User Guide for `lpinfer`

Conroy Lau* Alexander Torgovitsky†

September 2, 2021

1 Introduction

The `lpinfer` package for `R` contains a collection of tools for estimation and inference on the solutions for linear programs. Our motivation comes from the econometrics literature on partial identification or "bounds" approaches. A string of recent research has shown that bounds in many interesting models can be constructed by solving two linear programs:

$$\min_{x \in \mathbb{R}^d_+} / \max_{x \in \mathbb{R}^d_+} A_{\text{tgt}} x \quad \text{s.t.} \quad A_{\text{shp}} x = \beta_{\text{shp}} \quad \text{and} \quad A_{\text{obs}} x = \beta_{\text{obs}}. \tag{1}$$

The notation in (1) means the following:

- $x \in \mathbb{R}^d_+$ constitutes the unknown model parameters. In many settings, d will be quite large. The `lpinfer` package assumes that x is constrained to be non-negative. As we discuss in Section 8, it is possible to convert any linear program to such a form by adding slack variables.
- A_{tgt} is a $1 \times d$ dimensional matrix (a row vector) that encodes the **target parameter**, $A_{\text{tgt}} x$. The target parameter is the one–dimensional summary of x that the researcher wants to infer.
- A_{shp} and β_{shp} encode **shape constraints** on the possible values that x can take. These are used to enforce logical or *a priori* assumptions on the model.
- A_{obs} and β_{obs} encode the **data-matching** (sometimes called "observational equivalence") constraints. These require x to be consistent with the observed data; typically β_{obs} would be some moments estimated from the data, and $A_{\text{obs}} x$ would be the moments implied by the model if the parameters are x.

*Kenneth C. Griffin Department of Economics, University of Chicago.
†Kenneth C. Griffin Department of Economics, University of Chicago. Research supported by National Science Foundation grant SES-1846832.

A concrete example of how (1) arises from an econometric model is given in the next section.

Although the shape and data-matching constraints enter (1) in the same way, their conceptual difference will be important in the procedures ahead. The shape constraints will be *required* to hold. Subject to that requirement, x will be chosen so that the data-matching constraints are met as well as possible.

Often, it will not be possible to meet the data-matching constraints exactly. This is no different here than in other statistical models. The estimation procedure described in Section 4 modifies (1) to account for this possibility. Applying this estimation procedure yields bound estimates for the target parameter, $[\hat{\beta}_{\text{lb}}, \hat{\beta}_{\text{ub}}]$. These estimates can be expected to be consistent for the "true" population bounds, $[\beta_{\text{lb}}, \beta_{\text{ub}}]$, that would arise if the model were correctly-specified and the population data were known; see Mogstad, Santos, and Torgovitsky (2018) for more detail.

In addition to estimating the population bounds, we might also want to conduct frequentist statistical inference on them, either by testing the null hypothesis $H_0 : \beta_{\text{tgt}} \in [\beta_{\text{lb}}, \beta_{\text{ub}}]$ for some pre-specified value β_{tgt}, or by constructing a confidence interval $[c_{\text{lb}}(\alpha), c_{\text{ub}}(\alpha)]$ that contains $[\beta_{\text{lb}}, \beta_{\text{ub}}]$ in repeated samples with probability $1 - \alpha$. This problem falls into the large and ever-expanding literature on inference in partially identified models; see the recent surveys by Canay and Shaikh (2017) or Molinari (2020) for extensive reviews.

For the `lpinfer` package, we focus on the subset of these procedures that scale well both in terms of d (the dimension of the parameters, x), and in terms of the number of rows in A_{obs} and A_{shp}. This limits our focus to procedures specifically focused on *projection inference*. We limit our focus even further to those procedures that can be implemented reliably in the sense that they involve only calculations for which certificates are available. This means that we do not consider methods that involve solving unstructured nonlinear, non-convex optimization problems. We would, however, be amenable to considering methods that involve non-convex optimization problems for which certificates of optimality are available, such as linear integer programs or bilinear programs.

Even with these restrictions, there are still many procedures. Currently, we have implemented four in `lpinfer`; see Section 5 for details. We have a list of other applicable procedures that we intend to implement in the future, and we plan to continually add new procedures as they are proposed.

