AgEcon SEARCH
RESEARCH IN AGRICULTURAL & APPLIED ECONOMICS

The World's Largest Open Access Agricultural & Applied Economics Digital Library

This document is discoverable and free to researchers across the globe due to the work of AgEcon Search.

Help ensure our sustainability.

Give to AgEcon Search

AgEcon Search
http://ageconsearch.umn.edu
aesearch@umn.edu

*Papers downloaded from **AgEcon Search** may be used for non-commercial purposes and personal study only. No other use, including posting to another Internet site, is permitted without permission from the copyright owner (not AgEcon Search), or as allowed under the provisions of Fair Use, U.S. Copyright Act, Title 17 U.S.C.*

No endorsement of AgEcon Search or its fundraising activities by the author(s) of the following work or their employer(s) is intended or implied.

The Stata Journal (2004)
4, Number 2, pp. 180–189

From the help desk: Polynomial distributed lag models

Allen McDowell
StataCorp

Abstract. Polynomial distributed lag models (PDLs) are finite-order distributed lag models with the impulse–response function constrained to lie on a polynomial of known degree. You can estimate the parameters of a PDL directly via constrained ordinary least squares, or you can derive a reduced form of the model via a linear transformation of the structural model, estimate the reduced-form parameters, and recover estimates of the structural parameters via an inverse linear transformation of the reduced-form parameter estimates. This article demonstrates both methods using Stata.

Keywords: st0065, polynomial distributed lag, Almon, Lagrangian interpolation polynomials

1 Introduction

A polynomial distributed lag model is a pth-order distributed lag model of the form

$$y_t = \sum_{i=0}^{p} \beta_i x_{t-i} + \epsilon_t \tag{1}$$

where the impulse–response function is constrained to lie on a polynomial of degree q. Requiring the impulse–response function to lie on a polynomial imposes $p-q$ constraints on the structural parameters of the model. Following Fomby, Hill, and Johnson (1984) and Shiller (1973), we can determine the form of the constraints from the fact that if f_i is a polynomial of degree n whose domain is the integers, the first difference $Df_i = (1-L)f_i$ can be expressed as a polynomial of degree $n-1$ in i. Consequently, the $(n+1)$st difference $D^{n+1}f_i$ is the zero function. Thus, the constraints have the form

$$(1-L)^{q+1}\beta_i = 0 \qquad i = q+1, \ldots, p \tag{2}$$

Consistent and efficient estimates of the structural parameters, subject to the $p-q$ constraints, can be obtained via constrained ordinary least squares.

Alternatively, we can write the constraints as

$$\beta_i = a_0 + a_1 i + a_2 i^2 + \cdots + a_q i^q$$

© 2004 StataCorp LP

Substituting the constraints into the finite-order distributed lag model yields a reduced-form representation. Vandermonde matrices, which are square matrices of the form

$$\mathbf{V} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ \tau_0 & \tau_1 & \dots & \tau_n \\ \vdots & \vdots & \ddots & \vdots \\ \tau_0^n & \tau_1^n & \dots & \tau_n^n \end{pmatrix}$$

provide a convenient way to derive the reduced form of the model. For example, if we make $\mathbf{V}$ a $(p+1)\times(p+1)$ matrix and substitute the lag index i for the τ_j, the resulting matrix has the form

$$\mathbf{V} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 0 & 1 & \dots & p \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1^n & \dots & p^n \end{pmatrix}$$

