# PyHDFE

*Release 0.2.0*

**Jeff Gortmaker and Anya Tarascina**

**Feb 07, 2024**

# TABLE OF CONTENTS


I User Documentation 1

1 Introduction 3
1.1 Installation . . . 3
1.2 Bugs and Requests . . . 3

2 API Documentation 5
2.1 pyhdfe.create . . . 5
2.2 pyhdfe.Algorithm . . . 8
2.3 pyhdfe.Algorithm.residualize . . . 9

3 Tutorial 11
3.1 scikit-learn . . . 12
3.2 statsmodels . . . 15

4 References 19
4.1 Papers . . . 19
4.2 Software . . . 20

5 Legal 23

II Developer Documentation 25

6 Contributing 27

7 Testing 29
7.1 Testing Requirements . . . 29
7.2 Running Tests . . . 29
7.3 Test Organization . . . 29

8 Version Notes 31
8.1 0.2 . . . 31
8.2 0.1 . . . 31

III Indices 33

Index 35

# Part I

# User Documentation

CHAPTER

ONE

# INTRODUCTION

**Note:** This package is in beta. In future versions, the API may change substantially. Please use the GitHub issue tracker to report bugs or to request features.

PyHDFE is a Python 3 implementation of algorithms for absorbing high dimensional fixed effects. This package was created by Jeff Gortmaker in collaboration with Anya Tarascina.

What PyHDFE won't do is provide a convenient interface for running regressions. Instead, the package is meant to be incorporated into statistical projects that would benefit from performant fixed effect absorption. Another goal is facilitating fair comparison of algorithms that have been previously implemented in various languages with different convergence criteria.

Development of the package has been guided by code made publicly available by many researchers and practitioners. For a full list of papers and software cited in this documentation, refer to the references section of the documentation.

## 1.1 Installation

The PyHDFE package has been tested on Python versions 3.6 through 3.9. The SciPy instructions for installing related packages is a good guide for how to install a scientific Python environment. A good choice is the Anaconda Distribution, since, along with many other packages that are useful for scientific computing, it comes packaged with PyHDFE's only required dependencies: NumPy and SciPy.

You can install the current release of PyHDFE with pip:

```
pip install pyhdfe
```

You can upgrade to a newer release with the `--upgrade` flag:

```
pip install --upgrade pyhdfe
```

If you lack permissions, you can install PyHDFE in your user directory with the `--user` flag:

```
pip install --user pyhdfe
```

Alternatively, you can download a wheel or source archive from PyPI. You can find the latest development code on GitHub and the latest development documentation here.

## 1.2 Bugs and Requests

Please use the GitHub issue tracker to submit bugs or to request features.

CHAPTER

TWO

# API DOCUMENTATION

Algorithms for absorbing fixed effects should be created with the following function.

| | |
|---|---|
| *create*(ids[, cluster_ids, drop_singletons, ...]) | Initialize an algorithm for absorbing fixed effects. |

## 2.1 pyhdfe.create

pyhdfe.**create**(*ids*, *cluster_ids=None*, *drop_singletons=True*, *compute_degrees=True*, *degrees_method=None*, *residualize_method=None*, *options=None*)

Initialize an algorithm for absorbing fixed effects.

By default, simple de-meaning is used for a single fixed effect, and non-accelerated de-meaning is used for more than one dimension. This is the most conservative and simplest algorithm for fixed effect absorption. If it is taking a long time, consider switching to a faster `residualize_method` and using different `options`.

When an algorithm is initialized, by default, singletons are dropped and degrees of freedom are computed. If either behavior isn't needed, or if degrees of freedom computation is taking a long time, consider using a more conservative `degrees_method` or disabling these behaviors with `drop_singletons` and `compute_degrees`.

> **Warning:** This function assumes that all of your data have already been cleaned. For example, it will not drop observations with null values.

