Evaluation of and interface for Dynare++

Tamás K Papp

June 10, 2006

1 About Dynare++

When we met on Wednesday, I mentioned that I heard about two other k-order approximation frameworks, one written by Hehui Jin and another by Gary Anderson and others. It turns out Jin's framework is certainly not an option for us, I could not even locate his homepage, let alone the software, the only link to it is on Dynare++'s website, and it is broken.

Gary Anderson, Eric Swanson, Andrew Levin and lately Jinil Kim appear to be working on another framework for k-order approximation. All the information I found about this is in this presentation: `http://www.stanford.edu/group/SITE/papers2004/Anderson04.pdf`. Originally, the first three of these authors were developing PerturbationAIM,[1] which was written in Mathematica. The presentation suggests that it is now rewritten in Matlab, and is more efficient. I could not locate any of this code, perhaps you could contact the authors. But it appears that for the moment, we need to use Dynare++.

Dynare++ is written completely in C++. I don't think C++ is a good language, so I was a bit prejudiced against this program from the start. Some of my expectations were verified: the authors got carried away with developing a really "nice" OO framework using virtual classes and fancy inheritance for many objects, this made it hard to find concrete things, eg the implementation for storing the tensors compactly, see more on that later. Also, the program sometimes segfaults if there is something it doesn't like, mostly malformed model files.

Handling errors is not a strong side, perhaps this is because the program was made to run by an economist who would read the log file (they call it the journal file), so I have to catch errors (eg indeterminacy of the first order

[1] http://www.ericswanson.us/perturbation.html

model) by grepping through the logfile. It seems to work, but there is always the chance that there is some error message that I am not grepping for.

The user community of Dynare is not overwhelmingly helpful (for example, if compared to the R mailing lists). I posted a question on the forums and had to answer it myself two days later.

However, considering these shortcomings, Dynare++ should be able to do the job of finding the approximation for the transition function if we are careful.

2 The R interface

Below, I explain the interface I constructed for calling Dynare++ from R. The file should be well documented, but I explain design choices here. The code listed below is in the file `dynread.r`.

2.1 makedynframework

First, we need to setup a *framework* (something like the instance of a class, currently implemented as a list) for calling dynare. I store all relevant environment variables (eg the path to the program), model characteristics (equations, variables names) and other things relevant for calling dynare here. The idea is that the user just needs to supply this list and the parameters to functions below. The parameters are

tempdir The temporary directory for writing out the file. I suggest that you setup a ramdisk for this, it should be faster. The model file is named `tmp.mod`, so the journal will be in `tmp.jnl` and the Matlab results in `tmp.mat`. Default is `/tmp`.

dynpath Sets the path for the `dynare++` binary.

var, varexo Names of variables, and exogenous variables. Character vectors.

parameters Names of the parameters. Remember the order, as you will need to give parameters in this order from now on.

model The model equation. If you give a filename, it will be read from a file, and no further modification will be made. If you give a vector of strings, it will append `model;` and `end;`, and put semicolons at the end of the lines, so don't include these.

initval The initial values of the variables. Since we only need the approximation, this is irrelevant, but the dimension should be correct.

order The order of the approximation.

dynparams Command line parameters to the program, I set the number of simulations to zero by default.

statevars Not used at the moment, should be the state variables. See matchvarnames below.

The function does some basic checking, and returns a list.

2.2 calldynare & auxiliary functions

The function calldynare will call dynare++ with the given parameters (including a variance matrix for the error terms). First it writes a syntactically correct model file (using the auxiliary functions), then erases journal and mat files that remained from a possible previous iteration. This is important, as the absence of these files will be an indicator that something has failed. This will appear as an error, if you don't want this to stop the program, then you need to check for it or catch it in R (using **try**). However, if a journal file is missing, then dynare++ misbehaves so this should be considered a fatal error.

