Title

stata.com

solvenl( ) — Solve systems of nonlinear equations

Description

The `solvenl()` suite of functions finds solutions to systems of nonlinear equations.

`solvenl_init()` initializes the problem and returns *S*, a structure that contains information regarding the problem, including default values. If you declare a storage type for *S*, declare it to be a `transmorphic scalar`.

The `solvenl_init_*(`*S*`, ...)` functions allow you to modify those default values and specify other aspects of your problem, including whether your problem refers to finding a fixed point or a zero starting value to use, etc.

`solvenl_solve(`*S*`)` solves the problem. `solvenl_solve()` returns a vector that represents either a fixed point of your function or a vector at which your function is equal to a vector of zeros.

The `solvenl_result_*(`*S*`)` functions let you access other information associated with the solution to your problem, including whether a solution was achieved, the final Jacobian matrix, and diagnostics.

Aside: The `solvenl_init_*(`*S*`, ...)` functions have two modes of operation. Each has an optional argument that you specify to set the value and that you omit to query the value. For instance, the full syntax of `solvenl_init_startingvals()` is

void `solvenl_init_startingvals(`*S*`,` *real colvector ivals*`)`
real colvector `solvenl_init_startingvals(`*S*`)`

The first syntax sets the parameter values and returns nothing. The second syntax returns the previously set (or default, if not set) parameter values.

All the `solvenl_init_*(`*S*`, ...)` functions work the same way.

Syntax

```
S = solvenl_init()

(varies)        solvenl_init_type(S [, {"fixedpoint"|"zero"}])
(varies)        solvenl_init_startingvals(S [, real colvector ivals])
(varies)        solvenl_init_numeq(S [, real scalar nvars])
(varies)        solvenl_init_technique(S [, "technique"])
(varies)        solvenl_init_conv_iterchng(S [, real scalar itol])
(varies)        solvenl_init_conv_nearzero(S [, real scalar ztol])
(varies)        solvenl_init_conv_maxiter(S [, real scalar maxiter])
(varies)        solvenl_init_evaluator(S [, &evaluator()])
(varies)        solvenl_init_argument(S, real scalar k [, X])
(varies)        solvenl_init_narguments(S [, real scalar K])
(varies)        solvenl_init_damping(S [, real scalar damp])
(varies)        solvenl_init_iter_log(S [, {"on"|"off"}])
(varies)        solvenl_init_iter_dot(S [, {"on"|"off"}])
(varies)        solvenl_init_iter_dot_indent(S [, real scalar indent])

real colvector  solvenl_solve(S)
real scalar     _solvenl_solve(S)

real scalar     solvenl_result_converged(S)
real scalar     solvenl_result_conv_iter(S)
real scalar     solvenl_result_conv_iterchng(S)
real scalar     solvenl_result_conv_nearzero(S)
real colvector  solvenl_result_values(S)
real matrix     solvenl_result_Jacobian(S)
real scalar     solvenl_result_error_code(S)
real scalar     solvenl_result_return_code(S)
string scalar   solvenl_result_error_text(S)

void            solvenl_dump(S)
```

S, if it is declared, should be declared as

```
transmorphic S
```

and *technique* optionally specified in `solvenl_init_technique()` is one of the following:

	technique	Description
	gaussseidel	Gauss–Seidel
	dampedgaussseidel	Damped Gauss–Seidel
	broydenpowell	Broyden–Powell
*	newtonraphson	Newton–Raphson

* `newton` may also be abbreviated as `nr`.

For fixed-point problems, allowed *technique*s are `gaussseidel` and `dampedgaussseidel`. For zero-finding problems, allowed *technique*s are `broydenpowell` and `newtonraphson`. `solvenl_*()` exits with an error message if you specify a *technique* that is incompatible with the type of evaluator you declared by using `solvenl_init_type()`. The default technique for fixed-point problems is `dampedgaussseidel` with a damping parameter of 0.1. The default technique for zero-finding problems is `broydenpowell`.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

Let $\mathbf{x}$ denote a $k \times 1$ vector and let $\mathtt{F} : R^k \rightarrow R^k$ denote a function that represents a system of equations. The `solvenl()` suite of functions can be used to find fixed-point solutions $\mathbf{x}^* = \mathbf{F}(\mathbf{x}^*)$, and it can be used to find a zero of the function, that is, a vector $\mathbf{x}^*$ such that $\mathbf{F}(\mathbf{x}^*) = \mathbf{0}$.

