandle` gives the handles of patches regardless of which version of Matlab the user has prior to Matlab 2014b. The value 4 is the marker size and the option `'filled'` means that the markers for the scatter-plot will be filled. The call to the function `SetAxesFonts` ensures that the YADA defaults for label and title fonts will be used and that these objects will have the desired text. Finally, the code sets the X and Y-axis grids to both be either `'on'` or `'off'`.

Matlab 2014b is the first Matlab release based on the new graphics engine HG2. The option `'v6'` is not supported any longer and is consequently not used by YADA. At the same time, the units of the markers for the scatter series object differs from the units used with the older objects (patch objects). The units in 2014b and later is points squared so that 25 means 5 points times 5.

The final step in all of the four cases in `ProjectFunctions` is to call the above dialog function. The following code takes care of this for the posterior distribution:

```
SpatialDistortionsDLG('init',DSGEModel,CurrINI,SpatDist, ...
                                   'Posterior Distribution');
```

It now remains for you to decide exactly what the spatial distortions should do. It has been suggested in the physics literature that when a physical model of the distortion process is lacking or inadequate, bi-cubic splines may be suitable interpolating functions for defining approximate spatial distortion functions. This may indeed be a way forward.

Table 1. Accessing the handles to the top-level controls on the file menu.

File menu handle: `controls.filemenu`

Menu item	Handle	Accelerator	Alt Keys
Open Model	`controls.openmenu`	Ctrl+O	Alt+F+O
Save Model	`controls.savemenu`	Ctrl+S	Alt+F+S
Reload Model	`controls.reloadmenu`	Ctrl+R	Alt+F+R
Reopen Model	`controls.reopenmenu`		Alt+F+M
Model Sequence	`controls.modelsequencemenu`		Alt+F+D
Import YADA Settings	`controls.importsettingsmenu`		Alt+F+I
Open Text File	`controls.opentextfilemenu`	Ctrl+T	Alt+F+T
Print Setup	`controls.printsetup`	Ctrl+P	Alt+F+P
Close Model	`controls.closemenu`		Alt+F+C
Parallel Computing Toolbox	`controls.parallel.parent`		Alt+F+A
Quit	`controls.quitmenu`	Ctrl+Q	Alt+F+Q

Table 2. Accessing the handles to the controls on the edit menu (only available on computers with Matlab version 6 or earlier.

Edit menu handle: `controls.editmenu`

Menu item	Handle	Accelerator	Alt Keys
Undo	`controls.editundomenu`	Ctrl+Z	Alt+E+U
Cut	`controls.editcutmenu`	Ctrl+X	Alt+E+T
Copy	`controls.editcopymenu`	Ctrl+C	Alt+E+C
Paste	`controls.editpastemenu`	Ctrl+V	Alt+E+P
Select All	`controls.editselectallmenu`	Ctrl+A	Alt+E+A

TABLE 3. Accessing the handles to the top-level controls on the view menu.

View menu handle: `controls.viewmenu`

Menu item	Handle	Alt Keys
Open Graphics	`controls.viewgraphicsmenu`	`Alt+V+G`
Data Construction Information	`controls.datainfo`	`Alt+V+D`
AiM Model File	`controls.aimmodelinfo`	`Alt+V+A`
Prior Distribution Information	`controls.priorinfo`	`Alt+V+U`
State-Space Form	`controls.statespaceparent`	`Alt+V+F`
Graph-a-Prior	`controls.plotaprior`	`Alt+V+H`
Parameter Covariance Matrix	`controls.estimateparamcovmat`	`Alt+V+X`
Posterior Mode Results	`controls.modeinfo`	`Alt+V+P`
Posterior Mode Summary	`controls.modesummary`	`Alt+V+M`
Optimization Error Summary	`controls.opterrorsummary`	`Alt+V+Z`
Posterior Sampling Summary	`controls.postsamplesummary.parent`	`Alt+V+S`
Raw Posterior Draws	`controls.postsample.rawplots`	`Alt+V+W`
Scatter-Plot Posterior Draws	`controls.postsample.scatterplots`	`Alt+V+E`
Sequential Marginal Likelihood	`controls.postsample.marglike`	`Alt+V+Q`
Convergence	`controls.postsample.convergence`	`Alt+V+V`
Modesty Statistics	`controls.modesty.parent`	`Alt+V+Y`
Iterated Parameter Estimates	`controls.iteratedparameters`	`Alt+V+I`
Check Posterior Mode	`controls.checkpostmode`	`Alt+V+K`
Posterior Mode Surface	`controls.threedeepostmode.parent`	`Alt+V+C`
Prior Densities	`controls.viewprior`	`Alt+V+R`
Laplace Posterior Densities	`controls.viewlaplaceposterior`	`Alt+V+L`
Posterior Densities	`controls.viewposterior`	`Alt+V+T`
Prior and Log Jacobian Value	`controls.viewpriorvalue.parent`	`Alt+V+J`
Observed Variables	`controls.viewdata`	`Alt+V+O`
Annualized Observed Variables	`controls.viewannualdata`	`Alt+V+N`
Transformed Observed Variables	`controls.viewtransdata`	`Alt+V+B`

TABLE 4. Accessing the handles to the top-level controls on the tools menu.

