An Efficient Indexing Algorithm for Solving Large Fixed Effects Problems

Tom Balmat[*]
Jerome P. Reiter[†]

---------- DRAFT 1.4 6/20/2017 ----------

Abstract

The standard method for solving Ordinary Least Squares (OLS) regression problems in R is to use `lm()` and, for small problems, involving fewer than 1,000,000 observations and 25 independent variables, this is efficient. But for larger problems, involving tens of millions of observations and thousands of independent variables, execution of lm becomes computationally impractical due to system memory requirements and execution time. Large fixed effects regression problems, involving effects with thousands of levels, present a special performance opportunity because of the large proportion of entries in the expanded design matrix (fixed effect levels translated from single columns into dichotomous indicator columns, one for each level) that are zero. For many problems, the proportion of expanded design matrix entries that are zero is above 0.995, which would be considered sparse. In this paper, we present an efficient method for solving a large, sparse fixed effects OLS problem without creation of the expanded design matrix and avoiding computations involving zero-level effects. This leads to minimal memory usage and optimal execution. A feature, often desired in social science applications, is to estimate parameter standard errors clustered about a key identifier, such as employee ID, and large problems, with ID counts in the millions, present significant computational challenges. We present a sparse matrix indexing algorithm that produces clustered standard error estimates that, for large fixed effects problems, is many times more efficient than standard "sandwich" matrix operations.

Keywords: fixed effects least squares solution, sparse matrix methods, high performance computing, parallel computing, clustered standard error estimation

The authors would like to thank Alex Bolton, Andres (Felipe) Barrientos, and John de Figueiredo for sharing data and models from their research and to the IT staff at Duke University's Social Science Research Institute for high performance computing support.

Introduction

The classic method of estimating Ordinary Least Squares (OLS) model parameters is to solve

$X'X\hat{\beta} = X'Y$ for $\hat{\beta}$. For small designs, where X ($n \times p$) involves n and p on the order of 10^4 and

[*] Social Science Research Institute, Duke University, Durham, NC 27708 (thomas.balmat@duke.edu)
[†] Department of Statistical Science, Duke University, Durham, NC 27708 (jerry@stat.duke.edu);

10^1, respectively, this system is efficiently solved in R using the `lm()` function[*], while larger designs, where $n \times p$ is on the order of $10^7 \times 10^3$, present special storage and computational challenges. For example, creation of a 50,000,000 x 2,000 numeric design matrix in R, which encodes in double floating point format (eight bytes per cell), requires $10^{11} \times 8$ bytes, or approximately one terabyte, of memory. Further, to generate $X'X$ with standard R functions, that is using `t(X)%*%X`, requires an additional copy of X for the transpose[†], for a total memory requirement of approximately two terabytes (note that standard R functions require objects to be completely contained in on-line memory). Physical memory constraints are easily overcome by using R x64 (the 64 bit version with increased memory addressing) installed on a system with on-line memory sufficient for the problem to be solved. In addition to memory constraints, solution feasibility is practically limited by the processing time required to compose $X'X$ and $X'Y$ and finally to solve the corresponding OLS equation. For even large designs, on the order of $10^7 \times 10^3$, construction of $X'Y$ and solution of $X'X\hat{\beta} = X'Y$, using QR decomposition through standard R functions `qr()` and `qr.coef()`, are accomplished in minutes, leaving the majority of processing time to the construction of $X'X$. (*The story changes a bit with fixed effects of dimension above 30,000, where Cholesky decomposition gives a significant performance advantage over QR, but that seems a bit too much detail for the intro*). Given system memory sufficient to store X', X, and necessary intermediate computational values, the time required to execute necessary row and column products can be significant (our $10^7 \times 10^3$ example involves order 10^6 row-column products, each consisting of order 10^7 cell products and additions, for a total operation count of order 10^{13}). This paper presents a method for constructing $X'X$ that is

[*] `lm()` is the standard linear model regression function of R (cite R-docs)
[†] This can be confirmed by monitoring total memory usage while executing the R command `sum(t(X)%*%X)`

very efficient in a fixed effects OLS model environment, particularly in a sparse[*], high observation count (order 10^7 and above) and high dimension fixed effects (levels of order 10^2 and above) setting, achieving solution times measured in minutes. Additionally, an efficient algorithm is presented for estimating robust, clustered standard errors[†]. Traditional clustered standard error estimation involves the variance equation $V(\hat{\beta}) = (X'X)^{-1}X'UX(X'X)^{-1}$, where U is the $n \times n$ "sandwich" matrix of within cluster residual covariances[‡], and is estimated by setting all inter-cluster errors, $(\epsilon'\epsilon)_{cluster}$, to 0 (on the assumption of inter-cluster independence). For observation counts, n, on the order of 10^7 and fixed effects on the order of 10^3 the $V(\hat{\beta})$ equation involves operations on the order of 10^{19}, which is generally impracticable using standard linear algebra operations. The algorithm presented uses a sparse indexing method along with piece-wise summing of cluster row-column products in solving $V(\hat{\beta})$ to achieve solution times measured in tens of minutes for problems on the order of those mentioned above.

Data Sets, Models, and Research

Need statement from Alex on utility of fixed effects models

It is said that the social sciences are in the midst of a "data revolution," that "an explosion of information is upon [it]," and that new methods are emerging to "leverage the capacity to collect and analyze data with an unprecedented breadth and scale" (Connelly et al. 2016; Liu and Guo2016; Lazer et al. 2009). *Cite estimates of rates of data volume increase.* However, many

[*] For more on sparse matrix principles and methods, see (cite)

[†] Robust and clustered standard errors are popular in social science models, where influential independent variables are correlated within groups or subjects, but not explicitly included (generally not available) in the model (for more, see cite)

[‡] cite Huber-White or other "sandwich" references

recently developed methods, such as machine learning and text analysis, appear to be a direct response to the availability of and targeted to large, unstructured on-line data sets (cite). For research in the social sciences using large structured data sets, such as the Current Population Survey (U.S. Bureau of Labor Statistics, cite), GDP forecasts (U.S. Bureau of Economic Analysis, cite), and various socioeconomic data sets made available by the U.S. government (cite data.gov, NASA's Socioeconomic Data and Applications Center, http://sedac.ciesin.columbia.edu/) least squares regression, which estimates the expected value of a dependent variable given combinations of levels of a set of covarying independent variables, remains a popular model fitting method[*]. Since Gauss's time, additional, specialized methods, such as quantile and logistic regression, have been developed and Gauss's original method has become known as ordinary least squares (OLS) regression. A common feature in social science research is to employ models with categorical independent variables, such as gender, ethnicity, region, industry, and employment status. When all categorical variable levels of interest are represented in the data, they are termed fixed effects and the model becomes a fixed effects OLS model (verify, cite reference). Because fixed effects are not necessarily ordinal or even numerical, parameter estimates for individual levels provide an estimate of the difference in mean response between the group of observations for a given level and those for a corresponding reference level, for instance difference in mean annual wages between women and men, where men are the reference category. Also, because the fixed effects OLS model produces response estimates with minimum sum of squared deviations to within-level mean, when the effect of level itself is not being studied, a fixed effect can be considered a control variable, where the

[*] Credited to K.F. Gauss in the nineteenth century (cite Stigler). Is an estimate of the proportion of models fit using least squares available?

response is adjusted for mean effect of individual levels prior to estimating model parameters of interest. An example of this would be a model to measure disparity in pay by gender (one fixed effect), while controlling for geographic region, occupation, experience category, and race (controlling fixed effects). Interesting examples of models that use categorical variables for control or study include the Salk polio vaccine studies where individual family fixed effects were modeled to account for confounding of sibling health and hygiene practices (cite Salk polio studies) and a study of youth traffic fatalities with respect to the minimum drinking age imposed by state law, where differences in fatality rates were measured by a variety of categorical demographic classifiers, such as sex, race, urbanicity, and affluence (O'Malley and Wagenaar, 1990).

Value of Efficient Methods in Research

Cite a couple of references on valuation methods.

Later, under *Implementation*, eight methods of solving fixed effects problems are evaluated. For large problems, only two demonstrate efficiency sufficient to be considered practical. When applied to the largest problem considered, one of the efficient solutions outperforms the other by a factor of eight, solving the problem in ninety minutes as opposed to twelve hours. This difference in performance presents an opportunity to solve five problems per workday compared to two problems in three days. Alternatively, five researchers have an opportunity to solve one problem per day compared to two researchers in three days, enabling exploration of a greater variety of ideas, data sets, and verification plans, translating to an overall increase in research output (cite reference). Additionally, higher efficiency should reduce operating costs due to increased utilization of organizational information technology and computation resources (cite).

Motivation for Sparse Matrix Methods

Consider the trivial example of composing *X'X*, where *X* is a 1,000,000 x 2,000 matrix with all elements equal to 0. Of course, the result is a 2,000 x 2,000 matrix of zeros since all row and column products are zero, and we instantly translate the all element zero case to an all zero result. But R makes no such inference. It naively checks data types and dimensions then begins the task of summing the products of billions of row and column elements, each known by us to be zero. In trials, creation of *X'X* from a 10,000 x 2,000 matrix *X* with

```
X <- matrix(data=0, nrow=20000000, ncol=2000)
Z <- t(X)%*%X
```

requires over one hour to complete. A simple alternative solution to this trivial case is

```
Z <- matrix(data=rep(0, 4000000), ncol=2000)
```

which involves no math operations and executes immediately.