**Parameters**

- **ids** (*array-like*) – Two-dimensional array of fixed effect identifiers. Columns are fixed effect dimensions and rows are observations. Identifiers can be integers, strings, or other hashable data types. Columns after the first should have more than one unique value.
- **cluster_ids** (*array-like, optional*) – Two-dimensional array of cluster group identifiers, which if specified will be used when computing degrees of freedom. If a fixed effect (i.e., a column in `ids`) is nested within a cluster (i.e., a column of this matrix), it will not contribute towards degrees of freedom used by the fixed effects. For more information, see *Correia (2015)*.
- **drop_singletons** (*bool, optional*) – Whether to drop singleton groups or observations in `ids` when initializing the algorithm. Singletons groups are fixed effect groups with only one observation. By default, singletons are dropped. When dropped, the number of singleton groups is equal to the number of rows in `ids` minus *`Algorithm.observations`*. For more information about singletons and why they are typically dropped, see *Correia (2015)*.

- **compute_degrees** (*bool, optional*) – Whether to compute the number of degrees of freedom used by the fixed effects. By default, degrees of freedom are computed.
- **degrees_method** (*str, optional*) – How to compute or approximate the number of degrees of freedom used by the fixed effects that aren't nested within any `cluster_ids`. The following methods are supported:
  - `'none'` (default for one dimension) - Assume there are no redundant fixed effects. This method is exact for one dimension (i.e., for one column in `ids`). It provides the most conservative upper bound for multiple dimensions but requires no additional computation.

    For one dimension this method simply counts the number of fixed effect levels (i.e., the number of distinct values in `ids`). Each dimension after the first contributes its number of levels minus one.
  - `'pairwise'` (default for multiple dimensions) - Apply the algorithm of *Abowd, Creecy, and Kramarz (2002)* to each pair of fixed effect dimensions. This method is exact for two dimensions. It provides a smaller upper bound for more than two dimensions but can be computationally expensive.

    For one dimension this method is the same as `'none'`. However, the second dimension contributes its number of levels minus the number of connected components in the bipartite graph formed by the two dimensions. Each dimension after the second contributes its number of levels minus the maximum number of connected components in the bipartite graphs that it forms with prior dimensions. This is the method used by *reghdfe*.
  - `'exact'` - Apply `numpy.linalg.matrix_rank()` to dummy variables constructed from `ids`. This method is exact for any number of dimensions but is typically computationally infeasible. It is meant to be a benchmark.
- **residualize_method** (*str, optional*) – Type of algorithm to initialize. The following methods are supported:
  - `'within'` (default for one dimension) - Within transform. Matrix columns are demeaned within each fixed effect group (i.e., each unique value in `ids`). This algorithm only works for a single fixed effect dimension (i.e., one column in `ids`).
  - `'map'` (default for multiple dimensions) - Method of alternating projections applied to fixed effect absorption by *Guimarães and Portugal (2010)*, *Gaure (2013a)*, *Gaure (2013b)*, and *Correia (2017)*, among others. Matrix columns are iteratively de-meaned until convergence. This method works for any number of fixed effect dimensions but will be slower than `'within'` for one dimension. Variations on this method are used by *lfe* and *reghdfe*.
  - `'lsmr'` - LSMR method of *Fong and Saunders (2011)*. This implementation is taken from `scipy.sparse.linalg.lsmr()` and modified for simultaneous iteration over multiple matrix columns and custom convergence criteria. Matrix columns are iterated on until convergence. This method works for any number of fixed effect dimensions but will be slower than `'within'` for one dimension. This is the method used by *FixedEffectModels.jl*.
  - `'sw'` - Method of *Somaini and Wolak (2016)*. This non-iterative method only works for two dimensions (i.e., two columns in `ids`). To minimize memory usage, the first dimension of fixed effects should have fewer levels than the second dimension (i.e., the first column in `ids` should have fewer unique values than the second column). This is the method used by *res2fe*.
  - `'dummy'` - Matrix columns are replaced by residuals from regressions on dummy variables constructed from `ids`. This method works for any number of dimensions but is typically computationally infeasible. It is meant to be a benchmark.