2 Example: The Mixed Logit Model

In this section, we give a simple example of how (1) arises in an empirically-relevant econometric model. For more examples, see Fang et al. (2020, sec. 2) and the references cited therein.

Bajari, Fox, and Ryan (2007) and Fox et al. (2011) consider a class of discrete choice random coefficient models. The binary choice version of this model has the structure

$$Y_i = \mathbb{1}[V_i' W_i \geq U_i], \tag{2}$$

where i denotes an individual, Y_i is an observed binary random variable, W_i is an observed random vector, and both V_i and U_i are unobserved random variables that are assumed to be independent of W_i. The unobservable U_i is assumed to have a standard logistic distribution and be independent of V_i. Model (2) is like the latent variable representation of the well-known logit model for binary outcomes, with the important difference that V_i are unobserved random variables, not degenerate parameters.

Fox et al. (2011) consider approximating the distribution of V_i with a discrete distribution. This discrete distribution is assumed to have known support points $(v_1, \ldots, v_d)$. The unknown parameters of the model are the masses that the distribution places on each support points, which are the $x = (x_1, \ldots, x_d)$ in (1). These masses must be non-negative, so that $x \in \mathbb{R}^d_+$. They also must sum to unity, so that

$$\sum_{j=1}^{d} x_j \equiv A_{\text{shp}} x = \beta_{\text{shp}} \equiv 1. \tag{3}$$

These are the shape constraints of this simple model. More complicated models might have many more shape constraints.

If (2) were indeed the way that Y_i were generated, then we would have

$$\mathbb{P}[Y_i = 1 | W_i = w] = \sum_{j=1}^{d} x_j \ell(v_j' w), \tag{4}$$

where $\ell(v_j' w)$ are the logit probabilities conditional on $V_i = v_j$, that is,

$$\ell(z) \equiv \frac{1}{1 + \exp(-z)}.$$

The left-hand side of (4) is a population moment, which is not observed but can be estimated. For example, if W_i is discretely distributed, a simple estimator is

$$\beta_{\text{obs}}(w) \equiv \frac{\sum_{i:W_i = w} Y_i}{\sum_{i:W_i = w} 1}.$$

If we choose a set of supported points from the distribution of W_i, say $w_1, \ldots, w_p$, then we obtain the data-matching constraints

$$\beta_{\text{obs}} \equiv \begin{bmatrix} \beta_{\text{obs}}(w_1) \\ \vdots \\ \beta_{\text{obs}}(w_p) \end{bmatrix} = \begin{bmatrix} \ell(v_1' w_1) & \cdots & \ell(v_d' w_1) \\ \vdots & & \vdots \\ \ell(v_1' w_p) & \cdots & \ell(v_d' w_p) \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \equiv A_{\text{obs}} x. \tag{5}$$

If p is smaller than d, then we would generally expect there to be multiple solutions to (5), even under the shape constraint (3). This is the source of partial identification in this model, and the reason that the solutions to the minimization and maximization problems defined in (1) might be different.

There are many target parameters that might be of interest in this model. One example is the probability of choosing Y_i at some value $w^\star$ not in the support of W_i:

$$\mathbb{P}[Y_i = 1 | W_i = w^\star] = \sum_{j=1}^{d} x_j \ell(v_j' w^\star) = \begin{bmatrix} \ell(v_1' w^\star) & \cdots & \ell(v_d' w^\star) \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \equiv A_{\text{tgt}} x.$$

For a more elaborate example, consider the elasticity of choosing Y_i with respect to w_k at $W_i = w^\star$, and conditional on $V_i = v_j$:

$$\epsilon_j(w^\star) = \left(\frac{\partial}{\partial w_k} \ell(v_j' w^\star) \right) \frac{w_k^\star}{\ell(v_j' w^\star)} = v_{jk} w_k^\star (1 - \ell(v_j' w^\star)).$$

We might want to conduct inference on the average of this elasticity over V_i:

$$\sum_{j=1}^{d} x_j \epsilon_j(w^\star) \equiv A_{\text{tgt}} x,$$

or perhaps on the probability that it lies below some pre-specified value e:

$$\sum_{j=1}^{d} x_j \mathbb{1}[\epsilon_j(w^\star) \leq e] \equiv A_{\text{tgt}} x. \tag{6}$$

There are many other choices of target parameter that may be interesting, depending on the application.