Suppose that we are given a large, sparse matrix *X* (25,000,000 x 2,000 with a high proportion of elements equal to 0) along with a list of elements (row *i*, column *j*) that are non-zero. An efficient alternative to generating *X'X* with matrix operations[*] is to accumulate products of pairs of non-zero elements into corresponding positions of the resulting *X'X* matrix. Since *X'X* is symmetric, additional efficiency is gained by accumulating the upper triangle only, then copying its transpose to the lower triangle.

Example Data Set and Model

The U.S. Office of Personnel Management (OPM) maintains employment and human capital[†]

[*] `Z <- t(X)%*%X`

[†] For additional information on human capital from an OPM perspective, see https://www.opm.gov/policy-data-oversight/human-capital-management/

records for over one million non-DOD U.S. federal employees in what it calls the central personnel data file (CPDF)[‡]. CPDF "Status" records[§] for fiscal years 1988 through 2011 were supplied by OPM to the Human Capital Project at Duke University in response to a Freedom of Information Act request. Over 28,000,000 records were provided, each describing the human capital profile of active federal employees as of September 30 of the corresponding fiscal year; there exists one record per employee per year. Important data elements are: sex, race, age, education level, bureau employed in, occupation, and fiscal year of observation (terminology is from OPM's Guide to Data Standards[**]). Current research using these data have identified systematic changes in federal employee pay over time, along with disparities in pay by gender and race (cite Bolton and de Figueiredo 2015, 2016). For illustration, we will use a fixed effects model (similar to those used by Bolton and de Figueiredo) that measures disparity in pay by employee declared gender and race. The model is

$$y = \beta_0 + \beta_{sex} + \beta_{race} + \beta_{sex,race} * sex * race + \beta_{age} * age + \beta_{age2} * age^2 + \beta_{ed} * ed_{years} + \beta_{bur} + \beta_{occ} + \beta_{year} \quad (1)$$

where y is the logarithm of *basic pay* (OPM variable for pay) and the βs are linear effects of an employee's *sex*, *race*, *sex* and *race* interaction, *age*, square of *age*, *education* (years beyond HS), *bureau* (agency employed in), *occupation*, and *fiscal year*. *Sex*, *race*, *bureau*, *occupation*, and *year* are discrete fixed effects while *age* and *education* are continuous. Fixed effects parameter estimates measure the difference in expected value of log(*basic pay*) between a given level and a reference level. Table 1 lists reference levels for each effect. For this model, observations are limited to full time employees with non-zero *basic pay* values and valid, non-empty values in

[‡] cite OPM general CPDF description
[§] Status records, https://www.fedscope.opm.gov/datadefn/aehri_sdm.asp
[**] OPM GDS, https://catalog.data.gov/dataset/guide-to-data-standards-gds

each independent variable, giving a total included record count of 24,574,480. In constructing the design matrix X, each fixed effect C_j is expanded into p_j-1 indicator columns, one for each non-reference level of C_j, where p_j is the number of distinct levels of C_j. Element (row) i of indicator column $X_{j,k}$ corresponding to level k of fixed effect C_j contains a 1 if observation i is encoded with the kth level of C_j and contains a 0 otherwise. Table 1 lists the number of non-reference levels by effect. Including columns for the constant, continuous, and interaction variables gives a design matrix of dimension 24,574,480 x 1,228. Levels within a fixed effect are mutually exclusive since an observation can be encoded for a single level only, so that each row of the design matrix X has at most one column for each fixed effect coded as a 1, with the remainder necessarily containing 0. Large p_j values cause overall low density (proportion of 1 entries) in X and it is this property that we exploit in efficient construction of $X'X$. Table 1 lists mean density by fixed effect (average density of columns associated with each effect). Overall design matrix density in the OPM data set relative to model 1 is 0.006, making it quite sparse and a good candidate for algorithmic and solution comparison.

Table 1. Pay Disparity Fixed Effect Reference Levels, Number of Non-ref Levels, and Density

Effect	Reference Level	Non-Ref Levels	Density
Sex	M (Male)*	1	0.4800
Race	E (White)*	4	0.0821
Bureau	114009000 (Veteran's Administration)**	409	0.0020
Occupation	303 (Miscellaneous Clerk and Assistant)**	791	0.0012
Year	1988***	23	0.0420

* customary reference levels for pay equity research
** highest marginal frequency
*** first year in study data

Review of Available Solutions

Many statistical software programs recognize fixed effects, or categorical variables, in solving regression problems. Popular solutions such as `proc glm` from SAS (cite), the `regress` command of Stata (cite), and the `lm()` function of R (cite) implement a common method of expanding categorical variables into columns of the design matrix, one for each level, eliminating the column corresponding to a user declared or system selected reference level to avoid linear dependence, and finally solving the normal equations $X'X\hat{\beta} = X'Y$ for $\hat{\beta}$[††]. The *Review of Available Solutions* appendix evaluates popular software solutions for solving large OLS regression problems. Since this paper targets an R audience, solutions are limited to R functions and packages available at time of writing, including `lm()`, `biglm()`, `bigglm()`, `biglm.big.matrix()`, `SparseM.slm.fit()`, and `speedlm()`. Each addresses efficiency obstacles in a variety of ways, some targeting memory constraints only, others addressing both efficient computation and use of memory. Some employ parallel processing methods, others create external operating system files when a design matrix exceeds on-line memory capacity. Example data sets and models are employed to illustrate performance features of various solutions. Additionally, model 1 is fit, when possible, using the example OPM data set described above. Included in the *Solutions* appendix is a problem respecification alternative commonly known as *demeaning*, where continuous independent variables are transformed to deviations from within-level means. The technique has advantages and penalties (cite) and is included for completeness, covering the simple case where, say, only continuous variable parameter estimates are desired, without estimates of standard errors. Demeaning can transform a complex, high

[††] Verification of fixed effect expansion into binary columns with that of the reference level omitted can be accomplished with the `model.matrix()` function in R and the `base` option of the `regress` command in Stata.

dimension, computationally intensive fixed effects structure into a simple, low *p*-dimension OLS problem, easily computed with `lm()`, even with 100,000,000 or more observations.

Design Matrix and Operation Density

The concept of design matrix density has been introduced along with a suggested relationship to efficient solution methods. More specifically, a low density design matrix presents an opportunity to limit computations strictly to those that affect the final result, in the case of matrix multiplication, products that are non-zero. The idea is to identify matrix cells, or elements, that contain non-zero entries and operate solely on those. If the proportion of non-zero entries is low (a low density matrix) then the ratio of effective operations to total operations is high and computation is efficiently executed. Consider a design matrix of dimension $n \times p$, such that p consists of p_x columns for continuous variables (including a constant) and p_k columns, one for each level of a single fixed effect. Note that the fixed effect actually has p_k+1 levels, but one "reference" level is omitted to avoid linear dependence of the fixed effect and constant columns. Effect levels are mutually exclusive within observation since, at most, one level applies. Also, reference level observations have all fixed effect columns coded as 0. Therefore, within the np_k fixed effect cells, the maximum number of cells coded as 1 is n-1 (at least one reference level observation must exist), giving a density upper bound within the fixed effect columns of $D_{Mk} \leq \frac{n-1}{np_k} < \frac{n}{np_k} = \frac{1}{p_k}$. Letting d_j be the density of the jth continuous column, overall design matrix density has an upper bound of $D_M \leq \frac{n\left[\sum_{j=1}^{p_X} d_j\right] + n - 1}{np}$. Assuming high density in the continuous columns (near 1.0), this becomes

$$D_M \leq \frac{np_X + n - 1}{np} < \frac{p_X + 1}{p} = \frac{p_X + 1}{p_X + p_k}.$$

It follows that density decreases with increasing fixed effect levels. In fact, the rate of change per additional level (assuming high density continuous columns) of the upper bound of D_M is

$$\Delta D_M = \frac{np_X + n}{n(p+1)} - \frac{np_X + n}{np} = \frac{p_X + 1}{p+1} - \frac{p_X + 1}{p} = -\frac{p_X + 1}{p(p+1)} \propto -\frac{1}{p^2} \ (approximate)$$

$$= -\frac{1}{(p_X + p_k)^2}.$$

It is interesting to note that the upper bounds for both D_M and ΔD_M depend solely on p which, for high dimension problems, is dominated by p_k, making density predominantly a function of the number of fixed effect levels. The upper panel of figure 1 plots total design matrix density against number of fixed effect levels from 1 through 500 (less the reference level) for a model with 1, 2, 5, and 10 continuous variables (plus a constant term) and a single fixed effect. It is seen that density rapidly decreases as the number of levels increases, presenting efficiency opportunities for all but the lowest dimension problems. In a multiple fixed effects problem, with m effects and $p_K = \sum_{i=1}^{m} p_{ki}$ fixed effects columns, density is further reduced since np_K cells are populated with a maximum of $m(n\text{-}1)$ non-zero values. Given m fixed effects, an upper bound on matrix density is

$$D_M \leq \frac{np_X + m(n-1)}{n(p_X + p_K)} < \frac{p_X + m}{p_X + p_K} = \frac{p_X + m}{p}.$$

The lower panel of figure 1 shows, for models with 5 continuous variables and 1, 2, 5, or 10 fixed effects, significant reduction in density as the combined number of levels, p_k, increases.

