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:

```c
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

\[ S = \text{solvenl\_init()} \]

(varies) \text{solvenl\_init\_type}(S [, \{"fixedpoint" | "zero" \}])
(varies) \text{solvenl\_init\_startingvals}(S [, real colvector ivals])
(varies) \text{solvenl\_init\_numeq}(S [, real scalar nvars])
(varies) \text{solvenl\_init\_technique}(S [, "technique"])
(varies) \text{solvenl\_init\_conv\_iterchng}(S [, real scalar itol])
(varies) \text{solvenl\_init\_conv\_nearzero}(S [, real scalar ztol])
(varies) \text{solvenl\_init\_conv\_maxiter}(S [, real scalar maxiter])
(varies) \text{solvenl\_init\_evaluator}(S [, \&evaluator()])
(varies) \text{solvenl\_init\_argument}(S, real scalar k [, X])
(varies) \text{solvenl\_init\_narguments}(S [, real scalar K])
(varies) \text{solvenl\_init\_damping}(S [, real scalar damp])
(varies) \text{solvenl\_init\_iter\_log}(S [, \{"on" | "off" \}])
(varies) \text{solvenl\_init\_iter\_dot}(S [, \{"on" | "off" \}])
(varies) \text{solvenl\_init\_iter\_dot\_indent}(S [, real scalar indent])

\text{real colvector} \ \text{solvenl\_solve}(S)
\text{real scalar} \ \_\text{solvenl\_solve}(S)

\text{real scalar} \ \text{solvenl\_result\_converged}(S)
\text{real scalar} \ \text{solvenl\_result\_conv\_iter}(S)
\text{real scalar} \ \text{solvenl\_result\_conv\_iterchng}(S)
\text{real scalar} \ \text{solvenl\_result\_conv\_nearzero}(S)
\text{real colvector} \ \text{solvenl\_result\_values}(S)
\text{real matrix} \ \text{solvenl\_result\_Jacobian}(S)
\text{real scalar} \ \text{solvenl\_result\_error\_code}(S)
\text{real scalar} \ \text{solvenl\_result\_return\_code}(S)
\text{string scalar} \ \text{solvenl\_result\_error\_text}(S)

\text{void} \ \text{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:

<table>
<thead>
<tr>
<th>technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaussseidel</td>
<td>Gauss–Seidel</td>
</tr>
<tr>
<td>dampedgaussseidel</td>
<td>Damped Gauss–Seidel</td>
</tr>
<tr>
<td>broydenpowell</td>
<td>Broyden–Powell</td>
</tr>
<tr>
<td><em>newtonraphson</em></td>
<td>Newton–Raphson</td>
</tr>
</tbody>
</table>

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

For fixed-point problems, allowed *techniques* are *gaussseidel* and *dampedgaussseidel*. For zero-finding problems, allowed *techniques* 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**

Remarks are presented under the following headings:

- Introduction
- A fixed-point example
- A zero-finding example
- Writing a fixed-point problem as a zero-finding problem and vice versa
- Gauss–Seidel methods
- Newton-type methods
- Convergence criteria
- Exiting early

**Functions**

- *solvenl_init()
- *solvenl_init_type()
- *solvenl_init_startingvals()
- *solvenl_init_numeq()
- *solvenl_init_technique()
- *solvenl_init_conv_iterchg()
- *solvenl_init_conv_nearzero()
- *solvenl_init_conv_maxiter()
- *solvenl_init_evaluator()
- *solvenl_init_argument() and *solvenl_init_narguments()
- *solvenl_init_damping()
- *solvenl_init_iter_log()
- *solvenl_init_iter_dot()
- *solvenl_init_iter_dot_indent()
- *solvenl_solve() and _solvenl_solve()
- *solvenl_result_converged()
- *solvenl_result_conv_iter()
- *solvenl_result_conv_iterchg()
- *solvenl_result_conv_nearzero()
- *solvenl_result_values()
- *solvenl_result_Jacobian()
- *solvenl_result_error_code(), . . . _return_code(), and . . . _error_text()
- *solvenl_dump()
Introduction

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

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

A fixed-point example

We want to solve the system of equations

\[
\begin{align*}
x &= \frac{5}{3} - \frac{2}{3}y \\
y &= \frac{10}{3} - \frac{2}{3}x
\end{align*}
\]

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.

```cpp
: 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:

```cpp
: 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:  3.33333333
Iteration 2:  .833333333
(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 $z$:

\[
\begin{align*}
10 - xe^y - z &= 0 \\
12 - xe^{2y} - 2z &= 0 \\
15 - xe^{3y} - 3z &= 0
\end{align*}
\]

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