- **options** (*dict, optional*) – Configuration options for the chosen `method`. The `'within'`, `'sw'`, and `'dummy'` methods do not support any configuration options. The following options are supported by both `'map'` and `'lsmr'`:
  - **iteration_limit** : (*int, optional*) - Maximum number of iterations, after which an exception will be raised if the algorithm has not converged. By default, the maximum number of iterations is `1000000`.
  - **tol** : (*float, optional*) - Common convergence criteria based on the differences between two iterations' residualized matrices. By default, algorithms will converge when the maximum absolute value of these differences is less than `1e-8`. Convergence based on this criteria can be disabled by setting this value to `0`.
  - **converged** : (*callable or None, optional*) - Custom convergence criteria, which should be a function of the form `converged(last_matrix, matrix) -> bool` that accepts the current iteration's residualized `matrix` and the last iteration's residualized `last_matrix`. It should return a boolean indicating whether the routine has converged. When a custom convergence criteria is used, `tol` is ignored.

  The following options are supported only by `'map'`:

  - **transform** : (*str, optional*) - Transform operator $T$ that determines the order of projections $P_1, P_2, \ldots, P_n$ for each of the $n$ columns of fixed effects in `ids`. The following transforms are supported:
    - `'kaczmarz'` (default) - Kaczmarz or von Neumann-Halpering operator $T = P_n \cdots P_1$, which is asymmetric and hence does not support `'cg'` acceleration.
    - `'symmetric'` - Symmetric Kaczmarz operator $T = P_n \cdots P_1 \cdots P_n$.
    - `'cimmino'` - Symmetric Cimmino operator $T = (P_1 + \cdots + P_n)/n$.
  - **acceleration** : (*str, optional*) - Method used to accelerate fixed point iteration. The following methods are supported:
    - `'none'` (default) - Simple non-accelerated fixed point iteration.
    - `'gk'` - Line search method of *Gearhart and Koshy (1989)* applied to fixed effect absorption by *Gaure (2013a)*.
    - `'cg'` - Conjugate gradient method described by *Hernández-Ramos, Escalante, and Raydan (2011)*. This method is not supported by the asymmetric `'kaczmarz'` transform.
  - **acceleration_tol** : (*float, optional*) - Acceleration method-specific tolerance for when to stop accelerating the convergence of a vector and switch to simple iteration.

    For `'gk'`, each vector's convergence is accelerated only when the sum of squared residuals relative to the sum of squared vector values is greater than this value, which is by default `1e-16`.

    For `'cg'`, each vector's convergence is accelerated up until the first time that its sum of squared residuals is greater than this value.

  The following options are supported only by `'lsmr'`:

  - **residual_tol** : (*float, optional*) - Convergence criteria S2 from *Fong and Saunders (2011)* based on Stewart's backwards error estimate. This is by default `1e-8`. Convergence based on this criteria can be disabled by setting this value to `0`.
  - **condition_limit** : (*float, optional*) - Maximum estimated condition number of the matrix of fixed effects. For higher estimated condition numbers, an exception will be raised. By default, the maximum estimated condition number is `100000000`.

**Returns** Initialized *Algorithm* for absorbing fixed effects. Class attributes contain information about the number of observations, the number of fixed effect dimensions, and if computed, the number of singletons and degrees of freedom used by the fixed effects.

**Return type** *Algorithm*

### Examples

- *Tutorial*

Algorithm classes contain information about the fixed effects.

| | |
|---|---|
| *Algorithm* | Algorithm for absorbing fixed effects. |

## 2.2 pyhdfe.Algorithm

**class** pyhdfe.**Algorithm**

Algorithm for absorbing fixed effects. Class attributes contain counts of observations and fixed effect dimensions, and if computed, singletons and degrees of freedom used by the fixed effects.