We are checking for indeterminacy by grepping the journal file for the phrase "model not stable".[2] This function returns a list, with the following elements:

dyn this is all the data from the Matlab file

isunstable set TRUE if we detected indeterminacy

dynout the output dynare++ sent to the command line

2.3 unfoldtensor & auxiliary functions

The most important thing we need from dynare++ is the approximation to the transition function. This is a list of tensors with increasing order, stored

[2]From the source code in `kord/first_order.cweb`, we know that the two possible messages are "Blanchard-Kahn condition satisfied, model stable" and "Blanchard-Kahn condition not satisfied, model not stable".

as **dyn**.g.1, **dyn**.g.2, ..., and the "stochastic steady state"[3] in **dyn**.ss.[4]

The tensors are "folded" into these matrix: each row of the matrix corresponds to one variable of the transition function, and the columns correspond to elements in the tensor. The traversal mode for the tensor is going through the minor dimensions in the inner loop, and discarding repetitions.[5]. Mapping this back to an array to contain the tensor ("unfolding") is quite tricky, and is ultimately done by unfoldtensor. This function takes a *row* of the **dyn**.g.i matrices as its first argument, the order of this term (i here), and the total number of state variables. It returns a k-dimensional array.

2.4 matchvarnames

Quite awkwardly, dynare++ may change the order of the variables from the one supplied in the model file (the `var` line). Fortunately, it returns the new order in **dyn**.vars and **dyn**.state.vars, sometimes these are padded with spaces. The matchvarnames function establishes a mapping (a permutation) between these two and any order we would like to choose (it makes sense to store the latter in dynfw).

dynread.r

```
library("R.matlab")

### Makes a list of all relevant parameters that characterize the
### model we are iteration on, and also some environment variables
### relevant for calling dynare.
makedynframework <- function(tempdir="/tmp", # ramdisk recommended
                             dynpath, # path of dynare
                             var, # vector of variables (char)
                             varexo, # exogenous variables (char)
                             parameters, # parameter names (char)
                             model, # list of strings or filename
                             initval, # make up something
                             order=2, # order of approximation
                             dynparams="--sim␣0␣--per␣0",
                                                 # parameters for dynare
```

[3] I still don't know what this means. Juillard and Kamenik have a paper on it (`http://ideas.repec.org/p/sce/scecf5/106.html`) but I haven't been able to obtain a copy so far. It is certainly not the approach of Judd and Jin (also used by Anderson at al) which first does the expansion for deterministic models and then solves uncertain ones around that.

[4] All these variables are in the **dyn** element of the list returned by calldynare. The library **R**.matlab replaces the _ with . in the variable names.

