

# Package ‘GNE’

August 9, 2018

**Type** Package

**Title** Computation of Generalized Nash Equilibria

**Version** 0.99-2

**Author** Christophe Dutang

**Maintainer** Christophe Dutang <christophe.dutang@ensimag.fr>

**Description** Provide functions to compute standard and generalized Nash Equilibria.  
Optimization methods are available nonsmooth reformulation, fixed-point formulation,  
minimization problem and constrained-equation reformulation.  
See e.g. Kanzow and Facchinei (2010), <doi:10.1007/s10479-009-0653-x>.

**Depends** R (>= 3.0.0), alabama, nleqslv, BB, SQUAREM

**Suggests** sensitivity

**License** GPL (>= 2)

**URL** <http://optimizer.r-forge.r-project.org/>

**NeedsCompilation** yes

**Repository** CRAN

**Date/Publication** 2018-08-09 21:20:12 UTC

## R topics documented:

bench.GNE . . . . .	2
CER . . . . .	4
compl . . . . .	8
eqsolve . . . . .	10
GNE . . . . .	12
GNE.ceq . . . . .	14
GNE.fpeq . . . . .	18
GNE.minpb . . . . .	22
GNE.nseq . . . . .	24
NIR . . . . .	30
potential.reduction . . . . .	32
projector . . . . .	33

rejection . . . . .	35
SSR . . . . .	36
stepfunc . . . . .	44
VIR . . . . .	46
<b>Index</b>	<b>48</b>

---

bench.GNE	<i>Benchmark function</i>
-----------	---------------------------

---

## Description

Benchmark function to compare GNE computational methods.

## Usage

```

bench.GNE.nseq(xinit, ..., echo=FALSE, control=list())
bench.GNE.ceq(xinit, ..., echo=FALSE, control=list())
bench.GNE.fpeq(xinit, ..., echo=FALSE, control.outer=list(),
control.inner=list())
bench.GNE.minpb(xinit, ..., echo=FALSE, control.outer=list(),
control.inner=list())

```

## Arguments

<code>xinit</code>	a numeric vector for the initial point.
<code>...</code>	further arguments to be passed to <code>GNE.nseq</code> , <code>GNE.ceq</code> , <code>GNE.fpeq</code> or <code>GNE.minpb</code> . NOT to the functions <code>func1</code> and <code>func2</code> .
<code>echo</code>	a logical to get some traces of the benchmark computation.
<code>control</code> , <code>control.outer</code> , <code>control.inner</code>	a list with control parameters to be passed to <code>GNE.xxx</code> function.

## Details

Computing generalized Nash Equilibrium can be done in three different approaches.

- (i) **extended KKT system** It consists in solving the non smooth extended Karush-Kuhn-Tucker (KKT) system  $\Phi(z) = 0$ . `func1` is *Phi* and `func2` is *JacPhi*.
- (ii) **fixed point approach** It consists in solving equation  $y(x) = x$ . `func1` is *y* and `func2` is ?
- (iii) **gap function minimization** It consists in minimizing a gap function  $minV(x)$ . `func1` is *V* and `func2` is *GradV*.

**Value**

For `GNE.bench.ceq` and `GNE.bench.nseq`, a `data.frame` is returned with columns:

`method` the name of the method.  
`fctcall` the number of calls of the function.  
`jaccall` the number of calls of the Jacobian.  
`comptime` the computation time.  
`normFx` the norm of the merit function at the final iterate.  
`code` the exit code.  
`localmethods` the name of the local method.  
`globalmethods` the name of the globalization method.  
`x` the final iterate.

For `GNE.bench.minpb`, a `data.frame` is returned with columns:

`method` the name of the method.  
`minfncall.outer` the number of calls of the merit function.  
`grminfncall.outer` the number of calls of the gradient of the merit function.  
`gapfncall.inner` the number of calls of the gap function.  
`grgapfncall.outer` the number of calls of the gradient of the gap function.  
`comptime` the computation time.  
`normFx` the norm of the merit function at the final iterate.  
`code` the exit code.  
`x` the final iterate.

For `GNE.bench.fpeq`, a `data.frame` is returned with columns:

`method` the name of the method.  
`fpfncall.outer` the number of calls of the fixed-point function.  
`merfncall.outer` the number of calls of the merit function.  
`gapfncall.inner` the number of calls of the gap function.  
`grgapfncall.outer` the number of calls of the gradient of the gap function.  
`comptime` the computation time.  
`normFx` the norm of the merit function at the final iterate.  
`code` the exit code.  
`x` the final iterate.

**Author(s)**

Christophe Dutang

## References

F. Facchinei, A. Fischer & V. Piccialli (2009), *Generalized Nash equilibrium problems and Newton methods*, Math. Program.

A. von Heusinger (2009), *Numerical Methods for the Solution of the Generalized Nash Equilibrium Problem*, Ph. D. Thesis.

A. von Heusinger & J. Kanzow (2009), *Optimization reformulations of the generalized Nash equilibrium problem using Nikaido-Isoda-type functions*, Comput Optim Appl .

## See Also

See [GNE.fpeq](#), [GNE.minpb](#), [GNE.ceq](#) and [GNE.nseq](#) for other approaches.

---

CER

*Constrained Equation Reformulation*

---

## Description

functions of the Constrained Equation Reformulation of the GNEP

## Usage

```
funCER(z, dimx, dimlam,
grobj, arggrobj,
constr, argconstr,
grconstr, arggrconstr,
dimmu, joint, argjoint,
grjoint, arggrjoint,
echo=FALSE)
```

```
jacCER(z, dimx, dimlam,
heobj, argheobj,
constr, argconstr,
grconstr, arggrconstr,
heconstr, argheconstr,
dimmu, joint, argjoint,
grjoint, arggrjoint,
hejoint, arghejoint,
echo=FALSE)
```

## Arguments

z	a numeric vector z containing x then lambda values.
dimx	dimension of x.
dimlam	dimension of lambda.

grobj	gradient of the objective function, see details.
arggrobj	a list of additional arguments of the objective gradient.
constr	constraint function, see details.
argconstr	a list of additional arguments of the constraint function.
grconstr	gradient of the constraint function, see details.
arggrconstr	a list of additional arguments of the constraint gradient.
dimmu	a vector of dimension for $\mu$ .
joint	joint function, see details.
argjoint	a list of additional arguments of the joint function.
grjoint	gradient of the joint function, see details.
arggrjoint	a list of additional arguments of the joint gradient.
heobj	Hessian of the objective function, see details.
argheobj	a list of additional arguments of the objective Hessian.
heconstr	Hessian of the constraint function, see details.
argheconstr	a list of additional arguments of the constraint Hessian.
hejoint	Hessian of the joint function, see details.
arghejoint	a list of additional arguments of the joint Hessian.
echo	a logical to show some traces.

### Details

Compute the H function or the Jacobian of the H function defined in Dreves et al.(2009).

**Arguments of the H function** The arguments which are functions must respect the following features

grobj The gradient  $GradObj$  of an objective function  $Obj$  (to be minimized) must have 3 arguments for  $GradObj(z, playnum, ideriv)$ : vector  $z$ , player number, derivative index, and optionnally additional arguments in `arggrobj`.

constr The constraint function  $g$  must have 2 arguments: vector  $z$ , player number, such that  $g(z, playnum) \leq 0$ . Optionnally,  $g$  may have additional arguments in `argconstr`.

grconstr The gradient of the constraint function  $g$  must have 3 arguments: vector  $z$ , player number, derivative index, and optionnally additional arguments in `arggrconstr`.

**Arguments of the Jacobian of H** The arguments which are functions must respect the following features

heobj It must have 4 arguments: vector  $z$ , player number, two derivative indexes.

heconstr It must have 4 arguments: vector  $z$ , player number, two derivative indexes.

Optionnally, `heobj` and `heconstr` can have additional arguments `argheobj` and `argheconstr`.

See the example below.

### Value

A vector for `funcER` or a matrix for `jacER`.

**Author(s)**

Christophe Dutang

**References**

Dreves, A., Facchinei, F., Kanzow, C. and Sagratella, S. (2011), *On the solutions of the KKT conditions of generalized Nash equilibrium problems*, SIAM Journal on Optimization.

F. Facchinei, A. Fischer and V. Piccialli (2009), *Generalized Nash equilibrium problems and Newton methods*, Math. Program.

**See Also**

See also [GNE.ceq](#).

**Examples**

```
#-----
# (1) Example 5 of von Facchinei et al. (2007)
#-----

dimx <- c(1, 1)
#Gr_x_j 0_i(x)
grobj <- function(x, i, j)
{
  if(i == 1)
    res <- c(2*(x[1]-1), 0)
  if(i == 2)
    res <- c(0, 2*(x[2]-1/2))
  res[j]
}
#Gr_x_k Gr_x_j 0_i(x)
heobj <- function(x, i, j, k)
  2 * (i == j && j == k)

dimlam <- c(1, 1)
#constraint function g_i(x)
g <- function(x, i)
  sum(x[1:2]) - 1
#Gr_x_j g_i(x)
grg <- function(x, i, j)
  1
#Gr_x_k Gr_x_j g_i(x)
heg <- function(x, i, j, k)
  0

x0 <- rep(0, sum(dimx))
z0 <- c(x0, 2, 2, max(10, 5-g(x0, 1) ), max(10, 5-g(x0, 2) ) )
```

```

#true value is (3/4, 1/4, 1/2, 1/2)
funCER(z0, dimx, dimlam, grobj=grobj,
constr=g, grconstr=grg)

jacCER(z0, dimx, dimlam, heobj=heobj,
constr=g, grconstr=grg, heconstr=heg)

#-----
# (2) Duopoly game of Krawczyk and Stanislav Uryasev (2000)
#-----

#constants
myarg <- list(d= 20, lambda= 4, rho= 1)

dimx <- c(1, 1)
#Gr_x_j 0_i(x)
grobj <- function(x, i, j, arg)
{
res <- -arg$rho * x[i]
if(i == j)
res <- res + arg$d - arg$lambda - arg$rho*(x[1]+x[2])
-res
}
#Gr_x_k Gr_x_j 0_i(x)
heobj <- function(x, i, j, k, arg)
arg$rho * (i == j) + arg$rho * (j == k)

dimlam <- c(1, 1)
#constraint function g_i(x)
g <- function(x, i)
-x[i]
#Gr_x_j g_i(x)
grg <- function(x, i, j)
-1*(i == j)
#Gr_x_k Gr_x_j g_i(x)
heg <- function(x, i, j, k)
0

#true value is (16/3, 16/3, 0, 0)

x0 <- rep(0, sum(dimx))
z0 <- c(x0, 2, 2, max(10, 5-g(x0, 1) ), max(10, 5-g(x0, 2) ) )

funCER(z0, dimx, dimlam, grobj=grobj, arggrobj=myarg,
constr=g, grconstr=grg)

jacCER(z0, dimx, dimlam, heobj=heobj,

```

```
argheobj=myarg, constr=g, grconstr=grg, heconstr=heg)
```

---

 compl

*Complementarity functions*


---

### Description

Classic Complementarity functions

### Usage

```
phiFB(a, b)
GrAphiFB(a, b)
GrBphiFB(a, b)
```

```
phipFB(a, b, p)
GrAhipFB(a, b, p)
GrBhipFB(a, b, p)
```

```
phirFB(a, b)
GrAphirFB(a, b)
GrBphirFB(a, b)
```

```
phiMin(a, b)
GrAphiMin(a, b)
GrBphiMin(a, b)
```

```
phiMan(a, b, f, fprime)
GrAphiMan(a, b, f, fprime)
GrBphiMan(a, b, f, fprime)
```

```
phiKK(a, b, lambda)
GrAphiKK(a, b, lambda)
GrBphiKK(a, b, lambda)
```

```
phiLT(a, b, q)
GrAphiLT(a, b, q)
GrBphiLT(a, b, q)
```

```
compl.par(type=c("FB", "pFB", "rFB", "Min", "Man", "LT", "KK"),
p, f, fprime, q, lambda)
```



```
## S3 method for class 'compl.par'
print(x, ...)

## S3 method for class 'compl.par'
summary(object, ...)
```

### Arguments

a	first parameter.
b	second parameter.
f, fprime	a univariate function and its derivative.
lambda	a parameter in $[0, 2[$ .
q	a parameter $>1$ .
p	a parameter $>0$ .
type	a character string for the complementarity function type: either "FB", "Min", "Man", "LT" or "KK".
x, object	an object of class "compl.par".
...	further arguments to pass to print, or summary.