Four solution methods are available: Gauss–Seidel (GS), damped Gauss–Seidel (dGS), Newton's method (also known as the Newton–Raphson method), and the Broyden–Powell (BP) method. The first two methods are used to find fixed points, and the latter two are used to find zeros. However, as we discuss below, fixed-point problems can be rewritten as zero-finding problems, and many zero-finding problems can be rewritten as fixed-point problems.

Solving systems of nonlinear equations is inherently more difficult than minimizing or maximizing a function. The set of first-order conditions associated with an optimization problem satisfies a set of integrability conditions, while `solvenl_*()` works with arbitrary systems of nonlinear equations. Moreover, while one may be tempted to approach a zero-finding problem by defining a function

$$f(\mathbf{x}) = \mathbf{F}(\mathbf{x})'\mathbf{F}(\mathbf{x})$$

and minimizing $f(\mathbf{x})$, there is a high probability that the minimizer will find a local minimum for which $\mathbf{F}(\mathbf{x}) \neq \mathbf{0}$ (Press et al. 2007, 476). Some problems may have multiple solutions.

A fixed-point example

We want to solve the system of equations

$$x = \tfrac{5}{3} - \tfrac{2}{3}y$$
$$y = \tfrac{10}{3} - \tfrac{2}{3}x$$

First, we write a program that takes two arguments: a column vector representing the values at which we are to evaluate our function and a column vector into which we are to place the function values.

```
: void function myfun(real colvector from, real colvector values)
> {
>         values[1] =  5/3 - 2/3*from[2]
>         values[2] = 10/3 - 2/3*from[1]
> }
```

Our invocation of `solvenl_*()` proceeds as follows:

```
: S = solvenl_init()
: solvenl_init_evaluator(S, &myfun())
: solvenl_init_type(S, "fixedpoint")
: solvenl_init_technique(S, "gaussseidel")
: solvenl_init_numeq(S, 2)
: solvenl_init_iter_log(S, "on")
: x = solvenl_solve(S)
Iteration 1:    3.3333333
Iteration 2:    .83333333
  (output omitted)
: x
                    1
  +-----------------+
1 |  -.9999999981   |
2 |               4 |
  +-----------------+
```

In our equation with x on the left-hand side, x did not appear on the right-hand side, and similarly for the equation with y. However, that is not required. Fixed-point problems with left-hand-side variables appearing on the right-hand side of the same equation can be solved, though they typically require more iterations to reach convergence.

A zero-finding example

We wish to solve the following system of equations (Burden, Faires, and Burden 2016, 657) for the three unknowns x, y, and z:

$$10 - x\,e^{y} - z = 0$$
$$12 - x\,e^{2y} - 2z = 0$$
$$15 - x\,e^{3y} - 3z = 0$$

We will use Newton's method. We cannot use $x = y = z = 0$ as initial values because the Jacobian matrix is singular at that point; we will instead use $x = y = z = 0.2$. Our program is

```
: void function myfun2(real colvector x, real colvector values)
> {
>         values[1] = 10 - x[1]*exp(x[2]*1) - x[3]*1
>         values[2] = 12 - x[1]*exp(x[2]*2) - x[3]*2
>         values[3] = 15 - x[1]*exp(x[2]*3) - x[3]*3
> }
: S = solvenl_init()
: solvenl_init_evaluator(S, &myfun2())
: solvenl_init_type(S, "zero")
: solvenl_init_technique(S, "newton")
: solvenl_init_numeq(S, 3)
: solvenl_init_startingvals(S, J(3,1,.2))
: solvenl_init_iter_log(S, "on")
: x = solvenl_solve(S)
Iteration 0:  function = 416.03613
Iteration 1:  function = 63.014451  delta X = 1.2538445
Iteration 2:  function = 56.331397  delta X = .70226488
Iteration 3:  function = 48.572941  delta X = .35269647
Iteration 4:  function = 37.434106  delta X = .30727054
Iteration 5:  function = 19.737501  delta X = .38136739
Iteration 6:  function = .49995202  delta X =  .2299557
Iteration 7:  function = 1.164e-08  delta X = .09321045
Iteration 8:  function = 4.154e-16  delta X = .00011039
: x
                  1
  +-----------------+
1 |   8.771286448   |
2 |    .2596954499  |
3 |  -1.372281335   |
  +-----------------+
```