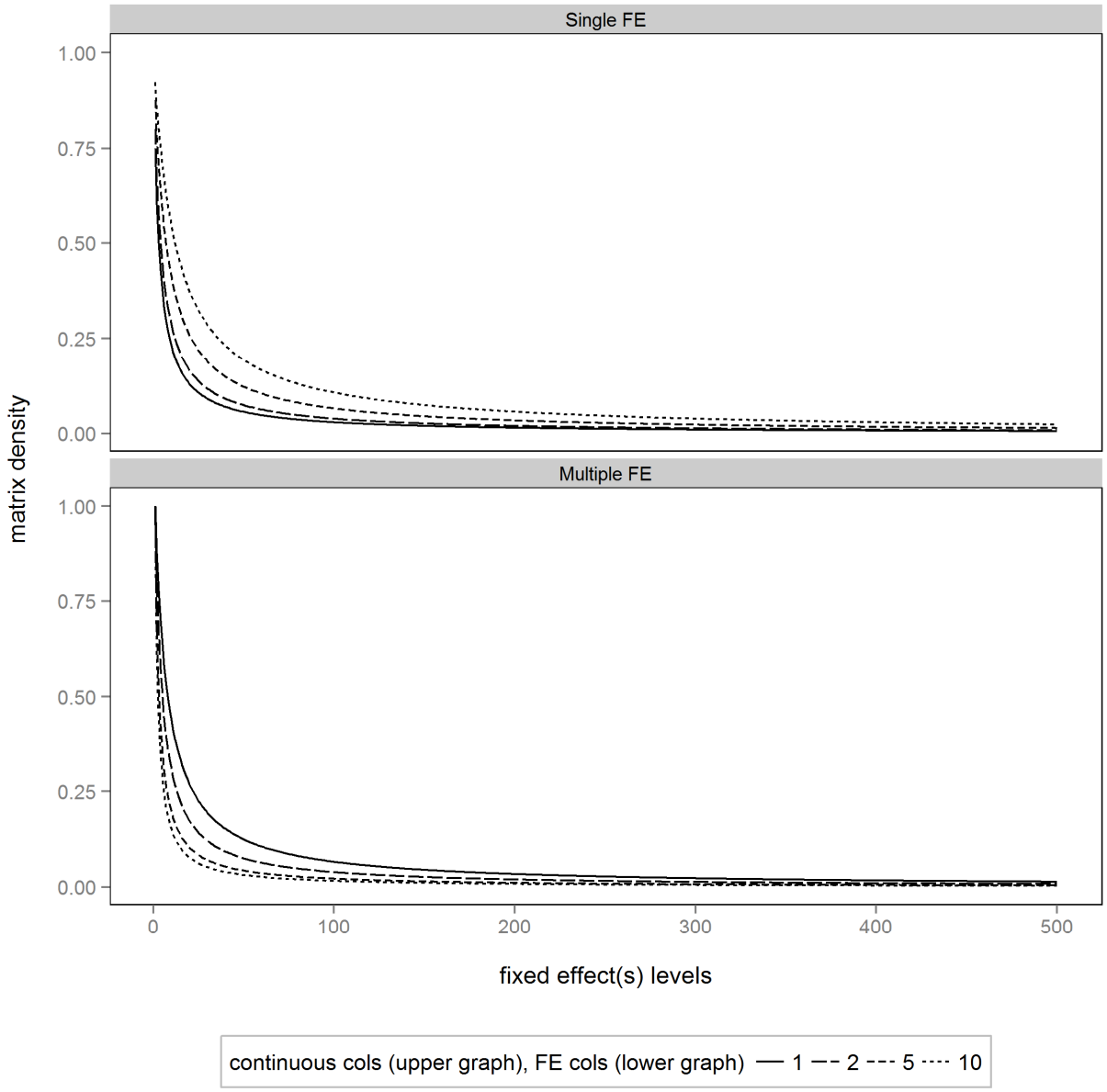

Figure 1. Design Matrix Density vs. Increase In Levels of Fixed Effects. Upper panel, one to ten continuous columns with a single fixed effect. Lower panel, a single continuous column with one to ten fixed effects with common dimension (number of levels).

Design matrix density measures the proportion of elements that contribute information to and cannot be eliminated from the $X'X$ operation. Figure 1 indicates that, for even low dimension

fixed effects, the proportion of essential matrix elements is relatively small, typically less than 0.01 for problems in the class we consider. However, design matrix density merely characterizes an opportunity for efficient computation. More important than matrix density itself is what we will call *operation density*, the proportion of standard matrix operations (sums of products of row and column elements) that contribute to the resulting $X'X$ product. Recall that, given X ($n \times p$), the matrix product $X'X$ has $p \times p$ cells, the ijth being the product $(X_i)'X_j$, which involves the addition of n element products, for a total of $n+n$-1 basic operations (n multiplication and n-1 addition operations). The total number of basic operations to produce $X'X$, using standard matrix methods, is then $(2n\text{-}1)p^2$. Consider a model with p_x continuous variables, each having (matrix) density of 1.0, and a single fixed effect with p_k levels that are uniformly distributed throughout observations (giving fixed effect indicator column density of $d_k=1/p_k$). The number of operations for products of continuous columns and continuous columns is $(2n-1)p_X^2$, for products of continuous and fixed effect indicator columns is $(2nd_k-1)p_Xp_k = (2n-p_k)p_X$, and for products of fixed effect and fixed effect columns is $(2nd_k^2-1)p_k^2 = 2n-p_k^2$. Operation density for this model is then

$$D_O = \frac{(2n-1)p_X^2 \; + \; (2n-p_k)p_X \; + \; 2n-p_k^2}{(2n-1)p^2}$$

$$\leq \frac{(2n-1)p_X^2 \; + \; (2n-1)p_X \; + \; 2n-1}{(2n-1)p^2}$$

$$= \frac{p_X^2 \; + \; p_X \; + \; 1}{p^2} \propto \frac{1}{p^2}.$$

The rate of change, per additional level of the fixed effect, of the upper bound of D_O is

$$\Delta D_O = (p_X^2 + p_X + 1)\left[\frac{1}{(p+1)^2} - \frac{1}{p^2}\right]$$

$$= -(p_X^2 + p_X + 1)\frac{2p+1}{p^2(p+1)^2} \propto -\frac{1}{p^3} \quad (approximate).$$

Figure 2 shows the relationship of operation density to fixed effect levels. The upper panel plots ΔD_O against fixed effect levels for a model with 1, 2, 5, or 10 continuous variables and one fixed effect. The lower panel plots ΔD_O against fixed effect levels for a model with 5 continuous variables and 1, 2, 5, or 10 fixed effects, each with dimension as indicated on the x-axis. A rapid decrease in D_o is apparent even when fixed effects are of low dimension, further evidence of an opportunity of computational efficiency by operating solely on non-zero cells of X. It should be noted that an increase in fixed effect levels does not increase the number of non-zero matrix elements or operations, but does increases the number of standard $X'X$ operations, hence the relative decrease in D_O. Another important feature of D_O is that since, within a given fixed effect, indicator columns are orthogonal, the p_k^2 elements of $X'X$ corresponding to it are necessarily diagonal ($p_k \times p_k$). This means that, for example, a model with four continuous variables (plus a constant) and one fixed effect with 96 levels has $95^2 - 95 = 8{,}930$ of 10,000 total $X'X$ elements known in advance to be zero. Any operations involving elements from columns i and j within a fixed effect, where $i \neq j$, are unnecessary. Also, since a fixed effect is represented in X as a series of indicator columns, one for each level, the p_k diagonal elements of $X'X$ corresponding to an effect are simply $(X_j)'X_j = n_j$, where j is the indicator column index and n_j is the observation count for the corresponding fixed effect and level. Thus, multiplication operations within fixed effects can be substituted with simple counts.

