Speeding Up the ARDL Estimation Command:
A Case Study in Efficient Programming in Stata and Mata

Sebastian Kripfganz\textsuperscript{1} \hspace{1cm} Daniel C. Schneider\textsuperscript{2}

\textsuperscript{1}University of Exeter
\textsuperscript{2}Max Planck Institute for Demographic Research

German Stata Users Group Meeting, June 23, 2017
Contents

Efficient Coding

Digression: A Tiny Bit of Asymptotic Notation

The ARDL Model

Optimal Lag Selection

Incremental Code Improvements
Introduction

- Long code execution times are more than a nuisance: they negatively affect the quality of research

- strategies for speeding up execution:
  - lower-level language
  - parallelization
  - writing efficient code

- Efficient coding is often the best choice.
  - Moving to lower-level languages is tedious.
  - In many settings, speed improvements are higher than through parallelization.
Introduction: Speed of Stata and Mata

- C is the reference
  - compiled to machine instructions
- Post of Bill Gould (2014) at the Stata Forum:
  - Stata (interpreted) code is 50-200 times slower than C.
  - Mata compiled byte-code 5-6 times slower than C.
    => Mata is 10-40 times faster than Stata.
  - In real-world applications, Mata is ~2 times slower than C.
    - Mata has built-in C routines based on very efficient code.
Introduction: Efficient Coding Strategies

- Using Common Sense
  - An if-condition requires at least $N$ comparisons. Use in-conditions instead, if possible.
  - Multiplying two 100x100 matrices requires about $2 \times 100^3 = 2,000,000$ arithmetic operations.

- Using Knowledge of Your Software (Stata, of course!)
  - Examples:
    - Mata: passing of arguments to functions
    - Efficient operators and functions (e.g. Mata's colon operator and its c-conformability)
    - Read the Stata and Mata programming manuals
Introduction: Efficient Coding Strategies

Using Knowledge of Matrix Algebra

- Translating mathematical formulas one-to-one into matrix language expressions is oftentimes (very!) inefficient.

- Examples:

  - diagonal matrices (D):
    - multiplication of a matrix by D: don’t do it!
      Mata: use c-conformability of the colon operator (see [M-2] \texttt{op_colon})
    - inverse: flip diagonal elements instead of calling a matrix solver / inverter function ($O(n)$ vs. $O(n^3)$)

  - block diagonal matrices:
    - multiplication: just multiply diagonal blocks; the latter is faster by $1/s^2$, where $s$ is the number of diagonal blocks
    - inverse: invert individual blocks

  - order of matrix multiplication / parenthesization
    - $b = (X'X)^{-1}(X' y)$ is faster than $b = (X'X)^{-1}X'y$
      e.g. for $k = 10$, $N = 10,000$: matrix multiplications are 11 times faster!
Asymptotic Notation

Definition

An algorithm with input size $n$ and running time $T(n)$ is said to be $\Theta(g(n))$ ("theta of g of n") or to have an asymptotically tight bound $g(n)$ if there exist positive real numbers $c_1, c_2, n_0 > 0$ such that

$$c_1 g(n) \leq T(n) \leq c_2 g(n) \quad \forall n \geq n_0$$

$$T(n)=0.8n^3 - 1000n^2 + 1000n + 10e9$$

is $O(n^3)$

# of arithmetic operations

0 500 1000 1500 2000

Algorithm input size $n$

0.2 * $n^3$

0.801 * $n^3$
Asymptotic Notation

- $O(g(n))$ ("(big) oh of g of n"), as opposed to $\Theta(g(n))$, is used here to only denote an upper bound. Notation differs in the literature.

- Technically, $\Theta(g(n))$ and $O(g(n))$ are sets of functions, so we write e.g. $T(n) \in O(g(n))$.

- For matrix operations, $g(n)$ is frequently $n$ raised to some low integer power.
  - $\Theta(n)$ is much better than $\Theta(n^2)$, which in turn is much better than $\Theta(n^3)$
  - (Square) matrix multiplication is $\Theta(n^3)$: each element of the new $n \times n$ matrix is a sum of $n$ terms. Costly!
  - Many types of matrix inversion, e.g. the LU-decomposition, are also $\Theta(n^3)$. Costly!
  - Inner vector products are $\Theta(n)$.

- When $T(n)$ is an $i$-th order polynomial, the leading term asymptotically dominates: $T(n) \in O(n^i)$.