The `lpinfer` package includes example code for the mixed logit. This example code includes a function that generates the matrix A_{obs} and the vector β_{obs} in (5), a function that draws a sample of data, and a function that generates the A_{tgt} vector that corresponds to the distribution of elasticity in (6). The following snippet illustrates these functions. They will be used ahead to demonstrate the core features of `lpinfer`.

```r
source("/Users/conroylau/Library/R/3.6/library/lpinfer/example/dgp_mixedlogit.R")
dgp <- mixedlogit_dgp(dimw = 4, dimv = 16)
dgp$wdist
```

```
##        pr w1 w2
## [1,] 0.25  1  0
## [2,] 0.25  1  1
## [3,] 0.25  1  2
## [4,] 0.25  1  3
```

```r
head(dgp$vdist)
```

```
##          pr        v1 v2
## [1,] 0.0625 0.0000000 -3
## [2,] 0.0625 0.1666667 -3
## [3,] 0.0625 0.3333333 -3
## [4,] 0.0625 0.5000000 -3
## [5,] 0.0625 0.0000000 -2
## [6,] 0.0625 0.1666667 -2
```

```r
dim(dgp$vdist)
```

```
## [1] 16  3
```

```r
mixedlogit_Aobs(dgp)
```

```
##              [,1]         [,2]         [,3]        [,4]        [,5]       [,6]
## [1,] 0.5000000000 0.5415704832 0.5825702065 0.622459331 0.500000000 0.54157048
## [2,] 0.0474258732 0.0555492602 0.0649691691 0.075858180 0.119202922 0.13784166
## [3,] 0.0024726232 0.0029197498 0.0034474513 0.004070138 0.017986210 0.02117911
## [4,] 0.0001233946 0.0001457702 0.0001722026 0.000203427 0.002472623 0.00291975
##             [,7]        [,8]       [,9]      [,10]      [,11]      [,12] [,13]
## [1,] 0.582570206 0.622459331 0.50000000 0.54157048 0.58257021 0.62245933   0.5
## [2,] 0.158869105 0.182425524 0.26894142 0.30294072 0.33924363 0.37754067   0.5
## [3,] 0.024924427 0.029312231 0.11920292 0.13784166 0.15886910 0.18242552   0.5
## [4,] 0.003447451 0.004070138 0.04742587 0.05554926 0.06496917 0.07585818   0.5
##          [,14]     [,15]     [,16]
```

```
## [1,] 0.5415705 0.5825702 0.6224593
## [2,] 0.5415705 0.5825702 0.6224593
## [3,] 0.5415705 0.5825702 0.6224593
## [4,] 0.5415705 0.5825702 0.6224593
```

```
set.seed(67)
df <- mixedlogit_draw(dgp, n = 4000)
mixedlogit_betaobs(df, dgp)
```

```
## [1] 0.5523256 0.2874372 0.1747104 0.1696905
```

```
Atgt <- mixedlogit_Atgt_dfelast(dgp, w2eval = 1, eeval = -2)
Atgt %*% dgp$vdist[,1] # The actual (unknown) value of the target parameter
```

```
##      [,1]
## [1,] 0.25
```

```
# Change the target parameter slightly
Atgt <- mixedlogit_Atgt_dfelast(dgp, w2eval = 1, eeval = -1)
Atgt %*% dgp$vdist[,1]
```

```
##      [,1]
## [1,]  0.5
```

3 The `lpmodel` Class

The `lpinfer` package is built around an S3 class called `lpmodel`. This class simply consists of the five objects necessary to specify (1): $A_{\text{tgt}}, A_{\text{shp}}, \beta_{\text{shp}}, A_{\text{obs}}, \beta_{\text{obs}}$. An instance of the class is created with the following syntax:

```
library("lpinfer")

lpm <- lpmodel(A.obs = mixedlogit_Aobs(dgp),
               beta.obs = function(d) mixedlogit_betaobs(d, dgp),
               A.shp = rep(1, nrow(dgp$vdist)),
               beta.shp = 1,
               A.tgt = mixedlogit_Atgt_dfelast(dgp, w2eval = 1, eeval = -1))
```

