Taylor Projection - A User Guide

Oren Levintal

June 30, 2016

1 Introduction

This note explains how to implement the MATLAB package provided by Levintal (2016) to solve a simple neoclassical growth model by Taylor projection.

2 Installation

The package is implemented by MATLAB/MEX files. You will need MATLAB, MATLAB symbolic toolbox, MATLAB optimization toolbox, MATLAB control system toolbox (optional), and Intel Fortran compiler (optional) that is configured to work with MATLAB.[1] The package was tested on MATLAB version R2014b with an Intel Visual Fortran Composer XE 2013 on Windows 64bit.

To install the package follow these steps:

- Add the folder `Solution_Methods` and its subfolders to the search path. For example, if the folder location is `c:/Solution_Methods` type:

[1] see `www.mathworks.com`

```
addpath(genpath('c:/Solution_Methods'))
```

in the command prompt.

- **MEX files**: The package includes MEX files that were compiled on WINDOWS10 (64bit). In case that these files do not work on your computer go to folder `Solution_Methods\Taylor_Projection\MEX_files` and run the file `do_mex.m` to compile the FORTRAN source codes on your system. To do so, you will need an Intel Fortran compiler that works with MATLAB.

3 Running the program

The folder `Simple_Example\Taylor projection` solves the neoclassical growth model in two stages. The first stage is performed by `prepare_model.m`. This file differentiates the model and prepares files and data that are used when the model is solved. The second stage is performed by `solve_model.m`, which solves the model for given parameter values.

4 The model

The model is defined in `prepare_model.m`. The notation of the model follows Schmitt-Grohé and Uribe (2004):

$$E_t f\left(y_{t+1}, y_t, x_{t+1}, x_t\right) = 0, \tag{1}$$

$$x_{t+1} = h\left(x_t\right) + \eta \epsilon_{t+1}, \tag{2}$$

$$y_t = g\left(x_t\right) \tag{3}$$

$$h\left(x_t\right) = \begin{pmatrix} \tilde{h}\left(x_t\right) \\ \Phi\left(x_t^2\right) \end{pmatrix}, \ \eta = \begin{pmatrix} 0 \\ \tilde{\eta} \end{pmatrix}. \tag{4}$$

The state and control variables are defined by the symbolic variables `x` and `y` for current period, and by `xp` and `yp` for next period. The equilibrium conditions are defined by the symbolic variable `f`. It is recommended to define `f` in a unit-free form to avoid scaling problems. This will also help later to compute the unit-free model residuals. The parameters are defined by the symbolic variable `symparams`, and the matrix `eta` is the same matrix as in Schmitt-Grohé and Uribe (2004).

The code differs from Schmitt-Grohé and Uribe (2004) by allowing to define the function `Phi`. This is a symbolic expression that defines the expected value of future exogenous state variables as a function of current state variables, i.e., the function $\Phi\left(x_t^2\right)$, which is the lower block of $h\left(x\right)$. Importantly, vector `x` should be defined `x=[x1,x2]`, where the first block is the endogenous state variables and the second block is the exogenous state variables.

4.1 Auxiliary variables

The code allows to define the model by using auxiliary variables. It is not necessary but highly recommended. The auxiliary variables are substituted out in the final system. For instance, you can declare a new symbolic variable `logy` and define the symbolic expression `logy_=log(y)`. Then, you can use `logy` throughout the code, but the algorithm will know to substitute it with `logy_`. Importantly, when the algorithm differentiates the model it DOES NOT substitute the variables. Instead, it applies the chain rule. This makes the differentiation much more efficient.

For example, consider the Euler condition:

```
Euler=BETA*exp(GAMMA*(logc-logcp))*(ALPHA*exp(logap+(ALPHA-1)*logkp)+1-DELTA)-1;
```

It can be defined also in the following way:

```
logmpkp_=log(ALPHA)+logap+(ALPHA-1)*logkp;
mpkp_=exp(logmpkp);
logmp_=log(BETA)+GAMMA*(logc-logcp);
mp_=exp(logmp);
Euler=mp*(mpkp+1-DELTA)-1;
```

Note that `mp` is the stochastic discount factor from current to future period, and `mpkp` is future marginal product of capital. Hence, the auxiliary variables make the model more economically meaningful.