Writing a fixed-point problem as a zero-finding problem and vice versa

Earlier, we solved the system of equations

$$x = \tfrac{5}{3} - \tfrac{2}{3}y$$
$$y = \tfrac{10}{3} - \tfrac{2}{3}x$$

by searching for a fixed point. We can rewrite this system as

$$x - \tfrac{5}{3} + \tfrac{2}{3}y = 0$$
$$y - \tfrac{10}{3} + \tfrac{2}{3}x = 0$$

and then use BP or Newton's method to find the solution. In general, we simply rewrite $\mathbf{x}^* = \mathbf{F}(\mathbf{x}^*)$ as $\mathbf{x}^* - \mathbf{F}(\mathbf{x}^*) = \mathbf{0}$.

Similarly, we may be able to rearrange the constituent equations of a system of the form $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ so that each variable is an explicit function of the other variables in the system. If that is the case, then GS or dGS can be used to find the solution.

Gauss–Seidel methods

Let $\mathbf{x}_{i-1}$ denote the previous iteration's values or the initial values, and let $\mathbf{x}_i$ denote the current iteration's values. The Gauss–Jacobi method simply iterates on $\mathbf{x}_i = \mathbf{F}(\mathbf{x}_{i-1})$ by evaluating each equation in order. The Gauss–Seidel method implemented in `solvenl_*()` instead uses the new, updated values of $\mathbf{x}_i$ that are available for equations 1 through $j-1$ when evaluating equation j at iteration i.

For damped Gauss–Seidel, again let $\mathbf{x}_i$ denote the values obtained from evaluating $\mathbf{F}(\mathbf{x}_{i-1})$. However, after evaluating $\mathbf{F}$, dGS calculates the new parameter vector that is carried over to the next iteration as

$$\mathbf{x}_i^{\#} = (1-\delta)\mathbf{x}_i + \delta\mathbf{x}_{i-1}$$

where δ is the damping factor. Not fully updating the parameter vector at each iteration helps facilitate convergence in many problems. The default value of δ for method dGS is 0.1, representing just a small amount of damping, which is often enough to achieve convergence. You can use `solvenl_init_damping()` to change δ; the current implementation uses the same value of δ for all iterations. Increasing the damping factor generally slows convergence by requiring more iterations.

Newton-type methods

Newton's method for solving $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is based on the approximation

$$\mathbf{F}(\mathbf{x}_i) \approx \mathbf{F}(\mathbf{x}_{i-1}) + \mathbf{J}(\mathbf{x}_{i-1}) \times (\mathbf{x}_i - \mathbf{x}_{i-1})$$

where $\mathbf{J}(\mathbf{x}_{i-1})$ is the Jacobian matrix of $\mathbf{F}(\mathbf{x}_{i-1})$. Rearranging and incorporating a step-length parameter α, we have the iteration

$$\mathbf{x}_i = \mathbf{x}_{i-1} - \alpha\,\mathbf{J}^{-1}(\mathbf{x}_{i-1}) \times \mathbf{F}(\mathbf{x}_{i-1})$$

We calculate $\mathbf{J}$ numerically by using the `deriv()` (see [M-5] **deriv( )**) suite of functions. In fact, we do not calculate the inverse of $\mathbf{J}$; we instead use LU decomposition to solve for $\mathbf{x}_i - \mathbf{x}_{i-1}$.

To speed up convergence, we define the function $f(\mathbf{x}) = \mathbf{F}(\mathbf{x})'\mathbf{F}(\mathbf{x})$ and then choose α between 0 and 1 such that $f(\mathbf{x}_i)$ is minimized. We use a golden-section line search with a maximum of 20 iterations to find α.

Because we must compute a $k \times k$ Jacobian matrix at each iteration, Newton's method can be slow. The BP method, similar to quasi-Newton methods for optimization, instead builds and updates an approximation $\mathbf{B}$ to the Jacobian matrix at each iteration. The BP update is

$$\mathbf{B}_i = \mathbf{B}_{i-1} + \frac{\mathbf{y}_i - \mathbf{B}_{i-1}\mathbf{d}_i}{\mathbf{d}_i'\mathbf{d}_i}\mathbf{d}_i'$$

where $\mathbf{d}_i = \mathbf{x}_i - \mathbf{x}_{i-1}$ and $\mathbf{y}_i = \mathbf{F}(\mathbf{x}_i) - \mathbf{F}(\mathbf{x}_{i-1})$. Our initial estimate of the Jacobian matrix is calculated numerically at the initial values by using `deriv()`. Other than how the Jacobian matrix is updated, the BP method is identical to Newton's method, including the use of a step-length parameter determined by using a golden-section line search at each iteration.