All of the statistical inference procedures utilize either the bootstrap or subsampling. The procedures differ on which components of an `lpmodel` are allowed to change across resampling, that is, which components are treated as "stochastic". Table 1 catalogues these differences. In implementing the procedures, we have tried to allow for as much flexibility as possible. If the paper does not explicitly require a given object to be deterministic, then we allow it to be stochastic. In some cases, there may be additional restrictions on the form of some of the components of an `lpmodel`.

Stochastic components can be specified in two ways. The first is to pass a function that accepts a `data.frame` as an argument and returns the realization of the object. This is what is being done for `beta.obs` in the syntax above, but it can be done for any of the other components as well. For `beta.obs` only, the function can also return two objects in a list, with the first object being a vector realization of β_{obs} and the second object being an estimate of the asymptotic variance matrix of β_{obs}. This is useful in some procedures that make use of the variance–covariance matrix of β_{obs}; if one is not passed, then `lpinfer` will estimate it via nonparametric bootstrap.

Alternatively, the user can create a list of objects ahead of time and pass this in lieu of a function. For example,

```
R <- 100 # number of bootstrap replications
beta.obs.list <- vector(mode = "list", length = R)
for (i in 1:R) {
    d <- mixedlogit_draw(dgp, n = 4000)
    beta.obs.list[[i]] <- mixedlogit_betaobs(d, dgp)
}

lpm2 <- lpmodel(A.obs = mixedlogit_Aobs(dgp),
                beta.obs = beta.obs.list,
                A.shp = rep(1, nrow(dgp$vdist)),
                beta.shp = 1,
                A.tgt = mixedlogit_Atgt_dfelast(dgp, w2eval = 1, eeval = -1))
```

When using this list syntax, the number of bootstrap replications is inferred from the length of `boot.obs.list`. This makes it more error-prone for procedures where multiple components can be stochastic, since each one would need to have a list of the same length. We include the list syntax to accommodate models in which it might be costly to compute `beta.obs`; using the list approach the user can just calculate these once at the outset before running various testing procedures. When using the function syntax, the number of bootstrap replications can be specified in the testing procedure itself.

Table 1: Requirements for 'lpmodel' by procedure

Procedure	Can it be a function?					Other notes
	A_{tgt}	A_{shp}	β_{shp}	A_{obs}	β_{obs}	
`estbounds`	✓	✓	✓	✓	✓	–
`fsst`					✓	–
`dkqs`					✓	Must have $A_{\text{shp}} = 1_{1\times d}$ and $\beta_{\text{shp}} = 1$
`subsample`	✓	✓	✓	✓	✓	–
`chorussell`	✓	✓	✓	✓	✓	–

4 Estimation

Estimation is conducted using the procedure developed in Mogstad, Santos, and Torgovitsky (2018). The command is called `estbounds`:

```
estbounds(data = df, lpmodel = lpm)
```

```
## Estimated bounds: [0.57875, 0.58024]
```

For a full description of the procedure and its theoretical justification, we refer the reader to Mogstad, Santos, and Torgovitsky (2018). However, the basic idea is intuitive. First we search over all $x \in \mathbb{R}^d_+$ that satisfy $A_{\text{shp}}x = \beta_{\text{shp}}$ to try to minimize the distance between $A_{\text{obs}}x$ and β_{obs}. By default, this distance is measured using the Euclidean (ℓ_2) norm, although in computationally difficult problems one might want to switch to the ℓ_1 norm using the `norm` option:

```
r <- estbounds(data = df, lpmodel = lpm, norm = 1)
print(r)
```

```
## Estimated bounds: [0.55677, 0.55677]
```

The outcome of this problem is saved in the field `mincriterion`:

```
print(r$mincriterion)
```

```
## [1] 0.01633628
```