An algorithm is initialized by *create()* with one or more dimensions of fixed effects specified by `ids`. Once initialized, *Algorithm.residualize()* absorbs the fixed effects into a matrix and returns the residuals from a regression of each matrix column on the fixed effects.

**observations**
Number of observations in the data (i.e., the number of rows in `ids`).

**Type** *int*

**dimensions**
Number of fixed effect dimensions (i.e., the number of columns in `ids`).

**Type** *int*

**singletons**
Number of singleton groups or observations. This will be `None` if there was no need to identify singletons (i.e., if `drop_singletons` and `compute_degrees` were both `False` in *create()*).

**Type** *int or None*

**singleton_indices**
Indices of any singleton observations. This will be `None` if there was no need to identify singletons.

**Type** *array or None*

**degrees**
Exact or approximate number of degrees of freedom used by the fixed effects computed according to `degrees_method` in *create()*. This will be `None` if `compute_degrees` was `False` in *create()*.

**Type** *int or None*

### Examples

- *Tutorial*

### Methods

| | |
|---|---|
| *residualize*(matrix[, weights, errors]) | Absorb the fixed effects into a matrix and return the residuals from a regression of each column of the matrix on the fixed effects. |

They can be used to absorb fixed effects (i.e., residualize matrices).

| | |
|---|---|
| *Algorithm.residualize*(matrix[, weights, errors]) | Absorb the fixed effects into a matrix and return the residuals from a regression of each column of the matrix on the fixed effects. |

## 2.3 pyhdfe.Algorithm.residualize

`Algorithm.`**`residualize`**(*matrix*, *weights=None*, *errors='raise'*)

Absorb the fixed effects into a matrix and return the residuals from a regression of each column of the matrix on the fixed effects.

> **Warning:** This function assumes that all of your data have already been cleaned. For example, it will not drop observations with null values. It will also not do any checks on provided weights (e.g., if they are all larger than zero).

**Parameters**

- **`matrix`** (*array-like*) – The two-dimensional array to residualize, which should have a number of rows equal to *`Algorithm.observations`* (i.e., the number of rows in the `ids` passed to *`create()`*).
- **`weights`** (*array-like, optional*) – Two-dimensional array with weights, which should have a number of rows equal to *`Algorithm.observations`* (i.e., the number of rows in the `ids` passed to *`create()`*), and one column. Currently supported algorithms are `'within'`, `'dummy'`, and non-accelerated `'map'`.
- **`errors`** (*str, optional*) – If *'raise'*, the default, any errors raise an exception. If *'warn'*, non-critical errors will generate a warning and residualization will continue. For example, if an iteration limit is hit, *'raise'* will raise an exception, while *'warn'* will warn that the limit is hit but still return the non-converged, partially residualized matrix.

**Returns** Residuals from a (potentially weighted) regression of each column of `matrix` on the fixed effects. This matrix has the same number of columns as `matrix`. If any singleton observations were dropped when initializing the *`Algorithm`* (this is the default behavior of *`create()`*), the residualized matrix will have correspondingly fewer rows.

**Return type** *ndarray*

### Examples

- *Tutorial*

CHAPTER

THREE

# TUTORIAL

This section uses a series of Jupyter Notebooks to demonstrate how pyhdfe can be used together with regression routines from other packages. Each notebook employs the Frisch-Waugh-Lovell (FWL) theorem of *Frisch and Waugh (1933)* and *Lovell (1963)* to run a fixed effects regression by residualizing (projecting) the variables of interest.

This tutorial is just meant to demonstrate how pyhdfe can be used in the simplest of applications. For detailed information about the different algorithms supported by pyhdfe, refer to *API Documentation*.

The online version of the following section may be easier to read.