Finding a solution to

$$\mathbf{V}'\mathbf{a} = \boldsymbol{\beta}$$

is equivalent to polynomial interpolation. Once a degree, q, for the polynomial has been chosen such that $q < p$, we can simply replace $\mathbf{V}$ in the expression above with a matrix consisting of the first $q+1$ rows of $\mathbf{V}$, and the pth-order impulse–response function is constrained to lie on a polynomial of degree q. Letting $\mathbf{V}_{q+1}$ denote a matrix consisting of the first $q+1$ rows of $\mathbf{V}$ and substituting $\mathbf{V'}_{q+1}\mathbf{a}$ for $\boldsymbol{\beta}$ in (1) yields the reduced-form representation of the model:

$$\begin{aligned} \mathbf{y} &= \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \\ &= \mathbf{X}\mathbf{V'}_{q+1}\mathbf{a} + \boldsymbol{\epsilon} \\ &= \mathbf{Z}\mathbf{a} + \boldsymbol{\epsilon} \end{aligned}$$

where $\mathbf{Z} = \mathbf{X}\mathbf{V'}_{q+1}$. The parameters of the reduced form can be estimated consistently and efficiently via OLS. Estimates of the structural parameters and their variances can be recovered via the relations

$$\widehat{\boldsymbol{\beta}} = \mathbf{V'}_{q+1}\widehat{\mathbf{a}}$$

and

$$\mathrm{Var}(\widehat{\boldsymbol{\beta}}) = \mathbf{V'}_{q+1}\mathrm{Var}(\widehat{\mathbf{a}})\mathbf{V}_{q+1}$$

Cooper (1972) refers to this as the direct method.

An extension to the method just described was introduced by Almon (1965). Note that the estimates of $\boldsymbol{\beta}$ are unique up to a nonsingular linear transformation. For example, let $\mathbf{J}$ be any nonsingular $(q+1)\times(q+1)$ matrix. Let

$$\mathbf{a} = \mathbf{J}^{-1}\boldsymbol{\gamma}$$

It follows that

$$\boldsymbol{\beta} = \mathbf{V'}_{q+1}\mathbf{J}^{-1}\boldsymbol{\gamma}$$

If $\mathbf{J}$ is the transpose of a $(q+1) \times (q+1)$ Vandermonde matrix, the elements of a general row of $\mathbf{V'}_{q+1}\mathbf{J}^{-1}$ are Lagrangian interpolation polynomials. Again, estimation of the reduced-form parameters, $\boldsymbol{\gamma}$, can be consistently and efficiently obtained via ordinary least squares. Estimates of the structural parameters, $\boldsymbol{\beta}$, and their variances are recovered via

$$\widehat{\boldsymbol{\beta}} = \mathbf{V'}_{q+1}\mathbf{J}^{-1}\widehat{\boldsymbol{\gamma}}$$

and

$$\mathrm{Var}(\widehat{\boldsymbol{\beta}}) = (\mathbf{V'}_{q+1}\mathbf{J}^{-1})\mathrm{Var}(\widehat{\boldsymbol{\gamma}})(\mathbf{V'}_{q+1}\mathbf{J}^{-1})'$$

According to Cooper (1972), the Almon method is preferred in estimation since the resulting weighting matrix will have a more irregular pattern of weights compared with $\mathbf{V'}_{q+1}$, reducing the likelihood that the artificial variables, $\mathbf{Z}$, will be collinear.

2 Estimation using Stata

To demonstrate how to fit a polynomial distributed lag model in Stata using each of the methods described above, let's consider an example of a PDL model with $p = 12$ and $q = 4$. First, let's simulate some data:

```
* generate the pdl(12,4) data

clear
set seed 2001
sim_arma x, ar(.9) spin(10000) nobs(300)
tsset _t
gen double y = .8*x + .8^2*L.x + .8^3*L2.x + .8^4*L3.x  ///
             + .8^5*L4.x + .8^6*L5.x + .8^7*L6.x        ///
             + .8^8*L7.x + .8^9*L8.x + .8^10*L9.x       ///
             + .8^11*L10.x + .8^12*L11.x + .8^13*L12.x  ///
             + invnorm(uniform())

save pdl, replace

exit
```

The input variable, x_t, follows an AR(1) process, and the impulse–response function is a simple geometric series. x_t was generated using the `sim_arma` command, which will simulate data from any ARMA process. `sim_arma` was written by Jeff Pitblado of StataCorp; it can be located and installed using the `findit` command or by issuing the command

```
. net install http://www.stata.com/users/jpitblado/sim_arma
```

To fit the structural model via constrained OLS, we use the `cnsreg` command. The `constraints` option of `cnsreg` accepts either a *numlist* that identifies the individual constraints or the name of an existing constraint matrix. To employ the *numlist*, we would have to define each constraint individually using the `constraint` command, and we would have to redefine the constraints each time we altered the PDL model specification. While this is not particularly difficult, it can become rather tedious. It is much simpler to write a general-purpose program that will construct an appropriate constraint matrix for any PDL model specification.