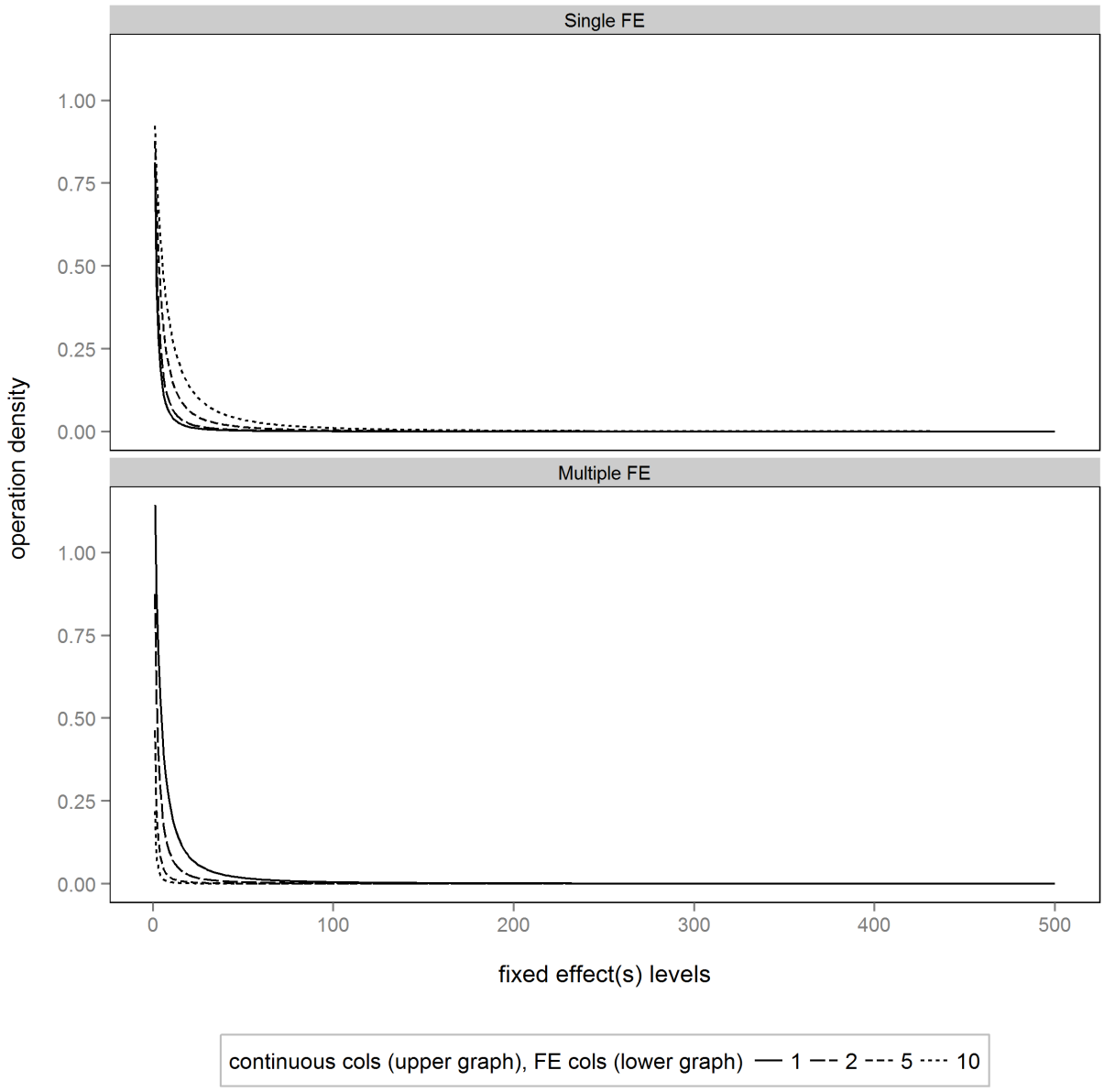

Figure 2. Operation Density vs. Increase in Fixed Effects Levels. Upper panel, one to ten continuous columns with a single fixed effect. Lower panel, a single continuous column with one to ten fixed effects with common dimension (number of levels).

We have established that matrix density is strongly influenced by fixed effect dimension, that operation density is a result of matrix density, and that theoretical measures of operation density indicate an opportunity, for sufficiently sparse problems, to eliminate a high proportion of

standard matrix operations in producing *X'X*. Analysis of figure 2 indicates that for models involving fixed effects with 50 or more levels, the proportion of effective operations (those that must be executed) approaches zero. Models with fixed effects in this range are very common (50 or more occupations, states, counties, schools, etc.), making implementation of a solution method both relevant and appealing. The example OPM data set and model, with three continuous and five fixed effects variables, has an operation density of approximately 0.00001[*], making it a candidate for computational efficiency improvement.

Method: Efficient Indexing

When a design matrix, *X*, exhibits low operation density, we know that relatively few operations are actually necessary to produce *X'X*, or more correctly stated, relatively few elements of *X* need to be operated on. The initial challenge, then, is to develop an efficient method of expanding fixed effects into a representation of design matrix indicator columns that index strictly non-zero effect levels in the design. The ideal solution to this is:

- efficient, solving in minutes, problems on the order of the example OPM federal pay disparity model, and
- general, using base R functions, making it readily portable and robust

Two base R functions that are useful for general purpose indexing are `which()`[†] and `match()`[‡]. `match()`[‡]. `which()` is generally easy to implement: to generate a vector of indices, positions within a column `X` that contain a particular value `a`, we simply execute `which(X==a)`. By

[*] Operation density can be derived from a constructed *X'X* matrix using the fixed effect indicator product entries, since they report the number of intersecting non-zero elements and indicate the number of pseudo-multiplication operations performed

[†] R `which()` function, https://stat.ethz.ch/R-manual/R-devel/library/base/html/which.html

[‡] R `match()` function, https://stat.ethz.ch/R-manual/R-devel/library/base/html/match.html

executing `which()` within `lapply()`[*], once for each unique non-reference level in a fixed effect column, we generate a list of p_k variable length vectors, x_1, ..., x_{pk}, one for each fixed effect level, where x_j contains the indices (observation row positions) corresponding to observations coded with effect level j. The density of x_j is n_j/n, where n_j is the number of observations coded with level j and n is the total number of observations. The x_j vectors indicate strictly non-zero, effective matrix elements and are the sole ones to be operated on. One drawback to using the `lapply(which())` method is that `which()` is invoked independently for each of the p_k fixed effect levels, requiring a complete scan of a fixed effect column for each level. This is computationally expensive with large p_k. Additional efficiency is gained by executing `lapply()` in parallel[†], instructing multiple worker processes (cores) to index blocks of levels simultaneously. Specifying n_c cores reduces execution time to slightly greater than $1/n_c$, but incurs, in a MS Windows sockets environment, additional cost in time and memory to export entire fixed effect vectors to each core. An alternative is to `order()`[‡] observation indices by fixed effect level, identify level transition positions, using `match()`[§], and parse all indices between transition points, giving sets of non-zero observation indices by effect level. For large n, the default implementation of `order()` is relatively inefficient, but its inefficiencies are compensated for by the speed of `match()`. An improvement of `order()` is to convert a fixed effect column to numeric, using `factor()`[**], then order the resulting indices using `sort.list(method="quick")`[††]. Since all indices are needed by `order()` and `sort.list()`

[*] R `lapply()` function, https://stat.ethz.ch/R-manual/R-devel/library/base/html/lapply.html
[†] using `parLapply()` from the `SNOW` or `parallel` packages (cite)
[‡] R `order()` function, cite https://stat.ethz.ch/R-manual/R-devel/library/base/html/order.html
[§] R `match()` function, cite
[**] R `factor()` function, cite https://stat.ethz.ch/R-manual/R-devel/library/base/html/factor.html
[††] `sort.list()` is a variant of `order()`, cite https://stat.ethz.ch/R-manual/R-devel/library/base/html/order.html

simultaneously, the `order()`, `sort.list()`, `match()` methods are not good candidates for parallelization. So, which method is most efficient? The answer depends on n, p_k, and n_c. Figure 3 shows simulated computation times, in minutes, to index n = 25,000,000 observations in p_k = 10, 50, 100, 250, 500, 1,000, and 5,000, 100,000, and 1,000,000 random, uniformly distributed fixed effect levels using n_c = 16 cores (for `parLapply(which())` method). Results are mean elapsed time (as reported by `proc.time()`) of ten iterations executed on a dedicated server. The clear winner is to convert a fixed effect to a factor then use `sort.list()` and `match()` to parse indices. The important lesson in this exercise is to measure computation time of critical elements of an algorithm and to test alternative methods that exploit high performance features of readily available functions. Although, for indexing, `which()` may come to mind first, it is generally not an efficient method for the problems of the size we consider and, although conversion to (numeric) factors and use of `sort.list()` and `match()` are not as intuitive, they remain base R functions and perform very well, eliminating minutes to hours from overall computation time by comparison.

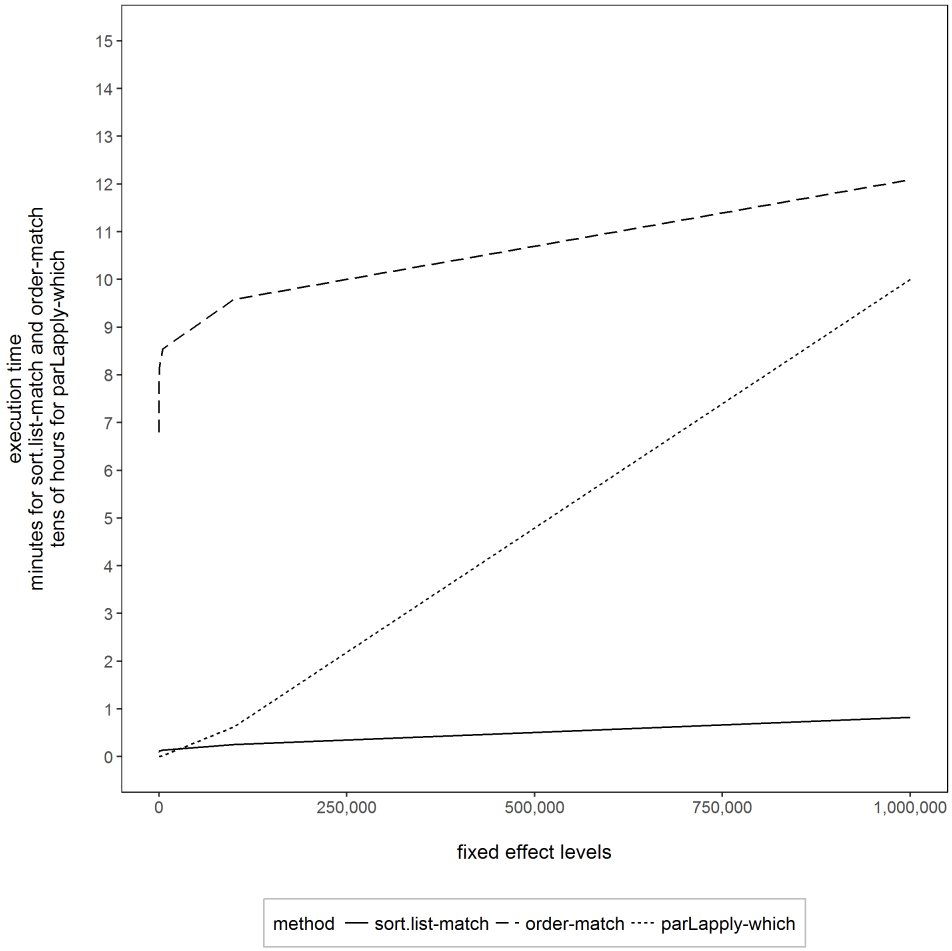

Figure 3. Three Indexing Methods: Comparison of Time to Index a Single High Dimension Fixed Effect Column. `sort.list-match` and `order-match` measured in minutes; due to lengthy execution time, `parallel-apply-which` is measured in hours divided by 10.