- $\Theta(a^n)$ is worse than $\Theta(n^a)$; $\Theta(lg \ n)$ is better than $\Theta(n)$
ARDL: Model Setup

- ARDL \((p, q_1, \ldots, q_k)\): autoregressive distributed lag model

- Popular, long-standing single-equation time-series model for continuous variables

- Linear model:

  \[
  y_t = c_0 + c_1 t + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{i=0}^{q} \beta_i' x_{t-i} + u_t, \quad u_t \sim iid (0, \sigma^2)
  \]

- \((y_t, x_t')\) can be purely \(I(0)\), purely \(I(1)\), or cointegrated: can be used to test for cointegration (bounds testing procedure). (Pesaran, Shin, and Smith, 2001).

  => econometrics of ARDL can be complicated.

- net install ardl, from(http://www.kripfganz.de/stata)

- This talk: programming; for the statistics of ardl, see Kripfganz/Schneider (2016).
ARDL: Computational Considerations

- Despite its complex statistical properties, estimating an ARDL model is just based on OLS!

- The computational costly parts are:
  - determination of optimal lag orders (e.g. via AIC or BIC)
    - treated at length in this talk
    - not covered by this talk
Optimal Lag Selection: The Problem

- For $k + 1$ variables (indepvars + depvar) and $\maxlag$ lags for each variable, run a regression and calculate an information criterion (IC) for each possible lag combination and select the model with the best IC value.

**Example:** 2 variables (v1 v2), $\maxlag = 2$
- `regress v1 L(1/1).v1 L(0/0).v2`
- `regress v1 L(1/2).v1 L(0/0).v2`
- `regress v1 L(1/1).v1 L(0/1).v2`
- `regress v1 L(1/2).v1 L(0/1).v2`
- `regress v1 L(1/1).v1 L(0/2).v2`
- `regress v1 L(1/2).v1 L(0/2).v2`

- # of regressions to run is exponential in $k$:
  \[ \text{maxlags} \cdot (\text{maxlags} + 1)^k \]

<table>
<thead>
<tr>
<th>$k + 1$</th>
<th>maxlags</th>
<th># regressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>$\sim 650$</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>$\sim 5,800$</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>$\sim 470,000$</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>$\sim 38,000,000$</td>
</tr>
</tbody>
</table>
Lag Selection: Preliminaries

- Lag combination matrix for $k = 3$ and maxlags = 2:

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 0 & 2 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
1 & 1 & 2 \\
1 & 2 & 0 \\
1 & 2 & 1 \\
\vdots \\
2 & 2 & 2
\end{bmatrix}
\]

- e.g. row 3: $[1 \ 0 \ 2]$ corresponds to regressors $L.v1 \ L(0/0).v2 \ L(0/2).v3 = [v_{1t-1} \ v_{2t} \ v_{3t} \ v_{3t-1} \ v_{3t-2}]$

- called “lagcombs” in pseudo-code to follow
Lag Selection: Naive Approach Using `regress`

Stata/Mata-like pseudocode:

```plaintext
// note: may contain incorrect syntax,
// fictitious commands/options/return values, etc.
lagcombs, k(3) maxlag(8)
// defines matrix with all lag combs

scalar ic = .
forvalues i=1/numrows(lagcombs) {
    matrix lags = lagcombs(`i',1...)
    regress v1 v2 v3 , lags(lags)
estat ic
    if r(aic)<ic {
        scalar ic = r(aic)
        matrix optimlag = lags
    }
}
```
## Lag Selection: Timings

Timings in seconds (2.5GHz, single core) for N=1000:

<table>
<thead>
<tr>
<th>$k + 1$</th>
<th>$maxlags$</th>
<th># regressions</th>
<th>regress</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>100</td>
<td>1.6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>$\sim 650$</td>
<td>12.5</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>$\sim 5,800$</td>
<td>132</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>$\sim 470,000$</td>
<td>$\sim 14000$</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>$\sim 38,000,000$</td>
<td>(13 days?)</td>
</tr>
</tbody>
</table>
Efficient Coding
Digression: A Tiny Bit of Asymptotic Notation
The ARDL Model
Optimal Lag Selection
Incremental Code Improvements