Convergence criteria

`solvenl_*()` stops if more than *maxiter* iterations are performed, where *maxiter* is `c(maxiter)` by default and can be changed by using `solvenl_init_conv_maxiter()`. Convergence is not declared after *maxiter* iterations unless one of the following convergence criteria is also met.

Let $\mathbf{x}_i$ denote the proposed solution at iteration *i*, and let $\mathbf{x}_{i-1}$ denote the proposed solution at the previous iteration. Then the parameters have converged when $\texttt{mreldif}(\mathbf{x}_i, \mathbf{x}_{i-1}) < \textit{itol}$, where *itol* is 1e–9 by default and can be changed by using `solvenl_init_conv_iterchng()`. Techniques GS and dGS use only this convergence criterion.

For BP and Newton's method, let $f(\mathbf{x}_i) = \mathbf{F}(\mathbf{x}_i)'\mathbf{F}(\mathbf{x}_i)$. Then convergence is declared if $\texttt{mreldif}(\mathbf{x}_i, \mathbf{x}_{i-1}) < \textit{itol}$ or $f(\mathbf{x}_i) < \textit{ztol}$, where *ztol* is 1e–9 by default and can be changed by using `solvenl_init_conv_nearzero()`.

Exiting early

In some applications, you might have a condition that indicates your problem either has no solution or has a solution that you know to be irrelevant. In these cases, you can return a column vector with zero rows. `solvenl()` will then exit immediately and return an error code indicating you have requested an early exit.

To obtain this behavior, include the following code in your evaluator:

```
: void function myfun(real colvector from, real colvector values)
>
>         ...
>         if (condition)
>                 values = J(0, 1, .)
>                 return
>
>         values[1] =  5/3 - 2/3*from[2]
>         values[2] = 10/3 - 2/3*from[1]
>         ...
>
```

Then if *condition* is true, `solvenl()` exits, `solvenl_result_error_code()` returns error code 27, and `solvenl_result_converged()` returns 0 (indicating a solution has not been found).

Functions

solvenl_init( )

```
solvenl_init()
```

`solvenl_init()` is used to initialize the solver. Store the returned result in a variable name of your choosing; we use the letter *S*. You pass *S* as the first argument to the other `solvenl()` suite of functions.

`solvenl_init()` sets all `solvenl_init_*()` values to their defaults. You can use the query form of the `solvenl_init_*()` functions to determine their values. Use `solvenl_dump()` to see the current state of the solver, including the current values of the `solvenl_init_*()` parameters.

solvenl_init_type( )

void `solvenl_init_type(`*S*`, {"fixedpoint"|"zero"})`

string scalar `solvenl_init_type(`*S*`)`

`solvenl_init_type(`*S*`,` *type*`)` specifies whether to find a fixed point or a zero of the function. *type* may be `fixedpoint` or `zero`.

If you specify `solvenl_init_type(`*S*`, "fixedpoint")` but have not yet specified a *technique*, then *technique* is set to `dampedgaussseidel`.

If you specify `solvenl_init_type(`*S*`, "zero")` but have not yet specified a *technique*, then *technique* is set to `broydenpowell`.

`solvenl_init_type(`*S*`)` returns `"fixedpoint"` or `"zero"` depending on how the solver is currently set.

solvenl_init_startingvals( )

void `solvenl_init_startingvals(`*S*`,` *real colvector ivals*`)`

real colvector `solvenl_init_startingvals(`*S*`)`

`solvenl_init_startingvals(`*S*`,` *ivals*`)` sets the initial values for the solver to *ivals*. By default, *ivals* is set to the zero vector.

`solvenl_init_startingvals(`*S*`)` returns the currently set initial values.

solvenl_init_numeq( )

void `solvenl_init_numeq(`*S*`,` *real scalar k*`)`

real scalar `solvenl_init_numeq(`*S*`)`

`solvenl_init_numeq(`*S*`,` *k*`)` sets the number of equations in the system to *k*.