## 3.1 scikit-learn

```
import pyhdfe
import numpy as np
from sklearn import datasets, linear_model

pyhdfe.__version__
```

```
'0.2.0'
```

In this tutorial, we'll use the boston data set from scikit-learn to demonstrate how pyhdfe can be used to absorb fixed effects before running regressions.

First, load the data set and create a matrix of fixed effect IDs. We'll use a dummy for the Charles river and an index of accessibility to radial highways.

```
boston = datasets.load_boston().data
ids = boston[:, [3, 8]]
ids
```

```
C:\Programs\Anaconda\envs\pyhdfe\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function load_boston␣
↪is deprecated; `load_boston` is deprecated in 1.0 and will be removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np

        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
```

(continues on next page)

(continued from previous page)

```
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.
  warnings.warn(msg, category=FutureWarning)
```

```
array([[0., 1.],
       [0., 2.],
       [0., 2.],
       ...,
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

Next, choose our variables: per capita crime rate, proportion of residential land zoned for lots over 25,000 square feet, and proportion of non-retail business acres per town.

```
variables = boston[:, :3]
variables
```

```
array([[6.3200e-03, 1.8000e+01, 2.3100e+00],
       [2.7310e-02, 0.0000e+00, 7.0700e+00],
       [2.7290e-02, 0.0000e+00, 7.0700e+00],
       ...,
       [6.0760e-02, 0.0000e+00, 1.1930e+01],
       [1.0959e-01, 0.0000e+00, 1.1930e+01],
       [4.7410e-02, 0.0000e+00, 1.1930e+01]])
```

The *create* function initializes an *Algorithm* for fixed effect absorption that can residualize matrices with *Algorithm.residualize*. We'll use the default algorithm. You may want to try other algorithms if it takes a long time to absorb fixed effects into your data.

```
algorithm = pyhdfe.create(ids)
residualized = algorithm.residualize(variables)
residualized
```

```
array([[-1.08723516e-01, -2.20167195e+01, -2.65583593e+00],
       [-5.59754167e-02, -2.04166667e+01, -2.56083333e+00],
       [-5.59954167e-02, -2.04166667e+01, -2.56083333e+00],
       ...,
       [-5.42835164e-02, -4.00167195e+01,  6.96416407e+00],
       [-5.45351644e-03, -4.00167195e+01,  6.96416407e+00],
       [-6.76335164e-02, -4.00167195e+01,  6.96416407e+00]])
```

We can now run a regression of per capita crime rate on the other two variables and our fixed effects.

```
y = residualized[:, [0]]
X = residualized[:, 1:]
regression = linear_model.LinearRegression()
regression.fit(X, y)
regression.coef_
```

```
array([[-6.97058632e-05,  5.53038164e-02]])
```

The online version of the following section may be easier to read.

## 3.2 statsmodels

```
import pyhdfe
import numpy as np
import statsmodels.api as sm
from sklearn import datasets

pyhdfe.__version__
```

```
'0.2.0'
```

In this tutorial, we'll use the boston data set from scikit-learn to demonstrate how pyhdfe can be used to absorb fixed effects before running regressions with statsmodels. We'll also demonstrate how pyhdfe can be used to compute degrees of freedom used by fixed effects.

First, load the data set and create a matrix of fixed effect IDs. We'll use a dummy for the Charles river and an index of accessibility to radial highways.

```
boston = datasets.load_boston().data
ids = boston[:, [3, 8]]
ids
```

```
C:\Programs\Anaconda\envs\pyhdfe\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function load_boston␣
↪is deprecated; `load_boston` is deprecated in 1.0 and will be removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np
```

(continues on next page)

(continued from previous page)

```
        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.
  warnings.warn(msg, category=FutureWarning)
```

```
array([[0., 1.],
       [0., 2.],
       [0., 2.],
       ...,
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