[5] Explained on page 90 (section 230) of `http://www.cepremap.cnrs.fr/juillard/mambo/download/manual/dynare++/tl.pdf`

```
                                statevars # state variables
                                ) {
  ## checks
  stopifnot(order >= 1)
  ## read model if given as a filename, otherwise append "model;",
  ## "end;", and semicolon at the end of each line
  if (length(model)==1) # this is a filename
    model <- readLines(model)
  else { # not a filename
    model <- sapply(model,function(a) sprintf("%s;",a)) # semicolons
    model <- c("model;",model,"end;")
  }
  ## construct framework
  dynfwork <- list(tempdir=tempdir,
                   dynpath=dynpath,
                   var=var,
                   varexo=varexo,
                   parameters=parameters,
                   model=model,
                   initval=initval,
                   order=as.integer(order),
                   dynparams=dynparams,
                   statevars=statevars
                   )
}

### Output lists of the form "keyword a, b, c, d;"
catenum <- function(con,keyword,var,append=TRUE) {
    cat(file=con,keyword,sapply(var[-length(var)], # a, b, c, ...
          function(a) sprintf("%s,",a)),
        sprintf("%s;\n",var[length(var)]),append=append) # d; (last variable)
}

### Output lists of the form "a=1; b=2, c=3;", with linebreaks
catvalues <- function(con, varnames, values) {
  stopifnot(length(varnames)==length(values))
  writeLines(con=con,mapply(function(a,b)
        sprintf("%s=%f;",a,b),varnames,values))
}

## This function generates a model file with the given parameters,
## calls dynare, analyzes the journal (eg for indeterminacy), reads
## the matrices in Matlab format, and returns a list.
calldynare <- function(dynfw,params,vcovmat) {
  with(dynfw, {
    ## test for consistency
    stopifnot(is.matrix(vcovmat) && (nrow(vcovmat)==ncol(vcovmat))
              && is.finite(vcovmat))
    stopifnot(is.vector(params) && is.finite(params))
```

```
    ## construct model file
    con <- file(sprintf("%s/tmp.mod",tempdir),"w") # open connection
    catenum(con,"var",var)
    catenum(con,"varexo",varexo)
    catenum(con,"parameters",parameters)
    catvalues(con,parameters,params) # parameters
    cat(file=con,"\n")
    writeLines(model,con) # model
    cat(file=con,"\n\ninitval;\n")
    catvalues(con,var,initval) # initval
    cat(file=con,"end;\n\nvcov␣=␣[\n")
    for (i in 1:nrow(vcovmat)) { # variance matrix
      cat("␣␣", vcovmat[i,], file=con)
      if (i != nrow(vcovmat)) # semicolon for all but last line
        cat(";", file=con)
    }
    cat(file=con,"\n];\n\n")
    cat(file=con,sprintf("order=%d;\n",order)) # order
    close(con) # close connection
    ## remove journal and mat files now (if any remained from previous
    ## calls), so if dynare fails, reading them will generate and error
    unlink(sprintf("%s/tmp.jnl",tempdir))
    unlink(sprintf("%s/tmp.mat",tempdir))
    ## call dynare
    dynout <- system(sprintf("%s␣%s␣%s/tmp.mod",dynpath,dynparams,
                             tempdir),TRUE,FALSE)
    ## analyze the journal file
    journal <- readLines(sprintf("%s/tmp.jnl",tempdir))
    ## the following is just an example
    isunstable <- length(grep("model␣not␣stable", journal)) > 0
    ## read the results, return as a list
    if (isunstable)
      dyn <- NULL
    else
      dyn <- readMat(sprintf("%s/tmp.mat",tempdir))
    list(dyn=dyn,isunstable=isunstable,dynout=dynout)
  })
}

## We need the mapping from the folded tensor to the unfolded one.
## The folding algorithm is explained in Section 230 (page 91) of the
## documentation file tl.pdf, url:
## http://www.cepremap.cnrs.fr/juillard/mambo/download/manual/dynare++/tl.
    pdf
## The function is actually a wrapper, to account for the fact that R
## starts indices from 1, not 0. ii is a vector if indices (starting
## from ONE!), n is the total number of variables. We also need to
## sort the variables.
```

```
getoffset <- function(ii,n) {
  getoffsetinternal(sort(ii)-1,n)+1
}

getoffsetinternal <- function(ii,n) { # this is for indexing from 0
  k <- length(ii)
  if (k==0)
    0
  else
    choose(n+k-1,k)-choose(n-ii[1]+k-1,k)+
      getoffsetinternal(ii[-1]-ii[1],n-ii[1])
}

## ## I wrote some basic functions to test the code above, they are not
## ## needed for anything else.
## ## This section is commented out, as I only used it for testing.
## genoffset <- function(n,k,i) {
## if (k==1)
## matrix(i:n,n-i+1,1)
## else
## do.call(rbind,lapply(i:n,function (j) cbind(j,genoffset(n,k-1,j))))
## }
## testoffset <- function(n,k) {
## oo <- genoffset(n,k,1)
## trueo <- as.vector(apply(oo,1,function(o)
## getoffset(o[sample(1:k)],n)))
## all.equal(trueo,1:choose(n+k-1,k))
## }
## testrandomoffsets <- function() {
## for (i in 1:1000) {
## if (isTRUE(testoffset(sample(6,1),sample(6,1))))
## cat(".") # good
## else
## cat("!") # bad
## }
## cat("\n")
## }

## Converts a flat vector index into an array index. k is the
## dimension, n is the number of elements along each dimension. As
## opposed to a base conversion, numbers are reversed for an array.
## Also, we add and subtract 1 (R indexing).
mapindex <- function(m,n,k) {
  mm <- vector(mode="numeric",length=k)
  m <- m-1
  for (i in seq(along=mm)) {
    mm[i] <- m %% n
    m <- m %/% n
  }
```

```
  mm+1
}

## unfolds a single tensor (a row of the folded matrix)
unfoldtensor <- function(folded, # the folded tensor (from dynare)
                         k, # the order
                         n) { # total number of variables
  a <- vector(mode="numeric",length=n^k)
  for (i in seq(along=a))
    a[i] <- folded[getoffset(mapindex(i,n,k),n)]
  array(a,dim=rep(n,k))
}

## match variables names
matchvarnames <- function(varnames,dynnames) {
  ## check
  stopifnot(length(varnames)==length(dynnames))
  ## strip padding spaces
  dynnames <- sapply(dynnames,function(a) gsub(" "," ",a))
  ## find matches
  pp <- vector(mode="numeric",length=length(varnames))
  for (i in seq(along=pp)) {
    m <- grep(varnames[i],dynnames) # find matches
    if (length(m)!=1)
      stop(sprintf("There are %d matches for variable %s in matchvarnames!\n",
                   length(m),varnames[i]))
    pp[i] <- m[1] # store match
  }
  pp
}
```