`solvenl_init_numeq(`*S*`)` returns the currently specified number of equations.

solvenl_init_technique()

void `solvenl_init_technique(`*S*`,` *technique*`)`

string scalar `solvenl_init_technique(`*S*`)`

`solvenl_init_technique(`*S*`,` *technique*`)` specifies the solver technique to use. For more information, see *technique* above.

If you specify *technique*s `gaussseidel` or `dampedgaussseidel` but have not yet called `solvenl_init_type()`, `solvenl_*()` assumes you are solving a fixed-point problem until you specify otherwise.

If you specify *technique*s `broydenpowell` or `newtonraphson` but have not yet called `solvenl_init_type()`, `solvenl_*()` assumes you have a zero-finding problem until you specify otherwise.

`solvenl_init_technique(`*S*`)` returns the currently set solver technique.

solvenl_init_conv_iterchng()

void `solvenl_init_conv_iterchng(`*S*`,` *itol*`)`

real scalar `solvenl_init_conv_iterchng(`*S*`)`

`solvenl_init_conv_iterchng(`*S*`,` *itol*`)` specifies the tolerance used to determine whether successive estimates of the solution have converged. Convergence is declared when `mreldif(`$\mathbf{x}(i), \mathbf{x}(i-1)$`)` $<$ *itol*. For more information, see *Convergence criteria* above. The default is 1e–9.

`solvenl_init_conv_iterchng(`*S*`)` returns the currently set value of *itol*.

solvenl_init_conv_nearzero()

void `solvenl_init_conv_nearzero(`*S*`,` *ztol*`)`

real scalar `solvenl_init_conv_nearzero(`*S*`)`

`solvenl_init_conv_nearzero(`*S*`,` *ztol*`)` specifies the tolerance used to determine whether the proposed solution to a zero-finding problem is sufficiently close to 0 based on the squared Euclidean distance. For more information, see *Convergence criteria* above. The default is 1e–9.

`solvenl_init_conv_nearzero(`*S*`)` returns the currently set value of *ztol*.

`solvenl_init_conv_nearzero()` only applies to zero-finding problems. `solvenl_*()` simply ignores this criterion when solving fixed-point problems.

solvenl_init_conv_maxiter( )

void solvenl_init_conv_maxiter(*S*, *maxiter*)

real scalar solvenl_init_conv_maxiter(*S*)

solvenl_init_conv_maxiter(*S*, *maxiter*) specifies the maximum number of iterations to perform. Even if *maxiter* iterations are performed, convergence is not declared unless one of the other convergence criteria is also met. For more information, see *Convergence criteria* above. The default is 16,000 or whatever was previously declared by using set maxiter (see [R] **maximize**).

solvenl_init_conv_maxiter(*S*) returns the currently set value of *maxiter*.

solvenl_init_evaluator( )

void solvenl_init_evaluator(*S*, *pointer(real function) scalar fptr*)

pointer(real function) scalar solvenl_init_evaluator(*S*)

solvenl_init_evaluator(*S*, *fptr*) specifies the function to be called to evaluate $\mathbf{F}(\mathbf{x})$. You must use this function. If your function is named myfcn(), then you specify solvenl_init_evaluator(*S*, &myfcn()).

solvenl_init_evaluator(*S*) returns a pointer to the function that has been set.

solvenl_init_argument( ) and solvenl_init_narguments( )

void solvenl_init_argument(*S*, *real scalar k*, *X*)

void solvenl_init_narguments(*S*, *real scalar K*)

pointer scalar solvenl_init_argument(*S*, *real scalar k*)

real scalar solvenl_init_narguments(*S*)

solvenl_init_argument(*S*, *k*, *X*) sets the *k*th extra argument of the evaluator function as *X*, where *k* can be 1, 2, or 3. If you need to pass more items to your evaluator, collect them into a structure and pass the structure. *X* can be anything, including a pointer, a view matrix, or simply a scalar. No copy of *X* is made; it is passed by reference. Any changes you make to *X* elsewhere in your program will be reflected in what is passed to your evaluator function.

solvenl_init_narguments(*S*, *K*) sets the number of extra arguments to be passed to your evaluator function. Use of this function is optional; initializing an additional argument by using solvenl_init_argument() automatically sets the number of arguments.

solvenl_init_argument(*S*, *k*) returns a pointer to the previously set *k*th additional argument.

