Webnucleo Technical Report: wn_sparse_solve Module

Bradley S. Meyer

March 31, 2024

wn_sparse_solve is a Webnucleo module that provides a convenient interface between Webnucleo matrix codes and SPARSKIT2, a toolkit for basic sparse matrix operations by Yousef Saad. As of version 0.8, wn_sparse_solve by default links to SPARSKIT2_F95, a version refactored appropriately for fortran95 by Jaad Tannous. This technical report provides information about the wn_sparse_solve structures, about the way users can provide their own preconditioner routines for solutions to sparse matrix equations, and about stopping criteria. The latest release of wn_sparse_solve is available at

http://wnsparsesolve.sourceforge.net/

1 wn_sparse_solve Structures Overview

wn_sparse_solve has three basic structures. These structures contain the settings for sparse matrix solutions, and the user can update the settings through wn_sparse_solve API routines. The structures are the following:

- WnSparseSolve__Mat

  This structure contains the settings for the SPARSKIT2 basic sparse solver routines, which solve the matrix equation

  $$Ax = b \tag{1}$$

  where A is a matrix, b is the known right-hand-side (rhs) vector, and x is the unknown vector. SPARSKIT2 has a variety of iterative solvers available to find solutions to Eq. (1), and WnSparseSolve__Mat permits Webnucleo module users a straightforward interface to them.

  The user creates a solver with the API routine *WnSparseSolve__Mat__new()*. The default settings for the solver are 1) the solver method is the biconjugate gradient method, 2) the maximum number of iterations the solver will use is 100, 3) the relative tolerance is 10^{-8}, 4) the absolute tolerance is 10^{-8}, 5) there is no preconditioning of the matrix, 6) the stopping criteria

are relative to the initial residual, and 7) debugging is turned off. The user may then update these settings through API routines.

- WnSparseSolve__Exp

  This structure contains the settings for use with SPARSKIT2's routines for solving the matrix equation

$$\frac{dY}{dt} = AY \tag{2}$$

  where Y is a vector, A is a matrix, and t is the time. In general one seeks the solution $Y(t+\Delta t)$ given $Y(t)$. The formal solution is by exponentiation of the matrix:

$$Y(t + \Delta t) = exp(A\Delta t)Y(t) \tag{3}$$

  To find the solution, wn_sparse_solve uses a modified form of the SPARSKIT2 unsupported routine exppro.f. The version of exppro.f used is in the src/ directory in the wn_sparse_solve installation.

  The user creates a new exponentiation solver with the API routine *WnSparseSolve__Exp__new()*. The default settings for the solver are 1) the maximum number of iterations is 100, 2) the tolerance is 10^{-8}, 3) the workspace setting is 40 (the workspace setting must be in the range between 1 and 60, inclusive), and 4) debugging is turned off. The user can update these settings through API routines.

- WnSparseSolve__Phi

  This structure contains the settings for use with SPARSKIT2's routines for solving the matrix equation

$$\frac{dY}{dt} = AY + P \tag{4}$$

  where Y is a vector, A is a matrix, P is a constant matrix, and t is the time. In general one seeks the solution $Y(t + \Delta t)$ given $Y(t)$.

  To find the solution, wn_sparse_solve uses a modified form of the SPARSKIT2 unsupported routine phipro.f. The version of phipro.f used is in the src/ directory in the wn_sparse_solve installation.

  The user creates a new exponentiation plus constant vector solver with the API routine *WnSparseSolve__Phi__new()*. The default settings for the solver are 1) the maximum number of iterations is 100, 2) the tolerance is 10^{-8}, 3) the workspace setting is 40 (the workspace setting must be in the range between 1 and 60, inclusive), and 4) debugging is turned off. The user can update these settings through API routines.

2 Preconditioning

The rate of convergence of an iterative matrix equation solver typically depends on the condition number of the matrix. A larger condition number will generally lead to a slower convergence. Suppose one is solving the matrix equation in Eq. (1). A preconditioner is a matrix P whose inverse is approximately the inverse of A. Applying P^{-1} to A can reduce the condition number and lead to a more rapid convergence of the solution.