### Details

We implement 5 complementarity functions From Facchinei & Pang (2003).

- (i) **phiFB** the Fischer-Burmeister complementarity function  $\sqrt{a^2 + b^2} - (a + b)$ . The penalized version is  $\text{phiFB}(a, b) - p * \max(a, 0) * \max(b, 0)$ , whereas the regularized version is  $\text{phiFB}(a, b) - \text{epsilon}$ .
- (ii) **phiMin** the minimum complementarity function  $\min(a, b)$ .
- (iii) **phiMan** the Mangasarian's family of complementarity function  $f(|a - b|) - f(a) - f(b)$ , typically  $f(t) = t$  or  $f(t) = t^3$ .
- (iv) **phiKK** the Kanzow-Kleinmichel complementarity function  $(\sqrt{((a - b)^2 + 2 * \lambda * a * b)} - (a + b)) / (2 - \lambda)$ .
- (v) **phiLT** the Luo-Tseng complementarity function  $(a^q + b^q)^{1/q} - (a + b)$ .

GrXXX and GrBXXX implements the derivative of the complementarity function XXX with respect to  $a$  and  $b$  respectively.

compl.par creates an object of class "compl.par" with attributes "type" a character string and "fun", "grA", "grB" the corresponding functions for a given type. Optional arguments are also available, e.g. lambda for the KK complementarity function.

### Value

A numeric or an object of class "compl.par".

**Author(s)**

Christophe Dutang

**References**

F. Facchinei and J.S. Pang, *Finite-Dimensional Variational Inequalities and Complementarity Problems*, Springer-Verlag (New York 2003).

**See Also**

See also [GNE.nseq](#).

**Examples**

```
phiFB(1, 2)
phiLT(1, 2, 2)
phiKK(1, 2, 1)

-2*phiMin(1, 2)
phiMan(1, 2, function(t) t)

complFB <- compl.par("FB")
summary(complFB)

complKK <- compl.par("KK", lambda=1)
summary(complKK)

complKK$fun(1, 1, complKK$lambda)
complFB$fun(1, 1)
```

---

eqsolve

*Solving non linear equations*

---

**Description**

Non linear Solving methods

**Usage**

```
eqsolve(xinit, f, jac,
        method=c("Newton", "Levenberg-Marquardt", "Broyden"),
        global=c("line search", "none"), control=list())
```

**Arguments**

<code>xinit</code>	initial point.
<code>f</code>	the function for which we search roots.
<code>jac</code>	the Jacobian of the function <code>f</code> .
<code>method</code>	a character string specifying the method to use: either "Newton", "Levenberg-Marquardt", or "Broyden".
<code>global</code>	a character string for the globalization method to be used: either "line search" or "none".
<code>control</code>	a list for the control parameters. See details.

**Details**

The `control` argument is a list that can supply any of the following components:

<code>tol</code>	The absolute convergence tolerance. Default to 1e-6.
<code>maxit</code>	The maximum number of iterations. Default to 100.
<code>echo</code>	A logical or an integer (0, 1, 2, 3, 4) to print traces. Default to FALSE, i.e. 0.
<code>echofile</code>	A character string to store the traces in that file. Default to NULL.
<code>echograph</code>	A character string to plot iter-by-iter information. Either "NULL" (default), or "line" for line search plot or "trust" for trust region plots.
<code>sigma</code>	Reduction factor for the geometric linesearch. Default to 0.5.
<code>btol</code>	The backtracking tolerance. Default to 0.01.
<code>delta</code>	The exponent parameter for the LM parameter, should in [1, 2]. Default to 2.
<code>initlnsrch</code>	The initial integer for starting the line search. Default to 0.
<code>minstep</code>	The minimal step. Default to 0.001.

**Value**

A list with components:

<code>par</code>	The best set of parameters found.
<code>counts</code>	A two-element integer vector giving the number of calls to <code>phi</code> and <code>jacphi</code> respectively.
<code>iter</code>	The iteration number.
<code>code</code>	0 if convergence, 1 if <code>maxit</code> is reached, 10 if <code>tol</code> is not reached and 11 for both.

**Author(s)**

Christophe Dutang

**See Also**

See [nleqslv](#) from the package of the same name.

GNE

*GNE package***Description**

Generalized Nash Equilibrium computational methods.

**Usage**

```
GNE(approach =
c("non smooth", "fixed point", "minimization", "constrained equation"),
method = "default", xinit, control=list(), ...)
```

**Arguments**

approach	a character string for the approach: either "non smooth", "fixed point", "minimization" or "constrained equation".
method	a character string for the computation method: either "default" or the name of the method.
xinit	a numeric vector for the initial point.
...	further arguments to be passed to <code>GNE.nseq</code> , <code>GNE.fpeq</code> or <code>GNE.minpb</code> .
control	a list with control parameters.

**Details**

Computing generalized Nash Equilibrium can be done in three different approaches.

**(i) extended KKT system** It consists in solving the non smooth extended Karush-Kuhn-Tucker (KKT) system  $\Phi(z) = 0$ .

**(ii) fixed point approach** It consists in solving equation  $y(x) = x$ .

**(iii) gap function minimization** It consists in minimizing a gap function  $\min V(x)$ .

**(iv) constrained equation** It consists in solving  $F(x)$  such that  $x$  belongs to a specific set.

The GNE function is a global function calling the appropriate function `GNE.nseq`, `GNE.fpeq`, `GNE.ceq` or `GNE.minpb`. Benchmark functions comparing all methods for a given reformulation are available: see `bench.GNE`.

Additional utility functions are also available: `rejection`, `projector`, `stepfunc`, `complementarity` and `funSSR`.

**Value**

A list with components:

`par` The best set of parameters found.

`value` The value of the merit function.

`counts` A two-element integer vector giving the number of calls to `phi` and `jacphi` respectively.

`iter` The outer iteration number.

`code` The values returned are

- 1 Function criterion is near zero. Convergence of function values has been achieved.
- 2  $x$ -values within tolerance. This means that the relative distance between two consecutive  $x$ -values is smaller than `xtol`.
- 3 No better point found. This means that the algorithm has stalled and cannot find an acceptable new point. This may or may not indicate acceptably small function values.
- 4 Iteration limit `maxit` exceeded.
- 5 Jacobian is too ill-conditioned.
- 6 Jacobian is singular.
- 100 an error in the execution.

`message` a string describing the termination code

`fvec` a vector with function values.

`approach` the name of the approach.

**Author(s)**

Christophe Dutang

**References**

F. Facchinei, A. Fischer and V. Piccialli (2009), *Generalized Nash equilibrium problems and Newton methods*, Math. Program.

A. von Heusinger (2009), *Numerical Methods for the Solution of the Generalized Nash Equilibrium Problem*, Ph. D. Thesis.

A. von Heusinger and C. Kanzow (2009), *Optimization reformulations of the generalized Nash equilibrium problem using Nikaido-Isoda-type functions*, Comput Optim Appl .

F. Facchinei and C. Kanzow (2009), *Generalized Nash Equilibrium problems*. Preprint 290.

C. Dutang (2013), *A survey of GNE computation methods: theory and algorithms*, preprint on HAL, <http://hal.archives-ouvertes.fr/hal-00813531>.

**See Also**

See [GNE.fpeq](#), [GNE.minpb](#), [GNE.ceq](#) and [GNE.nseq](#) for other approaches.

GNE.ceq

*Constrained equation reformulation of the GNE problem.***Description**

Constrained equation reformulation via the extended KKT system of the GNE problem.

**Usage**

```
GNE.ceq(init, dimx, dimlam, grobj, arggrobj, heobj, argheobj,
constr, argconstr, grconstr, arggrconstr, heconstr, argheconstr,
dimmu, joint, argjoint, grjoint, arggrjoint, hejoint, arghejoint,
method="PR", control=list(), silent=TRUE, ...)
```

**Arguments**

<code>init</code>	Initial values for the parameters to be optimized over: $z = (x, \lambda, \mu)$ .
<code>dimx</code>	a vector of dimension for $x$ .
<code>dimlam</code>	a vector of dimension for $\lambda$ .
<code>gobj</code>	gradient of the objective function (to be minimized), see details.
<code>arggrobj</code>	a list of additional arguments of the objective gradient.
<code>heobj</code>	Hessian of the objective function, see details.
<code>argheobj</code>	a list of additional arguments of the objective Hessian.
<code>constr</code>	constraint function ( $g^i(x) \leq 0$ ), see details.
<code>argconstr</code>	a list of additional arguments of the constraint function.
<code>grconstr</code>	gradient of the constraint function, see details.
<code>arggrconstr</code>	a list of additional arguments of the constraint gradient.
<code>heconstr</code>	Hessian of the constraint function, see details.
<code>argheconstr</code>	a list of additional arguments of the constraint Hessian.
<code>dimmu</code>	a vector of dimension for $\mu$ .
<code>joint</code>	joint function ( $h(x) \leq 0$ ), see details.
<code>argjoint</code>	a list of additional arguments of the joint function.
<code>grjoint</code>	gradient of the joint function, see details.
<code>arggrjoint</code>	a list of additional arguments of the joint gradient.
<code>hejoint</code>	Hessian of the joint function, see details.
<code>arghejoint</code>	a list of additional arguments of the joint Hessian.
<code>method</code>	a character string specifying the method "PR" or "AS".
<code>control</code>	a list with control parameters.
<code>...</code>	further arguments to be passed to the optimization routine. NOT to the functions H and jach.
<code>silent</code>	a logical to get some traces. Default to FALSE.

## Details

GNE.ceq solves the GNE problem via a constrained equation reformulation of the KKT system.

