

---

# **ipfp\_python**

***Release 1.0.0***

**Bernard Salanie**

**May 28, 2020**



# CONTENTS

1	Guide	1
2	Module <code>ipfp_solvers</code>	3
3	Module <code>ipfp_utils</code>	5
	Python Module Index	7
	Index	9



`ipfp_python` contains Ppython implementations of the Iterative Projection Fitting Procedure (IPFP) algorithm to solve for equilibrium and do comparative statics in several separable matching models of the [Choo and Siow 2006](#) variety.

This class of matching models is one-to-one, bipartite, separable with perfectly transferable utilities—see [Galichon and Salanié 2020](#) for a general study. For concreteness, I will use the terms *men* and *women* to describe the two sides of the market. The joint surplus created by a match between a man  $i$  who belongs to a discrete category  $x$  and a woman  $j$  who belongs to a discrete category  $y$  is

$$\tilde{\Phi}_{ij} = \Phi_{xy} + \varepsilon_y^i + \eta_x^j.$$

The original Choo and Siow model had the  $\varepsilon_y^i$  and  $\eta_x^j$  error terms drawn iid from a standard type I extreme value (multinomial logit) distribution. We call it the *homoskedastic* model. The function `ipfp_homo_solver()` solves for its equilibrium given the values of the joint surplus (the matrix  $\Phi$ ) and the margins (the numbers  $n_x$  and  $m_y$  of men and women in each discrete category).

The `ipfp_python` module also contains solvers for

- the homoskedastic model without singles (`ipfp_homo_solver_no_singles()`), for use when only data on realized matches is available)
- a gender-heteroskedastic model (`ipfp_hetero_solver()`), which allows for with a scale parameter on the error term for women (that is,  $\tau\eta_x^j$ )
- a gender- and type-heteroskedastic (`ipfp_heteroxy_solver()`), with type-dependent scale parameters on the error terms for men and for women:

$$\varepsilon_y^i + \eta_x^j \rightarrow \sigma_x \varepsilon_y^i + \tau_y \eta_x^j$$

In the heteroskedastic models, the scale parameters must also be provided as inputs to the algorithm.

Each solver has two tuning parameters that control when it stops:

- *tol* is a tolerance on the difference between candidate solutions at two successive iterations
- *maxiter* sets an upper limit on the number of iterations.

They are set at reasonable defaults, but you may want to change *tol* at least.

In addition to the equilibrium matching patterns by cell  $(\mu_{xy}, \mu_{x0}, \mu_{0y})$ , (only  $\mu_{xy}$  for `ipfp_homo_solver_no_singles()`), the solvers also return the adjustment errors on the margins, and, if the optional parameter *gr* is set to *True*, the derivatives of the equilibrium matching patterns with respect to the parameters: the joint surplus, the margins, and the scale parameters if any.

The algorithm and its properties are described in detail in [Galichon and Salanié 2020](#). It is extremely fast and robust.



## MODULE IPFP\_SOLVERS

Implementations of the IPFP algorithm to solve for equilibrium and do comparative statics in several variants of the Choo and Siow 2006 model:

- homoskedastic with singles (as in CS 2006)
- homoskedastic without singles
- gender-heteroskedastic: with a scale parameter on the error term for women
- gender- and type-heteroskedastic: with a scale parameter on the error term for women

each solver, when fed the joint surplus and margins, returns the equilibrium matching patterns, the adding-up errors on the margins, and if requested (`gr=True`) the derivatives of the matching patterns in all primitives.

`ipfp_solvers.ipfp_hetero_solver` (*Phi*, *men\_margins*, *women\_margins*, *tau*, *tol=1e-09*, *gr=False*,  
*verbose=False*, *maxiter=1000*)  
solve for equilibrium in a in a gender-heteroskedastic Choo and Siow market  
given systematic surplus and margins and a scale parameter `dist_params[0]`

### Parameters

- **Phi** (*np.array*) – matrix of systematic surplus, shape (ncat\_men, ncat\_women)
- **men\_margins** (*np.array*) – vector of men margins, shape (ncat\_men)
- **women\_margins** (*np.array*) – vector of women margins, shape (ncat\_women)
- **tau** (*float*) – a positive scale parameter for the error term on women
- **tol** (*float*) – tolerance on change in solution
- **gr** (*boolean*) – if True, also evaluate derivatives of `muxy` wrt `Phi`
- **verbose** (*boolean*) – prints stuff
- **maxiter** (*int*) – maximum number of iterations
- **dist\_params** (*np.array*) – array of one positive number (the scale parameter for women)