Example

I tested this interface with a very simple model. The consumer maximizes

$$E\sum_{t=0}^{\infty}\beta^t u(c_t)$$

subject to the budget constraint

$$k_t + c_t = e^{a_t}k_{t-1}^{\alpha} + (1-\delta)k_{t-1}$$

and the evolution of productivity,

$$a_t = \rho a_{t-1} + \varepsilon_t$$

Using the first order conditions, the following three equations characterize the model for dynare++:

$$\beta E\left[(c/c_{+1})^{\gamma}(\alpha e^{a_{+1}}k^{\alpha-1}+1-\delta)\right]=1$$
$$a=\rho a_{-1}+\varepsilon$$
$$k+c=e^{a}k_{-1}^{\alpha}+(1-\delta)k_{-1}$$

The code below shows how we set up the model framework, call dynare with some parameters (reasonable ones, but I did no "calibration"), then extract the 2nd order tensor of the first variable, which is k in this case (we would need to apply the permutation pp to have a as the first variable, as in dynfw).

dynexample.r

```
source("dynread.r")

## example of using these functions
model <- c("(c/c(1))^gamma*beta*(alpha*exp(a(1))*k^(alpha-1)+1-delta)=1",
"a=rho*a(-1)+eps",
"k+c=exp(a)*k(-1)^alpha+(1-delta)*k(-1)")

dynfw <- makedynframework(dynpath="/home/tpapp/bin/dynare++",
                          var=c("k","c","a"),
                          varexo="eps",
                          parameters=c("beta", "gamma", "rho",
                            "alpha", "delta"),
                          model=model,
                          initval=c(k=0.066,c=0.43,a=0.01),
                          order=2,
                          statevars=c("a","k","eps"))

dd <- calldynare(dynfw, c(.99, # beta
                          2, # gamma
                          .9, # rho
                          .3, # alpha
                          .025 # delta
                          ), matrix(0.001,1,1))

unfoldtensor(dd$dyn$dyn.g.2[1,],k=2,n=3) # 2nd order tensor for the
                                         # first variable

## match variables names
pp <- matchvarnames(dynfw$statevars,dd$dyn$dyn.state.vars)
dd$dyn$dyn.state.vars[pp]
```

This is how `tmp.mod`, the model file, looks like:

```
var k, c, a;
```

```
varexo  eps;
parameters beta, gamma, rho, alpha, delta;
beta=0.990000;
gamma=2.000000;
rho=0.900000;
alpha=0.300000;
delta=0.025000;

model;
(c/c(1))^gamma*beta*(alpha*exp(a(1))*k^(alpha-1)+1-delta)=1;
a=rho*a(-1)+eps;
k+c=exp(a)*k(-1)^alpha+(1-delta)*k(-1);
end;


initval;
k=0.066000;
c=0.430000;
a=0.010000;
end;

vcov = [
   0.001
];

order=2;
```

3 Suggestions

A couple of suggestions that might be useful:

- when trying to do MCMC, first write the model file by hand, and run dynare++ on it, with some simulations. That uncovered some typos for me. This also shows if dynare segfaults for some reason for your model.
- use this model file to write the "model" variable for the R interface. If you do not read from a file, include the model/end pairs and the semicolons.
- try a ramdisk for the temporary directory, and see it if speeds things up. I didn't notice a significant improvement for this small model though.