This approach consists in solving the extended Karush-Kuhn-Tucker (KKT) system denoted by  $H(z) = 0$ , for  $z \in \Omega$  where eqnz is formed by the players strategy  $x$ , the Lagrange multiplier  $\lambda$  and the slate variable  $w$ . The root problem  $H(z) = 0$  is solved by an iterative scheme  $z_{n+1} = z_n + d_n$ , where the direction  $d_n$  is computed in two different ways. Let  $J(x) = JacH(x)$ . There are two possible methods either "PR" for potential reduction algorithm or "AS" for affine scaled trust reduction algorithm.

**(a) potential reduction algorithm:** The direction solves the system  $H(z_n) + J(z_n)d = \sigma a_n a^T H(z_n) / \|a\|_2^2 a$ .

**(b) bound-constrained trust region algorithm:** The direction solves the system  $\min_p \|J(z_n)^T p + H(z_n)\|^2$ , for  $p$  such that  $\|p\| \leq \Delta_n$ .

... are further arguments to be passed to the optimization routine, that is `global`, `xscalm`, `silent`. A globalization scheme can be choosed using the `global` argument. Available schemes are

**(1) Line search:** if `global` is set to "qline" or "gline", a line search is used with the merit function being half of the L2 norm of *Phi*, respectively with a quadratic or a geometric implementation.

**(3) Trust-region:** if `global` is set to "pwl dog", the Powell dogleg method is used.

**(2) None:** if `global` is set to "none", no globalization is done.

The default value of `global` is "gline" when `method="PR"` and "pwl dog" when `method="AS"`. The `xscalm` is a scaling parameter to used, either "fixed" (default) or "auto", for which scaling factors are calculated from the euclidean norms of the columns of the jacobian matrix. The `silent` argument is a logical to report or not the optimization process, default to FALSE.

The control argument is a list that can supply any of the following components:

`xtol` The relative steplength tolerance. When the relative steplength of all scaled  $x$  values is smaller than this value convergence is declared. The default value is  $10^{-8}$ .

`ftol` The function value tolerance. Convergence is declared when the largest absolute function value is smaller than `ftol`. The default value is  $10^{-8}$ .

`btol` The backtracking tolerance. The default value is  $10^{-2}$ .

`maxit` The maximum number of major iterations. The default value is 100 if a global strategy has been specified.

`trace` Non-negative integer. A value of 1 will give a detailed report of the progress of the iteration, default 0.

`sigma`, `delta`, `zeta` Parameters initialized to 1/2, 1,  $\text{length}(\text{init})/2$ , respectively, when `method="PR"`.

`forcingpar` Forcing parameter set to 0.1, when `method="PR"`.

`theta`, `radiusmin`, `reducmin`, `radiusmax`, `radiusred`, `reduced`, `radiusexp`, `reducexp` Parameters initialized to 0.99995, 1, 0.1,  $1e10$ , 1/2, 1/4, 2, 3/4, when `method="AS"`.

**Value**

GNE . ceq returns a list with components:

par The best set of parameters found.

value The value of the merit function.

counts A two-element integer vector giving the number of calls to H and jacH respectively.

iter The outer iteration number.

code The values returned are

- 1 Function criterion is near zero. Convergence of function values has been achieved.
- 2 x-values within tolerance. This means that the relative distance between two consecutive x-values is smaller than xtol.
- 3 No better point found. This means that the algorithm has stalled and cannot find an acceptable new point. This may or may not indicate acceptably small function values.
- 4 Iteration limit maxit exceeded.
- 5 Jacobian is too ill-conditioned.
- 6 Jacobian is singular.
- 100 an error in the execution.

message a string describing the termination code.

fvec a vector with function values.

**Author(s)**

Christophe Dutang

**References**

J.E. Dennis and J.J. Moré (1977), *Quasi-Newton methods, Motivation and Theory*, SIAM review.

Monteiro, R. and Pang, J.-S. (1999), *A Potential Reduction Newton Method for Constrained equations*, SIAM Journal on Optimization 9(3), 729-754.

S. Bellavia, M. Macconi and B. Morini (2003), *An affine scaling trust-region approach to bound-constrained nonlinear systems*, Applied Numerical Mathematics 44, 257-280

A. Dreves, F. Facchinei, C. Kanzow and S. Sagratella (2011), *On the solutions of the KKT conditions of generalized Nash equilibrium problems*, SIAM Journal on Optimization 21(3), 1082-1108.

**See Also**

See [GNE.fpeq](#), [GNE.minpb](#) and [GNE.nseq](#) for other approaches; [funcER](#) and [jacER](#) for template functions of  $H$  and  $JacH$ .

**Examples**

```
#-----
# (1) Example 5 of von Facchinei et al. (2007)
```



```

#-----

dimx <- c(1, 1)
#Gr_x_j O_i(x)
grobj <- function(x, i, j)
{
  if(i == 1)
    res <- c(2*(x[1]-1), 0)
  if(i == 2)
    res <- c(0, 2*(x[2]-1/2))
  res[j]
}
#Gr_x_k Gr_x_j O_i(x)
heobj <- function(x, i, j, k)
2 * (i == j && j == k)

dimlam <- c(1, 1)
#constraint function g_i(x)
g <- function(x, i)
sum(x[1:2]) - 1
#Gr_x_j g_i(x)
grg <- function(x, i, j)
1
#Gr_x_k Gr_x_j g_i(x)
heg <- function(x, i, j, k)
0

x0 <- rep(0, sum(dimx))
z0 <- c(x0, 2, 2, max(10, 5-g(x0, 1) ), max(10, 5-g(x0, 2) ) )

#true value is (3/4, 1/4, 1/2, 1/2)
GNE.ceq(z0, dimx, dimlam, grobj=grobj, heobj=heobj,
constr=g, grconstr=grg, heconstr=heg, method="PR",
control=list(trace=0, maxit=10))

GNE.ceq(z0, dimx, dimlam, grobj=grobj, heobj=heobj,
constr=g, grconstr=grg, heconstr=heg, method="AS", global="pwldog",
xscal="auto", control=list(trace=0, maxit=100))

#-----
# (2) Duopoly game of Krawczyk and Stanislav Uryasev (2000)
#-----

#constants
myarg <- list(d= 20, lambda= 4, rho= 1)

dimx <- c(1, 1)
#Gr_x_j O_i(x)
grobj <- function(x, i, j, arg)

```

```

{
res <- -arg$rho * x[i]
if(i == j)
res <- res + arg$d - arg$lambda - arg$rho*(x[1]+x[2])
-res
}
#Gr_x_k Gr_x_j O_i(x)
heobj <- function(x, i, j, k, arg)
arg$rho * (i == j) + arg$rho * (j == k)

dimlam <- c(1, 1)
#constraint function g_i(x)
g <- function(x, i)
-x[i]
#Gr_x_j g_i(x)
grg <- function(x, i, j)
-1*(i == j)
#Gr_x_k Gr_x_j g_i(x)
heg <- function(x, i, j, k)
0

#true value is (16/3, 16/3, 0, 0)

x0 <- rep(0, sum(dimx))
z0 <- c(x0, 2, 2, max(10, 5-g(x0, 1) ), max(10, 5-g(x0, 2) ) )

GNE.ceq(z0, dimx, dimlam, grobj=gobj, heobj=heobj, arggrobj=myarg,
argheobj=myarg, constr=g, grconstr=grg, heconstr=heg,
method="PR", control=list(trace=0, maxit=10))

GNE.ceq(z0, dimx, dimlam, grobj=gobj, heobj=heobj, arggrobj=myarg,
argheobj=myarg, constr=g, grconstr=grg, heconstr=heg,
method="AS", global="pwlDog", xscalm="auto", control=list(trace=0, maxit=100))

```

---

GNE.fpeq

*Fixed point equation reformulation of the GNE problem.*


---

### Description

Fixed point equation reformulation via the NI function of the GNE problem.

### Usage

```

GNE.fpeq(init, dimx, obj, argobj, grobj, arggrobj,
heobj, argheobj, joint, argjoint, jacjoint, argjacjoint,

```

```
method = "default", problem = c("NIR", "VIR"),
merit = c("NI", "VI", "FP"), order.method=1, control.outer=list(),
control.inner=list(), silent=TRUE, param=list(), stepfunc, argstep, ...)
```

### Arguments

<code>init</code>	Initial values for the parameters to be optimized over: $z = (x, \lambda, \mu)$ .
<code>dimx</code>	a vector of dimension for $x$ .
<code>obj</code>	objective function (to be minimized), see details.
<code>argobj</code>	a list of additional arguments.
<code>grobj</code>	gradient of the objective function, see details.
<code>arggrobj</code>	a list of additional arguments of the objective gradient.
<code>heobj</code>	Hessian of the objective function, see details.
<code>argheobj</code>	a list of additional arguments of the objective Hessian.
<code>joint</code>	joint function ( $h(x) \leq 0$ ), see details.
<code>argjoint</code>	a list of additional arguments of the joint function.
<code>jacjoint</code>	Jacobian of the joint function, see details.
<code>argjacjoint</code>	a list of additional arguments of the Jacobian.
<code>method</code>	either "pure", "UR", "vH", "RRE", "MPE", "SqRRE" or "SqMPE" method, see details. "default" corresponds to "MPE".
<code>problem</code>	either "NIR", "VIP", see details.
<code>merit</code>	either "NI", "VI", "FP", see details.
<code>order.method</code>	the order of the extrapolation method.
<code>control.outer</code>	a list with control parameters for the fixed point algorithm.
<code>control.inner</code>	a list with control parameters for the fixed point function.
<code>silent</code>	a logical to show some traces.
<code>param</code>	a list of parameters for the computation of the fixed point function.
<code>stepfunc</code>	the step function, only needed when method="UR".
<code>argstep</code>	additional arguments for the step function.
<code>...</code>	further arguments to be passed to the optimization routine. NOT to the functions.

### Details

Functions in argument must respect the following template:

- `obj` must have arguments the current iterate  $z$ , the player number  $i$  and optionally additional arguments given in a list.
- `grobj` must have arguments the current iterate  $z$ , the player number  $i$ , the derivative index  $j$  and optionally additional arguments given in a list.
- `heobj` must have arguments the current iterate  $z$ , the player number  $i$ , the derivative indexes  $j, k$  and optionally additional arguments given in a list.

- joint must have arguments the current iterate  $z$  and optionnally additional arguments given in a list.
- jacjoint must have arguments the current iterate  $z$ , the derivative index  $j$  and optionnally additional arguments given in a list.

The fixed point approach consists in solving equation  $y(x) = x$ .

**(a) Crude or pure fixed point method:** It simply consists in iterations  $x_{n+1} = y(x_n)$ .

**(b) Polynomial methods: - relaxation algorithm (linear extrapolation):** The next iterate is computed as

$$x_{n+1} = (1 - \alpha_n)x_n + \alpha_n y(x_n).$$

The step  $\alpha_n$  can be computed in different ways: constant, decreasing serie or a line search method. In the literature of game theory, the decreasing serie refers to the method of Ursayev and Rubinstein (method="UR") while the line search method refers to the method of von Heusinger (method="vH"). Note that the constant step can be done using the UR method.