**Returns** (`muxy`, `mux0`, `mu0y`), errors on margins `marg_err_x`, `marg_err_y`, and gradients of (`muxy`, `mux0`, `mu0y`) wrt (`men_margins`, `women_margins`, `Phi`, `dist_params[0]`) if `gr=True`

`ipfp_solvers.ipfp_heteroxy_solver` (*Phi*, *men\_margins*, *women\_margins*, *sigma\_x*, *tau\_y*,  
*tol=1e-09*, *gr=False*, *maxiter=1000*, *verbose=False*)  
solve for equilibrium in a in a gender- and type-heteroskedastic Choo and Siow market  
given systematic surplus and margins and a scale parameter `dist_params[0]`

### Parameters

- **Phi** (*np.array*) – matrix of systematic surplus, shape (ncat\_men, ncat\_women)
- **men\_margins** (*np.array*) – vector of men margins, shape (ncat\_men)
- **women\_margins** (*np.array*) – vector of women margins, shape (ncat\_women)
- **sigma\_x** (*np.array*) – an array of positive numbers of shape (ncat\_men)
- **tau\_y** (*np.array*) – an array of positive numbers of shape (ncat\_women)
- **tol** (*float*) – tolerance on change in solution
- **gr** (*boolean*) – if True, also evaluate derivatives of muxy wrt Phi
- **verbose** (*boolean*) – prints stuff
- **maxiter** (*int*) – maximum number of iterations

**Returns** (muxy, mux0, mu0y), errors on margins marg\_err\_x, marg\_err\_y, and gradients of (muxy, mux0, mu0y) wrt (men\_margins, women\_margins, Phi, dist\_params) if gr=True

`ipfp_solvers.ipfp_homo_nosingles_solver` (*Phi, men\_margins, women\_margins, tol=1e-09, gr=False, verbose=False, maxiter=1000*)

solve for equilibrium in a Choo and Siow market without singles

given systematic surplus and margins

#### Parameters

- **Phi** (*np.array*) – matrix of systematic surplus, shape (ncat\_men, ncat\_women)
- **men\_margins** (*np.array*) – vector of men margins, shape (ncat\_men)
- **women\_margins** (*np.array*) – vector of women margins, shape (ncat\_women)
- **tol** (*float*) – tolerance on change in solution
- **gr** (*boolean*) – if True, also evaluate derivatives of muxy wrt Phi
- **verbose** (*boolean*) – prints stuff
- **maxiter** (*int*) – maximum number of iterations

**Returns** muxy, marg\_err\_x, marg\_err\_y and gradients of muxy wrt Phi if gr=True

`ipfp_solvers.ipfp_homo_solver` (*Phi, men\_margins, women\_margins, tol=1e-09, gr=False, verbose=False, maxiter=1000*)

solve for equilibrium in a Choo and Siow market

given systematic surplus and margins

#### Parameters

- **Phi** (*np.array*) – matrix of systematic surplus, shape (ncat\_men, ncat\_women)
- **men\_margins** (*np.array*) – vector of men margins, shape (ncat\_men)
- **women\_margins** (*np.array*) – vector of women margins, shape (ncat\_women)
- **tol** (*float*) – tolerance on change in solution
- **gr** (*boolean*) – if True, also evaluate derivatives of muxy wrt Phi
- **verbose** (*boolean*) – prints stuff
- **maxiter** (*int*) – maximum number of iterations

**Returns** (muxy, mux0, mu0y), errors on margins marg\_err\_x, marg\_err\_y, and gradients of (muxy, mux0, mu0y) wrt (men\_margins, women\_margins, Phi) if gr=True



## MODULE IPFP\_UTILS

some utility programs used by ipfp\_solvers

`ipfp_utils.der_npexp` (*arr*: *numpy.array*, *bigx*: *float* = 30.0, *verbose*: *bool* = *False*) → *numpy.array*  
derivative of  $C^2$  extension of  $\exp(a)$  above *bigx*

**Parameters**

- **arr** (*np.array*) – a Numpy array
- **bigx** (*float*) – upper bound

**Returns** derivative of  $\exp(a)$   $C^2$ -extended above *bigx*

`ipfp_utils.der_npln` (*arr*: *numpy.array*, *eps*: *float* = 1e-30, *verbose*: *bool* = *False*) → *numpy.array*  
derivative of  $C^2$  extension of  $\ln(a)$  below *eps*