A left preconditioner can be applied to the original equation to give

$$P^{-1}Ax = P^{-1}b. \tag{5}$$

Since $P^{-1} \approx A^{-1}$, $P^{-1}b$ is an approximation to x. For a right preconditioner, one writes

$$AP^{-1}Px = b. \tag{6}$$

One again finds $x \approx P^{-1}b$ for $P^{-1} \approx A^{-1}$.

wn_sparse_solve allows a user to supply his or her own left preconditioner. In particular, a user supplies two routines. One takes in a vector y and returns x, where x is the result of solving

$$Px = y. \tag{7}$$

P in this equation is an appropriate left preconditioner. The other routine takes in y and returns x, where x is the result of solving

$$P^T x = y. \tag{8}$$

Here P^T is the transpose of P, the preconditioner. The prototype for both routines is:

```
gsl_vector *
user_routine(
  gsl_vector *p_input_vector,
  void *p_user_data
);
```

The routines take in the vector y as a pointer to a gsl_vector. The user also supplies the preconditioner through his or her own data structure. The routines take in the pointer to the user's data structure, appropriately cast. Typically the user's data structure will supply either P, or, more likely, P^{-1}. Note that the formal solution to Eq. (8) is

$$x = \left(P^T\right)^{-1} y. \tag{9}$$

However,

$$\left(PP^{-1}\right)^T = \left(P^{-1}\right)^T P^T = I, \tag{10}$$

where I is the identity matrix. Hence,

$$\left(P^T\right)^{-1} = \left(P^{-1}\right)^T. \tag{11}$$

A logical strategy then is to provide P^{-1} and to write routines that return $P^{-1}y$ and $\left(P^{-1}\right)^T y$, that is, a matrix vector multiply and a transpose matrix vector multiply. For example, if the matrix A is diagonally dominant, a good choice for P is a simply the matrix with the diagonal elements of A along the diagonals of P and the remaining elements of P equal to zero, that is, the element on row i, column j of P is $P_{i,j} = A_{i,i}\delta_{i,j}$, where $\delta_{i,j}$ is the Krönecker delta. The elements of P^{-1} then are $\left(P^{-1}\right)_{i,j} = \delta_{i,j}/A_{i,j}$.

Once the user has constructed the preconditioner solver and preconditioner transpose solver, he or she passes these in to the sparse matrix solver via the API routines

WnSparseSolve__Mat__updatePreconditionerSolver()

and

WnSparseSolve__Mat__updatePreconditionerTransposeSolver().

The user's data structure is passed into the sparse matrix solver via the API routine

WnSparseSolve__Mat__updatePreconditionerUserData().

The wn_sparse_solve API documentation and the example codes provided in the wn_sparse_solve distribution demonstrate how to carry out these steps.

3 Stopping Criteria

The standard stopping criterion for an iterative solver is when the residual, defined as

$$\text{residual} = ||Ax - b|| \tag{12}$$

reaches a sufficiently small value. In Eq. (12), the $||.||$ denotes the two-norm of a vector. If $v \equiv (v_1, v_2, ..., v_n)$, then

$$||v|| = \sqrt{\sum_{i=1}^{n} v_i^2}. \tag{13}$$

Clearly if the residual is precisely zero, x is the solution to $Ax = b$.

wn_sparse_solve provides three convergence methods. The default method is "initial residual" method. Here convergence is determined by comparing the

residual relative to the initial residual. Thus, if x_{guess} is the user-supplied guess vector, then the initial residual is

$$\text{residual} = ||Ax_{guess} - b|| \tag{14}$$

After i iterations, the solution vector is x_i, and iteration stops if

$$||Ax_i - b|| \leq \epsilon_{rel}||Ax_{guess} - b|| + \epsilon_{abs}. \tag{15}$$

Here ϵ_{rel} is the relative tolerance and ϵ_{abs} is the absolute tolerance.

Instead of the "initial residual" method, the user can use the "rhs" (right-hand-side) method. Here the convergence criterion is determined from the condition

$$||Ax_i - b|| \leq \epsilon_{rel}||b|| + \epsilon_{abs}. \tag{16}$$

The user can switch between these methods via the API routine

WnSparseSolve__Mat__updateConvergenceMethod().

For certain solvers, the user can also choose to supply his or her own convergence tester. The prototype for the user's routine is

```
int
user_tester_routine(
  gsl_vector *p_change_vector,
  gsl_vector *p_solution_vector,
  void *p_user_data
);
```

Here p_solution_vector is a pointer to a gsl_vector containing the current solution vector and p_change_vector is a pointer to a gsl_vector containing the change in the solution vector over the last iteration. p_user_data is a pointer to a user-defined data structure containing any extra data the user may wish to use in his or her routine. The user's routine must return 0 (false) if the user's convergence criterion is not met and 1 (true) if it is.

Once the user-supplied convergence tester routine is written, the user provides it to the matrix solver via the API routine

WnSparseSolve__Mat__updateConvergenceTester().

The extra data for the convergence tester are set through the API routine

WnSparseSolve__Mat__updateConvergenceTesterUserData().

The wn_sparse_solve API documentation and an example in the wn_sparse_solve distribution demonstrate how to supply a user-defined convergence tester.