- **RRE and MPE method:** Reduced Rank Extrapolation and Minimal Polynomial Extrapolation methods are polynomial extrapolation methods, where the monomials are functional "powers" of the  $y$  function, i.e. function composition of  $y$ . Of order 1, RRE and MPE consists of

$$x_{n+1} = x_n + t_n(y(x_n) - x_n),$$

where  $t_n$  equals to  $\langle v_n, r_n \rangle / \langle v_n, v_n \rangle$  for RRE1 and  $\langle r_n, r_n \rangle / \langle v_n, r_n \rangle$  for MPE1, where  $r_n = y(x_n) - x_n$  and  $v_n = y(y(x_n)) - 2y(x_n) + x_n$ . To use RRE/MPE methods, set method = "RRE" or method = "MPE".

- **squaring method:** It consists in using an extrapolation method (such as RRE and MPE) after two iteration of the linear extrapolation, i.e.

$$x_{n+1} = x_n - 2t_n r_n + t_n^2 v_n.$$

The squared version of RRE/MPE methods are available via setting method = "SqRRE" or method = "SqMPE".

**(c) Epsilon algorithms:** Not implemented.

For details on fixed point methods, see Varadhan & Roland (2004).

The control.outer argument is a list that can supply any of the following components:

merit="FP" **and** method="pure" see [fpiter](#). the default parameters are list(tol=1e-6, maxiter=100, trace=TRUE).

merit="FP" **and** method!="pure" see [squarem](#). the default parameters are list(tol=1e-6, maxiter=100, trace=TRUE).

merit!="FP" parameters are

tol The absolute convergence tolerance. Default to 1e-6.

maxit The maximum number of iterations. Default to 100.

echo A logical or an integer (0, 1, 2, 3) to print traces. Default to FALSE, i.e. 0.

sigma, beta parameters for von Heusinger algorithm. Default to 9/10 and 1/2 respectively.

**Value**

A list with components:

`par` The best set of parameters found.

`value` The value of the merit function.

`outer.counts` A two-element integer vector giving the number of calls to fixed-point and merit functions respectively.

`outer.iter` The outer iteration number.

`code` The values returned are

1 Function criterion is near zero. Convergence of function values has been achieved.

4 Iteration limit `maxit` exceeded.

100 an error in the execution.

`inner.iter` The iteration number when computing the fixed-point function.

`inner.counts` A two-element integer vector giving the number of calls to the gap function and its gradient when computing the fixed-point function.

`message` a string describing the termination code

**Author(s)**

Christophe Dutang

**References**

A. von Heusinger (2009), *Numerical Methods for the Solution of the Generalized Nash Equilibrium Problem*, Ph. D. Thesis.

A. von Heusinger and C. Kanzow (2009), *Optimization reformulations of the generalized Nash equilibrium problem using Nikaido-Isoda-type functions*, *Comput Optim Appl* .

S. Uryasev and R.Y. Rubinstein (1994), *On relaxation algorithms in computation of noncooperative equilibria*, *IEEE Transactions on Automatic Control*.

R. Varadhan and C. Roland (2004), *Squared Extrapolation Methods (SQUAREM): A New Class of Simple and Efficient Numerical Schemes for Accelerating the Convergence of the EM Algorithm*, Johns Hopkins University, Dept. of Biostatistics Working Papers.

**See Also**

See [GNE.ceq](#), [GNE.minpb](#) and [GNE.nseq](#) for other approaches.

---

GNE.minpb

*Non smooth equation reformulation of the GNE problem.*


---

### Description

Non smooth equation reformulation via the extended KKT system of the GNE problem.

### Usage

```
GNE.minpb(init, dimx, obj, argobj, grobj, arggrobj,
heobj, argheobj, joint, argjoint, jacjoint, argjacjoint,
method="default", problem = c("NIR", "VIP"), control.outer=list(),
control.inner=list(), silent=TRUE, param=list(),
optim.type=c("free", "constr"), ...)
```

### Arguments

<code>init</code>	Initial values for the parameters to be optimized over: $z = (x, \lambda, \mu)$ .
<code>dimx</code>	a vector of dimension for $x$ .
<code>obj</code>	objective function (to be minimized), see details.
<code>argobj</code>	a list of additional arguments.
<code>grobj</code>	gradient of the objective function, see details.
<code>arggrobj</code>	a list of additional arguments of the objective gradient.
<code>heobj</code>	Hessian of the objective function, see details.
<code>argheobj</code>	a list of additional arguments of the objective Hessian.
<code>joint</code>	joint function ( $h(x) \leq 0$ ), see details.
<code>argjoint</code>	a list of additional arguments of the joint function.
<code>jacjoint</code>	Jacobian of the joint function, see details.
<code>argjacjoint</code>	a list of additional arguments of the Jacobian.
<code>method</code>	either "BB", "CG" or "BFGS", see details.
<code>problem</code>	either "NIR", "VIP", see details.
<code>optim.type</code>	either "free", "constr", see details.
<code>control.outer</code>	a list with control parameters for the minimization algorithm.
<code>control.inner</code>	a list with control parameters for the minimization function.
<code>...</code>	further arguments to be passed to the optimization routine. NOT to the functions <code>phi</code> and <code>jacphi</code> .
<code>silent</code>	a logical to show some traces.
<code>param</code>	a list of parameters for the computation of the minimization function.

## Details

Functions in argument must respect the following template:

- `obj` must have arguments the current iterate  $z$ , the player number  $i$  and optionnally additional arguments given in a list.
- `grobj` must have arguments the current iterate  $z$ , the player number  $i$ , the derivative index  $j$  and optionnally additional arguments given in a list.
- `heobj` must have arguments the current iterate  $z$ , the player number  $i$ , the derivative indexes  $j, k$  and optionnally additional arguments given in a list.
- `joint` must have arguments the current iterate  $z$  and optionnally additional arguments given in a list.
- `jacjoint` must have arguments the current iterate  $z$ , the derivative index  $j$  and optionnally additional arguments given in a list.

The gap function minimization consists in minimizing a gap function  $\min V(x)$ . The function `minGap` provides two optimization methods to solve this minimization problem.

**Barzilai-Borwein algorithm** when `method = "BB"`, we use Barzilai-Borwein iterative scheme to find the minimum.

**Conjugate gradient algorithm** when `method = "CG"`, we use the CG iterative scheme implemented in R, an Hessian-free method.

**Broyden-Fletcher-Goldfarb-Shanno algorithm** when `method = "BFGS"`, we use the BFGS iterative scheme implemented in R, a quasi-Newton method with line search.

In the game theory literature, there are two main gap functions: the regularized Nikaido-Isoda (NI) function and the regularized QVI gap function. This correspond to `type="NI"` and `type="VI"`, respectively. See von Heusinger & Kanzow (2009) for details on the NI function and Kubota & Fukushima (2009) for the QVI regularized gap function.

The `control.outer` argument is a list that can supply any of the following components:

`tol` The absolute convergence tolerance. Default to  $1e-6$ .

`maxit` The maximum number of iterations. Default to 100.

`echo` A logical or an integer (0, 1, 2, 3) to print traces. Default to FALSE, i.e. 0.

`stepinit` Initial step size for the BB method (should be small if gradient is "big"). Default to 1.

Note that the Gap function can return a numeric or a list with computation details. In the latter case, the object return must be a list with the following components `value`, `counts`, `iter`, see the example below.

## Value

A list with components:

`par` The best set of parameters found.

`value` The value of the merit function.

`outer.counts` A two-element integer vector giving the number of calls to `Gap` and `gradGap` respectively.

`outer.iter` The outer iteration number.

`code` The values returned are

- 1 Function criterion is near zero. Convergence of function values has been achieved.
- 2 x-values within tolerance. This means that the relative distance between two consecutive x-values is smaller than `xtol`.
- 3 No better point found. This means that the algorithm has stalled and cannot find an acceptable new point. This may or may not indicate acceptably small function values.
- 4 Iteration limit `maxit` exceeded.
- 5 Jacobian is too ill-conditioned.
- 6 Jacobian is singular.
- 100 an error in the execution.

`inner.iter` The iteration number when computing the minimization function.

`inner.counts` A two-element integer vector giving the number of calls to the gap function and its gradient when computing the minimization function.

`message` a string describing the termination code

### Author(s)

Christophe Dutang

### References

A. von Heusinger (2009), *Numerical Methods for the Solution of the Generalized Nash Equilibrium Problem*, Ph. D. Thesis.

A. von Heusinger and C. Kanzow (2009), *Optimization reformulations of the generalized Nash equilibrium problem using Nikaido-Isoda-type functions*, *Comput Optim Appl* .

K. Kubota and M. Fukushima (2009), *Gap function approach to the generalized Nash Equilibrium problem*, *Journal of Optimization theory and applications*.

### See Also

See [GNE.fpeq](#), [GNE.ceq](#) and [GNE.nseq](#) for other approaches.

---

GNE.nseq

*Non smooth equation reformulation of the GNE problem.*

---

### Description

Non smooth equation reformulation via the extended KKT system of the GNE problem.

### Usage

```
GNE.nseq(init, dimx, dimlam, grobj, arggrobj, heobj, argheobj,
constr, argconstr, grconstr, arggrconstr, heconstr, argheconstr,
compl, gcompla, gcomplb, argcompl,
dimmu, joint, argjoint, grjoint, arggrjoint, hejoint, arghejoint,
method="default", control=list(), silent=TRUE, ...)
```



**Arguments**

init	Initial values for the parameters to be optimized over: $z = (x, \lambda, \mu)$ .
dimx	a vector of dimension for $x$ .
dimlam	a vector of dimension for $\lambda$ .
grobj	gradient of the objective function (to be minimized), see details.
arggrobj	a list of additional arguments of the objective gradient.
heobj	Hessian of the objective function, see details.
argheobj	a list of additional arguments of the objective Hessian.
constr	constraint function ( $g^i(x) \leq 0$ ), see details.
argconstr	a list of additional arguments of the constraint function.
grconstr	gradient of the constraint function, see details.
arggrconstr	a list of additional arguments of the constraint gradient.
heconstr	Hessian of the constraint function, see details.
argheconstr	a list of additional arguments of the constraint Hessian.
compl	the complementarity function with (at least) two arguments: $\text{compl}(a, b)$ .
argcompl	list of possible additional arguments for $\text{compl}$ .
gcompla	derivative of the complementarity function w.r.t. the first argument.
gcomplb	derivative of the complementarity function w.r.t. the second argument.
dimmu	a vector of dimension for $\mu$ .
joint	joint function ( $h(x) \leq 0$ ), see details.
argjoint	a list of additional arguments of the joint function.
grjoint	gradient of the joint function, see details.
arggrjoint	a list of additional arguments of the joint gradient.
hejoint	Hessian of the joint function, see details.
arghejoint	a list of additional arguments of the joint Hessian.
method	a character string specifying the method "Newton", "Broyden", "Levenberg-Marquardt" or "default" which is "Newton".
control	a list with control parameters.
...	further arguments to be passed to the optimization routine. NOT to the functions $\text{phi}$ and $\text{jacphi}$ .
silent	a logical to get some traces. Default to FALSE.