Equipped with minimal sets of fixed effects observation indices, those that must be included in matrix operations, we now proceed with $X'X$ construction. First, knowing that $X'X$ is symmetric, we will construct the diagonal and upper triangle only, then copy the transpose of the upper triangle to the lower triangle. Ignoring (for simplicity, although the algorithm supports) interactions, there are four types of "products" that must be accommodated:

- the transpose of the constant (1) vector "multiplied" by continuous and fixed effects columns

- the transpose of a continuous column "multiplied" by a continuous column
- the transpose of a continuous column "multiplied" by a fixed effect column
- the transpose of fixed effect column "multiplied" by a fixed effect column

Multiplied is quoted to indicate that a result identical to that of standard matrix multiplication operations will be produced, but using where possible, fewer and simpler operations. Each of the four "product" types uses a distinct combination of operations:

- Row 1 (constant times continuous and fixed effects columns) consists of sums of corresponding columns. Column 1 is simply *n*, the total number of observations. For continuous columns, `sum(`x_j`)` is used. For fixed effects columns, `length(`x_{ki}`)` is used, where x_{ki} is the previously constructed non-zero index vector for the *i*th level of the *k*th fixed effect.
- Products involving two continuous columns are composed with `sum(`x_i`*`x_j`)`, which is equivalent to the standard matrix operation `(`x_i`)'`x_j.
- Products of continuous and fixed effects columns equate to the sum of continuous x_j elements corresponding to non-zero positions of fixed effects and use `sum(`x_j`[`x_{ki}`])`, where x_{ki} is the previously composed index vector for the *i*th level of the *k*th fixed effect.
- Products of fixed effects with fixed effects have three types:
    - On-diagonal products are simply the number of observations coded with the corresponding level (since the product of a fixed effect indicator variable by itself is the sum of n_{ki} products of 1.0 and 1.0, where n_{ki} is the number of observations coded with level *i* of fixed effect *k*).
    - Off-diagonal products of columns from within a single fixed effect are zero, due to orthogonality.

- Off-diagonal products of columns from distinct fixed effects use the length of the intersection of corresponding index columns, equating to the count (sum of products of 1.0 and 1.0) of positions where each index is non-zero. Specifically, $(X'X)_{k1i,k2j}$ = `length(intersect(`x_{k1i}`, `x_{k2j}`))` is used, where x_{k1i} is the index vector for the *i*th level of fixed effect 1 and x_{k2j} is the index vector for the *j*th level of fixed effect 2.

Note the limited requirement for matrix and standard multiplication operations. In fact, of the p^2 vector products required in standard matrix multiplication, the proposed method performs only $\frac{p_x(p_x-1)}{2}$, those for continuous variables. The remaining operations (`length(), intersect(), sum()`) substitute multiplication with very efficient base R vector functions applied to sparse vectors.

Explain construction of interaction X'X elements

Explain structure of parallel implementation

Solving the OLS Normal Equations

With *X'X* constructed (after copying the transpose of the constructed upper triangle to the lower triangle[*]), we now turn to solving the OLS normal equations, $X'X\hat{\beta} = X'Y$ for $\hat{\beta}$. We see that, in addition to *X'X*, *X'Y* is need and it should be noted that a complete expanded *X* matrix (consisting of columns for continuous and fixed effect indicators) is not available to us. However, *X'Y* is simply a p x 1 vector with p_x positions consisting of inner-products of continuous *X* columns and

[*] with `X'X <- X'X + t(X'X)-diag(x=diag(X'X), nrow=p)`; note that `X'X` is initially 0_{pxp}; also note the subtraction of a redundant diagonal

Y, followed by p_k fixed effect positions, one for each non-reference level of each effect and containing the sum of Y elements corresponding to the associated effect and level. $X'Y$ positions corresponding to fixed effects are efficiently constructed using the non-zero indicators and indexing methods developed in composing $X'X$.

Of the various methods for solving $X'X\hat{\beta} = X'Y$ for $\hat{\beta}$, two that are well established and known for computational efficiency are QR decomposition[†] and Cholesky decomposition[‡] (for a discussion of linear algebra methods and efficiency, see *cite*). The efficiency of QR and Cholesky decomposition derive from a strategy of forming pairs of triangular matrices such their product represents the left hand matrix in the system being solved, $X'X$ in our case. In the case of Cholesky decomposition, an upper triangular matrix R is computed such that $R'R = X'X$, then $R'R\hat{\beta} = X'Y$ is solved in two stages: $R'W = X'Y$ for W then $R\hat{\beta} = W$ for $\hat{\beta}$. In addition to facilitating solution of $X'X\hat{\beta} = R'R\hat{\beta} = X'Y$, R can be used to solve $X'XW = I$, where $W = (X'X)^{-1}$, the computationally significant component of the homoskedastic parameter covariance equation $\text{cov}(\hat{\beta}) = \hat{\sigma}^2(X'X)^{-1}$. QR decomposition follows a similar strategy, but with matrices Q and R, where $QR = X'X$, Q is orthogonal, and R is upper triangular. In practice, the computational effort required to generate R (or Q and R) combined with solving two simpler systems tends to be less than in solving $X'X\hat{\beta} = X'Y$ using direct Gaussian elimination and the resulting equations and solution tend to exhibit improved numerically stability[§]. Although both QR decomposition and Cholesky decomposition produce near mathematically equivalent results, it has been the authors'

[†] For more on QR decomposition, see (cite). QR decomposition is implemented by the R function `qr()` (cite) and is the default algorithm used by `lm()` (cite).
[‡] for more on Cholesky decomposition, see (cite). Cholesky decomposition is implemented by the R function `chol()` (cite).
[§] cite

experience that, for large regression problems, Cholesky decomposition requires less time to solve both systems of equations, one for $\hat{\beta}$ and one for cov($\hat{\beta}$), than does QR decomposition. As an example, table CCC compares times to compute the QR decomposition, Cholesky decomposition, and inverse of *X'X* from two design matrices (one of size 5,000,000 x 6,200; the other 10,000,000 x 16,000) using the standard R functions `qr()`, `qr.solve()`[**], `chol()`, and `chol2inv()`[††]. The apparent four to one performance ratio gives Cholesky decomposition a significant advantage.

Table CCC. Time (in minutes) to compute decomposition and inverse of *X'X* using QR and Cholesky methods

Method	6,200 matrix	16,000 matrix
`qr()+qr.solve()`*	10.0	172.7
`chol()+chol2inv()`	2.5	44.1

* Computed using the slightly more efficient `LAPACK` option, as opposed to the default `LINPACK`

We will restricting our discussion to Cholesky decomposition. While solving various large regression problems, it was observed that as much as three-fourths of the entire computation cycle [composition of *X'X* (using efficient indexing as presented), computation of *R*, solution of $\hat{\beta}$, solution of cov($\hat{\beta}$)] was devoted to the final three items, all involving *R*. Although, algorithmically, *R* can be computed in simultaneous parallel operations, the base R functions `chol()` and `chol2inv()`are implemented as a sequence of row and column operations, with a single row or column being computed before subsequent rows or columns are begun. As part of

[**] `qr.solve(`*X*`)` computes the inverse of *X*. cite R reference.
[††] `chol2inv(`*R*`)` computes the inverse of *X* = *R'R*. cite R reference.

the current project, two Cholesky related functions were developed: one to compute the decomposition *R*, given a composed *X'X*, and one to compute $(X'X)^{-1}$ using *R*. Both are implemented in the C language (using package `Rcpp`[‡‡]) and employ parallel methods (using `OpenMP`[§§]). Source listings are in appendices *Parallel Cholesky Decomposition Algorithm* and *Parallel Algorithm to Compute Inverse of X Using the Cholesky Decomposition*. Function `choleskyDecomp()` computes *R* and function `cholInvDiag()` computes $(X'X)^{-1}$. Figure AAA compares the performance of the custom parallel algorithms to that of their base R counterparts, solid lines indicate execution times of custom functions, dashed lines for R functions. With performance of approximately eight times that of base R functions, equating to a savings of over five hours for the largest problem tested, the benefit of custom, parallel algorithm implementation is significant.