`cnsreg` fits a linear model, $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, subject to $\mathbf{R}\boldsymbol{\beta} = \mathbf{r}$. If we include an intercept in the model, there will be $p+2$ estimated parameters and $p-q$ constraints. Therefore, $\mathbf{R}$ will be a matrix of dimension $(p-q) \times (p+2)$, and $\mathbf{r}$ will be a matrix of dimension $(p-q) \times 1$. As documented in [P] **matrix constraint**, Stata expects the constraints to be represented by a single matrix constructed by concatenating $\mathbf{R}$ and $\mathbf{r}$, with $\mathbf{r}$ represented by the rightmost column of the constraint matrix.

A close inspection of (2) tells us that the elements of $\mathbf{R}$ will form a simple pattern. There will be $q+2$ nonzero elements in each row of $\mathbf{R}$. In the first row of $\mathbf{R}$, the first $q+2$ elements will be nonzero, their specific values being determined by a binomial expansion, and the remaining elements will be zeros. In each subsequent row of $\mathbf{R}$, the same $q+2$ nonzero elements will appear, but each time we move down a row, the $q+2$ nonzero elements will be shifted one column to the right. The following program constructs such an $\mathbf{R}$ matrix, adding two columns of zeros; the first additional column of zeros accounts for the intercept, and the second additional column represents $\mathbf{r}$. The program requires three arguments, p, q, and a matrix name. It is sufficiently general for use with any PDL specification.

```
program pdlconstraints
        version 8.2
        args p q matname
        local r = `p' - `q'
        local m = `q' + 1
        matrix `matname' = J(`r',`p'+3,0)
        forvalues i = 1/`r' {
                local x = `i' + `q' + 1
                local k = -1
                local d = 1
                forvalues j = `x'(-1)`i' {
                        local k = `k' + 1
                        matrix `matname'[`i',`j'] = `d'*comb(`m',`k')
                        local d = -1*`d'
                }
        }
end
```

(*Continued on next page*)

With our simulated data and a program to construct the constraint matrix, we can now fit the PDL model by issuing the following commands:

```
. pdlconstraints 12 4 A
. cnsreg y  L(0/12).x, constraints(A)
Constrained linear regression                   Number of obs =     288
                                                F(  5,   282) = 4117.51
                                                Prob > F      =  0.0000
                                                Root MSE      =  .95335
 ( 1) - x + 5 L.x - 10 L2.x + 10 L3.x - 5 L4.x + L5.x = 0
 ( 2) - L.x + 5 L2.x - 10 L3.x + 10 L4.x - 5 L5.x + L6.x = 0
 ( 3) - L2.x + 5 L3.x - 10 L4.x + 10 L5.x - 5 L6.x + L7.x = 0
 ( 4) - L3.x + 5 L4.x - 10 L5.x + 10 L6.x - 5 L7.x + L8.x = 0
 ( 5) - L4.x + 5 L5.x - 10 L6.x + 10 L7.x - 5 L8.x + L9.x = 0
 ( 6) - L5.x + 5 L6.x - 10 L7.x + 10 L8.x - 5 L9.x + L10.x = 0
 ( 7) - L6.x + 5 L7.x - 10 L8.x + 10 L9.x - 5 L10.x + L11.x = 0
 ( 8) - L7.x + 5 L8.x - 10 L9.x + 10 L10.x - 5 L11.x + L12.x = 0
```