Lag Selection: Mata I

Stata/Mata-like pseudocode:

```mata
// note: may contain incorrect syntax,
// fictitious commands/options/return values, etc.
lagcombs, k(3) maxlag(8)
// defines matrix with all lag combs

mata:
lagcombs = st_matrix("lagcombs")
ic = .
for (i=1; i<=rows(lagcombs); i++) {
    y = st_data( DEFVAR )
    X = st_data( PULL DATA FOR SPECIFIC LAG COMBINATION )

    // calculate AIC
    ee = y'y - y'X*invsym(X'X)*X'y    // sum of squared residuals
    ic = T*log(2*pi()) + T*log(ee/T) + T + 2*k

    if (ic<ic_min) {
        ic_min = ic
        optimrow = i
    }
}
end
```

Kripfganz/Schneider  Uni Exeter & MPIDR  Speeding Up ARDL  June 23, 2017  15 / 27
Efficient Coding Digression: A Tiny Bit of Asymptotic Notation

The ARDL Model

Optimal Lag Selection

Incremental Code Improvements

Lag Selection: Mata II (no redundant calculations)

```mata
// note: may contain incorrect syntax, fictitious commands/options/return values, etc.
lagcombs, k(3) maxlag(8)

mata:

ey = st_data( DEFVAR )
X = st_data( PULL ALL DATA : ALL VARIABLES, ALL LAGS)

// calculate terms for full lag specification
XX = X'X ; Xy = X'y ; yy = y'y

lagcombs = st_matrix("lagcombs")
ic = .

for (i=1; i<=rows(lagcombs); i++) {
    // cross-products for current iteration
    idx = [ GET INDEX VECTOR FOR CURRENT LAG STRUCTURE ]
    XXi = XX[idx,idx] ; Xyi = Xy[idx]

    ee = yy - Xyi'*invsym(XXi)*Xyi  // calculate sum of squared residuals for AIC

    ic = T*log(2*pi()) + T*log(ee/T) + T + 2*k
    if (ic<ic_min) {
        ic_min = ic ;
        optimrow = i
    }
}
end
```

Kripfganz/Schneider
Uni Exeter & MPIDR
Speeding Up ARDL

June 23, 2017
Lag Selection: Mata III

- A sticky point are the many matrix inversions, which are $\Theta(n^3)$.
- We will further improve matters by using results from linear algebra.
- We will introduce and use pointer variables in the process.
- The following will put forth a somewhat complicated algorithm that affects many parts of the loop.
- In this talk, we could have focused our attention on many smaller changes for code optimization, but both things are not possible within the time window for this presentation.
Updating $(X'X)^{-1}$ Using Partitioned Matrices

- For $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, with $A$, $A_{11}$ and $A_{22}$ square and invertible:

$$A^{-1} = \begin{bmatrix} D & -DA_{12}A_{22}^{-1} \\ -A_{22}^{-1}A_{21}D & -A_{22}^{-1} + -A_{22}^{-1}A_{21}DA_{12}A_{22}^{-1} \end{bmatrix}$$

with $D = A_{11}^{-1} + A_{11}^{-1}A_{12}(A_{22} - A_{21}A_{11}^{-1}A_{12})^{-1}A_{21}A_{11}^{-1}$

- Here: Let $X_v = \begin{bmatrix} X \\ v \end{bmatrix}$. The cross-product matrix becomes

$$X_v'X_v = \begin{bmatrix} A_{11} = X'X & A_{12} = X'v \\ A_{21} = A_{12}' & A_{22} = v'v \end{bmatrix}$$

- Task: calculate $(X_v'X_v)^{-1}$ based on the known terms of: $X'X$, $(X'X)^{-1}$, $X'v$, $v'v$

  - Slight complication: Inserting a column to $X$, not just appending.
  - Can be solved by permutation vectors (see $[M-1]$ permutation).

- Let’s call this procedure PMAC (partitioned matrices / append column) to ease exposition.
Updating \((X'X)^{-1}\) Using Partitioned Matrices

- Problem: columns are sometimes deleted, not just added.

- Lag combination matrix \((\text{maxlags} = 2\) for all variables):

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 0 & 2 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
1 & 1 & 2 \\
1 & 2 & 0 \\
1 & 2 & 1 \\
2 & 2 & 2
\end{bmatrix}
\]

- e.g. moving from row 3: \(\begin{bmatrix} 1 & 0 & 2 \end{bmatrix}\) to row 4: \(\begin{bmatrix} 1 & 1 & 0 \end{bmatrix}\) deletes two lags of the last regressor

- Solution: store matrices the algorithm can jump back to using pointers.
Pointer Variables

- General and “advanced” programming concept, but the basics are easy to understand and apply.
  
- Each variable has a name and a type.
  
  - The name really is just a device to refer to a specific location in memory; every location in memory has a unique address.
  