Tools menu handle: `controls.toolsmenu`

Menu item	Handle	Alt Keys
DSGE Model Eigenvalues	`controls.dsgeeigenvalues.parent`	`Alt+T+G`
Monte Carlo Filtering	`controls.montecarlofiltering`	`Alt+T+F`
Poor Man's Invertibility Condition	`controls.dsgetovareigenvalues.parent`	`Alt+T+Y`
Information Matrix	`controls.informationmatrix.parent`	`Alt+T+X`
Simulate Data	`controls.simulatedata.parent`	`Alt+T+U`
Observed Variable Correlations	`controls.obsvarcorrs.parent`	`Alt+T+O`
Conditional Correlations	`controls.condcorrs.parent`	`Alt+T+R`
State Variable Correlations	`controls.statevarcorrs.parent`	`Alt+T+C`
State Shock Correlations	`controls.stateshockcorrs.parent`	`Alt+T+T`
Measurement Error Correlations	`controls.measurementerrorcorrs.parent`	`Alt+T+A`
1-Step Ahead Forecasts	`controls.obsvar.parent`	`Alt+T+1`
State Variables	`controls.statevariables.parent`	`Alt+T+S`
State Shocks	`controls.stateshocks.parent`	`Alt+T+K`
Measurement Errors	`controls.measurementerror.parent`	`Alt+T+M`
Log-likelihood Function	`controls.loglikelihood.parent`	`Alt+T+L`
Parameter Scenarios	`controls.controls.ps.parent`	`Alt+T+N`
Predictive Distributions	`controls.predict.parent`	`Alt+T+P`
Decompositions	`controls.decomp.parent`	`Alt+T+D`
Impulse Responses	`controls.irfunctions.parent`	`Alt+T+I`

TABLE 5. Accessing the handles to the top-level controls on the actions menu.

Actions menu handle: `controls.actionsmenu`

Menu item	Handle	Alt Keys
Run AiM Parser	`controls.runaimmenu`	`Alt+A+A`
Estimate Posterior Mode	`controls.postmodemenu`	`Alt+A+E`
Posterior Sampling	`controls.postsamplingmenu`	`Alt+A+P`
Prior Sampling	`controls.priorsamplingmenu`	`Alt+A+M`
Estimate System Prior Mode	`controls.systempriormodemenu`	`Alt+A+Y`
Set State Variables	`controls.statevarmenu`	`Alt+A+V`
Set State Equations	`controls.stateeqmenu`	`Alt+A+Q`
Set State Shocks	`controls.stateshockmenu`	`Alt+A+S`
Configure Shocks	`controls.configureshocksmenu`	`Alt+A+F`
Set Shock Groups	`controls.shockgroupmenu`	`Alt+A+G`
Set Observed Variable Groups	`controls.obsvargroupsmenu`	`Alt+A+R`
Set Initial State Values	`controls.initialstatevaluesmenu`	`Alt+A+I`
Specify Unit-Root State Variables	`controls.unitrootstatemenu`	`Alt+A+U`
Confidence Band Base Color	`controls.confbandcolor`	`Alt+A+C`
View Conditioning Variables	`controls.viewconditionalvars`	`Alt+A+W`
Select Conditioning Variables	`controls.setconditionalvars`	`Alt+A+L`
Select Conditioning Shocks	`controls.setconditionalshocks`	`Alt+A+O`
Reorder Conditioning Shocks	`controls.reorderconditionalshocks`	`Alt+A+H`
State Conditioning Variables	`controls.stateconditionalvarsparent`	`Alt+A+N`

TABLE 6. Accessing the handles to the top-level controls on the DSGE-VAR menu.

DSGE-VAR menu handle: `controls.dsgevarmenu`

Menu item	Handle	Alt Keys
Estimate Marginal Posterior Mode	`controls.dsgevarmargmodemenu`	`Alt+D+E`
Estimate Joint Posterior Mode	`controls.dsgevarjointmodemenu`	`Alt+D+J`
DSGE Posterior Sampling	`controls.dsgevarpostsamplingmenu`	`Alt+D+D`
VAR Posterior Sampling	`controls.dsgevarvarpostsamplingmenu`	`Alt+D+P`
Prior Sampling	`controls.dsgevarpriorsamplingmenu`	`Alt+D+M`
View	`controls.dsgevarviewparent`	`Alt+D+V`
Tools	`controls.dsgevartoolsparent`	`Alt+D+T`
Set DSGE-VAR Shocks	`controls.dsgevarshocksmenu`	`Alt+D+S`
Lag Order	`controls.dsgevarlagordermenu`	`Alt+D+L`

TABLE 7. Accessing the handles to the top-level controls on the BVAR menu.

BVAR menu handle: `controls.bvarmenu`

Menu item	Handle	Alt Keys
Estimate Posterior Mode	`controls.bvarmodemenu`	`Alt+B+E`
Posterior Sampling	`controls.gibbssamplingmenu`	`Alt+B+P`
Posterior Mode Results	`controls.bvarmoderesultsmenu`	`Alt+B+M`
Posterior Sampling Results	`controls.bvargibbsresultsmenu`	`Alt+B+S`
Eigenvalues	`controls.bvareigenvalues`	`Alt+B+G`
Predictive Distributions	`controls.bvarpredict.parent`	`Alt+B+C`
Raw Posterior Draws	`controls.bvarrawdraws`	`Alt+B+R`
Sequential Marginal Likelihood	`controls.bvarmarglike`	`Alt+B+L`
Convergence	`controls.bvarconvergence`	`Alt+B+V`
Modesty Statistics	`controls.bvarmodesty.parent`	`Alt+B+Y`
Prior Densities	`controls.bvarpriordensity`	`Alt+B+I`
Posterior Densities	`controls.bvarpostdensity`	`Alt+B+D`

TABLE 8. Accessing the handles to the controls on the help menu.

Help menu handle: `controls.helpmenu`		
Menu item	Handle	Alt Keys
Help	`controls.helpfilemenu`	`Alt+H+H`
Browser-Based Help	`controls.htmlhelpfilemenu`	`Alt+H+B`
Manual - Computational Details (PDF)	`controls.yadamanual`	`Alt+H+M`
Extending YADA (PDF)	`controls.extendyada`	`Alt+H+E`
YADA Cyberspace Connection	`controls.yadahomepage`	`Alt+H+Y`
License	`controls.license`	`Alt+H+L`
About	`controls.aboutmenu`	`Alt+H+A`

TABLE 9. DSGEModel fields related to user files.