[‡‡] cite `Rcpp` reference
[§§] cite `OpenMP` reference

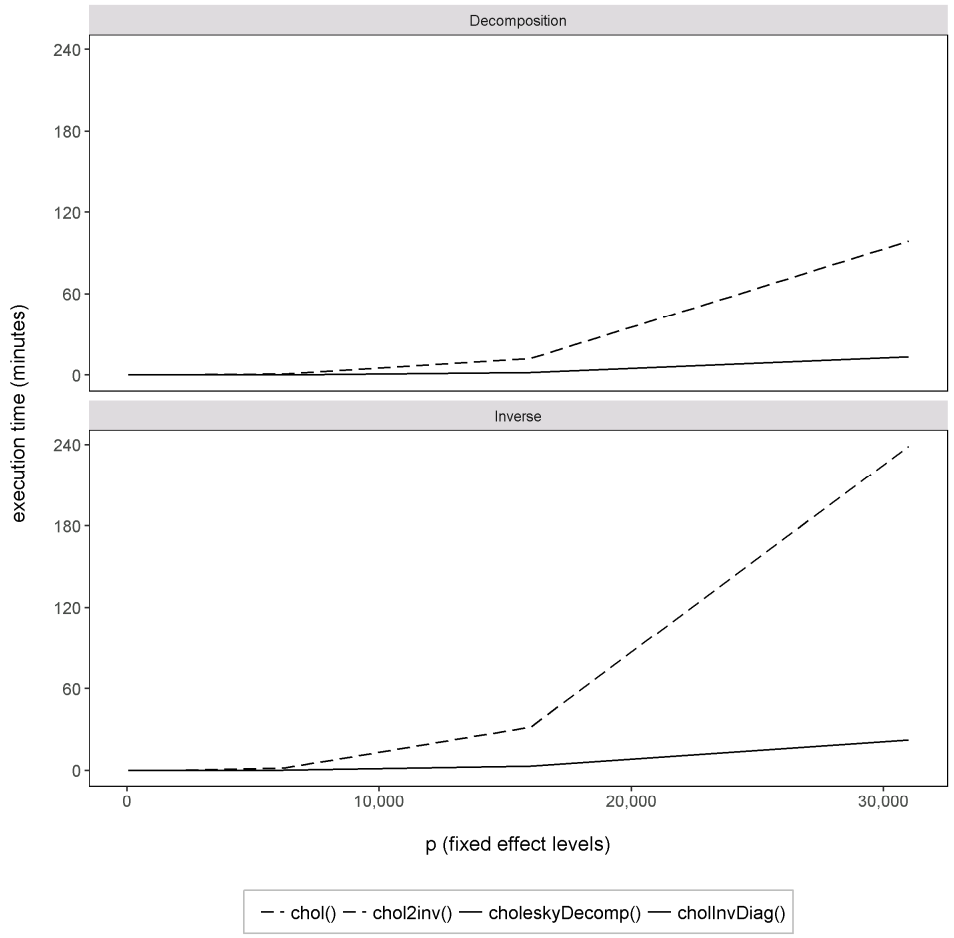

Figure AAA. Comparison of Execution Times to Compute Cholesky Decomposition and $(X'X)^{-1}$ Using Custom Parallel Algorithm and Base R Functions. Five independent data sets were generated for each level of p_k (number of fixed effect levels). Observation counts ranged from 10,000 (small p_k) to 25,000,000 (large p_k).

To verify consistency of results between the custom, parallel and base R functions, element-wise differences in computed R or $(X'X)^{-1}$ matrices of absolute value greater than 10^{-10} were tested for using `length(which(abs(R1-R2)>1e-10))` and `length(which(abs(iR1-iR2)>1e-10))`, where `R1` and `R2` are custom and base Cholesky decomposition results, respectively, and `iR1` and `iR2` are custom and base inverses, respectively. None were reported.

Implementation: `feXTX()` with parallel `choleskyDecomp` and `cholInvDiag()`

While conducting research into U.S. federal employee pay disparity, the Human Capital Project at Duke University developed a requirement for fitting high dimension fixed effects OLS models to a large set of historical federal employee human capital data, described under *Example Data Set*, above. A simultaneous requirement arose from the Synthetic Data Project at Duke, to certify a synthetic U.S. federal human capital data set to be used in a public use verification server system (cite Reiter verification measure work and synthetic data disparity VM paper). After various attempts to use the available solutions previously described, with limited success, the teams decided to develop a solution tailored to solving general large fixed effects problems while taking advantage of the performance and efficiency opportunities mentioned so far in this paper. The result is an implementation of the *X'X* construction and Cholesky decomposition methods presented in previous sections that the teams call `feXTX()`. With the exception of parallel `apply()`[***] and the custom Cholesky decomposition functions, `feXTX()` uses base R functions. The only necessary package is for parallel operations (`parallel` and `SNOW` have been tested). `feXTX()` takes the following parameters:

- `data`: source data frame containing observations (vectors must be named)
- `Y`: the (character string) name of the dependent variable – vector with this name must appear in `data`
- `contX`: vector of (character string) names of continuous variables (corresponding columns must exist in `data`); specify `NULL` if none

[***] `parApply()` and `parLapply()` (cite parallel package reference)

- `fixedX`: vector of (character string) names of fixed effect variables (corresponding columns must exist in `data`); must contain at least one column name
- `refLevel`: vector of (character string) fixed effect reference levels in order of `fixedX`
- `interactionX`: two column array of interactions, each row an interaction; specify `NULL` if none
- `estBetaVar`: `"stdOLS"` for homoskedastic variances, `"robust"` for robust variances, `"cluster"` for clustered variances, or `"none"` for none
- `robustVarID`: (character string) column name in `data` containing values about which to cluster variances (vector with this name must appear in `data`)
- `nCoreXTX`: number of parallel cores to be used in beta solution (for the example pay disparity model, a value of 6 has proven to give good results – benefits of increasing cores is offset by the requirement to export data to them in a Windows environment)
- `nCoreVar`: number of parallel cores to be used in clustered variance estimation (`stdOLS` variances are not executed in parallel, the `robust` variance method uses data and cores created during the *X'X* construction step); for `clustered` variance estimates, a value of 4 (perhaps less) is optimal here, since clustered variance computations require significant export of data (Windows), while the computations are relatively efficient within each parallel process
- `solMethod`:
    - "`QR`" (default) for QR decomposition
    - "`Chol`" for Cholesky decomposition using base R `chol()` and `chol2inv()` functions

- "`Chol-Parallel`" for Cholesky decomposition using custom parallel `choleskyDecomp()` and `cholInvDiag()` functions

The value returned by `feXTX()` is a list with the following members:

- `status`: (character string) completion status ("" for success, otherwise error message)
- `Y`: (character string) echo of input `Y` parameter
- `contX`: (character string) echo of input `contX` parameter
- `fixedX`: (character string) echo of input `fixedX` parameter
- `refLevel`: (character string vector) echo of input `refLevel` parameter
- `interactionX`: (two column array or `NULL`) echo of input `interactionX` parameter
- `nX`: (numeric scalar) number of rows in `data`
- `beta`: (numeric vector) OLS parameter estimates for `Y ~ b0 + contX + indicator(fixedX|not ref) + interactionX`
- `vBeta`: (numeric vector) parameter estimate variances, as instructed in `estBetaVar`
- `estBetaVar`: (character string) echo of input `estBetaVar` parameter
- `XTX`: (matrix) $X'X$ matrix (the primary effort of this function!)
- `dimXTX`: (numeric scalar) column (and row) dimension of $X'X$
- `rankXTX`: (numeric scalar) rank of $X'X$ as reported by `qr()` or `chol()`
- `XTY`: $X'Y$
- `yEst`: (numeric vector) estimated `Y` values
- `yVar`: (numeric scalar) variance of observed errors, `sum(Y-estY)^2/df`
- `time`: matrix of execution time, row 1 for compute, row 2 for memory export (Windows)

A complete program listing of `feXTX()` is in the *R `feXTX()` Program Listing appendix*.

Include notes on loading and calling `feXTX()`

Program listings for `choleskyDecomp()` and `cholInvDiag()` are in appendices *Parallel Cholesky Decomposition Algorithm* and *Parallel Algorithm to Compute Inverse of X Using the Cholesky Decomposition*, respectively.

Performance of Solutions with Simulated Data and Models

We now compare the performance and results of `feXTX()` with other solutions presented. Before using the example OPM data set, since it is not available to the public, and to give the reader a ready means of testing various methods, simulated data sets with specifiable covariate relationships will be generated. The *Sample Data Generation* appendix contains an R script that produces a data frame, `feDat`, with one dependent (continuous numeric) vector `Y` and independent vectors X_1, X_2, X_3, X_4, and X_5, where X_1 and X_2 are continuous numeric vectors and X_3, X_4, and X_5 are categorical (character) fixed effects. `Y` is simulated as a linear function of the `X` values plus normally distributed error such that

$$Y = b_0 + b_1X_1 + b_2X_2 + b_{3i}X_{3i} + b_{4j}X_{4j} + b_{5k}X_{5k} + \epsilon \qquad (2)$$

where `i`, `j`, and `k` indicate levels of the fixed effects. Values for b_j and ϵ are available in the appendix. Table 2 lists test parameters (number of observations and number of levels for X_3, X_4, and X_5) used to generate simulated data sets for evaluation of execution times.