**Details**

Functions in argument must respect the following template:

- `constr` must have arguments the current iterate  $z$ , the player number  $i$  and optionally additional arguments given in a list.
- `grobj`, `grconstr` must have arguments the current iterate  $z$ , the player number  $i$ , the derivative index  $j$  and optionally additional arguments given in a list.

- `heobj`, `heconstr` must have arguments the current iterate  $z$ , the player number  $i$ , the derivative indexes  $j$ ,  $k$  and optionnally additional arguments given in a list.
- `compl`, `gcompl`, `gcomplb` must have two arguments  $a$ ,  $b$  and optionnally additional arguments given in a list.
- `joint` must have arguments the current iterate  $z$  and optionnally additional arguments given in a list.
- `grjoint` must have arguments the current iterate  $z$ , the derivative index  $j$  and optionnally additional arguments given in a list.
- `hejoint` must have arguments the current iterate  $z$ , the derivative indexes  $j$ ,  $k$  and optionnally additional arguments given in a list.

`GNE.nseq` solves the GNE problem via a non smooth reformulation of the KKT system. `bench.GNE.nseq` carries out a benchmark of the computation methods (Newton and Broyden direction with all possible global schemes) for a given initial point. `bench.GNE.nseq.LM` carries out a benchmark of the Levenberg-Marquardt computation method.

This approach consists in solving the extended Karush-Kuhn-Tucker (KKT) system denoted by  $\Phi(z) = 0$ , where  $eqnz$  is formed by the players strategy  $x$  and the Lagrange multiplier  $\lambda$ . The root problem  $\Phi(z) = 0$  is solved by an iterative scheme  $z_{n+1} = z_n + d_n$ , where the direction  $d_n$  is computed in three different ways. Let  $J(x) = Jac\Phi(x)$ .

- (a) **Newton:** The direction solves the system  $J(z_n)d = -\Phi(z_n)$ , generally called the Newton equation.
- (b) **Broyden:** It is a quasi-Newton method aiming to solve an approximate version of the Newton equation  $d = -\Phi(z_n)W_n$  where  $W_n$  is computed by an iterative scheme. In the current implementation,  $W_n$  is updated by the Broyden method.
- (c) **Levenberg-Marquardt:** The direction solves the system

$$[J(z_n)^T J(z_n) + \lambda_n^\delta I] d = -J(z_n)^T \Phi(z_n)$$

where  $I$  denotes the identity matrix,  $\delta$  is a parameter in  $[1,2]$  and  $\lambda_n = \|\Phi(z_n)\|$  if `LM.param="merit"`,  $\|J(z_n)^T \Phi(z_n)\|$  if `LM.param="jacmerit"`, the minimum of both preceding quantities if `LM.param="min"`, or an adaptive parameter according to Fan(2003) if `LM.param="adaptive"`.

In addition to the computation method, a globalization scheme can be choosed using the `global` argument, via the `...` argument. Available schemes are

- (1) **Line search:** if `global` is set to `"qline"` or `"gline"`, a line search is used with the merit function being half of the L2 norm of `Phi`, respectively with a quadratic or a geometric implementation.
- (2) **Trust region:** if `global` is set to `"dbl dog"` or `"pwl dog"`, a trust region is used respectively with a double dogleg or a Powell (simple) dogleg implementation. This global scheme is not available for the Levenberg-Marquardt direction.
- (3) **None:** if `global` is set to `"none"`, no globalization is done.

The default value of `global` is `"gline"`. Note that in the special case of the Levenberg-Marquardt direction with adaptive parameter, the global scheme must be `"none"`.

In the GNEP context, details on the methods can be found in Facchinei, Fischer & Piccialli (2009), "Newton" corresponds to method 1 and "Levenberg-Marquardt" to method 3. In a general non-linear equation framework, see Dennis & Moree (1977), Dennis & Schnabel (1996) or Nocedal & Wright (2006),

The implementation relies heavily on the `nleqslv` function of the package of the same name. So full details on the control parameters are to be found in the help page of this function. We briefly recall here the main parameters. The `control` argument is a list that can supply any of the following components:

`xtol` The relative steplength tolerance. When the relative steplength of all scaled `x` values is smaller than this value convergence is declared. The default value is  $10^{-8}$ .

`ftol` The function value tolerance. Convergence is declared when the largest absolute function value is smaller than `ftol`. The default value is  $10^{-8}$ .

`delta` A numeric `delta` in  $[1, 2]$ , default to 2, for the Levenberg-Marquardt method only.

`LM.param` A character string, default to "merit", for the Levenberg-Marquardt method only.

`maxit` The maximum number of major iterations. The default value is 150 if a global strategy has been specified.

`trace` Non-negative integer. A value of 1 will give a detailed report of the progress of the iteration, default 0.

... are further arguments to be passed to the optimization routine, that is `global`, `xscal`, `silent`. See above for the globalization scheme. The `xscal` is a scaling parameter to used, either "fixed" (default) or "auto", for which scaling factors are calculated from the euclidean norms of the columns of the jacobian matrix. See `nleqslv` for details. The `silent` argument is a logical to report or not the optimization process, default to FALSE.

## Value

`GNE.nseq` returns a list with components:

`par` The best set of parameters found.

`value` The value of the merit function.

`counts` A two-element integer vector giving the number of calls to `phi` and `jacphi` respectively.

`iter` The outer iteration number.

`code` The values returned are

- 1 Function criterion is near zero. Convergence of function values has been achieved.
- 2 `x`-values within tolerance. This means that the relative distance between two consecutive `x`-values is smaller than `xtol`.
- 3 No better point found. This means that the algorithm has stalled and cannot find an acceptable new point. This may or may not indicate acceptably small function values.
- 4 Iteration limit `maxit` exceeded.
- 5 Jacobian is too ill-conditioned.
- 6 Jacobian is singular.
- 100 an error in the execution.

`message` a string describing the termination code.

fvec a vector with function values.

bench.GNE.nseq returns a list with components:

compres a data.frame summarizing the different computations.

reslist a list with the different results from GNE.nseq.

### Author(s)

Christophe Dutang

### References

J.E. Dennis and J.J. Moree (1977), *Quasi-Newton methods, Motivation and Theory*, SIAM review.

J.E. Dennis and R.B. Schnabel (1996), *Numerical methods for unconstrained optimization and nonlinear equations*, SIAM.

F. Facchinei, A. Fischer and V. Piccialli (2009), *Generalized Nash equilibrium problems and Newton methods*, Math. Program.

J.-Y. Fan (2003), *A modified Levenberg-Marquardt algorithm for singular system of nonlinear equations*, Journal of Computational Mathematics.

B. Hasselman (2011), *nleqslv: Solve systems of non linear equations*, R package.

A. von Heusinger and C. Kanzow (2009), *Optimization reformulations of the generalized Nash equilibrium problem using Nikaido-Isoda-type functions*, Comput Optim Appl .

J. Nocedal and S.J. Wright (2006), *Numerical Optimization*, Springer Science+Business Media

### See Also

See [GNE.fpeq](#), [GNE.ceq](#) and [GNE.minpb](#) for other approaches; [funSSR](#) and [jacSSR](#) for template functions of  $\Phi$  and  $Jac\Phi$  and [complementarity](#) for complementarity functions.

See also [nleqslv](#) for some optimization details.

### Examples

```
#-----
# (1) Example 5 of von Facchinei et al. (2007)
#-----

dimx <- c(1, 1)
#Gr_x_j 0_i(x)
grobj <- function(x, i, j)
{
  if(i == 1)
    res <- c(2*(x[1]-1), 0)
  if(i == 2)
    res <- c(0, 2*(x[2]-1/2))
  res[j]
}
```

```

#Gr_x_k Gr_x_j O_i(x)
heobj <- function(x, i, j, k)
2 * (i == j && j == k)

dimlam <- c(1, 1)
#constraint function g_i(x)
g <- function(x, i)
sum(x[1:2]) - 1
#Gr_x_j g_i(x)
grg <- function(x, i, j)
1
#Gr_x_k Gr_x_j g_i(x)
heg <- function(x, i, j, k)
0

#true value is (3/4, 1/4, 1/2, 1/2)

z0 <- rep(0, sum(dimx)+sum(dimlam))

funSSR(z0, dimx, dimlam, grobj=grobj, constr=g, grconstr=grg, compl=phiFB, echo=FALSE)

jacSSR(z0, dimx, dimlam, heobj=heobj, constr=g, grconstr=grg,
heconstr=heg, gcompla=GrAphiFB, gcomplb=GrBphiFB)

GNE.nseq(z0, dimx, dimlam, grobj=grobj, NULL, heobj=heobj, NULL,
constr=g, NULL, grconstr=grg, NULL, heconstr=heg, NULL,
compl=phiFB, gcompla=GrAphiFB, gcomplb=GrBphiFB, method="Newton",
control=list(trace=1))

GNE.nseq(z0, dimx, dimlam, grobj=grobj, NULL, heobj=heobj, NULL,
constr=g, NULL, grconstr=grg, NULL, heconstr=heg, NULL,
compl=phiFB, gcompla=GrAphiFB, gcomplb=GrBphiFB, method="Broyden",
control=list(trace=1))

#-----
# (2) Duopoly game of Krawczyk and Stanislav Uryasev (2000)
#-----

#constants
myarg <- list(d= 20, lambda= 4, rho= 1)

dimx <- c(1, 1)
#Gr_x_j O_i(x)
grobj <- function(x, i, j, arg)
{
res <- -arg$rho * x[i]

```

```

if(i == j)
res <- res + arg$d - arg$lambda - arg$rho*(x[1]+x[2])
-res
}
#Gr_x_k Gr_x_j O_i(x)
heobj <- function(x, i, j, k, arg)
arg$rho * (i == j) + arg$rho * (j == k)

dimlam <- c(1, 1)
#constraint function g_i(x)
g <- function(x, i)
-x[i]
#Gr_x_j g_i(x)
grg <- function(x, i, j)
-1*(i == j)
#Gr_x_k Gr_x_j g_i(x)
heg <- function(x, i, j, k)
0

#true value is (16/3, 16/3, 0, 0)

z0 <- rep(0, sum(dimx)+sum(dimlam))

funSSR(z0, dimx, dimlam, grobj=grobj, myarg, constr=g, grconstr=grg, compl=phiFB, echo=FALSE)

jacSSR(z0, dimx, dimlam, heobj=heobj, myarg, constr=g, grconstr=grg,
heconstr=heg, gcompla=GrAphiFB, gcomplb=GrBphiFB)

GNE.nseq(z0, dimx, dimlam, grobj=grobj, myarg, heobj=heobj, myarg,
constr=g, NULL, grconstr=grg, NULL, heconstr=heg, NULL,
compl=phiFB, gcompla=GrAphiFB, gcomplb=GrBphiFB, method="Newton",
control=list(trace=1))

GNE.nseq(z0, dimx, dimlam, grobj=grobj, myarg, heobj=heobj, myarg,
constr=g, NULL, grconstr=grg, NULL, heconstr=heg, NULL,
compl=phiFB, gcompla=GrAphiFB, gcomplb=GrBphiFB, method="Broyden",
control=list(trace=1))

```

## Description

functions of the Nikaido Isoda Reformulation of the GNEP

**Usage**

```
gapNIR(x, y, dimx, obj, argobj, param=list(), echo=FALSE)
gradxgapNIR(x, y, dimx, grobj, arggrobj, param=list(), echo=FALSE)
gradygapNIR(x, y, dimx, grobj, arggrobj, param=list(), echo=FALSE)
fpNIR(x, dimx, obj, argobj, joint, argjoint,
      grobj, arggrobj, jacjoint, argjacjoint, param=list(),
      echo=FALSE, control=list(), yinit=NULL, optim.method="default")
```

**Arguments**

<code>x,y</code>	a numeric vector.
<code>dimx</code>	a vector of dimension for <code>x</code> .
<code>obj</code>	objective function (to be minimized), see details.
<code>argobj</code>	a list of additional arguments.
<code>grobj</code>	gradient of the objective function, see details.
<code>arggrobj</code>	a list of additional arguments of the objective gradient.
<code>joint</code>	joint function, see details.
<code>argjoint</code>	a list of additional arguments of the joint function.
<code>jacjoint</code>	gradient of the joint function, see details.
<code>argjacjoint</code>	a list of additional arguments of the joint Jacobian.
<code>param</code>	a list of parameters.
<code>control</code>	a list with control parameters for the fixed point algorithm.
<code>yinit</code>	initial point when computing the fixed-point function.
<code>optim.method</code>	optimization method when computing the fixed-point function.
<code>echo</code>	a logical to show some traces.