Field	Data Type	Handle to Control	Description
AIMFile	string	controls.dsge.aimfile	The string property of the control holds then full path and name of the AiM model file. The default value is pwd.
DataConstructionFile	string	controls.dsge.datafile	The string property of the control holds the full path and name of the data construction file. The default value is pwd.
InitializeParameterFile	string	controls.dsge.initialparameterfile	The string property of the control holds the full path and name of the file with parameters to initialize. The default value is an empty string.
MeasurementEquationFile	string	controls.dsge.measurefile	The string property of the control holds the full path and name of the measurement equation file. The default value is pwd.
NameOfModel	string	controls.dsge.modelname	The string property of the control gives the name of the DSGE model. The default value is an empty string, but takes on the name of the AIMFile once this file has been selected.
OutputDirectory	string	controls.outputdir	The string property of the control holds the full path of the output directory.
PriorFile	string	controls.dsge.priorfile	The string property of the control holds the full path and name of the prior distribution specification file. The default value is pwd.
PriorFileSheet	string	controls.dsge.priorfilesheet	The string property of the control holds the name of the sheet of the spreadsheet file in PriorFile when this file is an Excel file. The default value is an empty string.
RunInitializeFirst	boolean	controls.dsge.runinitialfirst	The value property of the control holds the data that determines if the file with parameters to initialize should be executed before (1) or after (0) the file with parameters to update. This control is only enabled when InitializeParameterFile exists and the value property is 0 by default.
SystemPriorFile	string	controls.dsge.systempriorfile	The string property of the control holds the full path and name of the system prior density file. The default value is pwd.
UpdateParameterFile	string	controls.dsge.updateparameterfile	The string property of the control holds the full path and name of the file with parameters to update. The default value is an empty string.

Table 10. `DSGEModel` fields related to observed data.

Field	Data Type	Description
`Actuals`	structure	A vector structure with subfields `data` and `title`. The former contains alternative data for the observed variables that may be used in forecasting exercises, while the latter is a string with the name (or title) of the dataset.
`annual`	vector	A vector of length equal to the number of observed variables (n). A value of 1 means that the variable is already annualized, 4 that its in quarterly changes so that adding 4 consecutive quarters gives annual changes, and 12 that the variable is measured in monthly changes.
`annualscale`	vector	A vector of length n with coefficients that should be multiplied by the variables after the data in `annual` is accounted for.
`BeginPeriod`	string	The period of the first observation of the observed (endogenous) variables.
`BeginYear`	string	The year of the first observation of the observed variables.
`BVARX`	vector	Determines which exogenous variables are included in a Bayesian VAR with steady-state prior.
`BVARY`	vector	Determines which endogenous (observed) variables are included in a Bayesian VAR with steady-state prior.
`ConditionalShocks`	vector	Determines which structural shocks that can be used in conditional forecasting exercises.
`ConditionalVariables`	vector	Determines which conditioning assumptions to use in conditonal forecasting exercises.
`DataFrequency`	string	Determines the frequency of the data, with `a` being annual, `q` quarterly, and `m` monthly.
`EndPeriod`	string	The period of the last observation of the observed (endogenous) variables.
`EndYear`	string	The year of the last observation of the observed (endogenous) variables.
`k`	integer	The number of the k exogenous variables.
`K1`	matrix	An $n \times m$ matrix linking the current value of the m possible conditioning assumptions to the current value of the observed variables.
`K2`	matrix	An $n \times m$ (or empty) matrix linking the current value of the m possible conditioning assumptions to lags of the observed variables.
`K3`	matrix	An $r \times q_z$ (or empty) matrix linking the current value of the q_z possible conditioning assumptions on the state variables to the current value of the r state variables.
`levels`	vector	A vector of length equal to the number of observed variables that has zero values for all variables that appear in first differences and unit values for the variables that already appear in levels.
`MixedConditionalShocks`	vector	Determines which structural shocks that can be used when the control of the distribution of a subset of the shocks conditioning method is used.

TABLE 10. `DSGEModel` fields related to observed data (continued).

Field	Data Type	Description
`n`	integer	The number of the n observed (endogenous) variables.
`ObsVarGroupNames`	string	A string matrix with the names of all observed variable groups.
`ObsVarGroups`	vector	A vector with integer values that maps each observed variable to an observed variable group.
`Percentiles`	vector	Determines the percentiles to use when creating plots that involve distributional data, such as condifidence bands.
`PredictedX`	matrix	A $k \times T_p$ matrix with out-of-sample data on the exogenous variables.
`T`	integer	The number of data points of the observed (endogenous) variables.
`U`	matrix	An $m \times T_z$ matrix with initial conditions when mapping the observed variables into the conditioning assumptions. The first observation is measured in the same time period as the first observation in the field `Y`.
`VariableNames`	string	A string matrix with n rows holding the names of the observed (endogenous) variables.
`X`	matrix	A $k \times T$ matrix with data on the exogenous variables. The data points in this matrix appear directly before the data points in `PredictedX`. The first observation is measured in the same time period as the first observation of the field `Y`.
`XVariablesNames`	string	A string matrix with k rows holding the names of the exogenous variables.
`Y`	matrix	An $n \times T$ matrix with data on the observed (endogenous) variables.
`YNaNs`	boolean	Indicates if there are missing observations among the observed variables in `Y` (1) or not (0).
`YTransformation`	structure	Determines how to transform the observed variables data. The structure has field names given by the names of the observed variables.
`YTransMatrix`	matrix	An $n \times n$ matrix with linear combinations that are applied to the vector of observed variables after the individual transformations in `YTransformation` have been applied.
`Z`	matrix	An $m \times T_z$ matrix with data for the conditioning assumptions that can be used. The first observation is measured in the same time period as the first observation of the field `Y`.