Table 2. Simulated Data Set Parameters for Execution Time Evaluation

Parameter Set	n (Observations)	Levels X_3	Levels X_4	Levels X_5
1	10,000	10	15	20
2	25,000	10	20	40
3	50,000	15	30	50
4	100,000	15	40	60
5	150,000	25	40	60
6	250,000	25	50	75
7	500,000	50	75	100
8	1,000,000	75	150	250
9	1,500,000	100	250	500
10	2,000,000	100	500	1,000
11	5,000,000	200	1,000	5,000
12	10,000,000	1,000	5,000	10,000
13	25,000,000	1,000	5,000	25,000

Also in the *Sample Data* appendix are functions for fitting model 2 to simulated data, using `lm()`, `biglm()`, `bigglm()`, `biglm.big.matrix()`, `SparseM.slm.fit()`, `speedlm()`, `lfe()`, and `feXTX()`. Note that appropriate packages must be installed in order to execute related model fitting functions. Figure 4 plots mean execution time by method using five independently simulated data sets for parameter sets 1 through 9. In addition to execution efficiency, the total on-line memory used by an algorithm is important, since it is generally limited and shared by multiple concurrent users and processes. As a quick assessment of memory required by each method, table 3 lists maximum differential of memory in use (from baseline prior to execution, as reported by Windows task manager) while the algorithms were executed using a single instance of data generated by parameter set 9 (the highest dimension set that all methods were executed with).

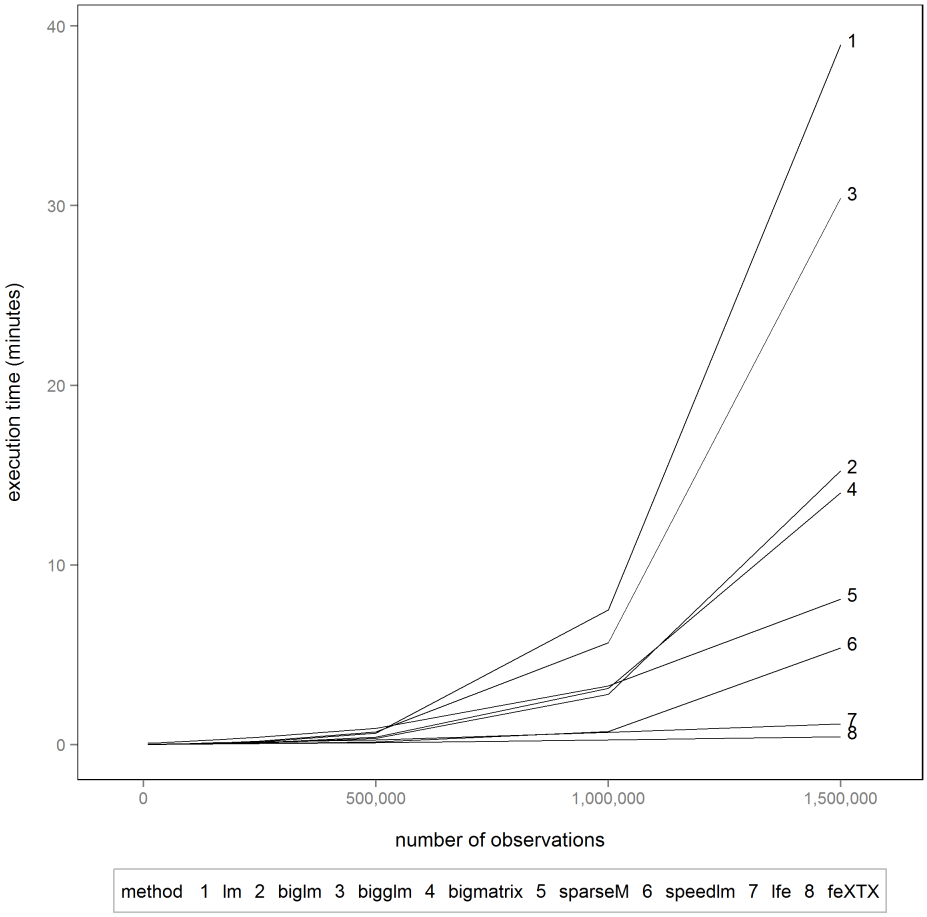

Figure 4. Mean Execution Time to Solve Graduated Fixed Effects Problems by Method. Execution time is the differential of "elapsed" time as reported by `proc.time()`. All processing was performed on a single dedicated server.

Table 3. Maximum On-line Memory used while fitting simulation parameter set 9

Method	Memory used (Gb)
`lm()`	19.3
`biglm()`	19.3
`bigglm()`	19.4
`biglm.big.matrix()`	0.80*
`SparseM.slm.fit()`	20.3
`speedlm()`	28.8
`lfe()`	10.4
`feXTX()`	0.50

* backing file created

Efficient execution is important, but accuracy of results is of the utmost importance. Considering the computational challenges of fitting high dimension models to large data sets, obtaining an authoritative set of parameter estimates for objective comparison is problematic. As an alternative, we might take the approach of comparing estimates to a consensus of those from established methods. If no major deviations are observed then we can state that, with data sets and models tested, our method introduces no error beyond that observed in methods currently in use. Table BBB shows, by method, the number of parameter estimates, using simulation parameter set 9, that differ with those generated by `feXTX()`, grouped by absolute value ranges listed in the column headers. There does seem to be consensus, with all methods except `lfe()` having all deviations of parameter estimates less than 10^{-7}. By default, `lfe()` does not estimate an intercept and fixed effect reference levels are not omitted, causing significant differences in estimated values. Alternate parameter estimates can be requested, but only by constructing estimable function specifications, which was not pursued.

Table BBB. Comparison of Parameter Estimates, Evaluated Solutions vs. `feXTX()`

Method	$0 < 10^{-12}$	$10^{-12} < 10^{-10}$	$10^{-10} < 10^{-7}$	$10^{-7} < 10^{-5}$	$> 10^{-5}$
`lm()`	369	478	3	0	0
`biglm()`	339	508	3	0	0
`bigglm()`	339	508	3	0	0
`bigmatrix()`	339	508	3	0	0
`sparseM()`	0	2	848	0	0
`speedlm()`	0	2	848	0	0
`lfe()`	0	0	2	1	847

In terms of execution and memory efficiency, `lfe()` and `feXTX()` appear to significantly outperform the others and maintain efficiency throughout the range of smaller data sets. Continuing with more demanding data sets (parameter sets 10, 11, 12, and 13) and limiting our study to `lfe()` and `feXTX()`, we see in figure 5 a continued divergence of execution time, with `lfe()` exhibiting exponential increase with respect to problem size (as observed with the other methods and smaller problems), while `feXTX()`remains nearly linear, executing the largest problem, with 25,000,000 observations and 31,000 fixed effects levels, in $1/8^{th}$ the time required by `lfe()`. Memory requirements to fit model 2 using parameters set 13 were ??? Gb and ??? Gb for `lfe()` and `feXTX()`, respectively.

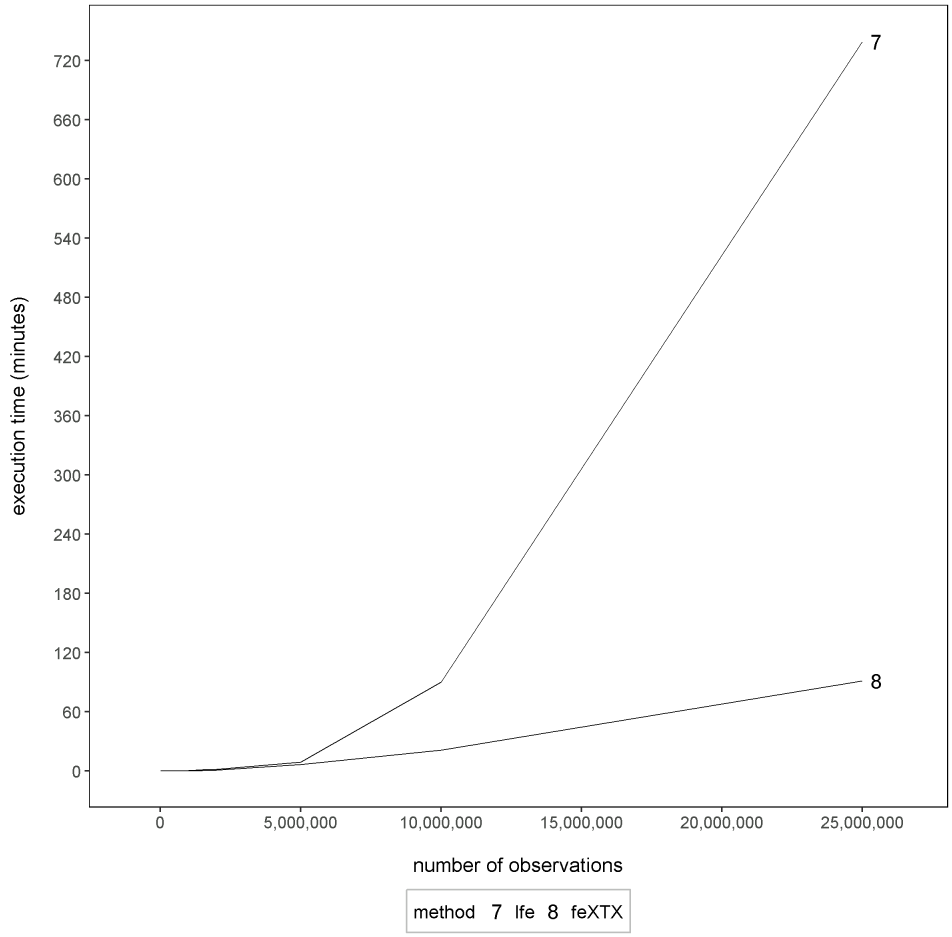

Figure 5. Mean Execution Time to Solve Graduated Fixed Effects Problems. Higher dimension effects, for `lfe()` and fe`XTX()` methods. Execution time is the differential of "elapsed" time as reported by `proc.time()`. All processing was performed on a single dedicated server.

