

---

# **ipfp\_python**

***Release 1.0.0***

**Bernard Salanie**

**Apr 26, 2021**



# CONTENTS

1	Module <code>ipfp_solvers</code>	1
2	Module <code>ipfp_utils</code>	5
3	Module <code>estimate_cs_fuvl</code>	7
	Python Module Index	9
	Index	11



## 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_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* the matching patterns, shape (ncat\_men, ncat\_women)
- *marg\_err\_x*, *marg\_err\_y* the errors on the margins
- and the 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) the matching patterns
- marg\_err\_x, marg\_err\_y the errors on the margins
- and the gradients of (muxy, mux0, mu0y) wrt (men\_margins, women\_margins, Phi) if gr=True

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

ipfp\_solvers.**print\_simulated\_ipfp**(muxy, marg\_err\_x, marg\_err\_y)





## MODULE IPFP\_UTILS

some utility programs used by ipfp\_solvers

`ipfp_utils.print_stars` (*title: Optional[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

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

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

**Returns** a float

`ipfp_utils.nplog` (*arr*: *numpy.array*, *eps*: *float* =  $1e-30$ , *verbose*: *bool* = *False*) → *numpy.array*  
 $C^2$  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.der_nplog` (*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.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**  $\exp(a)$   $C^2$ -extended above *bigx*

`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.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.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, ...]*) – if an array, should have the same shape as *a*

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

## MODULE ESTIMATE\_CS\_FUVL

Estimation of the [Choo and Siow 2006](#) model: in its original version (homoskedastic with singles).

We minimize the  $F(u, v, \lambda) - \hat{\mu} \cdot \Phi^\lambda$  function of Galichon–Salanie (2020, Proposition 5.)

```
estimate_cs_fufl.estimate_cs_fufl (mxy:      numpy.ndarray,  nx:      numpy.ndarray,  
                                   my:      numpy.ndarray,  bases:   numpy.ndarray) →  
                                   scipy.optimize.optimize.OptimizeResult
```

this estimates the parameters and equilibrium utilities in a semilinear homoskedastic Choo-Siow model.

### Parameters

- **mxy** (*np.ndarray*) – the numbers of matches in each (*x,y*) cell, a (*X,Y*) matrix
- **nx** (*np.ndarray*) – the numbers of men in each *x* cell, a *X*-vector
- **my** (*np.ndarray*) – the numbers of women in each *y* cell, a *Y*-vector
- **bases** (*np.ndarray*) – the values of the *K* basis functions in each cell, a (*X,Y,K*) array

### Returns

a *scipy.optimize.OptimizeResult* object *resus*. *resus.x* has the estimates of *u*, *v*, and  $\lambda$  in that order:

- $u_x$  the expected utility of men of type *x*
- $v_y$  the expected utility of women of type *y*
- $\lambda_k$  the coefficient of basis function *k*



## PYTHON MODULE INDEX

### e

`estimate_cs_fuvl`, 7

### i

`ipfp_solvers`, 1

`ipfp_utils`, 5



## INDEX

### D

`der_npexp()` (in module *ipfp\_utils*), 6  
`der_nplog()` (in module *ipfp\_utils*), 6  
`der_nppow()` (in module *ipfp\_utils*), 6  
`describe_array()` (in module *ipfp\_utils*), 5

### E

`estimate_cs_fuvl`  
    module, 7  
`estimate_cs_fuvl()` (in module *estimate\_cs\_fuvl*),  
    7

### I

`ipfp_hetero_solver()` (in module *ipfp\_solvers*), 2  
`ipfp_heteroxy_solver()` (in module  
    *ipfp\_solvers*), 2  
`ipfp_homo_nosingles_solver()` (in module  
    *ipfp\_solvers*), 1  
`ipfp_homo_solver()` (in module *ipfp\_solvers*), 1  
`ipfp_solvers`  
    module, 1  
`ipfp_utils`  
    module, 5

### M

module  
    *estimate\_cs\_fuvl*, 7  
    *ipfp\_solvers*, 1  
    *ipfp\_utils*, 5

### N

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

### P

`print_simulated_ipfp()` (in module  
    *ipfp\_solvers*), 3  
`print_stars()` (in module *ipfp\_utils*), 5