Table 10. `DSGEModel` fields related to observed data (continued).

Field	Data Type	Description
`Zeta`	matrix	A $q_z \times T_z$ matrix with data for the conditioning assumptions on the state variables. The first observation is measured in the same time period as the first observation of the field `Y`.
`ZetaConditionalShocks`	vector	Determines which structural shocks that can be used in forecasting exercises with state variable assumptions.
`ZetaConditionalVariables`	vector	Determines which conditioning assumptions for the state variables to use in forecasting exercises.
`ZetaMixedConditionalShocks`	vector	Determines which structural shocks that can be used in forecasting exercises with state variable assumptions when the control of the distribution of a subset of the shocks conditioning method is used.
`ZetaVariableNames`	string	A string matrix with q_z rows holding the names of the conditioning assumptions for the state variables.
`ZLBData`	Vector	A T_{zlb}-dimensional vector with data for the zero lower bound. May also be a scalar.
`ZLBPosition`	integer	Determines the position of the monetary policy rate among the n observed variabvles.
`ZVariableNames`	string	A string matrix with m rows holding the names of the conditioning assumptions that can be used.

Table 11. DSGEModel fields related to AiM.

Field	Data Type	Handle to Control	Description
AIMDataFile	string		Gives the full path and the name of the mat-file that holds the AiM data, i.e., the output from running the function compute_aim_data that the AiM parser creates.
AIMTolerance	integer	controls.dsge.aimtolerance	Determines the tolerance level for AiM. The AIMTolerance value is equal to the row in the string matrix holding the allowed tolerance levels.
ModelSolver	integer	controls.dsge.modelsolver	Determines which algorithm is used to solve the DSGE model.
ShockAliases	string		A string matrix where alternative names for the selected state (structural) shocks are located in the rows.
ShockGroupNames	string		A string matrix with the names of all shock groups.
ShockGroups	vector		A vector with integer values that maps each state (structural) shock to a shock group.
Solution	structure		A structure with field names A, H, R, F, and B_0 which may locally hold the state-space matrices that are the solution to the DSGE model and the measurement equations. The field is typically missing, but may be used by a system prior file.
StateEquationPositions	vector		Determines in which integer positions the state equations are located among all the equations that are listed in AIMDataFile.
StateShockNames	string		A string matrix where the names of the selected state (structural) shocks are located in the rows.
StateShockPositions	vector		Determines in which integer positions the state (structural) shocks are located among all the variables that are listed in AIMDataFile.
StateVariableNames	string		A string matrix where the names of the selected state variables are located in the rows.
StateVariablePositions	vector		Determines in which integer positions the state variables are located among all the variables that are listed in AIMDataFile.

TABLE 12. DSGEModel fields related to sample selection and Kalman filter.

Field	Data Type	Handle to Control	Description
AllowUnitRoot	boolean	controls.dsge.allowunitroot	Makes it possible to allow the DSGE model to have an arbitrary number of unit roots.
DAMaximumIterationsValue	integer	controls.dsge.doublingalgorithm	Determines the maximum number of iterations to use with the doubling algorithm when computing the covariance matrix of the state variables. The value is equal to the row number in the string matrix with the possible number of iterations.
DAToleranceValue	integer	controls.dsge.doublingtolerance	Determines the tolerance value when the doubling algorithm is used to compute the covariance matrix of the state variables. The value is equal to the row number in the string matrix with possible tolerance levels.
InitialStateValues	vector		The field holds alternative initial values for the state variables when initializing the Kalman filter. The default values are zero for all state variables.
KalmanAlgorithm	integer	controls.dsge.kalmanalgorithm	Determines which Kalman filter algorithm is used when computing the log-likelihood function.
KalmanFirstObservation	integer	controls.dsge.initializekalman	Determines the first observation to use after the training sample. The default is unity, i.e., the first period in the selected sample (meaning that there is no training sample).
PeriodStrMatrix	string	controls.dsge.subbeginperiod	String matrix with all the period values determined by the field DataFrequency. For quarterly (monthly) data the string matrix has 4 (12) rows, while for annual data it has 1 row.
StateCovConst	integer	controls.dsge.statecovariance	Determines the constant that is multiplied by the identity matrix when the state covariance matrix for the Kalman filter is initialized by the value 3 for UseDoublingAlgorithm.
SubBeginPeriod	string		The period string value for the first observation of the selected sample. For quarterly data the possible values are '1' until '4'.
SubBeginPeriodValue	integer	controls.dsge.subbeginperiod	The period integer value for the first observation of the selected sample. For quarterly data the possible values are 1 until 4.

Table 12. DSGEModel fields related to sample selection and Kalman filter (continued).