Next, choose our variables: per capita crime rate, proportion of residential land zoned for lots over 25,000 square feet, and proportion of non-retail business acres per town.

```
variables = boston[:, :3]
variables
```

```
array([[6.3200e-03, 1.8000e+01, 2.3100e+00],
       [2.7310e-02, 0.0000e+00, 7.0700e+00],
       [2.7290e-02, 0.0000e+00, 7.0700e+00],
       ...,
       [6.0760e-02, 0.0000e+00, 1.1930e+01],
       [1.0959e-01, 0.0000e+00, 1.1930e+01],
       [4.7410e-02, 0.0000e+00, 1.1930e+01]])
```

The *create* function initializes an *Algorithm* for fixed effect absorption that can residualize matrices with *Algorithm.residualize*. We'll use the default algorithm. You may want to try other algorithms if it takes a long time to absorb fixed effects into your data.

```
algorithm = pyhdfe.create(ids)
residualized = algorithm.residualize(variables)
residualized
```

```
array([[-1.08723516e-01, -2.20167195e+01, -2.65583593e+00],
       [-5.59754167e-02, -2.04166667e+01, -2.56083333e+00],
       [-5.59954167e-02, -2.04166667e+01, -2.56083333e+00],
       ...,
       [-5.42835164e-02, -4.00167195e+01,  6.96416407e+00],
       [-5.45351644e-03, -4.00167195e+01,  6.96416407e+00],
       [-6.76335164e-02, -4.00167195e+01,  6.96416407e+00]])
```

We can now run a regression of per capita crime rate on the other two variables and our fixed effects.

```
y = residualized[:, [0]]
X = residualized[:, 1:]
ols = sm.OLS(y, X)
result = ols.fit()
result.params
```

```
array([-6.97058632e-05,  5.53038164e-02])
```

Standard errors can be adjusted to account for the degrees of freedom that are lost because of the fixed effects. By default, fixed effect degrees of freedom are computed when *create* initializes an algorithm and are stored in *Algorithm.degrees*.

```
se = result.HC0_se
se
```

```
array([0.00109298, 0.00962226])
```

```
se_adjusted = np.sqrt(np.square(se) * result.df_resid / (result.df_resid - algorithm.degrees))
se_adjusted
```

```
array([0.00110398, 0.00971916])
```

CHAPTER

FOUR

# REFERENCES

## 4.1 Papers

### 4.1.1 Abowd, Creecy, and Kramarz (2002)

Abowd, John M., Robert H. Creecy, and Francis Kramarz (2002). Computing person and firm effects using linked longitudinal employer-employee data. Longitudinal Employer-Household Dynamics Technical Papers 2002-06, Center for Economic Studies, U.S. Census Bureau.

### 4.1.2 Correia (2015)

Correia, Sergio (2015). Singletons, cluster-robust standard errors and fixed effects: A bad mix. Technical Note, Duke University.

### 4.1.3 Correia (2017)

Correia, Sergio (2017). Linear models with high-dimensional fixed effects: An efficient and feasible estimator. Working Paper.

### 4.1.4 Fong and Saunders (2011)

Fong, David Chin-Lung, and Michael Saunders (2011). LSMR: An iterative algorithm for sparse least-squares problems. *SIAM Journal on Scientific Computing, 33* (5), 2950–2971.

### 4.1.5 Frisch and Waugh (1933)

Frisch, Ragnar, and Frederick V. Waugh (1933). Partial time regressions as compared with individual trends. *Econometrica, 1* (4), 387-401.

### 4.1.6 Gaure (2013a)

Gaure, Simen (2013a). OLS with multiple high dimensional category variables. *Computational Statistics & Data Analysis, 66* (0), 8-18.

### 4.1.7 Gaure (2013b)

Gaure, Simen (2013b). lfe: Linear group fixed effects. *The R Journal, 5* (2), 104-117.

### 4.1.8 Gearhart and Koshy (1989)

Gearhart, William B., and Mathew Koshy (1989). Acceleration schemes for the method of alternating projections. *Journal of Computational and Applied Mathematics, 26* (3), 235-249.