Performance with OPM Data and Models

However compelling the case for efficiency is made using simulated data, ultimately we must assess utility with actual data and models. Performance and issues regarding execution of `lm()`, `biglm()`, `bigglm()`, `biglm.big.matrix()`, `SparseM.slm.fit()`, and `speedlm()` using the OPM data and models are discussed in the *Review of Available Solutions* appendix. Due to

obstacles and limitations, these solutions are not considered here. As previously stated, the example OPM data set contains 24,574,480 observations and two moderate dimension fixed effects (*bureau* and *occupation*, with 409 and 791 levels, respectively). With our understanding of matrix and operation density, it is of interest to measure algorithm performance using a somewhat large, higher dimension data set and an actual model. Before fitting model 1 we will use

$$y = \beta_0 + \beta_{sex} + \beta_{race} + \beta_{age} * age + \beta_{age2} * age^2 + \beta_{ed} * ed_{years} + \beta_{bur} + \beta_{occ} + \beta_{year} \quad (3)$$

which is identical to model 1 without the *sex*race* interaction term (this model appears in Bolton and de Figueiredo, 2016 - cite). Table 4 lists execution times and memory requirements to fit model 3 using `lfe()` and `feXTX()`.

Table 4. Execution Time and Memory Required to Fit Model 3 to OPM Data

Method	Execution Time (min)	Memory used (Gb)
`lfe()`	39.45	128*
`feXTX()`	9.6	15.3

* exceeds memory available on 64 Gb test server; execution accomplished on alternate dedicated server with 256 Gb of memory; "non-estimable function" warnings observed on alternate server

List table of parameter estimates in comparison to those from the Bolton Wage Disparity paper

Include notes on estimable function requirements for lfe()

Finally, execution times and memory requirements to fit model 1 to the example OPM data are listed in table 5. Although efficient in a sparse context, the `feXTX()` indexing algorithm for

interactions involving fixed effect columns is somewhat complex and adds inordinately to execution time[†††]. An alternative to specifying interactions in the function call to `feXTX()`, when low dimension fixed effects are involved (*sex* and *race* have 1 and 4 non-reference levels, respectively), is to construct indicator columns for each intersection of interacting levels and include these in the input data frame as continuous variables[‡‡‡]. Table 5 lists execution time to fit model 1 using `lfe()` (with interactions in model specification – the preferred method) and `feXTX()` (with interactions expanded as described).

Table 5. Execution Time and Memory Required to Fit Model 1 to OPM Data

Method	Execution Time (min)	Memory used (Gb)
`lfe()`	40.5	128*
`feXTX()`, interactions expanded	13.2	20

* exceeds memory available on 64 Gb test server; execution accomplished on alternate dedicated server with 256 Gb of memory; "non-estimable function" warnings observed on alternate server

Again, we observe a favorable difference in execution and memory efficiency when comparing `feXTX()` to `lfe()`, which itself has been observed to be reasonably efficient.

[†††] Ironically, density of fixed effect interactions is expected to be low, $d_1 * d_2$ for two interacting levels of distinct fixed effects, but the algorithm involves exhaustive, nested calls to `apply()` functions. At present, interaction elements of $X'X$ are processed sequentially; a future release will compute them in parallel.

[‡‡‡] Approximately two seconds of compute time is required to construct *sex* and *race* interaction vectors

Robust and Clustered Standard Errors

To compensate for heteroskedasticity of within-group errors, analysts often report parameter estimates with respect to robust or clustered standard errors (cite). `feXTX()` computes robust and clustered standard errors using the fundamental equation for $V(\hat{\beta})$ derived from the OLS normal equations:

$$V(\hat{\beta}) = (X'X)^{-1}X'UX(X'X)^{-1} \quad (4)$$

where U is the $n \times n$ matrix of within cluster residual covariances, and is estimated by setting all inter-cluster errors, $(\epsilon'\epsilon)_{\text{cluster}}$, to 0 (on the assumption of inter-cluster independence, cite). Note that U is $n \times n$ which, for the problems we consider, can be quite large, 24,574,480 x 24,574,480 in our example OPM data set. Further, it is involved in operations with another large matrix, X (24,574,480 x p). Standard matrix operations involving objects of this size are computationally impractical, in both time and memory requirements. However, indexing methods similar to those used in composing $X'X$ can be used to compute $V(\hat{\beta})$ very efficiently.

Briefly describe significant features and opportunities of robust/clustered error spare matrix indexing

Provide some details on the difference in robust and clustered methods of evaluating $V(B)$

Adapt program comments to an explanation of the algorithm (include in appendix)

Robust standard error algorithm comments from script:

```
# robust, uncorrelated, heteroskedastic (independent, non-identically-distributed) errors
# estimate robust variances (square of standard errors) as v(beta) = i(X'X)*X'uu'X*i(X'X)
# where i() is the inverse function and u is the vector of observation response to
# predicted value errors (y - yEstimate)
# this derives from the expression for parameter variances from the normal OLS equations
# uu' forms an n X n diagonal error variance-covariance matrix with all off-diagonal
```

```
# elements set to 0, which asserts the assumption of independent (uncorrelated) errors
# but, since the diagonal elements are expected to be non-constant, this method models
# parameter standard errors in a heteroskedastic (independent, but not identically
# distributed) error setting
# since X'uu'X is symmetric, construct upper triangle only then copy transpose to lower
# triangle
```

Clustered error algorithm comments from script:

```
# heteroskedastic, correlated within cluster, independent (uncorrelated) between cluster,
# non-identically-distributed (heteroskedastic) errors
# estimate clustered standard errors using v(beta) = i(X'X)*X'uu'X*i(X'X),
# where i() implies inverse and uu' is the var-cov matrix of observation to estimate
# errors (y-yest)
# this is the expression for var(beta) from ordinary least squares estimates of beta
# on the assumption that X is non-stochastic and uu' is an estimate of error covariance
# and, further, that covariance observed within clusters (groups) estimates that in the
# population while covariance between groups is 0, uu' is altered such that all off-
# diagonal inter-group elements, that is uu'(ij) where
# indices i and j belong to different groups, are set to 0
# uu' has the form:
#
#     uu'(g1)    0       0    0 0 0 ...   0
#        0    uu'(g2)    0    0 0 0 ...   0
#        0       0    uu'(g3) 0 0 0 ...   0
#                                 ...   0
#        0       0       0      0   ... uu'(gk)
#
# where uu'(gi) is the n(gi) X n(gi) sub-matrix of the error covariace matrix
# corresponding to group i
# note that with this construction of uu', X'uu'X is the sum (over all groups i) of
# (Xgi)'*uu'(gi)*Xgi,
# where Xgi and uu'(gi) are the rows of X and sub-matrix of uu' corresponding to group i
# [to see this, imagine multiplying X' by uu' and consider the effect of uu' elements of
# rows that do not correspond to a particular group - each resulting column corresponds
# to products from a single group -
# then multiplying that by X (each row belongs to a single group) gives a p X p sum of
# individual group products]
# note that (Xg)'*ug = [(ug)'*Xg]', making (Xg)'*uu'(g)*Xg symmetric
# also, (ug)'*Xg is a 1 X p vector, where p is the column dimension of X (the design
# matrix)
# for each ID, construct v = (ug)'*Xg then use it to compute v'v = (Xg)'*uu'(g)*Xg
```

Table 6 lists execution times to fit model 1 and compute robust or clustered standard errors using `lfe()`, `feXTX()`, and Stata (Stata/MP version 13.1). All processing was executed on a single, dedicated, 24 core Windows 7 server. The approximate 2 to 1 performance ratio of `feXTX()` to both `lfe()` and Stata indicates an advantage to computing heteroskedastic robust and clustered standard errors by solving the analytical OLS variance equation (eq 4) using efficient indexing methods. The largest (absolute value) pair-wise deviation in Stata and `feXTX()` computed standard errors was less than 10^{-5}, lending credence to accurate analytical equation evaluation. `lfe()` and `feXTX()` computed values were not compared due to the default `lfe()`behavior of

omitting the intercept and including fixed effect reference levels when estimable functions are not specified.

Table 6. Execution Time (in minutes) to Fit Model 1 and Compute Robust and Clustered Standard Errors

Method	Robust SE Time (min)	Clustered SE Time (min)	Memory (Gb)
`lfe()`*	-	51.9	128
`feXTX()`**	13.2	27.6	20/38
`Stata`***	51.5	55.5	<20

* bootstrap estimates computed; robust standard errors require use of ancillary bootstrap estimation functions which were not pursued; executed on alternate server; message received: "`chol.default() - matrix is either rank-deficient or indefinite`" `warning; "non-estimable function`"
** SEs are the result of evaluation of analytical standard error equation
*** Using Stata/MP Version 13.1

Compare clustered errors to those in Bolton wage disparity paper

Further Development

While developing `feXTX()` and associated algorithms, several opportunities for improvement of performance or utility were identified but, due to practical limitations of time and striving to hold a course of completing primary features, they are withheld for future development. Among them are:

- Improved sub-setting and transmission of fixed effect vectors to parallel cores – transmit only levels to be operated on by a given core
- Elimination of fixed effect level index vector export (however, passing large lists as parallel function parameters is also expensive)
- Parallelization of interaction vector-products

- Encode composed *X'X* matrix in SparseM format, since it is itself sparse
- Implement parallel algorithm for forward and back solving of system of equation using computed Cholesky decomposition

Conclusion

Conclusion