Field	Data Type	Handle to Control	Description
SubBeginYear	string		The year string value for the first observation of the selected sample.
SubBeginYearValue	integer	controls.dsge.subbeginyear	The row number of YearStrMatrix that locates the string which is equal SubBeginYear.
SubEndPeriod	string		The period string value for the last observation of the selected sample.
SubEndPeriodValue	integer	controls.dsge.subendperiod	The period integer value for the last observation of the selected sample.
SubEndYear	string		The year string value for the last observation of the selected sample.
SubEndYearValue		controls.dsge.subendyear	The row number of YearStrMatrix that locates the string which is equal SubEndYear.
UnitRootStates	vector		Vector with the positions of state variables that have a unit root.
UseDoublingAlgorithm	integer	controls.dsge.usedoublingalgorithm	Determines which method is used for computing the initial state covariance matrix for the Kalman filter. The value 1 means that the covariance matris is computed from the state equation matrices using an analytical method (vectorization), the value 2 (default) that the doubling algorithm is used, 3 that the initial covariance matrix is equal to a constant times the identity (where the constant is determined via the field StateCovConst), and 4 that exact diffuse initialization is performed.
UseOwnInitialState	boolean	controls.dsge.useowninitialstate	Determines if the state variables are initialized in the Kalman filter by user selected values (1) or the default of 0.
YearStrMatrix	string	controls.dsge.subbeginyear	String matrix with all the year values for the full sample. The first row value is therefore equal to the field BeginYear and the last to EndYear.

TABLE 13. DSGEModel fields related to forecasting.

Field	Data Type	Handle to Control	Description
AdjustPredictionPaths	boolean	controls.forecast.adjustpredictionpaths	Determines if the simulated sample paths from the prediction distribution are adjusted such that the sample mean of the sample paths is equal to the theoretical (population) mean (unit value) or not (zero value).
KsiUseCondData	boolean	controls.forecast.ksiuseconddata	Determines if the conditioning assumptions will be used when determining the mean and covariance of the distribution from which the state variables in period T are drawn for conditional forecasts from period $T+1$ and onwards.
MaxForecastHorizon	integer	controls.forecast.maxhorizon	Determines the maximum horizon for out-of-sample forecasts.
NumPredPathsValue	integer	controls.forecast.predpaths	Determines the number of prediction paths that are computed for each parameter value that the preditive distribution should account for. This field is also used whenever data are simulated from the model and a given number of paths per parameter value is needed.
PostDrawsUsageValue	integer	controls.posterior.postdrawsusage	Determines the maximum number of draws from the posterior distribution to use for simulation exercises such as estimation of the preditive distribution.
PredictionEvent	matrix		Determines the lower and the upper bound of the prediction events for all observed variables, as well as the length of the event.
RunPredictionEvent	boolean		Indicates if prediction event calculations should be performed.
ShockControlMethod	integer	controls.forecast.shockcontrols	Indicates which method to use in conditional forecasting. If the field is one, then the "values for shocks" method of directly manipulating the values for certain shocks is used, the value two means that the user has selected the Waggoner-Zha (cf. Waggoner and Zha, 1999) method of manipulation the distribution of the shocks, while three means that the subset method is used.

TABLE 14. DSGEModel fields related to optimization.

Field	Data Type	Handle to Control	Description
CheckOptimum	boolean	controls.optimize.checkoptimum	Determines whether (unit value) or not (zero value) YADA should check the curvature around of the posterior mode of the estimated parameters once it has completed posterior mode estimation.
CheckTransformedOptimum	boolean	controls.optimize	If this field is unity then YADA will only check the curvature around the posterior mode for the transformed parameters, while if zero then it will also check the curvature around the mode for the original parameters.
CsminwelExtraRuns	integer	controls.optimize.csminwelextraruns	Determines the maximum number of times the csminwel routine may be executed after the original optimization run.
FiniteDifferenceHessian	boolean	controls.optimize.finitediffhessian	This field is unity if the inverse Hessian at the posterior mode should be computed also with finite difference methods, and zero otherwise.
GridWidth	integer	controls.optimize.gridwidth	Determines the number of "standard deviations" to use when calculating a lower and an upper bound for the grid around the posterior mode. YADA uses the square root of the diagonal elements of the estimated inverse Hessian at the mode as standard deviation.
InitializeHessian	integer	controls.optimize.initcsminwel	Determines how the inverse Hessian is initialized when using the csminwel optimization routine.
MaximizeAlgorithmValue	integer	controls.optimize.maxroutine	Determines which optimization routine is used by YADA. A value of one or two means the csminwel routine, a value of three or four that newrat is applied, five or six that the Monte Carlo based routine gmhmaxlik from Dynare is run, and a value of seven or eight that the fminunc routine of the Optimization Toolbox in Matlab is used. Odd values refer to the transformed parameters and even to the original parameters.
NumberOfGridPoints	integer	controls.optimize.numbergridpoints	Determines the number of points around the posterior mode in the grid when checking the optimum.

TABLE 14. DSGEModel fields related to optimization (continued).

Field	Data Type	Handle to Control	Description
OptMaxIterationsValue	integer	controls.optimize.maximumiterations	The value determines the maximum number of iterations to use for posterior mode estimation.
OptToleranceValue	integer	controls.optimize.tolerance	Determines the tolerance level for the posterior mode estimation.
ShowProgress	boolean	controls.dsge.progress	If the field is unity then the progress dialog is displayed during, e.g., posterior mode estimation. The progress dialog is shown in Figure 6.
ShowProgressClock	boolean	controls.dsge.progressclock	A clock will be displayed on the progress dialog only when this field is unity.
StepLengthHessian	integer	controls.optimize.steplength	Determines the step length that is used when computing the inverse Hessian with finite differences.

Table 15. DSGEModel fields related to the posterior distribution.