Next, (1) is solved with the hard constraint $A_{\text{obs}}x = \beta_{\text{obs}}$ replaced by the soft-constraint that x is such that the norm of $A_{\text{obs}}x - \beta_{\text{obs}}$ is within some tolerance of `mincriterion`. This tolerance is determined by a non-negative regularization parameter `kappa`. By default this parameter is set to 0, but in some problems it may be theoretically important to set it to a small number.

```
estbounds(data = df, lpmodel = lpm, kappa = 1e-3)
```

```
## Estimated bounds: [0.57583, 0.58359]
```

5 Statistical Inference: Testing

In this section, we briefly outline the syntax and options for the testing procedures that are currently available in `lpinfer`. For details on the theory, please consult the original paper. All of the tests are for the null hypothesis discussed in Section 1.

An important comment—applicable to all of the procedures—is that the number of bootstrap or subsampling replications is, by default, set to 100. This should be increased to a conventional size for production runs. The optional argument for all procedures is called `R`. Examples of changing it are given ahead.

5.1 Fang et al. (2020, "FSST")

The basic syntax for testing $\beta_{\text{tgt}} = .2 \in [\beta_{\text{lb}}, \beta_{\text{ub}}]$ is this:

```
fsst(data = df, lpmodel = lpm, beta.tgt = .2)
```

```
## p-value (by data-driven 'lambda'): 0.04
```

```
fsst(data = df, lpmodel = lpm, beta.tgt = .2, R = 500) # more bootstraps
```

```
## p-value (by data-driven 'lambda'): 0.038
```

The tuning parameter in the procedure is called `lambda`, which is a number between 0 and 1. By default, a data-driven choice of `lambda` is used, which corresponds to `lambda = NA`. The test can be run for multiple values of `lambda` simultaneously by passing a vector:

```
fsst(data = df, lpmodel = lpm, beta.tgt = .2, lambda = c(.1, .2, .5, NA, .9))
```

```
## p-values:
##     lambda p-value
##        0.1    0.15
##        0.2    0.07
##        0.5    0.05
##        0.9    0.04
```

```
##  0.84564 *    0.04
##
## * refers to the data-driven 'lambda' parameter.
```

Another choice that one might want to change in some settings is the weighting matrix used in an internal step of `fsst`. By default, it is set to the inverse of the diagonal of the asymptotic variance matrix of β_{obs}, but one can also use the identity matrix, or the full asymptotic variance matrix.

```
set.seed(25)
fsst(data = df, lpmodel = lpm, beta.tgt = .2, weight.matrix = "diag")
```

```
## p-value (by data-driven 'lambda'): 0.01
```

```
set.seed(25)
fsst(data = df, lpmodel = lpm, beta.tgt = .2, weight.matrix = "identity")
```

```
## p-value (by data-driven 'lambda'): 0.01
```

```
set.seed(25)
fsst(data = df, lpmodel = lpm, beta.tgt = .2, weight.matrix = "avar")
```

```
## p-value (by data-driven 'lambda'): 0.01
```

The function used to compute the matrix square root can be specified through the `sqrtm.method` option. This function can only have one argument and return a matrix. An example of using `pracma::sqrtm` with `kmax = 100` is as follows.

```
set.seed(25)
fsst(data = df, lpmodel = lpm, beta.tgt = .2,
     sqrtm.method = function(m) pracma::sqrtm(m, kmax = 100)$B)
```

```
## p-value (by data-driven 'lambda'): 0.01
```

More detailed information on test statistics and critical values can be obtained by calling `summary` on the output of `fsst`.

5.2 Deb et al. (2018, "DKQS")

The basic syntax for testing $\beta_{\text{tgt}} = .2 \in [\beta_{\text{lb}}, \beta_{\text{ub}}]$ is similar to `fsst`:

```
dkqs(data = df, lpmodel = lpm, beta.tgt = .2)
```

```
## p-value: 0.24
## tau: 1
```

```
dkqs(data = df, lpmodel = lpm, beta.tgt = .2, R = 500) # more bootstraps
```

```
## p-value: 0.338
## tau: 1
```