The auxiliary variables are defined by vector `auxvars` and the auxiliary functions by `auxfuns`. In the example above, these vectors would be defined as follows:

```
auxvars=[logmpkp;mpkp;logmp;mp]

auxfuns=[logmpkp_;mpkp_;logmp_;mp_]
```

If you type `[auxvars,auxfuns]` you get all the auxiliary equations. The algorithm applies the chain rule on these equations to obtain the required derivatives.

It is useful to denote the auxiliary function by the same name as the auxiliary variable, with some common suffix (e.g. underscore).

4.2 Differentiating the model

After the model is defined, the function `prepare_tp` is called. This function differentiates the model and generates files and data. The files are stored in an automatically generated folder `files`. Data are stored in the output variable `model`. This variable is used later to solve the model so it has to be saved.

When you call `prepare_tp`, you have to specify the order of the system by the variable `order`. For instance, `order=3` will produce a third order Taylor projection system (which is the largest order available by the package). In this case, the policy

functions are approximated by 3rd order polynomials.

The package can also compute a perturbation solution (up to 4th order), which can be used as the initial guess. By default, the order of the perturbation solution is the same as the order of the Taylor projection solution. However, if you are interested in a perturbation solution of a different order, you should specify it as the second element of `order`. For instance `order=[2,4]` will produce a 2nd-order Taylor projection solution and a 4th-order perturbation solution. A higher-order perturbation solution would be required if the low order solution is not sufficiently accurate as an initial guess. For example, in models with strong volatility, the high-order derivatives w.r.t the perturbation parameter may be economically important to get a sufficiently accurate initial guess.

5 Solve the model

Open and run the file `solve_model.m` in folder `Simple_Example\Taylor projection`. This file solves the model by Taylor projection, using a perturbation solution as the initial guess.

The file starts by adding the folder `files` to the search path. This folder was generated automatically in the first stage by the function `prepare_tp`. It stores files that are necessary to solve the model. Second, load the variable `model` that was generated in the first step.

Next, you should provide the discrete distribution of the $n_\epsilon \times 1$ vector of shocks ϵ. The discrete distribution is defined by the realization matrix `nep` and the probability

vector `P`. The matrix `nep` is of size $n_\epsilon \times n_{nodes}$, where n_{nodes} is the number of nodes of the discrete distribution. The vector `P` is of length n_{nodes}. For instance, `nep(:,i)` is a realization with probability `P(i)`. If the shocks are continuous (as is usually the case), you should discretize them. The file `solve_model.m` uses monomial rules written by Judd, Maliar, Maliar, and Valero (2014).

By default, the solution uses an exact Jacobian. If you have a large model, you may want to use the approximate Jacobian discussed in Levintal (2016). To do so, define `model.jacobian='approximate'`.

We are now ready to solve the model. First, we need to obtain an initial guess. The package can compute a perturbation solution for the initial guess. This solution is usually very good for solving the model near the steady state.

5.1 Perturbation solution

A perturbation solution is obtained by the function `get_pert`. The inputs include parameter values (`params`), steady state values (`nxss`, `nyss`), the matrix η (`eta`) and the cross moments (`M`). The cross moments are obtained by the function `get_moments`. This function computes the moments from the discrete distribution. If you know the analytic cross moments, you can use them instead.[2] For example, if the shocks are independent standard normal you can use the function `gaussian_moments(n_e)` that computes the cross-moments of a vector of `n_e` standard normal iid shocks.

In addition, you need to choose a solver for the Sylvester equation solved by the perturbation algorithm. There are three possibilities: 1. `'vectorize'` is good only

[2] `M.M2` is $E\epsilon^{\otimes 2}$, `M.M3` is $E\epsilon^{\otimes 2}$, and `M.M4` is $E\epsilon^{\otimes 4}$.

for small models. 2. `'dlyap'` is good for larger models, but requires the MATLAB control system toolbox. 3. `'gensylv'` is recommended for very large models. It applies the algorithm of Kameník (2005), which is provided by `Dynare` as a compiled MEX file.[3] The current package includes the WINDOWS (64bit) version of this MEX file. If you work on a different operating system, you can get the appropriate version by downloading `Dynare` from `www.dynare.org` and searching for the MEX file `gensylv`. Then, add this file to folder `Solution_Methods\Perturbation\gensylv`.