Field	Data Type	Handle to Control	Description
BlockSize	vector		The minimum and maximum number of parameter blocks when random blocking samplers are used.
BurnIn	integer		The number of burn-in draws selected directly by the user.
BurnInValue	integer	controls.posterior.burnin	Determines the number of posterior draws at the beginning of a Markov chain that are discarded as a burn-in period.
CoverageEnd	integer	controls.dsge.coverageend	Gives the largest coverage probability that is used when computing the marginal likelihood with the modified harmonic mean estimator.
CoverageIncrement	integer	controls.dsge.coverageincrement	Gives the increment for a sequence of coverage probabilities that are used when computing the marginal likelihood with the modified harmonic mean estimator.
CoverageStart	integer	controls.dsge.coveragestart	Gives the smallest coverage probability that is used when computing the marginal likelihood with the modified harmonic mean estimator.
FixedNumParamBlocks	integer		Gives the number of parameter blocks when fixed blocking samplers are used.
InverseHessianEstimator	integer	controls.posterior.invhessian	Determines which estimator of the inverse Hessian that is used as the covariance matrix of the proposal density.
MarginalLikelihoodValue	integer	controls.dsge.marginallikelihood	Gives the algorithm(s) that is (are) used for estimating the marginal likelihood in connection with posterior sampling.
MaxCorrelationValue	integer	controls.posterior.maxcorrelation	Determines the maximum correlation that can appear in the covariance matrix of the proposal density.
MHInitialScaleFactor	integer	controls.posterior.initialscale	The value determines the constant c_0 used when computing the initial value of the parameters for the random walk Metropolis algorithm; see, e.g., Warne (2017, Section 9).
MHScaleFactor	integer	controls.posterior.scalefactor	Gives the constant c that is used when parameterizing the proposal density for the random walk Metropolis algorithm; see, e.g., Warne (2017, Section 9).
MixedDistWeightValue	integer	controls.posterior.mixeddistweight	Determines the weight for the mixed normal proposal density used in the mutation step of the SMC with likelihood tempering posterior sampler.

TABLE 15. `DSGEModel` fields related to the posterior distribution (continued).

Field	Data Type	Handle to Control	Description
`ModifiedHessian`	boolean	`controls.optimize.modifiedhessian`	Determines if the correlations from the inverse Hessian at the mode are applied when transforming the conditional standard deviations of the modified Hessian into marginal standard deviations.
`NumParamBlocks`	integer	`controls.posterior.numparamblocks`	Gives the number of parameter blocks for the mutation step of the SMC with likelihood tempering posterior sampler.
`OverwriteDraws`	boolean	`controls.dsge.overwritedraws`	When this field is unity, YADA will overwrite posterior draws on disk when they are based on the identical settings.
`ParallelChainsValue`	integer	`controls.posterior.chains`	Determines the number of Markov chains to run.
`ParameterCovMatrix`	string		Given the full path and name of the mat-file where an estimated covariance matrix for the transformed parameters is located. This matrix can be used as an estimator of the inverse Hessian.
`PosteriorDraws`	integer		The number of posterior draws selected directly by the user.
`PosteriorDrawsValue`	integer	`controls.posterior.draws`	Yields the total number of draws from the posterior distribution.
`PostDrawsPercentValue`	integer	`controls.posterior.usepostdraws`	Makes it possible to determine the percent of the the post burn-in sample draws from the posterior distribution to use when computing functions of the parameters, such as impulse responses.
`PosteriorSampler`	integer	`controls.posterior.posteriorsampler`	Determines which posterior sampler to use, random walk Metropolis with a normal proposal density (1); slice sampler (2); random walk Metropolis with a Student-t proposal density (3); fixed block RWM with a normal proposal density (4); fixed block RWM with a Student-t proposal density (5); random block RWM with a normal proposal density (5); random block RWM with a Student-t proposal density (7); SMC with likelihood tempering (8).
`RandomizeDraws`	boolean	`controls.tools.randomizedraws`	Determines if draws from the posterior should be selected randomly (unit value) or not (zero value). The field only applies to functions that use a subset of the posterior draws.

Table 15. DSGEModel fields related to the posterior distribution (continued).

Field	Data Type	Handle to Control	Description
RandomNumberValue	boolean	controls.tools.randomnumber	Determines if the random number generators are initialized with a fixed (unit value) or a variable state (zero value).
RandomWeightValue	integer	controls.posterior.randomweight	Determines the weight (between 0 and 1) on random draws relative to the posterior mode when initializing the parameters for multiple Markov chains.
ResamplingAlgorithm	integer	controls.posterior.resamplingalgorithm	Determines the resampling algorithm used in the selection step of the SMC with likelihood tempering posterior sampler.
ResamplingThresholdValue	integer	controls.posterior.resamplingthreshold	Determines the threshold value of the effective sample size for running a resampling algorithm during the selection step of the SMC with likelihood tempering posterior sampler.
SampleBatchValue	integer	controls.posterior.batch	Gives the number of sample batches per Markov chain.
ScenarioParameters	vector		Determines which estimated parameters can vary when posterior (and prior) draws are used. The field is not used by all tools.
SequentialML	boolean	controls.dsge.sequentialml	Determines if the marginal likelihood should be estimated sequentially in connection with posterior sampling.
SequentialStartIteration	integer	controls.dsge.sequentialstart	Gives the number of posterior draws used for the first sequential estimate.
SequentialStepLength	integer	controls.dsge.sequentialstep	Gives the number of posterior draws to add from one sequential estimate to the next.
SMCInitialScaleFactor	integer	controls.posterior.smcinitialscale	Determines the initial value of the scale factor for the proposal density for the SMC with likelihood tempering posterior sampler.
SMCNumMHSteps	integer	controls.posterior.smcmhsteps	Determines the number of Metropolis-Hastings steps of the mutation step of the SMC with likelihood posterior sampler.
StudenttDegFree	integer		Gives the number of degrees of freedom of the Student-t proposal density when the posterior sampler is the random walk Metropolis with such a proposal.

Table 15. `DSGEModel` fields related to the posterior distribution (continued).