The tuning parameter in DKQS is called `tau`, which is a number between 0 and 1. Kamat (2018) observed that a key optimization problem solved in the DKQS procedure can be infeasible unless `tau` is smaller than some maximum value. This value can be computed, and the default behavior of `dkqs` is to compute and use this value. Alternatively, the user can also pass their own values of `tau` as a list:

```
set.seed(34)
dkqs(data = df, lpmodel = lpm, beta.tgt = .2, tau = c(0, .5, .75))
```

```
## p-values:
##   tau p-value
##  0.00    0.03
##  0.50    0.07
##  0.75    0.17
```

One thing to note about the DKQS procedure is that it is only designed for a specific choice of A_{shp} and β_{shp}, as shown in Table 1. For this reason, `dkqs` ignores these fields of the `lpmodel`:

```
lpm2 <- lpm
lpm2$A.shp <- NULL
set.seed(34)
dkqs(data = df, lpmodel = lpm2, beta.tgt = .2, tau = .5) # still works
```

```
## p-value: 0.07
## tau: 0.5
```

5.3 Profiled Subsampling (Romano and Shaikh 2008)

Profiled subsampling is based on a criterion function that measures the distance between $A_{\text{obs}}x$ and β_{obs} while constraining $x \in \mathbb{R}^d_+$, $A_{\text{shp}}x = \beta_{\text{shp}}$ and $A_{\text{tgt}}x = \beta_{\text{tgt}}$ for a hypothetical β_{tgt}. The syntax is this:

```
subsample(data = df, lpmodel = lpm, beta.tgt = .2)
```

```
## p-value: 0.06
```

```
subsample(data = df, lpmodel = lpm, beta.tgt = .2, R = 500) # more bootstraps
```

```
## p-value: 0.092
```

As in `estbounds`, the user can change the norm used to measure distance to the ℓ_1 norm, although for `subsample` there are no computational benefits from doing so. The main tuning parameter is `phi`, which is a number between 0 and 1 that determines the subsample size. If n is the original number of observations, then the subsample size is taken to be n^ϕ. The default value is `phi = 2/3`. The user can also change `replace` from its default of `FALSE` to `TRUE` to draw with replacement. Note that `replace = TRUE` and `phi = 1` is a standard nonparametric bootstrap.

```
subsample(data = df, lpmodel = lpm, beta.tgt = .2, phi = 6/9)
```

```
## p-value: 0.03
```

```
subsample(data = df, lpmodel = lpm, beta.tgt = .2, phi = 1,
          replace = TRUE) # bootstrap
```

```
## p-value: 0.63
```

5.4 Cho and Russell (2019, "CR")

The basic syntax for testing $\beta_{\text{tgt}} = .2 \in [\beta_{\text{lb}}, \beta_{\text{ub}}]$ is similar to the previous methods:

```
chorussell(data = df, lpmodel = lpm, beta.tgt = .2)
```

```
## p-value: 6e-05
```

```
chorussell(data = df, lpmodel = lpm, beta.tgt = .2, R = 500) # more bootstraps
```

```
## p-value: 6e-05
```

The CR procedure directly bootstraps estimates of the bounds. To handle situations when $A_{\text{obs}}x = \beta_{\text{obs}}$ cannot be met perfectly in sample, we use `estbounds` to construct these estimates. Thus, the `kappa` and `norm` tuning parameters from `estbounds` are also tuning parameters for `chorussell`.

```
chorussell(data = df, lpmodel = lpm, beta.tgt = .2, kappa = 1e-3, norm = 1)
```

```
## p-value: 6e-05
```

The CR procedure also differs from the others because it is designed to directly construct confidence intervals, rather than test specific points. The examples above actually do the opposite procedure of inverting confidence intervals to obtain p–values for a specific point. Alternatively, the user can directly obtain a level $1 - \alpha$ confidence interval by just not passing `beta.tgt`, as in

```
chorussell(data = df, lpmodel = lpm, alpha = .1)
```

```
## 90% confidence interval: [0.47031, 0.74173]
```

6 Statistical Inference: Confidence Intervals

All of the procedures described in the previous section (other than CR) are designed for testing a given null hypothesis. The `lpinfer` package also provides a bisection routine that constructs a level $1 - \alpha$ confidence interval by inverting level α tests. For example,

```
set.seed(5)
invertci(f = fsst, farg = list(data = df, lpmodel = lpm), progress = FALSE)
```

```
## 95% confidence interval: [0.19711, 0.7753]
```