**Details**

`gapNIR` computes the Nikaido Isoda function of the GNEP, while `gradxgapNIR` and `gradygapNIR` give its gradient with respect to  $x$  and  $y$ . `fpNIR` computes the fixed-point function.

**Value**

A vector for `funSSR` or a matrix for `jacSSR`.

**Author(s)**

Christophe Dutang

**References**

A. von Heusinger & J. Kanzow (2009), *Optimization reformulations of the generalized Nash equilibrium problem using Nikaido-Isoda-type functions*, Comput Optim Appl .

F. Facchinei, A. Fischer and V. Piccialli (2009), *Generalized Nash equilibrium problems and Newton methods*, Math. Program.

**See Also**

See also [GNE.fpeq](#).

---

potential.reduction    *Potential reduction algorithm utility functions*

---

**Description**

Functions for the potential reduction algorithm

**Usage**

potential.ce(u, n, zeta)

gradpotential.ce(u, n, zeta)

psi.ce(z, dimx, dimlam, Hfinal, argfun, zeta)

gradpsi.ce(z, dimx, dimlam, Hfinal, jacHfinal, argfun, argjac, zeta)

**Arguments**

u	a numeric vector : $u = (u_1, u_2)$ where $u_1$ is of size n.
n	a numeric for the size of $u_1$ .
zeta	a positive parameter.
z	a numeric vector : $z = (x, lambda, w)$ where dimx is the size of components of $x$ and dimlam is the size of components of $lambda$ and $w$ .
dimx	a numeric vector with the size of each components of $x$ .
dimlam	a numeric vector with the size of each components of $lambda$ . We must have $\text{length}(\text{dimx}) == \text{length}(\text{dimlam})$ .
Hfinal	the root function.
argfun	a list of additionnals arguments for Hfinal.
jacHfinal	the Jacobian of the root function.
argjac	a list of additionnals arguments for jacHfinal.



**Details**

potential . ce is the potential function for the GNEP, and gradpotential . ce its gradient. psi . ce is the application of the potential function for Hfinal, and gradpsi . ce its gradient.

**Value**

A numeric or a numeric vector.

**Author(s)**

Christophe Dutang

**References**

S. Bellavia, M. Macconi, B. Morini (2003), *An affine scaling trust-region approach to bound-constrained nonlinear systems*, Applied Numerical Mathematics 44, 257-280

A. Dreves, F. Facchinei, C. Kanzow and S. Sagratella (2011), *On the solutions of the KKT conditions of generalized Nash equilibrium problems*, SIAM Journal on Optimization 21(3), 1082-1108.

**See Also**

See also [GNE . ceq](#).

---

projector

*Projection of a point on a set*

---

**Description**

Projection of a point  $z$  on the set defined by the constraints  $g(x) \leq 0$ .

**Usage**

```
projector(z, g, jacg, bounds=c(0, 10), echo=FALSE, ...)
```

**Arguments**

<code>z</code>	The point to project.
<code>g</code>	The constraint function.
<code>jacg</code>	The jacobian of the constraint function.
<code>bounds</code>	bounds for the randomized initial iterate.
<code>echo</code>	a logical to plot traces.
<code>...</code>	further arguments to pass to g function.

**Details**

Find a point  $x$  in the set  $K$  which minimizes the Euclidean distance  $\|z - x\|^2$ , where the set  $K$  is  $x, g(x) \leq 0$ . The Optimization is carried out by the `constrOptim.nl` function of the package `alabama`.

**Value**

A vector  $x$ .

**Author(s)**

Christophe Dutang

**See Also**

See also [GNE](#).

**Examples**

```
# 1. the rectangle set
#

g <- function(x)
c(x - 3, 1 - x)

jacg <- function(x)
rbind(
diag( rep(1, length(x)) ),
diag( rep(-1, length(x)) )
)

z <- runif(2, 3, 4)

#computation
projz <- projector(z, g, jacg)

#plot
plot(c(1, 3), c(1, 1), xlim=c(0, 4), ylim=c(0,4), type="l", col="blue")
lines(c(3, 3), c(1, 3), col="blue")
lines(c(3, 1), c(3, 3), col="blue")
lines(c(1, 1), c(3, 1), col="blue")

points(z[1], z[2], col="red")
points(projz[1], projz[2], col="red", pch="+")

z <- runif(2) + c(1, 0)
projz <- projector(z, g, jacg)

points(z[1], z[2], col="green")
points(projz[1], projz[2], col="green", pch="+")
```

```

# 2. the circle set
#

g <- function(x) sum((x-2)^2)-1
jacg <- function(x) as.matrix( 2*(x-2) )

z <- runif(2) + c(1, 0)

#computation
projz <- projector(z, g, jacg)

#plot
plot(c(1, 3), c(1, 1), xlim=c(0, 4), ylim=c(0,4), type="n", col="blue")
symbols(2, 2, circles=1, fg="blue", add=TRUE, inches=FALSE)

points(z[1], z[2], col="red")
points(projz[1], projz[2], col="red", pch="+")

z <- c(runif(1, 3, 4), runif(1, 1, 2))
projz <- projector(z, g, jacg)

points(z[1], z[2], col="green")
points(projz[1], projz[2], col="green", pch="+")

```

---

rejection

*Rejection method for random generation.*


---

### Description

Generate random variate satisfying the constraint function by the Rejection algorithm.

### Usage

```
rejection(constr, nvars, LB=0, UB=1, ..., echo=FALSE,
method=c("unif", "norm", "normcap"), control=list())
```

### Arguments

constr	Constraint function
nvars	Number of variables
LB	Lower bound
UB	Upper bound
...	further arguments to pass to constr function.

echo            a logical to plot traces.  
 method        the distribution to draw random variates, either "unif", "norm", "normcap".  
 control        a named list containing the mean and the standard deviation of the normal distribution used if method!="unif".

### Details

Draw random variates  $x$  until all the components of  $\text{constr}(x)$  are negative. The distribution to draw random variates can be the uniform distribution on the hypercube defined by LB and UB, the normal distribution centered in  $(LB + UB)/2$  and standard deviation  $(UB - LB) / (4 \times 1.9600)$  and the capped normal distribution (intended for debug use).

### Value

A vector  $x$  which verifies the constraints  $\text{constr}(x) \leq 0$ .

### Author(s)

Christophe Dutang

### See Also

See also [GNE](#).

### Examples

```
f <- function(x) x[1]^2 + x[2]^2 - 1
rejection(f, 2, -3, 3, method="unif")
rejection(f, 2, -3, 3, method="norm")
```

---

 SSR

*SemiSmooth Reformulation*


---

### Description

functions of the SemiSmooth Reformulation of the GNEP

### Usage

```
funSSR(z, dimx, dimlam, grobj, arggrobj, constr, argconstr, grconstr, arggrconstr,
compl, argcompl, dimmu, joint, argjoint, grjoint, arggrjoint, echo=FALSE)
jacSSR(z, dimx, dimlam, heobj, argheobj, constr, argconstr, grconstr, arggrconstr,
heconstr, argheconstr, gcompla, gcomplb, argcompl, dimmu, joint, argjoint,
grjoint, arggrjoint, hejoint, arghejoint, echo=FALSE)
```

**Arguments**

<code>z</code>	a numeric vector <code>z</code> containing $(x, \lambda, \mu)$ values.
<code>dimx</code>	a vector of dimension for $x$ .
<code>dimlam</code>	a vector of dimension for $\lambda$ .
<code>grobj</code>	gradient of the objective function, see details.
<code>arggrobj</code>	a list of additional arguments of the objective gradient.
<code>constr</code>	constraint function, see details.
<code>argconstr</code>	a list of additional arguments of the constraint function.
<code>grconstr</code>	gradient of the constraint function, see details.
<code>arggrconstr</code>	a list of additional arguments of the constraint gradient.
<code>compl</code>	the complementarity function with (at least) two arguments: <code>compl(a, b)</code> .
<code>argcompl</code>	list of possible additional arguments for <code>compl</code> .
<code>dimmu</code>	a vector of dimension for $\mu$ .
<code>joint</code>	joint function, see details.
<code>argjoint</code>	a list of additional arguments of the joint function.
<code>grjoint</code>	gradient of the joint function, see details.
<code>arggrjoint</code>	a list of additional arguments of the joint gradient.
<code>heobj</code>	Hessian of the objective function, see details.
<code>argheobj</code>	a list of additional arguments of the objective Hessian.
<code>heconstr</code>	Hessian of the constraint function, see details.
<code>argheconstr</code>	a list of additional arguments of the constraint Hessian.
<code>gcompla</code>	derivative of the complementarity function w.r.t. the first argument.
<code>gcomplb</code>	derivative of the complementarity function w.r.t. the second argument.
<code>hejoint</code>	Hessian of the joint function, see details.
<code>arghejoint</code>	a list of additional arguments of the joint Hessian.
<code>echo</code>	a logical to show some traces.

**Details**

Compute the SemiSmooth Reformulation of the GNEP: the Generalized Nash equilibrium problem is defined by objective functions  $Obj$  with player variables  $x$  defined in `dimx` and may have player-dependent constraint functions  $g$  of dimension `dimlam` and/or a common shared joint function  $h$  of dimension `dimmu`, where the Lagrange multiplier are  $\lambda$  and  $\mu$ , respectively, see F. Facchinei et al.(2009) where there is no joint function.

**Arguments of the Phi function** The arguments which are functions must respect the following features

`grobj` The gradient  $GradObj$  of an objective function  $Obj$  (to be minimized) must have 3 arguments for  $GradObj(z, playnum, ideriv)$ : vector `z`, player number, derivative index, and optionally additional arguments in `arggrobj`.