The function `get_pert` generates 4 outputs. The first output `derivs` contains the derivatives of g and h w.r.t x and σ, where σ is the perturbation parameter.[4] The second output `stoch_pert` is a standard perturbation solution transformed into a vector of unique polynomial coefficients. The third output `nonstoch_pert` is a perturbation solution of a deterministic model in the form of unique polynomial coefficients. This solution does not correct for the model volatility. Both vectors `stoch_pert` and `nonstoch_pert` can be used as an initial guess. However, in models with strong volatility the latter may not be sufficiently accurate. If `stoch_pert` is also not sufficiently accurate, try to produce a higher-order perturbation solution, as explained in section 4.2.

If this is not good either, try to start with a model with no volatility, namely, a model where η is all zero. The vector `nonstoch_pert` should be the exact solution of the nonlinear system at the steady state for this particular case. Then, you can increase η gradually to the required value and solve the model at each step using the

[3]For a description of `Dynare` see Adjemian, Bastani, Juillard, Mihoubi, Ratto, and Villemot (2011).

[4]Note that the perturbation parameter is added to the vector of state variables `x`, as done in Levintal (2015). For example, the derivatives of g w.r.t σ are in the last column of `derivs.gx`.

previous solution as the initial guess (as in homotopy).

5.2 Taylor projection solution

Having the initial guess, we are now ready to solve the model by the function `tpsolve`. The arguments of the function are: (1) the initial guess `coeffs`; (2) the point `x0` at which we solve the model; (3) the precomputed variable `model`; (4) the vector of parameter values `params`; (5) the matrix `eta`; (6) the point `c0` at which the initial guess is centered;[5] (7) the discretized shocks `nep` and probabilities `P`; (8) tolerance parameters of the Newton solver: `tolX`, `tolF` and `maxiter`. The function returns the following outputs: (1) the solution `ncoeffs`; (2) the variable `model` which contains some additional data computed during the first iteration of the Newton method (see below).

`tpsolve` solves the nonlinear system by a simple Newton solver. If the algorithm does not converge within the specified tolerance parameters, it switches automatically to `fsolve`, with the same tolerance parameters. You can control the tolerance of `fsolve` by the function `optimoptions`. You can also choose the `lsqnonlin` algorithm instead (as shown in the example). `lsqnonlin` is a nonlinear least squares algorithm. It is more appropriate when the nonlinear system does not have an exact solution.

[5]If the initial guess is a perturbation solution obtained by `get_pert`, then `c0` should be the steady state.

5.3 Moving to adjacent points

Once the model is solved at x_0, the solution can be used as an initial guess for solving the model at an adjacent point x_1. Note that you do not need to change the center c_0 when you move to other points. The algorithm will do it automatically, and return the solution as a power series centered at c_0.

6 More options

6.1 The variable `model`

The variable `model` stores data that speed up the computation of the solution. Some of the data were generated in the first step by the function `prepare_tp`. The remaining data are generated on the first call to `get_pert` and the first Newton iteration of `tpsolve`. After you run these functions for the first time and return the variable `model`, you can save it. Further calls to these (or other) functions will not generate any new data.[6]

6.2 The nonlinear system

The function `tp` is the main part of the algorithm. It computes the nonlinear system T and the Jacobian J by the command:

```
[T,J,model]=tp(coeffs,x0,model,params,eta,c0,nep,P);
```

[6] If a perturbation solution is not used, you just need to call `tpsolve`.

The first call to this function (like the first call to `tpsolve`) computes all the necessary data for `model`. Note that if $x_0 \neq c_0$, the function automatically shifts the center of the initial guess to x_0. It issues a warning, because the Jacobian in this case refers to the shifted coefficients, not the original coefficients.

6.3 Policy functions and model residuals

You can use the function `eval_model` to evaluate the model residuals over a grid. The function also outputs the policy functions, the function Φ and the auxiliary variables.

If you only want to evaluate the policy functions over a grid `x_grid`, use `eval_policy`.

6.4 The variable `coeffs`

The solution is given by a vector `coeffs` of length n_Θ. This vector stores the unique Taylor coefficients of n_f power series centered at `c0`. By typing:

```
coeffs=reshape(coeffs,model.n_f,model.n_b),
```

we get a $n_f \times n_b$ matrix, where n_f denotes the number of endogenous variables and n_b is the size of the basis function. The upper block `coeffs(1:model.n_y,:)` contains the solution of the control variables y_t as a function of x_t, and the lower block `coeffs(model.n_y+1:end,:)` is the solution of the endogenous state variables x^1_{t+1} as a function of x_t.