The first argument gives the function that controls the test, while the second argument passes any arguments needed to run the test as a list. The third argument `progress` is option and used here for brevity; when `progress = TRUE` the sequential progress of the bisection procedure is printed. The default is to construct a level 95% ($= 1 - .05$) confidence interval, but this can be changed with the `alpha` option:

```
set.seed(5)
invertci(f = fsst, farg = list(data = df, lpmodel = lpm), progress = FALSE,
         alpha = .10)
```

```
## 90% confidence interval: [0.23538, 0.76181]
```

The bisection procedure attempts to find appropriate initial brackets, but the user can override this to help speed up computation. Although it won't be clear here, since `progress = FALSE`, the following requires many fewer iterations than the previous command.

```
set.seed(5)
invertci(f = fsst, farg = list(data = df, lpmodel = lpm), progress = FALSE,
         init.lb = c(.1, .5), init.ub = c(.75, .8))
```

```
## 95% confidence interval: [0.19712, 0.77534]
```

The `invertci` also returns previous p–values it has computed, which can then be used as an input to future calls of `invertci` in order to speed up computation. For example, the second `invertci` here runs much more quickly with the optional `pvals` argument than it would without:

```
set.seed(5)
ci <- invertci(f = fsst, farg = list(data = df, lpmodel = lpm), progress = FALSE)
set.seed(5)
invertci(f = fsst, farg = list(data = df, lpmodel = lpm), progress = FALSE,
         alpha = .10, pvals = ci$pvals) # this runs more quickly
```

```
## 90% confidence interval: [0.23538, 0.76181]
```

7 Parallelization

All of the procedures discussed in Sections 5 and 6 can be parallelized in `lpinfer` using the `future` package.

```
library("future")

# 3 workers
plan(multisession, workers = 3)
t_start <- Sys.time()
```

```
set.seed(1)
fsst(df, lpm, .2, R = 5000)
```

```
## p-value (by data-driven 'lambda'): 0.0362
```

```
print(sprintf("That took %s seconds.",
              round(difftime(Sys.time(), t_start, units = "secs"),
                    digits = 3)))
```

```
## [1] "That took 41.953 seconds."
```

```
# 1 worker
plan(multisession, workers = 1)
t_start <- Sys.time()
set.seed(1)
fsst(df, lpm, .2, R = 5000)
```

```
## p-value (by data-driven 'lambda'): 0.0362
```

```
print(sprintf("That took %s seconds.",
              round(difftime(Sys.time(), t_start, units = "secs"),
                    digits = 3)))
```

```
## [1] "That took 102.903 seconds."
```

8 Converting Linear Programs to "Standard Form"

All linear programs can be converted to a form that looks like (1) by including appropriate slack variables. We call this "standard form" with the scare quotes because it's not the standard "standard form" seen in most textbooks on linear programming. At the same time, it's also not non-standard to see (1) called "standard form"; one reference we know of is Bertsimas and Tsitsiklis (1997, page. 4).

Semantics aside, we have included a function that helps one translate expressive linear programs in extensive form into the format (1) required by `lpinfer`. The idea is to use another S3 class in `lpinfer` called `lpmodel.natural`, which allows one to explicitly give bounds on variables and declare different "senses" of equality and inequality constraints. This functionality is still experimental, but here's an example of how it works:

```
### Step 1: Create an object in the 'lpmodel.natural' class
# Obs
Aobs0 <- matrix(c(1, 2), nrow = 1)
bobs0 <- c(10)

# Shp
Ashp0 <- matrix(c(3, 4, 5, 6), nrow = 2, byrow = TRUE)
bshp0 <- matrix(c(15, 100))
sshp0 <- matrix(c(">=", "<="))

# Tgt
Atgt0 <- matrix(c(1, 1), nrow = 1)

# Upper bounds
xub0 <- c(200, 200)

# Lower bounds
xlb0 <- c(0.1, 0.1)

# Formulate the 'lpmodel.natural' object
lpmn0 <- lpmodel.natural(A.obs = Aobs0,
                         A.shp = Ashp0,
                         A.tgt = Atgt0,
                         beta.obs = bobs0,
                         beta.shp = bshp0,
                         sense.shp = sshp0,
                         x.ub = xub0,
                         x.lb = xlb0)

### Step 2: Apply the 'standard.lpmodel' function
lpm1 <- standard.lpmodel(lpmn0)
```