solvenl_init_narguments(*S*) returns the number of extra arguments that are passed to the evaluator function.

solvenl_init_damping( )

void solvenl_init_damping(*S*, *real scalar d*)

real scalar solvenl_init_damping(*S*)

solvenl_init_damping(*S*, *d*) sets the damping parameter used by the damped Gauss–Seidel technique to *d*, where $0 \leq d < 1$. That is, $d = 0$ corresponds to no damping, which is equivalent to plain Gauss–Seidel. As *d* approaches 1, more damping is used. The default is $d = 0.1$. If the dGS technique is not being used, this parameter is ignored.

solvenl_init_damping(*S*) returns the currently set damping parameter.

solvenl_init_iter_log( )

void solvenl_init_iter_log(*S*, {"on"|"off"})

string scalar solvenl_init_iter_log(*S*)

solvenl_init_iter_log(*S*, *onoff*) specifies whether an iteration log should or should not be displayed. *onoff* may be on or off. By default, an iteration log is displayed.

solvenl_init_iter_log(*S*) returns the current status of the iteration log indicator.

solvenl_init_iter_dot( )

void solvenl_init_iter_dot(*S*, {"on"|"off"})

string scalar solvenl_init_iter_dot(*S*)

solvenl_init_iter_dot(*S*, *onoff*) specifies whether an iteration dot should or should not be displayed. *onoff* may be on or off. By default, an iteration dot is not displayed.

Specifying solvenl_init_iter_dot(*S*, on) results in the display of a single dot without a new line after each iteration is completed. This option can be used to create a compact status report when a full iteration log is too detailed but some indication of activity is warranted.

solvenl_init_iter_dot(*S*) returns the current status of the iteration dot indicator.

solvenl_init_iter_dot_indent( )

void solvenl_init_iter_dot_indent(*S*, *real scalar indent*)

string scalar solvenl_init_iter_dot_indent(*S*)

solvenl_init_iter_dot_indent(*S*, *indent*) specifies how many spaces from the left edge iteration dots should begin. This option is useful if you are writing a program that calls solvenl() and if you want to control how the iteration dots appear to the user. By default, the dots start at the left edge (indent = 0). If you do not turn on iteration dots with solvenl_init_iter_dot(), this option is ignored.

solvenl_init_iter_dot_indent(*S*) returns the current amount of indentation.

solvenl_solve( ) and _solvenl_solve( )

real colvector `solvenl_solve(`*S*`)`

void `_solvenl_solve(`*S*`)`

`solvenl_solve(`*S*`)` invokes the solver and returns the resulting solution. If an error occurs, `solvenl_solve()` aborts with error.

`_solvenl_solve(`*S*`)` also invokes the solver. Rather than returning the solution, this function returns an error code if something went awry. If the solver did find a solution, this function returns `0`. See below for a list of the possible error codes.

Before calling either of these functions, you must have defined your problem. At a minimum, this involves calling the following functions:

`solvenl_init()`

`solvenl_init_numeq()`

`solvenl_init_evaluator()`

`solvenl_init_type()` or `solvenl_init_technique()`

solvenl_result_converged( )

real scalar `solvenl_result_converged(`*S*`)`

`solvenl_result_converged(`*S*`)` returns `1` if the solver found a solution to the problem and `0` otherwise.

solvenl_result_conv_iter( )

real scalar `solvenl_result_conv_iter(`*S*`)`

`solvenl_result_conv_iter(`*S*`)` returns the number of iterations required to obtain the solution. If a solution was not found or the solver has not yet been called, this function returns missing.

solvenl_result_conv_iterchng( )

real scalar `solvenl_result_conv_iterchng(`*S*`)`

`solvenl_result_conv_iterchng(`*S*`)` returns the final tolerance achieved for the parameters if a solution has been reached. Otherwise, this function returns missing. For more information, see *Convergence criteria* above.

solvenl_result_conv_nearzero( )

real scalar `solvenl_result_conv_nearzero(`*S*`)`

`solvenl_result_conv_nearzero(`*S*`)` returns the final distance the solution lies from zero if a solution has been reached. Otherwise, this function returns missing. This function also returns missing if called after either GS or dGS was used because this criterion does not apply. For more information, see *Convergence criteria* above.

solvenl_result_values( )

real colvector `solvenl_result_values(`*S*`)`

`solvenl_result_values(`*S*`)` returns the column vector representing the fixed- or zero-point of the function if a solution was found. Otherwise, it returns a 0×1 vector of missing values.