Field	Data Type	Handle to Control	Description
`TargetAcceptanceRateValue`	integer	`controls.posterior.targetacceptancerate`	Determines the value of the target acceptance rate in the mutation step of the SMC posterior sampler
`TemperingLambdaValue`	integer	`controls.posterior.temperinglambda`	Determines the bending parameter (λ) for the likelihood tempering schedule of the SMC posterior sampler.
`TemperingStagesValue`	integer	`controls.posterior.temperingstages`	Determines the number of tempering stages for the SMC algorithm with likelihood tempering.

Table 16. `DSGEModel` fields related to graphical selections.

Field	Data Type	Handle to Control	Description
`ConfidenceRegionMethod`	integer		Determines if confidence sets are constructed based on equal tails (1) or highest probability density (2).
`ConfidenceBandBaseColor`	vector		Determines the base color for confidence bands. This field has three elements with real numbers between 0 and 1 representing red, green, and blue (RGB). The base color is multiplied by suitable scalars between 0 and 1 to produce shades of the base color. The default is [1 1 1], i.e., white.
`IRHorizon`	integer	`controls.tools.irhorizon`	Gives the value for the number of periods for impulse responses and variance decompositions.
`KernelDensityEstimator`	string		Gives a short name of the kernel density estimator for prior draws. The possible values are `kepan` (Epanechnikov), `knorm` (normal), `krect` (rectangular), `ktria` (triangular), `kbiwe` (bi-weight), `ktriw` (tri-weight), `klapl` (Laplace), and `klogi` (logistic).
`KernelDensityValue`	integer	`controls.graphics.kerneldensity`	Determines which kernel density estimation method to use for prior draws.
`ObsVarGroupColors`	matrix		Each row is a vector with an RGB triple (0-1) for a certain observed variable group. The number of rows is equal to the number of observed variable groups; see also Table 10.
`PosteriorDensityValue`	integer	`controls.graphics.posteriorkernel`	Gives the choice of kernel density estimator for posterior draws.
`PriorKernel`	boolean	`controls.graphics.priorkernel`	When the field is unity (zero) the prior density is computed with a kernel (grid) density estimator.
`ShockColors`	matrix		Each row is a vector with an RGB triple (0-1) for a certain shock. The number of rows is equal to the number of shocks.
`ShockGroupColors`	matrix		Each row is a vector with an RGB triple (0-1) for a certain shock group. The number of rows is equal to the number of shock groups; see also Table 11.

Table 17. DSGEModel fields related to DSGE-VAR models.

Field	Data Type	Description
DSGEVARShocks	vector	Determines which of the economic shocks in the DSGE model that will also be included in the DSGE-VAR. The number of such shocks must be equal to the number of observed variables or equal to zero. In the latter case, YADA will not support DSGE-VAR functions that require structural shocks.
JointLambda	vector	Determines which of the λ values in Lambda that will be used for joint posterior mode estimation. The default is all values.
Lambda	vector	Determines which possible values of the λ hyperparameter that can be used for DSGE-VAR models. The default is [0.25 0.5 0.75 1 5 Inf].
MarginalLambda	vector	Determines which of the λ values in Lambda that will be used for marginal posterior mode estimation. The default is all values.

Table 18. Miscellaneous `DSGEModel` fields.

Field	Data Type	Handle to Control	Description
`AdditionalDraws`	integer		Determines the percentage of additional draws of shocks, initial states, and measurement errors to make when running the forward-back shooting algorithm under stoachastic simulation of the DSGE model subject to the zero lower bound of the monetary policy rate.
`MonteCarloFilterDraws`	integer		Determines the default number of draws from the prior distribution to use for Monte Carlo filtering.
`ParameterScenario`	vector		Vector with ones and zero that determines which parameters are changed in a parameter scenario.
`ParameterScenarioValue`	integer		Determines the default start period of a parameter scenario.
`REqPosition`	integer		Position of the monetary policy rule among the DSGE model equations.
`RiccatiMaxIterations`	integer	`controls.tools.riccatiiteration`	Gives the maximum number of iterations when running the solver for the Riccati equations.
`RiccatiToleranceValue`	integer	`controls.tools.riccatitolerance`	Determines the tolerance level that is used for the Riccati equation solver. The tolerance level is equal to $10^{-(x+1)}$, with x being the value of the field.
`RtildePosition`	integer		Position of the monetary policy rate among the state variables in the DSGE model.
`ZLBSampleT`	integer		Determines the length of the prediction sample during which the zero lower bound may be binding.

TABLE 19. DSGEModel fields related to a Bayesian VAR model with a steady-state prior.

Field	Data Type	Handle to Control	Description
BVARLags	integer	controls.bvar.lag	Determines the lag order of VAR and DSGE-VAR models.
CrossEqTightnessValue	integer	controls.bvar.crosseqtightness	Gives the cross-equation tightness hyperparameter when a Minnesota-style prior is used for parameters on lagged variables.
HarmonicLagDecayValue	integer	controls.bvar.harmoniclagdecay	Determines the harmonic lag decay hyperparameter for proper priors of the parameters on lagged variables.
OmegaPriorType	integer	controls.bvar.omegapriortype	Gives the type of prior to use for the covariance matrix of the VAR residuals.
OverallTightnessValue	integer	controls.bvar.overalltightness	Determines the overall tightness hyperparameter for proper priors of the parameters on lagged variables.
PriorDiffMeanValue	integer	controls.bvar.priordiffmean	Gives the mean of parameters on the first own lag of first differences variables for proper priors.
PriorLevelMeanValue	integer	controls.bvar.priorlevelmean	Gives the mean of parameters on the first own lag of levels variables for proper priors.
PriorType	integer	controls.bvar.priortype	Determines the type of prior that is used for parameters on lagged variables.
StationaryVAR	boolean	controls.bvar.stationaryvar	Determines if the VAR parameter posterior draws are required to be consistent with stationarity (1) or not (0).
SteadyStatePriorFile	string	controls.bvar.steadystatefile	Gives the full path and name of the steady-state parameter prior file.
VarianceTightnessValue	integer	controls.bvar.variancetightness	Determines the value of the variance tightness hyperparameter when the inverse Wishart prior is used for the covariance matrix of the residuals and the distribution is parameterized by this hyperparameter times the identity.
WishartDFValue	integer	controls.bvar.wishartdf	Gives the number of degrees of freedom of the inverse Wishart prior.
WishartType	integer	controls.bvar.wisharttype	Determines how the location matrix of the inverse Wishart prior is parameterized.