**constr** The constraint function  $g$  must have 2 arguments: vector  $z$ , player number, such that  $g(z, \text{playnum}) \leq 0$ . Optionnally,  $g$  may have additional arguments in `argconstr`.

**grconstr** The gradient of the constraint function  $g$  must have 3 arguments: vector  $z$ , player number, derivative index, and optionnally additional arguments in `arggrconstr`.

**compl** It must have two arguments and optionnally additional arguments in `argcompl`. A typical example is the minimum function.

**joint** The constraint function  $h$  must have 1 argument: vector  $z$ , such that  $h(z) \leq 0$ . Optionnally,  $h$  may have additional arguments in `argjoint`.

**grjoint** The gradient of the constraint function  $h$  must have 2 arguments: vector  $z$ , derivative index, and optionnally additional arguments in `arggrjoint`.

**Arguments of the Jacobian of Phi** The arguments which are functions must respect the following features

**heobj** It must have 4 arguments: vector  $z$ , player number, two derivative indexes and optionnally additional arguments in `argheobj`.

**heconstr** It must have 4 arguments: vector  $z$ , player number, two derivative indexes and optionnally additional arguments in `argheconstr`.

**gcompla,gcomplb** It must have two arguments and optionnally additional arguments in `argcompl`.

**hejoint** It must have 3 arguments: vector  $z$ , two derivative indexes and optionnally additional arguments in `arghejoint`.

See the example below.

## Value

A vector for `funSSR` or a matrix for `jacSSR`.

## Author(s)

Christophe Dutang

## References

F. Facchinei, A. Fischer and V. Piccialli (2009), *Generalized Nash equilibrium problems and Newton methods*, Math. Program.

## See Also

See also [GNE.nseq](#).

## Examples

```
# (1) associated objective functions
#
dimx <- c(2, 2, 3)
#Gr_x_j O_i(x)
grfullob <- function(x, i, j)
```

```

{
x <- x[1:7]
if(i == 1)
{
grad <- 3*(x - 1:7)^2
}
if(i == 2)
{
grad <- 1:7*(x - 1:7)^(0:6)
}
if(i == 3)
{
s <- x[5]^2 + x[6]^2 + x[7]^2 - 5
grad <- c(1, 0, 1, 0, 4*x[5]*s, 4*x[6]*s, 4*x[7]*s)
}
grad[j]
}

#Gr_x_k Gr_x_j O_i(x)
hefullob <- function(x, i, j, k)
{
x <- x[1:7]
if(i == 1)
{
he <- diag( 6*(x - 1:7) )
}
if(i == 2)
{
he <- diag( c(0, 2, 6, 12, 20, 30, 42)*(x - 1:7)^c(0, 0:5) )
}
if(i == 3)
{
s <- x[5]^2 + x[6]^2 + x[7]^2

he <- rbind(rep(0, 7), rep(0, 7), rep(0, 7), rep(0, 7),
c(0, 0, 0, 0, 4*s+8*x[5]^2, 8*x[5]*x[6], 8*x[5]*x[7]),
c(0, 0, 0, 0, 8*x[5]*x[6], 4*s+8*x[6]^2, 8*x[6]*x[7]),
c(0, 0, 0, 0, 8*x[5]*x[7], 8*x[6]*x[7], 4*s+8*x[7]^2) )
}
he[j,k]
}

# (2) constraint linked functions
#

dimlam <- c(1, 2, 2)

#constraint function g_i(x)
g <- function(x, i)

```

```

{
x <- x[1:7]
if(i == 1)
res <- sum( x^(1:7) ) -7
if(i == 2)
res <- c(sum(x) + prod(x) - 14, 20 - sum(x))
if(i == 3)
res <- c(sum(x^2) + 1, 100 - sum(x))
res
}

#Gr_x_j g_i(x)
grfullg <- function(x, i, j)
{
x <- x[1:7]
if(i == 1)
{
grad <- (1:7) * x ^ (0:6)
}
if(i == 2)
{
grad <- 1 + sapply(1:7, function(i) prod(x[-i]))
grad <- cbind(grad, -1)
}
if(i == 3)
{
grad <- cbind(2*x, -1)
}

if(i == 1)
res <- grad[j]
if(i != 1)
res <- grad[j,]
as.numeric(res)
}

#Gr_x_k Gr_x_j g_i(x)
hefullg <- function(x, i, j, k)
{
x <- x[1:7]
if(i == 1)
{
he1 <- diag( c(0, 2, 6, 12, 20, 30, 42) * x ^ c(0, 0, 1:5) )
}
if(i == 2)
{
he1 <- matrix(0, 7, 7)
he1[1, -1] <- sapply(2:7, function(i) prod(x[-c(1, i)]))
he1[2, -2] <- sapply(c(1, 3:7), function(i) prod(x[-c(2, i)]))
}

```



```

he1[3, -3] <- sapply(c(1:2, 4:7), function(i) prod(x[-c(3, i)]))
he1[4, -4] <- sapply(c(1:3, 5:7), function(i) prod(x[-c(4, i)]))
he1[5, -5] <- sapply(c(1:4, 6:7), function(i) prod(x[-c(5, i)]))
he1[6, -6] <- sapply(c(1:5, 7:7), function(i) prod(x[-c(6, i)]))
he1[7, -7] <- sapply(1:6, function(i) prod(x[-c(7, i)]))

he2 <- matrix(0, 7, 7)

}
if(i == 3)
{
he1 <- diag(rep(2, 7))
he2 <- matrix(0, 7, 7)
}
if(i != 1)
return( c(he1[j, k], he2[j, k]) )
else
return( he1[j, k] )
}

# (3) compute Phi
#

z <- rexp(sum(dimx) + sum(dimlam))

n <- sum(dimx)
m <- sum(dimlam)
x <- z[1:n]
lam <- z[(n+1):(n+m)]

resphi <- funSSR(z, dimx, dimlam, grobj=grfullob, constr=g, grconstr=grfullg, compl=phiFB)

check <- c(grfullob(x, 1, 1) + lam[1] * grfullg(x, 1, 1),
grfullob(x, 1, 2) + lam[1] * grfullg(x, 1, 2),
grfullob(x, 2, 3) + lam[2:3] %% grfullg(x, 2, 3),
grfullob(x, 2, 4) + lam[2:3] %% grfullg(x, 2, 4),
grfullob(x, 3, 5) + lam[4:5] %% grfullg(x, 3, 5),
grfullob(x, 3, 6) + lam[4:5] %% grfullg(x, 3, 6),
grfullob(x, 3, 7) + lam[4:5] %% grfullg(x, 3, 7),
phiFB( -g(x, 1), lam[1]),
phiFB( -g(x, 2)[1], lam[2]),
phiFB( -g(x, 2)[2], lam[3]),
phiFB( -g(x, 3)[1], lam[4]),
phiFB( -g(x, 3)[2], lam[5]))

#check
cat("\n\n_____ \n\n")

```

```

#part A
print(cbind(check, res=as.numeric(resphi))[1:n, ])
#part B
print(cbind(check, res=as.numeric(resphi))[(n+1):(n+m), ])

# (4) compute Jac Phi
#

resjacphi <- jacSSR(z, dimx, dimlam, heobj=hefullob, constr=g, grconstr=grfullg,
heconstr=hefullg, gcompla=GrAphiFB, gcomplb=GrBphiFB)

#check
cat("\n\n_____ \n\n")

cat("\n\npart A\n\n")

checkA <-
rbind(
c(hefullob(x, 1, 1, 1) + lam[1]*hefullg(x, 1, 1, 1),
hefullob(x, 1, 1, 2) + lam[1]*hefullg(x, 1, 1, 2),
hefullob(x, 1, 1, 3) + lam[1]*hefullg(x, 1, 1, 3),
hefullob(x, 1, 1, 4) + lam[1]*hefullg(x, 1, 1, 4),
hefullob(x, 1, 1, 5) + lam[1]*hefullg(x, 1, 1, 5),
hefullob(x, 1, 1, 6) + lam[1]*hefullg(x, 1, 1, 6),
hefullob(x, 1, 1, 7) + lam[1]*hefullg(x, 1, 1, 7)
),
c(hefullob(x, 1, 2, 1) + lam[1]*hefullg(x, 1, 2, 1),
hefullob(x, 1, 2, 2) + lam[1]*hefullg(x, 1, 2, 2),
hefullob(x, 1, 2, 3) + lam[1]*hefullg(x, 1, 2, 3),
hefullob(x, 1, 2, 4) + lam[1]*hefullg(x, 1, 2, 4),
hefullob(x, 1, 2, 5) + lam[1]*hefullg(x, 1, 2, 5),
hefullob(x, 1, 2, 6) + lam[1]*hefullg(x, 1, 2, 6),
hefullob(x, 1, 2, 7) + lam[1]*hefullg(x, 1, 2, 7)
),
c(hefullob(x, 2, 3, 1) + lam[2:3] %*% hefullg(x, 2, 3, 1),
hefullob(x, 2, 3, 2) + lam[2:3] %*% hefullg(x, 2, 3, 2),
hefullob(x, 2, 3, 3) + lam[2:3] %*% hefullg(x, 2, 3, 3),
hefullob(x, 2, 3, 4) + lam[2:3] %*% hefullg(x, 2, 3, 4),
hefullob(x, 2, 3, 5) + lam[2:3] %*% hefullg(x, 2, 3, 5),
hefullob(x, 2, 3, 6) + lam[2:3] %*% hefullg(x, 2, 3, 6),
hefullob(x, 2, 3, 7) + lam[2:3] %*% hefullg(x, 2, 3, 7)
),
c(hefullob(x, 2, 4, 1) + lam[2:3] %*% hefullg(x, 2, 4, 1),
hefullob(x, 2, 4, 2) + lam[2:3] %*% hefullg(x, 2, 4, 2),
hefullob(x, 2, 4, 3) + lam[2:3] %*% hefullg(x, 2, 4, 3),
hefullob(x, 2, 4, 4) + lam[2:3] %*% hefullg(x, 2, 4, 4),
hefullob(x, 2, 4, 5) + lam[2:3] %*% hefullg(x, 2, 4, 5),
hefullob(x, 2, 4, 6) + lam[2:3] %*% hefullg(x, 2, 4, 6),
hefullob(x, 2, 4, 7) + lam[2:3] %*% hefullg(x, 2, 4, 7)
),
),