solvenl_result_Jacobian()

real matrix `solvenl_result_Jacobian(`*S*`)`

`solvenl_result_Jacobian(`*S*`)` returns the last-calculated Jacobian matrix if BP or Newton's method was used to find a solution. The Jacobian matrix is returned even if a solution was not found because we have found the Jacobian matrix to be useful in pinpointing problems. This function returns a 1×1 matrix of missing values if called after either GS or dGS was used.

solvenl_result_error_code( ), . . . _return_code( ), and . . . _error_text( )

real scalar `solvenl_result_error_code(`*S*`)`

real scalar `solvenl_result_return_code(`*S*`)`

string scalar `solvenl_result_error_text(`*S*`)`

`solvenl_result_error_code(`*S*`)` returns the unique `solvenl_*()` error code generated or zero if there was no error. Each error that can be produced by the system is assigned its own unique code.

`solvenl_result_return_code(`*S*`)` returns the appropriate return code to be returned to the user if an error was produced.

`solvenl_result_error_text(`*S*`)` returns an appropriate textual description to be displayed if an error was produced.

The error codes, return codes, and error text are listed below.

Error code	Return code	Error text
0	0	(no error encountered)
1	0	(problem not yet solved)
2	111	did not specify function
3	198	invalid number of equations specified
4	504	initial value vector has missing values
5	503	initial value vector length does not equal number of equations declared
6	430	maximum iterations reached; convergence not achieved
7	416	missing values encountered when evaluating function
8	3498	invalid function type
9	3498	function type ... cannot be used with technique ...
10	3498	invalid log option
11	3498	invalid solution technique
12	3498	solution technique *technique* cannot be used with function type { `"fixedpoint"` | `"zero"` }
13	3498	invalid iteration change criterion
14	3498	invalid near-zeroness criterion
15	3498	invalid maximum number of iterations criterion
16	3498	invalid function pointer
17	3498	invalid number of arguments
18	3498	optional argument out of range
19	3498	could not evaluate function at initial values
20	3498	could not calculate Jacobian at initial values
21	3498	iterations found local minimum of $\mathbf{F}'\mathbf{F}$; convergence not achieved
22	3498	could not calculate Jacobian matrix
23	198	damping factor must be in $[0, 1)$
24	198	must specify a function type, technique, or both
25	3498	invalid `solvenl_init_iter_dot()` option
26	3498	`solvenl_init_iter_dot_indent()` must be a nonnegative integer less than 78
27	498	the function evaluator requested that `solvenl_solve()` exit immediately

solvenl_dump( )

void `solvenl_dump(`*S*`)`

`solvenl_dump(`*S*`)` displays the current status of the solver, including initial values, convergence criteria, results, and error messages. This function is particularly useful while debugging.

Conformability

All functions' inputs are 1×1 and return 1×1 or *void* results except as noted below:

`solvenl_init_startingvals(`*S*`,` *ivals*`)`:

S:	*transmorphic*
ivals:	$k \times 1$
result:	*void*

`solvenl_init_startingvals(`*S*`)`:

S:	*transmorphic*
result:	$k \times 1$

`solvenl_init_argument(`*S*`,` *k*`,` *X*`)`:

S:	*transmorphic*
k:	1×1
X:	*anything*
result:	*void*

`solvenl_init_argument(`*S*`,` *k*`)`:

S:	*transmorphic*
k:	1×1
result:	*anything*

`solvenl_solve(`*S*`)`:

S:	*transmorphic*
result:	$k \times 1$

`solvenl_result_values(`*S*`)`:

S:	*transmorphic*
result:	$k \times 1$

`solvenl_result_Jacobian(`*S*`)`:

S:	*transmorphic*
result:	$k \times k$

Diagnostics

All functions abort with an error if used incorrectly.

`solvenl_solve()` aborts with an error if it encounters difficulties. `_solvenl_solve()` does not; instead, it returns a nonzero error code.

The `solvenl_result_*()` functions return missing values if the solver encountered difficulties or else has not yet been invoked.

References

Burden, R. L., D. J. Faires, and A. M. Burden. 2016. *Numerical Analysis*. 10th ed. Boston: Cengage.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. New York: Cambridge University Press.

Also see

[M-4] **mathematical** — Important mathematical functions