The solution is given in the form of unique Taylor coefficients about `c0` (the

center of the power series). For example, a second order solution to $g(x)$ contains the following **unique** Taylor coefficients:

$$\left[g(x), \frac{\partial g(x)}{\partial x_1}, \ldots, \frac{\partial g(x)}{\partial x_{n_x}}, \frac{1}{2!}\frac{\partial^2 g(x)}{\partial x_1 \partial x_1}, \frac{1}{2!}\frac{\partial^2 g(x)}{\partial x_2 \partial x_1}, \ldots, \frac{1}{2!}\frac{\partial^2 g(x)}{\partial x_{n_x} \partial x_1}, \frac{1}{2!}\frac{\partial^2 g(x)}{\partial x_2 \partial x_2}, \frac{1}{2!}\frac{\partial^2 g(x)}{\partial x_3 \partial x_2}, \ldots, \frac{1}{2!}\frac{\partial^2 g(x)}{\partial x_{n_x} \partial x_{n_x}}\right]$$

where g and all its derivatives are evaluated at `c0`. Note that these are only the **unique** coefficients. To get the full Taylor coefficients you need to type (it can also be done automatically, as explained below):

```
coeffs=reshape(coeffs,model.n_f,[]);
GH0=coeffs(:,1);
GH1=coeffs(:,2:1+model.n_x);
GH2=coeffs(:,2+model.n_x:1+model.n_x+nchoosek(model.n_x+1,2))*model.W{2};
GH3=coeffs(:,2+model.n_x+nchoosek(model.n_x+1,2):1+model.n_x+...
    nchoosek(model.n_x+1,2)+nchoosek(model.n_x+2,3))*model.W{3};
```

Then, you can evaluate these Taylor series at any point `x0` by typing:

```
vars=GH0+GH1*(x0-c0)+GH2*kron(x0-c0,x0-c0)+GH3*kron(x0-c0,kron(x0-c0,x0-c0));
```

These are the values of the endogenous variables. The endogenous control variables are in `vars(1:model.n_y)` and the endogenous state variables are in `vars(model.n_y+1:end)`. However, a more efficient way to get the endogenous variables is to call the function `eval_policy`, as shown in the example.

6.5 Transform derivatives into unique Taylor coefficients

Suppose you have a guess for the functions g and $\tilde{h}$ and you want to use it as an initial guess. You can use the function `derivs2coeffs` to convert the guess into a column vector of the unique Taylor coefficients. To do so, type:

```
coeffs=derivs2coeffs(model,[g0;h0],[gx;hx],[gxx;hxx],[gxxx;hxxx]).
```

Here, `[g0;h0]` is the value of the guess at the point of interest x_0. `[gx;hx]` is a matrix of first derivatives at x_0. `[gxx;hxx]` is a tensor of second derivatives, and so on. The vector `coeffs` is the unique Taylor coefficients about x_0. You can use it as an initial guess for `tpsolve` by setting $c_0 = x_0$.

References

Adjemian, S., H. Bastani, M. Juillard, F. Mihoubi, M. Ratto, and S. Villemot (2011): "Dynare: Reference Manual, Version 4," Dynare Working Papers 1, CEPREMAP.

Judd, K. L., L. Maliar, S. Maliar, and R. Valero (2014): "Smolyak Method for Solving Dynamic Economic Models: Lagrange Interpolation, Anisotropic Grid and Adaptive Domain," *Journal of Economic Dynamics and Control*, 44, 92–123.

Kameník, O. (2005): "Solving SDGE Models: A New Algorithm for the Sylvester Equation," *Computational Economics*, 1412.8659v1, 167–187.

Levintal, O. (2015): "Fifth Order Perturbation Solution to DSGE Models," *Manuscript, Interdisciplinary Center Herzliya*.

——— (2016): "Taylor Projection: A New Solution Method for Dynamic General Equilibrium Models," *Manuscript, Interdisciplinary Center Herzliya*.

Schmitt-Grohé, S., and M. Uribe (2004): "Solving Dynamic General Equilibrium Models Using a Second-Order Approximation to the Policy Function," *Journal of Economic Dynamics and Control*, 28, 755–775.