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
x						
--	.7709161	.0382041	20.18	0.000	.6957146	.8461176
L1	.6378791	.0140968	45.25	0.000	.6101307	.6656275
L2	.5162648	.0216048	23.90	0.000	.4737377	.5587919
L3	.4100719	.0179107	22.90	0.000	.3748162	.4453276
L4	.32173	.0123709	26.01	0.000	.297379	.346081
L5	.2520991	.0139051	18.13	0.000	.2247281	.27947
L6	.2004697	.0159642	12.56	0.000	.1690456	.2318938
L7	.1645632	.0138539	11.88	0.000	.1372931	.1918333
L8	.1405315	.0123681	11.36	0.000	.116186	.1648769
L9	.1229569	.0179885	6.84	0.000	.0875481	.1583657
L10	.1048526	.021662	4.84	0.000	.0622129	.1474923
L11	.0776623	.0141234	5.50	0.000	.0498617	.1054629
L12	.0312602	.0385907	0.81	0.419	-.0447023	.1072227
_cons	-.000655	.0566763	-0.01	0.991	-.1122174	.1109073

We can now fit the same model using the reduced-form representation of the model. The basic algorithm for both the direct method and the Almon method is to

1. generate a weighting matrix,
2. generate the artificial variables,
3. fit the reduced-form model via OLS, and
4. reverse the transformation.

There are a number of ways we can proceed. For pedagogical reasons, the method I have chosen will result in code that very closely follows the mathematical treatment presented above. From a programming perspective, this is not the most efficient method. For example, since both methods use Vandermonde matrices, I have produced a fairly general program for generating Vandermonde matrices. This means that, along the way, I will have to perform several operations on these matrices that could be avoided

if I were attempting to write production quality code, but then the code would not be as easy to follow, and its correspondence with the earlier discussion would be less transparent.

The following program generates a Vandermonde matrix. It requires the user to supply a name for the resulting matrix and a *numlist* that corresponds to the τ_j in the discussion above.

```
program vandermonde
        version 8.2
        syntax name, Numlist(numlist)
        local p: word count `numlist'
        tokenize `numlist'
        matrix `namelist' = J(`p',`p',0)
        forvalues c = 1/`p' {
                forvalues r = 1/`p' {
                        matrix `namelist'[`r',`c'] = (``c'')^(`r' - 1)
                }
        }
end
```

To get the matrix that we need for our PDL(12,4), we issue the command

```
. vandermonde V, n(0/12)
```

which produces a 13×13 Vandermonde matrix. Since we only need the first five rows of $\mathbf{V}$ to constrain the parameters to lie along a fourth degree polynomial, and since we want the transpose of this matrix for our weighting matrix, we issue the commands

```
. matrix V = V[1..5,1..13]
. matrix W = V'
```

to get the matrix we want. The second step generates the artificial variables of the reduced-form representation. We need to generate the variables $\mathbf{Z}$, such that $\mathbf{Z} = \mathbf{X}\mathbf{V}'_{q+1}$, or now, in terms of our program, we need to generate $\mathbf{Z} = \mathbf{X}\mathbf{W}$, which is what the following program does.

```
program zvars
        version 8.2
        syntax varname, Matrix(name)
        local n = colsof(`matrix')
        local k = rowsof(`matrix')
        forvalues i = 1/`n' {
                local z`i' `matrix'[1,`i']*`varlist'
        }
        forvalues j = 2/`k' {
                forvalues i = 1/`n' {
                        local m = `j' - 1
                        local z`i' `z`i'' + `matrix'[`j',`i']*L`m'.`varlist'
                }
        }
        forvalues i = 1/`n' {
                generate double z`i' = `z`i''
        }
end
```

When executing the program, we must pass the names of the input variable and the weighting matrix as arguments; that is, we issue the command

```
. zvars x, matrix(W)
```

The program is written so that we can use it again, without modification, when implementing the Almon method.