The new object, `lpm1` has updated `A.shp` and `beta.shp` matrices that incorporate the appropriate slack variables to fit into form (1).

```
print(lpm1$A.shp)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    5    6    1    0    0    0    0    0
## [2,]    1    0    0    1    0    0    0    0
## [3,]    0    1    0    0    1    0    0    0
## [4,]    3    4    0    0    0   -1    0    0
## [5,]    1    0    0    0    0    0   -1    0
## [6,]    0    1    0    0    0    0    0   -1
```

```
print(lpm1$beta.shp)
```

```
##       [,1]
## [1,] 100.0
## [2,] 200.0
## [3,] 200.0
## [4,]  15.0
## [5,]   0.1
## [6,]   0.1
```

For example, the first row corresponds to the `<=` components in the original `Ashp0` and `bshp0` specification, while the second and third rows correspond to the upper bounds on `x`. The `lpm1` instance can now be used as an `lpmodel` for all of the routines in `lpinfer`.

9 Further Examples

Further syntax examples are contained in the installation directory for `lpinfer` under the `example` subdirectory.

References

Bajari, Patrick, Jeremy T. Fox, and Stephen P. Ryan. 2007. "Linear Regression Estimation of Discrete Choice Models with Nonparametric Distributions of Random Coefficients." *American Economic Review* 97 (2): 459–63. doi:10.1257/aer.97.2.459.

Bertsimas, Dimitris, and John N Tsitsiklis. 1997. *Introduction to Linear Optimization*. Vol. 6. Athena Scientific Belmont, MA.

Canay, Ivan A., and Azeem M. Shaikh. 2017. "Practical and Theoretical Advances in Inference for Partially Identified Models." In *Advances in Economics and Econometrics*, edited by Bo Honore, Ariel Pakes, Monika Piazzesi, and Larry Samuelson, 271–306. Cambridge University Press. doi:10.1017/9781108227223.009.

Cho, JoonHwan, and Thomas M. Russell. 2019. "Simple Inference on Functionals of Set-Identified Parameters Defined by Linear Moments." *arXiv:1810.03180 [Econ]*, September. http://arxiv.org/abs/1810.03180.

Deb, Rahul, Yuichi Kitamura, John K. -H. Quah, and Jorg Stoye. 2018. "Revealed Price Preference: Theory and Empirical Analysis." *Working Paper*, January. http://arxiv.org/abs/1801.02702v1.

Fang, Zheng, Andres Santos, Azeem M. Shaikh, and Alexander Torgovitsky. 2020. "Inferece for Large-Scale Linear Systems with Known Coefficients." *Working Paper*.

Fox, Jeremy T., Kyoo il Kim, Stephen P. Ryan, and Patrick Bajari. 2011. "A Simple Estimator for the Distribution of Random Coefficients." *Quantitative Economics* 2 (3). The Econometric Society: 381–418. doi:10.3982/qe49.

Kamat, Vishal. 2018. "Identification with Latent Choice Sets: The Case of the Head Start Impact Study." *Working Paper*, December. http://arxiv.org/abs/1711.02048v1.

Mogstad, Magne, Andres Santos, and Alexander Torgovitsky. 2018. "Using Instrumental Variables for Inference About Policy Relevant Treatment Parameters." *Econometrica* 86 (5). The Econometric Society: 1589–1619. doi:10.3982/ecta15463.

Molinari, Francesca. 2020. "Microeconometrics with Partial Identification." *arXiv:2004.11751 [Econ]*, April. http://arxiv.org/abs/2004.11751.

Romano, Joseph P., and Azeem M. Shaikh. 2008. "Inference for Identifiable Parameters in Partially Identified Econometric Models." *Journal of Statistical Planning and Inference* 138 (9): 2786–2807.