```

```

c(hefullob(x, 3, 5, 1) + lam[4:5] %% hefullg(x, 3, 5, 1),
hefullob(x, 3, 5, 2) + lam[4:5] %% hefullg(x, 3, 5, 2),
hefullob(x, 3, 5, 3) + lam[4:5] %% hefullg(x, 3, 5, 3),
hefullob(x, 3, 5, 4) + lam[4:5] %% hefullg(x, 3, 5, 4),
hefullob(x, 3, 5, 5) + lam[4:5] %% hefullg(x, 3, 5, 5),
hefullob(x, 3, 5, 6) + lam[4:5] %% hefullg(x, 3, 5, 6),
hefullob(x, 3, 5, 7) + lam[4:5] %% hefullg(x, 3, 5, 7)
),
c(hefullob(x, 3, 6, 1) + lam[4:5] %% hefullg(x, 3, 6, 1),
hefullob(x, 3, 6, 2) + lam[4:5] %% hefullg(x, 3, 6, 2),
hefullob(x, 3, 6, 3) + lam[4:5] %% hefullg(x, 3, 6, 3),
hefullob(x, 3, 6, 4) + lam[4:5] %% hefullg(x, 3, 6, 4),
hefullob(x, 3, 6, 5) + lam[4:5] %% hefullg(x, 3, 6, 5),
hefullob(x, 3, 6, 6) + lam[4:5] %% hefullg(x, 3, 6, 6),
hefullob(x, 3, 6, 7) + lam[4:5] %% hefullg(x, 3, 6, 7)
),
c(hefullob(x, 3, 7, 1) + lam[4:5] %% hefullg(x, 3, 7, 1),
hefullob(x, 3, 7, 2) + lam[4:5] %% hefullg(x, 3, 7, 2),
hefullob(x, 3, 7, 3) + lam[4:5] %% hefullg(x, 3, 7, 3),
hefullob(x, 3, 7, 4) + lam[4:5] %% hefullg(x, 3, 7, 4),
hefullob(x, 3, 7, 5) + lam[4:5] %% hefullg(x, 3, 7, 5),
hefullob(x, 3, 7, 6) + lam[4:5] %% hefullg(x, 3, 7, 6),
hefullob(x, 3, 7, 7) + lam[4:5] %% hefullg(x, 3, 7, 7)
)
)

print(resjacphi[1:n, 1:n] - checkA)

cat("\n\n_____ \n\n")

cat("\n\npart B\n\n")

checkB <-
rbind(
cbind(c(grfullg(x, 1, 1), grfullg(x, 1, 2)), c(0, 0), c(0, 0), c(0, 0), c(0, 0)),
cbind(c(0, 0), rbind(grfullg(x, 2, 3), grfullg(x, 2, 4)), c(0, 0), c(0, 0)),
cbind(c(0, 0, 0), c(0, 0, 0), c(0, 0, 0),
  rbind(grfullg(x, 3, 5), grfullg(x, 3, 6), grfullg(x, 3, 7)))
)

print(resjacphi[1:n, (n+1):(n+m)] - checkB)

cat("\n\n_____ \n\n")
cat("\n\npart C\n\n")

gx <- c(g(x,1), g(x,2), g(x,3))

```

```

checkC <-
- t(
  cbind(
    rbind(
      grfullg(x, 1, 1) * GrAphiFB(-gx, lam)[1],
      grfullg(x, 1, 2) * GrAphiFB(-gx, lam)[1],
      grfullg(x, 1, 3) * GrAphiFB(-gx, lam)[1],
      grfullg(x, 1, 4) * GrAphiFB(-gx, lam)[1],
      grfullg(x, 1, 5) * GrAphiFB(-gx, lam)[1],
      grfullg(x, 1, 6) * GrAphiFB(-gx, lam)[1],
      grfullg(x, 1, 7) * GrAphiFB(-gx, lam)[1]
    ),
    rbind(
      grfullg(x, 2, 1) * GrAphiFB(-gx, lam)[2:3],
      grfullg(x, 2, 2) * GrAphiFB(-gx, lam)[2:3],
      grfullg(x, 2, 3) * GrAphiFB(-gx, lam)[2:3],
      grfullg(x, 2, 4) * GrAphiFB(-gx, lam)[2:3],
      grfullg(x, 2, 5) * GrAphiFB(-gx, lam)[2:3],
      grfullg(x, 2, 6) * GrAphiFB(-gx, lam)[2:3],
      grfullg(x, 2, 7) * GrAphiFB(-gx, lam)[2:3]
    ),
    rbind(
      grfullg(x, 3, 1) * GrAphiFB(-gx, lam)[4:5],
      grfullg(x, 3, 2) * GrAphiFB(-gx, lam)[4:5],
      grfullg(x, 3, 3) * GrAphiFB(-gx, lam)[4:5],
      grfullg(x, 3, 4) * GrAphiFB(-gx, lam)[4:5],
      grfullg(x, 3, 5) * GrAphiFB(-gx, lam)[4:5],
      grfullg(x, 3, 6) * GrAphiFB(-gx, lam)[4:5],
      grfullg(x, 3, 7) * GrAphiFB(-gx, lam)[4:5]
    )
  )
)

print(resjacphi[(n+1):(n+m), 1:n] - checkC)

cat("\n\n_____ \n\n")

cat("\n\npart D\n\n")

checkD <- diag(GrBphiFB(-gx, lam))

print(resjacphi[(n+1):(n+m), (n+1):(n+m)] - checkD)

```

**Description**

Step functions for relaxation methods

**Usage**

```
purestep(k)
decrstep(k, param)
decrstep5(k)
decrstep10(k)
decrstep20(k)
```

**Arguments**

k	iteration number.
param	parameter for the decreasing step function after which the step decreases.

**Details**

The decrstep function is a decreasing step serie such that  $\text{decrstep}(k)$  equals to  $1/2/(k - \text{param})$  when  $k > \text{param}$ ,  $1/2$ , otherwise. Functions `decrstep5`, `decrstep10`, `decrstep20` are just wrappers of `decrstep`.

The `purestep` function implements a constant step serie equaled to 1.

**Value**

A numeric.

**Author(s)**

Christophe Dutang

**See Also**

See also [GNE](#) and [GNE.fpeq](#).

**Examples**

```
cbind(
  purestep(1:20),
  decrstep(1:20, 7),
  decrstep5(1:20),
  decrstep10(1:20),
  decrstep20(1:20)
)
```

**Description**

functions of the Nikaido Isoda Reformulation of the GNEP

**Usage**

```
gapVIR(x, y, dimx, grobj, arggrobj, param=list(), echo=FALSE)
gradxgapVIR(x, y, dimx, grobj, arggrobj, heobj, argheobj, param=list(), echo=FALSE)
gradygapVIR(x, y, dimx, grobj, arggrobj, param=list(), echo=FALSE)
fpVIR(x, dimx, obj, argobj, joint, argjoint,
      grobj, arggrobj, jacjoint, argjacjoint, param=list(),
      echo=FALSE, control=list(), yinit=NULL, optim.method="default")
```

**Arguments**

<code>x,y</code>	a numeric vector.
<code>dimx</code>	a vector of dimension for $x$ .
<code>obj</code>	objective function (to be minimized), see details.
<code>argobj</code>	a list of additional arguments.
<code>grobj</code>	gradient of the objective function, see details.
<code>arggrobj</code>	a list of additional arguments of the objective gradient.
<code>heobj</code>	Hessian of the objective function, see details.
<code>argheobj</code>	a list of additional arguments of the objective Hessian.
<code>joint</code>	joint function, see details.
<code>argjoint</code>	a list of additional arguments of the joint function.
<code>jacjoint</code>	gradient of the joint function, see details.
<code>argjacjoint</code>	a list of additional arguments of the joint Jacobian.
<code>param</code>	a list of parameters.
<code>control</code>	a list with control parameters for the fixed point algorithm.
<code>yinit</code>	initial point when computing the fixed-point function.
<code>optim.method</code>	optimization method when computing the fixed-point function.
<code>echo</code>	a logical to show some traces.

**Details**

`gapVIR` computes the Nikaido Isoda function of the GNEP, while `gradxgapVIR` and `gradygapVIR` give its gradient with respect to  $x$  and  $y$ . `fpVIR` computes the fixed-point function.

**Value**

A vector for funSSR or a matrix for jacSSR.

**Author(s)**

Christophe Dutang

**References**

A. von Heusinger & J. Kanzow (2009), *Optimization reformulations of the generalized Nash equilibrium problem using Nikaido-Isoda-type functions*, Comput Optim Appl .

F. Facchinei, A. Fischer and V. Piccialli (2009), *Generalized Nash equilibrium problems and Newton methods*, Math. Program.

**See Also**

See also [GNE.fpeq](#).

# Index

## \*Topic **math**

bench.GNE, 2  
CER, 4  
GNE, 12  
NIR, 30  
SSR, 36  
stepfunc, 44  
VIR, 46

## \*Topic **nonlinear**

compl, 8  
eqsolve, 10  
GNE.ceq, 14  
GNE.fpeq, 18  
GNE.minpb, 22  
GNE.nseq, 24  
potential.reduction, 32  
projector, 33

## \*Topic **optimize**

CER, 4  
compl, 8  
eqsolve, 10  
GNE.ceq, 14  
GNE.fpeq, 18  
GNE.minpb, 22  
GNE.nseq, 24  
NIR, 30  
potential.reduction, 32  
projector, 33  
rejection, 35  
SSR, 36  
VIR, 46

bench.GNE, 2, 12

CER, 4  
compl, 8  
complementarity, 12, 28  
complementarity (compl), 8

decrstep (stepfunc), 44

decrstep10 (stepfunc), 44  
decrstep20 (stepfunc), 44  
decrstep5 (stepfunc), 44

eqsolve, 10

fpiter, 20  
fpNIR (NIR), 30  
fpVIR (VIR), 46  
funCER, 16  
funCER (CER), 4  
funSSR, 12, 28  
funSSR (SSR), 36

gapNIR (NIR), 30  
gapVIR (VIR), 46  
GNE, 12, 34, 36, 45  
GNE.ceq, 4, 6, 12, 13, 14, 21, 24, 28, 33  
GNE.fpeq, 4, 12, 13, 16, 18, 24, 28, 32, 45, 47  
GNE.minpb, 4, 12, 13, 16, 21, 22, 28  
GNE.nseq, 4, 10, 12, 13, 16, 21, 24, 24, 38  
gradpotential.ce (potential.reduction), 32

gradpsi.ce (potential.reduction), 32  
gradxgapNIR (NIR), 30  
gradxgapVIR (VIR), 46  
gradygapNIR (NIR), 30  
gradygapVIR (VIR), 46  
GrAphiFB (compl), 8  
GrAphiKK (compl), 8  
GrAphiLT (compl), 8  
GrAphiMan (compl), 8  
GrAphiMin (compl), 8  
GrAphiFB (compl), 8  
GrAphiRFB (compl), 8  
GrBphiFB (compl), 8  
GrBphiKK (compl), 8  
GrBphiLT (compl), 8  
GrBphiMan (compl), 8  
GrBphiMin (compl), 8



GrBhipFB (compl), 8  
GrBphirFB (compl), 8

jacCER, 16  
jacCER (CER), 4  
jacSSR, 28  
jacSSR (SSR), 36

NIR, 30  
nleqslv, 11, 27, 28

phiFB (compl), 8  
phiKK (compl), 8  
phiLT (compl), 8  
phiMan (compl), 8  
phiMin (compl), 8  
hipFB (compl), 8  
phirFB (compl), 8  
potential.ce (potential.reduction), 32  
potential.reduction, 32  
print.compl.par (compl), 8  
Projector (projector), 33  
projector, 12, 33  
psi.ce (potential.reduction), 32  
purestep (stepfunc), 44

rejection, 12, 35

squarem, 20  
SSR, 36  
stepfunc, 12, 44  
summary.compl.par (compl), 8

VIR, 46