```cpp
: 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
  8.771286448
  .2596954499
  -1.372281335
```

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

Earlier, we solved the system of equations

\[
\begin{align*}
x &= \frac{5}{3} - \frac{2}{3}y \\
y &= \frac{10}{3} - \frac{2}{3}x
\end{align*}
\]
by searching for a fixed point. We can rewrite this system as
\[
\begin{align*}
x - \frac{5}{3} + \frac{2}{3}y &= 0 \\
y - \frac{10}{3} + \frac{2}{3}x &= 0
\end{align*}
\]
and then use BP or Newton’s method to find the solution. In general, we simply rewrite \(x^* = F(x^*)\) as \(x^* - F(x^*) = 0\).

Similarly, we may be able to rearrange the constituent equations of a system of the form \(F(x) = 0\) 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 \(x_{i-1}\) denote the previous iteration’s values or the initial values, and let \(x_i\) denote the current iteration’s values. The Gauss–Jacobi method simply iterates on \(x_i = F(x_{i-1})\) by evaluating each equation in order. The Gauss–Seidel method implemented in `solvenl()` instead uses the new, updated values of \(x_i\) that are available for equations 1 through \(j-1\) when evaluating equation \(j\) at iteration \(i\).

For damped Gauss–Seidel, again let \(x_i\) denote the values obtained from evaluating \(F(x_{i-1})\). However, after evaluating \(F\), dGS calculates the new parameter vector that is carried over to the next iteration as
\[
x_i^\# = (1 - \delta)x_i + \delta x_{i-1}
\]
where \(\delta\) is the damping factor. Not fully updating the parameter vector at each iteration helps facilitate convergence in many problems. The default value of \(\delta\) 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 \(\delta\); the current implementation uses the same value of \(\delta\) for all iterations. Increasing the damping factor generally slows convergence by requiring more iterations.

**Newton-type methods**

Newton’s method for solving \(F(x) = 0\) is based on the approximation
\[
F(x_i) \approx F(x_{i-1}) + J(x_{i-1}) \times (x_i - x_{i-1})
\]
where \(J(x_{i-1})\) is the Jacobian matrix of \(F(x_{i-1})\). Rearranging and incorporating a step-length parameter \(\alpha\), we have the iteration
\[
x_i = x_{i-1} - \alpha J^{-1}(x_{i-1}) \times F(x_{i-1})
\]
We calculate \(J\) numerically by using the `deriv()` (see [M-5] deriv()) suite of functions. In fact, we do not calculate the inverse of \(J\); we instead use LU decomposition to solve for \(x_i - x_{i-1}\).

To speed up convergence, we define the function \(f(x) = F(x)'F(x)\) and then choose \(\alpha\) between 0 and 1 such that \(f(x_i)\) is minimized. We use a golden-section line search with a maximum of 20 iterations to find \(\alpha\).
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 $B$ to the Jacobian matrix at each iteration. The BP update is

$$B_i = B_{i-1} + \frac{y_i - B_{i-1}d_i}{d_i'd_i}d_i'$$

where $d_i = x_i - x_{i-1}$ and $y_i = F(x_i) - F(x_{i-1})$. Our initial estimate of the Jacobian matrix is calculated numerically at the initial values by using \texttt{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**

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

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

For BP and Newton’s method, let $f(x_i) = F(x_i)'F(x_i)$. Then convergence is declared if $\texttt{mreldif}(x_i,x_{i-1}) < \texttt{itol}$ or $f(x_i) < \texttt{ztol}$, where \texttt{ztol} is $1e^{-9}$ by default and can be changed by using \texttt{solvenl_init_conv_nearzero()}.\n
**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. \texttt{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:

```c
: 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 \texttt{condition} is true, \texttt{solvenl()} exits, \texttt{solvenl_result_error_code()} returns error code 27, and \texttt{solvenl_result_converged()} returns 0 (indicating a solution has not been found).
Functions

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()

\[
\begin{align*}
\text{void} & \quad \text{solvenl_init_type}(S, \{"fixedpoint" | "zero" \}) \\
\text{string scalar} & \quad \text{solvenl_init_type}(S)
\end{align*}
\]

solvenl_init_type\((S, \text{type})\) specifies whether to find a fixed point or a zero of the function. \( \text{type} \) 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()

\[
\begin{align*}
\text{void} & \quad \text{solvenl_init_startingvals}(S, \text{real colvector ivals}) \\
\text{real colvector} & \quad \text{solvenl_init_startingvals}(S)
\end{align*}
\]

solvenl_init_startingvals\((S, \text{ivals})\) 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()

\[
\begin{align*}
\text{void} & \quad \text{solvenl_init_numeq}(S, \text{real scalar k}) \\
\text{real scalar} & \quad \text{solvenl_init_numeq}(S)
\end{align*}
\]

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 techniques 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 techniques 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(x(i), 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()**

```c
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 `maxiter` is the number set using `set maxiter`, which is 300 by default.

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

**solvenl_init_evaluator()**

```c
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 \( F(x) \). 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()**

```c
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()

```cpp
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()

```cpp
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 unless `set iterlog` is set to off; see [R] set iter.

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

solvenl_init_iter_dot()

```cpp
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()

```cpp
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()

\[
\begin{align*}
\text{real colvector} & \quad \text{solvenl\_solve}(S) \\
\text{void} & \quad \_\text{solvenl\_solve}(S)
\end{align*}
\]

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

_\text{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:

\[
\begin{align*}
\text{solvenl\_init()} \\
\text{solvenl\_init\_numeq()} \\
\text{solvenl\_init\_evaluator()} \\
\text{solvenl\_init\_type()} \text{ or solvenl\_init\_technique()}
\end{align*}
\]

solvenl_result_converged()

\[
\begin{align*}
\text{real scalar} & \quad \text{solvenl\_result\_converged}(S)
\end{align*}
\]

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

solvenl_result_conv_iter()

\[
\begin{align*}
\text{real scalar} & \quad \text{solvenl\_result\_conv\_iter}(S)
\end{align*}
\]

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()

\[
\begin{align*}
\text{real scalar} & \quad \text{solvenl\_result\_conv\_iterchng}(S)
\end{align*}
\]

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()

\[
\begin{align*}
\text{real scalar} & \quad \text{solvenl\_result\_conv\_nearzero}(S)
\end{align*}
\]

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

solvenl_init_startingvals(S, vals):
    S: transmorphic
    vals: $k \times 1$
    result: void

solvenl_init_startingvals(S):
    S: transmorphic
    result: $k \times 1$

solvenl_init_argument(S, k, X):
    S: transmorphic
    k: $1 \times 1$
    X: anything
    result: void

solvenl_init_argument(S, k):
    S: transmorphic
    k: $1 \times 1$
    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


Also see

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

[R] **set iter** — Control iteration settings