The third step is to fit the reduced-form model via OLS. To do this, we issue the command

```
. regress y z*
```

Source	SS	df	MS	Number of obs = 288
				F(5, 282) = 4117.51
Model	18711.7058	5	3742.34116	Prob > F = 0.0000
Residual	256.305306	282	.908884064	R-squared = 0.9865
				Adj R-squared = 0.9862
Total	18968.0111	287	66.0906311	Root MSE = .95335

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
z1	.7709161	.0382041	20.18	0.000	.6957146	.8461176
z2	-.137023	.0628266	-2.18	0.030	-.2606917	-.0133543
z3	.0029926	.0231034	0.13	0.897	-.0424845	.0484697
z4	.0010588	.0029704	0.36	0.722	-.0047881	.0069058
z5	-.0000654	.0001233	-0.53	0.596	-.0003082	.0001774
_cons	-.000655	.0566763	-0.01	0.991	-.1122174	.1109073

The final step is to perform the reverse transformation to recover the estimates of the structural parameters and their variances. Once the transformation has been performed, we can also repost the results so that we get a nice tabular display. This is what the following program does.

```
program recover, eclass
        version 8.2
        syntax name, Matrix(name)
        tempname alpha v w B V
        matrix `alpha' = e(b)
        matrix `v' = e(V)
        local r = rowsof(`matrix')
        matrix `w' = J(`r',1,0)
        matrix `matrix' = `matrix',`w'
        local c = colsof(`matrix')
        matrix `w' = J(1,`c',0)
        matrix `w'[1,`c'] = 1
        matrix `matrix' = `matrix'\(`w')
        matrix `B' = (`matrix'*(`alpha')')'
        matrix `V' = `matrix'*`v'*(`matrix')'
        matrix rownames `B' = y
        local r = `r'-1
        local names  `namelist'
        forvalues i = 1/`r' {
                local names `names' L`i'.`namelist'
        }
        local names `names' _cons
        matrix colnames `B' = `names'
```

```
        matrix rownames 'V' = 'names'
        matrix colnames 'V' = 'names'
        ereturn post 'B' 'V'
        ereturn local cmd recover
        ereturn display
end
```

Immediately following the `tempname` command, the next two lines of code collect the reduced-form parameter estimates and the reduced-form variance–covariance matrix. The next seven lines adjust the weighting matrix to account for an intercept being included in the model; this is preparation for the reverse transformation needed to recover the structural parameter estimates, which is what the next two lines accomplish. Once the transformation is complete, the program renames the rows and columns of the matrices holding the structural parameter estimates and the associated variance–covariance matrix. Finally, the program posts those two matrices and displays the output.

We execute the command by typing

```
. recover x, matrix(W)
```

	Coef.	Std. Err.	z	P>|z|	[95% Conf.	Interval]
x						
--	.7709161	.0382041	20.18	0.000	.6960374	.8457948
L1	.6378791	.0140968	45.25	0.000	.6102498	.6655084
L2	.5162648	.0216048	23.90	0.000	.4739202	.5586094
L3	.4100719	.0179107	22.90	0.000	.3749676	.4451763
L4	.32173	.0123709	26.01	0.000	.2974835	.3459765
L5	.2520991	.0139051	18.13	0.000	.2248456	.2793525
L6	.2004697	.0159642	12.56	0.000	.1691805	.2317589
L7	.1645632	.0138539	11.88	0.000	.1374101	.1917163
L8	.1405315	.0123681	11.36	0.000	.1162905	.1647724
L9	.1229569	.0179885	6.84	0.000	.0877	.1582138
L10	.1048526	.021662	4.84	0.000	.0623959	.1473093
L11	.0776623	.0141234	5.50	0.000	.049981	.1053436
L12	.0312602	.0385907	0.81	0.418	-.0443763	.1068967
_cons	-.000655	.0566763	-0.01	0.991	-.1117386	.1104285

To summarize, given the various programs that have been introduced, we fit a PDL(12,4) model to our simulated data using the direct method by issuing the following sequence of commands:

```
. vandermonde V, n(0/12)
. matrix V = V[1..5,1..13]
. matrix W = V'
. zvars x, matrix(W)
. regress y z*
. recover x, matrix(W)
```