**Parameters**

- **arr** (*np.array*) – a Numpy array
- **eps** (*float*) – lower bound

**Returns** derivative of  $\ln(a)$   $C^2$ -extended below *eps*

`ipfp_utils.der_nppow` (*a*: *numpy.array*, *b*: *Union[int, float, numpy.array]*) → *numpy.array*  
evaluates the derivatives in *a* and *b* of element-by-element  $a^{**}b$

**Parameters**

- **a** (*np.array*) –
- **float, np.array]** **b** (*Union[int, float, numpy.array]*) – if an array, should have the same shape as *a*

**Returns** a pair of two arrays of the same shape as *a*

`ipfp_utils.describe_array` (*v*: *numpy.array*, *name*: *str* = 'v')  
descriptive statistics on an array interpreted as a vector

**Parameters**

- **v** (*np.array*) – the array
- **name** (*str*) – its name

**Returns** the *scipy.stats.describe* object

`ipfp_utils.npexp` (*arr*: *numpy.array*, *bigx*: *float* = 30.0, *verbose*: *bool* = *False*) → *numpy.array*  
 $C^2$  extension of  $\exp(a)$  above *bigx*

**Parameters**

- **arr** (*np.array*) – a Numpy array

- **bigx** (*float*) – upper bound

**Returns** :math:\exp(a)^{C^2}-extended above *bigx*

`ipfp_utils.nplog` (*arr: numpy.array, eps: float = 1e-30, verbose: bool = False*) → *numpy.array*  
*C*<sup>2</sup> extension of  $\ln(a)$  below *eps*

**Parameters**

- **arr** (*np.array*) – a Numpy array
- **eps** (*float*) – lower bound

**Returns**  $\ln(a)^{C^2}$ -extended below *eps*

`ipfp_utils.npmaxabs` (*arr: numpy.array*) → *float*  
maximum absolute value in an array

**Parameters** **arr** (*np.array*) – Numpy array

**Returns** a float

`ipfp_utils.nppow` (*a: numpy.array, b: Union[int, float, numpy.array]*) → *numpy.array*  
evaluates  $a*b$  element-by-element

**Parameters**

- **a** (*np.array*) –
- **float, np.array]** **b** (*Union[int, ...]*) – if an array, should have the same shape as *a*

**Returns** an array of the same shape as *a*

`ipfp_utils.nprepeat_col` (*v: numpy.array, n: int*) → *numpy.array*  
create a matrix with *n* columns equal to *v*

**Parameters**

- **v** (*np.array*) – a 1-dim array of size *m*
- **n** (*int*) – number of columns requested

**Returns** a 2-dim array of shape (*m, n*)

`ipfp_utils.nprepeat_row` (*v: numpy.array, m: int*) → *numpy.array*  
create a matrix with *m* rows equal to *v*

**Parameters**

- **v** (*np.array*) – a 1-dim array of size *n*
- **m** (*int*) – number of rows requested

**Returns** a 2-dim array of shape (*m, n*)

`ipfp_utils.print_stars` (*title: str = None, n: int = 70*) → *None*  
prints a starred line, or two around the title

**Parameters**

- **title** (*str*) – title
- **n** (*int*) – number of stars on line

**Returns** nothing

## PYTHON MODULE INDEX

### i

`ipfp_solvers`, 3

`ipfp_utils`, 5



## INDEX

### D

`der_npexp()` (in module *ipfp\_utils*), 5  
`der_nplog()` (in module *ipfp\_utils*), 5  
`der_nppow()` (in module *ipfp\_utils*), 5  
`describe_array()` (in module *ipfp\_utils*), 5

### I

`ipfp_hetero_solver()` (in module *ipfp\_solvers*), 3  
`ipfp_heteroxy_solver()` (in module *ipfp\_solvers*), 3  
`ipfp_homo_nosingles_solver()` (in module *ipfp\_solvers*), 4  
`ipfp_homo_solver()` (in module *ipfp\_solvers*), 4  
`ipfp_solvers` (module), 3  
`ipfp_utils` (module), 5

### N

`npexp()` (in module *ipfp\_utils*), 5  
`nplog()` (in module *ipfp\_utils*), 6  
`npmaxabs()` (in module *ipfp\_utils*), 6  
`nppow()` (in module *ipfp\_utils*), 6  
`nprepeat_col()` (in module *ipfp\_utils*), 6  
`nprepeat_row()` (in module *ipfp\_utils*), 6

### P

`print_stars()` (in module *ipfp\_utils*), 6