  - Since the type of the variable is known to Mata, it knows how big of a memory range a variable name refers to, and how to interpret the value (the bits stored there).
  
  - Think in these terms: each variable has an address and a value.

- Pointer variables hold memory addresses of other variables. Pointer variables can point to anything: scalars, matrices, pointers, objects, functions ...
Efficient Coding Digression: A Tiny Bit of Asymptotic Notation The ARDL Model Optimal Lag Selection Incremental Code Improvements

Pointer Variables

- Pointers are often assigned to using “&”; they are dereferenced using “*”.

- Read:
  - & : “the address of”
  - * : “the thing pointed to by”

- `mata:
  s = J(2,2,1)
p = &s
*p // outputs something like 0xcb3cb60
*p = J(2,2,-7)
s // now contains the matrix of -7s
end`

- See [M-2] pointers for many more details.
Using Pointers for Updating \((X'X)^{-1}\)

- Lag combination matrix:

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 0 & 2 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
1 & 1 & 2 \\
1 & 2 & 0 \\
\ldots
\end{bmatrix}
\]

- 3-element vector of pointers \(\text{vec} = [p_1 \ p_2 \ p_3]\); each element points to a matrix. Then calculate \((X'X)^{-1}\) for...

  - lags (1 0 0) by ordinary matrix inversion; store using \(p_1\)
  - lags (1 0 1) by PMAC using \(*p_1\); store using \(p_3\)
  - lags (1 0 2) by PMAC using \(*p_3\)
  - lags (1 1 0) by PMAC using \(*p_1\); store using \(p_2\)
  - lags (1 1 1) by PMAC using \(*p_2\); store using \(p_3\)
  - lags (1 1 2) by PMAC using \(*p_3\)
  - lags (1 2 0) by PMAC using \(*p_2\); store using \(p_2\) ... and so forth.
Lag Selection: Mata III (update inverses)

```mata
// note: may contain incorrect syntax, fictitious commands/options/return values, etc.
lagcombs, k(3) maxlag(8)
mata:
    y = st_data( DEPVAR )
    X = st_data( PULL ALL DATA : ALL VARIABLES, ALL LAGS)

    // calculate terms for full lag specification
    XX = X'X ; Xy = X'y ; yy = y'y

    lagcombs = st_matrix("lagcombs")
    ic = .
    for (i=1; i<=rows(lagcombs); i++) {
        idx = [ GET INDEX VECTOR FOR CURRENT LAG STRUCTURE ]
        XXi = XX[idx,idx] ; Xyi = Xy[idx]

        if (i==1) XXinvi = invsym(XX)
        else XXinvi = update_XXinv( XXiold, XXinvi, ... )

        ee = yy - Xyi'*XXinvi*Xyi

        XXiold = XXi
    }
end
```
Lag Selection: Timings

Timings in seconds (2.5GHz, single core) for N=1000:

Mata 2: no redundancies

Mata 3: no redundancies + inverse updating

<table>
<thead>
<tr>
<th>$k + 1$</th>
<th>maxlags</th>
<th># regressions</th>
<th>$\text{regress}$</th>
<th>Mata 1</th>
<th>Mata 2</th>
<th>Mata 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>100</td>
<td>1.6</td>
<td>0.36</td>
<td>0.11</td>
<td>0.14</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>$\sim650$</td>
<td>12.5</td>
<td>1.33</td>
<td>0.09</td>
<td>0.13</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>$\sim5,800$</td>
<td>132</td>
<td>11.8</td>
<td>0.31</td>
<td>0.27</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>$\sim470,000$</td>
<td>$\sim14,000$</td>
<td>$\sim1,400$</td>
<td>53</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>$\sim38,000,000$</td>
<td>(13 days?)</td>
<td>$\sim146,000$</td>
<td>$\sim6,500$</td>
<td>$\sim3,200$</td>
</tr>
</tbody>
</table>
Recap

In this talk, we have discussed

- Potential strategies for improving code performance
- Basic asymptotic notation for the computing time of algorithms
- Quick look at the ARDL model
- Optimal lag selection
- Moving Stata code to Mata and optimizing the Mata code
- An advanced way of using linear algebra results to improve code performance
- Pointer variables

We have tried to illustrate that mindful code creation can be superior to the “brute force” methods of low-level programming languages and parallelization.
Thank you!

Questions? Comments?

S.Kripfganz@exeter.ac.uk
schneider@demogr.mpg.de
References