To implement the Almon method, we need to construct a different weighting matrix; otherwise, the procedure is identical to the direct method. We begin with the same Vandermonde matrix as before:

```
. vandermonde V, n(0/12)
. matrix V = V[1..5,1..13]
```

Now, we need to construct a second Vandermonde matrix, i.e., the matrix **J** from the discussion above. **J** will be a $(q+1)\times(q+1)$ matrix. We can choose any five distinct points in the interval $[0, p]$ for the τ_j, so I will use five equidistant points, i.e., 0, 3, 6, 9, and 12. The command to generate the matrix is then

```
. vandermonde J, n(0 3 6 9 12)
```

and the weighting matrix is therefore

```
. matrix W = V'*inv(J')
```

Everything proceeds exactly as before from this point on:

```
. drop z*
. zvars x, matrix(W)
. regress y z*
```

Source	SS	df	MS	Number of obs =	288
				F(5, 282) =	4117.51
Model	18711.7058	5	3742.34116	Prob > F =	0.0000
Residual	256.305306	282	.908884064	R-squared =	0.9865
				Adj R-squared =	0.9862
Total	18968.0111	287	66.0906311	Root MSE =	.95335

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
z1	.7709161	.0382041	20.18	0.000	.6957146	.8461176
z2	.4100719	.0179107	22.90	0.000	.3748162	.4453276
z3	.2004697	.0159642	12.56	0.000	.1690456	.2318938
z4	.1229569	.0179885	6.84	0.000	.0875481	.1583657
z5	.0312602	.0385907	0.81	0.419	-.0447023	.1072227
_cons	-.000655	.0566763	-0.01	0.991	-.1122174	.1109073

```
. recover x, matrix(W)
```

	Coef.	Std. Err.	z	P>|z|	[95% Conf.	Interval]
x						
--	.7709161	.0382041	20.18	0.000	.6960374	.8457948
L1	.6378791	.0140968	45.25	0.000	.6102498	.6655084
L2	.5162648	.0216048	23.90	0.000	.4739202	.5586094
L3	.4100719	.0179107	22.90	0.000	.3749676	.4451763
L4	.32173	.0123709	26.01	0.000	.2974835	.3459765
L5	.2520991	.0139051	18.13	0.000	.2248456	.2793525
L6	.2004697	.0159642	12.56	0.000	.1691805	.2317589
L7	.1645632	.0138539	11.88	0.000	.1374101	.1917163
L8	.1405315	.0123681	11.36	0.000	.1162905	.1647724
L9	.1229569	.0179885	6.84	0.000	.0877	.1582138
L10	.1048526	.021662	4.84	0.000	.0623959	.1473093
L11	.0776623	.0141234	5.50	0.000	.049981	.1053436
L12	.0312602	.0385907	0.81	0.418	-.0443763	.1068967
_cons	-.000655	.0566763	-0.01	0.991	-.1117386	.1104285

Throughout the literature on PDLs, we find numerous assertions that the Almon method is the preferred method. However, we can clearly see that the constrained OLS method requires less effort and produces identical estimates. The argument is often made that the Almon method has better numerical properties. However, informal simulation studies indicate that the Almon method offers no advantage at all compared with the constrained OLS estimator, and the only instance in which it would offer an advantage over the direct method is the unlikely case where the artificial variables that are generated using the direct method result in a data matrix that is singular.

3 References

Almon, S. 1965. The distributed lag between capital appropriations and expenditures. *Econometrica* 33: 178–196.

Cooper, P. J. 1972. Two approaches to polynomial distributed lags estimation: an expository note and comment. *American Statistician* 26: 32–35.

Fomby, T. B., R. C. Hill, and S. R. Johnson. 1984. *Advanced Econometric Methods*. New York: Springer.

Shiller, R. J. 1973. A distributed lag estimator derived from smoothness priors. *Econometrics* 41: 775–788.

About the Author

Allen McDowell is Director of Technical Services at StataCorp.