### 4.1.9 Guimarães and Portugal (2010)

Guimarães, Paulo, and Pedro Portugal (2010). A simple feasible procedure to fit models with high-dimensional fixed effects. *Stata Journal, 10* (4), 628-649.

### 4.1.10 Hernández-Ramos, Escalante, and Raydan (2011)

Hernández-Ramos, Luis M., René Escalante, and Marcos Raydan (2011). Unconstrained optimization techniques for the acceleration of alternating projection methods. *Numerical Functional Analysis and Optimization, 32* (10), 1041-1066.

### 4.1.11 Lovell (1963)

Lovell, Michael C. (1963). Seasonal adjustment of economic time series and multiple regression analysis. *Journal of the American Statistical Association, 58* (304), 993-1010.

### 4.1.12 Somaini and Wolak (2016)

Somaini, Paulo, and Frank A. Wolak (2016). An algorithm to estimate the two-way fixed effects model. *Journal of Econometric Methods, 5* (1), 143-152.

## 4.2 Software

### 4.2.1 FixedEffectModels.jl

FixedEffectModels.jl. Julia. Matthieu Gomez. Implements a version of *Guimarães and Portugal (2010)*, *Gaure (2013a)*, *Gaure (2013b)*, and *Correia (2017)*.

### 4.2.2 lfe

lfe. R. Simen Gaure. Implements *Guimarães and Portugal (2010)*, *Gaure (2013a)*, and *Gaure (2013b)*.

### 4.2.3 reghdfe

reghdfe. Stata. Sergio Correia. Implements *Correia (2017)*, which augments *Guimarães and Portugal (2010)*, *Gaure (2013a)*, and *Gaure (2013b)*.

### 4.2.4 res2fe

res2fe. Matlab, SAS, and Stata. Paulo Somaini and Frank Wolak. Implements *Somaini and Wolak (2016)*.

CHAPTER

FIVE

# LEGAL

# Part II

# Developer Documentation

CHAPTER

SIX

# CONTRIBUTING

Please use the GitHub issue tracker to report bugs or to request features. Contributions are welcome. Examples include:

- Code optimizations.
- Documentation improvements.
- Alternate algorithms that have been implemented in the literature but not in PyHDFE.

CHAPTER

SEVEN

# TESTING

Testing is done with the tox automation tool, which runs a pytest-backed test suite in the `tests` module.

## 7.1 Testing Requirements

In addition to the installation requirements for the package itself, running tests and building documentation requires additional packages specified by the `tests` and `docs` extras in `setup.py`, along with any other explicitly specified `deps` in `tox.ini`.

## 7.2 Running Tests

Defined in `tox.ini` are environments that test the package under different python versions, check types, enforce style guidelines, verify the integrity of the documentation, and release the package. The following command can be run in the top-level pyfwl directory to run all testing environments:

```
tox
```

You can choose to run only one environment, such as the one that builds the documentation, with the `-e` flag:

```
tox -e docs
```

## 7.3 Test Organization

Fixtures, which are defined in `tests.conftest`, configure the testing environment and load data according to a range of specifications.

Tests in `tests.test_hdfe` verify that different algorithms yield the same solutions.

CHAPTER

EIGHT

# VERSION NOTES

These notes will only include major changes.

## 8.1 0.2

- Initial support for weights.

## 8.2 0.1

- Initial release.

# Part III

# Indices

# INDEX


## A

Algorithm (*class in pyhdfe*), 8

## C

create() (*in module pyhdfe*), 5

## D

degrees (*pyhdfe.Algorithm attribute*), 8
dimensions (*pyhdfe.Algorithm attribute*), 8

## O

observations (*pyhdfe.Algorithm attribute*), 8

## R

residualize() (*pyhdfe.Algorithm method*), 9

## S

singleton_indices (*pyhdfe.Algorithm attribute*), 8
singletons (*pyhdfe.Algorithm attribute*), 8