References

An, S. and Schorfheide, F. (2007), "Bayesian Analysis of DSGE Models," *Econometric Reviews*, 26, 113–172.

Anderson, B. D. O. and Moore, J. M. (1979), *Optimal Filtering*, Prentice-Hall, Englewook Cliffs.

Anderson, G. and Moore, G. (1985), "A Linear Algebraic Procedure for Solving Linear Perfect Foresight Models," *Economics Letters*, 17, 247–252.

Anderson, G. S. (2008), "Solving Linear Rational Expectations Models: A Horse Race," *Computational Economics*, 31, 95–113.

Anderson, G. S. (2010), "A Reliable and Computationally Efficient Algorithm for Imposing the Saddle Point Property in Dynamic Models," *Journal of Economic Dynamics and Control*, 34, 472–489.

Bernardo, J. M. and Smith, A. F. M. (2000), *Bayesian Theory*, John Wiley, Chichester.

Chib, S. and Jeliazkov, I. (2001), "Marginal Likelihood from the Metropolis-Hastings Output," *Journal of the American Statistical Association*, 96, 270–281.

Christoffel, K., Coenen, G., and Warne, A. (2008), "The New Area-Wide Model of the Euro Area: A Micro-Founded Open-Economy Model for Forecasting and Policy Analysis," ECB Working Paper Series No. 944.

Croushore, D. (2011a), "Forecasting with Real-Time Data Vintages," in M. P. Clements and D. F. Hendry (Editors), *The Oxford Handbook of Economic Forecasting*, 247–267, Oxford University Press, New York.

Croushore, D. (2011b), "Frontiers of Real-Time Data Analysis," *Journal of Economic Literature*, 49, 72–110.

Croushore, D. and Stark, T. (2001), "A Real-Time Data Set for Macroeconomists," *Journal of Econometrics*, 105, 111–130.

Del Negro, M. and Schorfheide, F. (2004), "Priors from General Equilibrium Models," *International Economic Review*, 45, 643–673.

Del Negro, M. and Schorfheide, F. (2006), "How Good Is What You've Got? DSGE-VAR as a Toolkit for Evaluating DSGE Models," *Federal Reserve Bank of Atlanta Economic Review*, 91, 21–37.

Del Negro, M. and Schorfheide, F. (2009), "Monetary Policy Analysis with Potentially Misspecified Models," *American Economic Review*, 99, 1415–1450.

Del Negro, M., Schorfheide, F., Smets, F., and Wouters, R. (2007), "On the Fit of New-Keynesian Models," *Journal of Business & Economic Statistics*, 25, 123–143, with discussion, p. 143–162.

Durbin, J. and Koopman, S. J. (2012), *Time Series Analysis by State Space Methods*, Oxford University Press, Oxford, 2nd edition.

Fernández-Villaverde, J., Rubio-Ramírez, J. F., Sargent, T. J., and Watson, M. W. (2007), "ABCs (and Ds) of Understanding VARs," *American Economic Review*, 97, 1021–1026.

Geweke, J. (1999), "Using Simulation Methods for Bayesian Econometric Models: Inference, Development, and Communication," *Econometric Reviews*, 18, 1–73.

Hamilton, J. D. (1994), *Time Series Analysis*, Princeton University Press, Princeton.

Harvey, A. C. (1989), *Forecasting, Structural Time Series Models and the Kalman Filter*, Cambridge University Press, Cambridge.

Kalman, R. E. (1960), "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, 82, 35–45.

Klein, P. (2000), "Using the Generalized Schur Form to Solve a Multivariate Linear Rational Expectations Model," *Journal of Economic Dynamics and Control*, 24, 1405–1423.

Koopman, S. J. and Durbin, J. (2000), "Fast Filtering and Smoothing for State Space Models," *Journal of Time Series Analysis*, 21, 281–296.

Morf, M., Sidhu, G. S., and Kailath, T. (1974), "Some New Algorithms for Recursive Estimation in Constant, Linear, Discrete-Time Systems," *IEEE Transaction on Automatic Control*, 19, 315–323.

Ratto, M. (2008), "Analysing DSGE Models with Global Sensitivity Analysis," *Computational Economics*, 31, 115–139.

Sheather, S. J. and Jones, M. C. (1991), "A Reliable Data-Based Bandwidth Selection Method for Kernel Density Estimation," *Journal of the Royal Statistical Society Series B*, 53, 683–690.

Silverman, B. (1986), *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, London.

Sims, C. A. (2002), "Solving Linear Rational Expectations Models," *Computational Economics*, 20, 1–20.

Sköld, M. and Roberts, G. O. (2003), "Density Estimation for the Metropolis-Hastings Algorithm," *Scandinavian Journal of Statistics*, 30, 699–718.

Villani, M. (2009), "Steady-State Priors for Vector Autoregressions," *Journal of Applied Econometrics*, 24, 630–650.

Waggoner, D. F. and Zha, T. (1999), "Conditional Forecasts in Dynamic Multivariate Models," *Review of Economics and Statistics*, 81, 639–651.

Warne, A. (2017), "YADA Manual — Computational Details," Manuscript, European Central Bank. Available with the YADA distribution.