Module 5.7: nag_sparse_lin_sys
Sparse Linear System Iterative Solvers

`nag_sparse_lin_sys` provides procedures for the solution of sparse linear systems using iterative methods.

Contents

Introduction

This module contains procedures for the iterative solution of a system of linear equations

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

with or without preconditioning. For this release, methods for those cases where A is real non-symmetric or complex non-Hermitian are provided. The sparse matrix A may be supplied explicitly (see the initialization procedures in the module `nag_sparse_mat`) or you may supply a function to return the result of the multiplication of the sparse matrix by a vector. If preconditioning is used, the preconditioner may be supplied explicitly (see the module `nag_sparse_prec`) or you may supply a procedure to return the solution of the preconditioned system for a given right-hand-side vector.

1 Background

For real non-symmetric and complex non-Hermitian matrices the following methods are available:

restarted generalized minimum residual method (GMRES(m));

conjugate gradient squared method (CGS);

stabilized bi-conjugate gradient method of order ℓ (Bi-CGSTAB (ℓ));

transpose-free quasi-minimal residual method (TFQMR).

1.1 Restarted Generalized Minimum Residual Method (GMRES(m))

The restarted generalized minimum residual method (GMRES(m)) (see Saad and Schultz [2], Barrett *et al.* [1], Dias da Cunha and Hopkins [3]) starts from the residual $r_0 = b - Ax_0$, where x_0 is an initial estimate for the solution (often $x_0 = 0$). An orthogonal basis for the Krylov subspace, span$\{A^k r_0\}$, for $k = 0, 1, 2, \ldots$, is generated explicitly: this is referred to as Arnoldi's method (see Arnoldi [4]). The solution is then expanded onto the orthogonal basis so as to minimize the residual norm $\|b - Ax\|_2$. The lack of symmetry of A implies that the orthogonal basis is generated by applying a 'long' recurrence relation, whose length increases linearly with the iteration count. For all but the most trivial problems, computational and storage costs can quickly become prohibitive as the iteration count increases. GMRES(m) limits these costs by employing a restart strategy: every m iterations at most, the Arnoldi process is restarted from $r_l = b - Ax_l$, where the subscript l denotes the last available iterate. Each group of m iterations is referred to as a 'super-iteration'. The value of m is chosen in advance and is fixed throughout the computation. Unfortunately, an optimum value of m cannot easily be predicted. A smaller number of basis vectors than specified may be generated and used when the stability of the solution process requires this.

1.2 Conjugate Gradient Squared Method (CGS)

The conjugate gradient squared method (CGS) (see Sonneveld [6], Barrett *et al.* [1], Dias da Cunha and Hopkins [3]) is a development of the bi-conjugate gradient method where the non-symmetric Lanczos method is applied to reduce the coefficients matrix to real tridiagonal form: two bi-orthogonal sequences of vectors are generated starting from the residual $r_0 = b - Ax_0$, where x_0 is an initial estimate for the solution (often $x_0 = 0$) and from the *shadow residual* $\hat{r}_0$ corresponding to the arbitrary problem $A^T\hat{x} = \hat{b}$, where $\hat{b}$ can be any vector, but in practice is chosen so that $r_0 = \hat{r}_0$. In the course of the iteration, the residual and shadow residual $r_i = P_i(A)r_0$ and $\hat{r}_i = P_i(A^T)\hat{r}_0$ are generated, where P_i is a polynomial of order i, and bi-orthogonality is exploited by computing the vector product $\rho_i = (\hat{r}_i, r_i) = (P_i(A^T)\hat{r}_0, P_i(A)r_0) = (\hat{r}_0, P_i^2(A)r_0)$. Applying the 'contraction' operator $P_i(A)$ twice, the iteration coefficients can still be recovered without advancing the solution of the shadow problem, which is of no interest. The CGS method often provides fast convergence; however, there is no reason why the contraction operator should also reduce the once reduced vector $P_i(A)r_0$: this may well lead to a highly irregular convergence which may result in large cancellation errors.

1.3 Stabilized Bi-Conjugate Gradient Method of Order ℓ (Bi-CGSTAB (ℓ))

The stabilized bi-conjugate gradient method of order ℓ (Bi-CGSTAB (ℓ)) (see van der Vorst [7], Sleijpen and Fokkema [8], Dias da Cunha and Hopkins [3]) is similar to the CGS method above. However, instead of generating the sequence $\{P_i^2(A)r_0\}$, it generates the sequence $\{Q_i(A)P_i(A)r_0\}$ where the $Q_i(A)$ are polynomials chosen to minimize the residual after the application of the contraction operator $P_i(A)$. Two main steps can be identified for each iteration: an OR (Orthogonal Residuals) step where a basis of order ℓ is generated by a Bi-CG iteration and an MR (Minimum Residuals) step where the residual is minimized over the basis generated, by a method akin to GMRES. For $\ell = 1$, the method corresponds to the Bi-CGSTAB method of van der Vorst [7]. For $\ell > 1$, more information about complex eigenvalues of the iteration matrix can be taken into account, and this may lead to improved convergence and robustness. However, as ℓ increases, numerical instabilities may arise. For this reason, a maximum value of $\ell = 10$ is imposed, but probably $\ell = 4$ is sufficient in most cases. A smaller value of ℓ than specified may be used when the stability of the solution process requires this.

1.4 Transpose-Free Quasi-Minimal Residual Method (TFQMR)

The transpose-free quasi-minimal residual method (TFQMR) (see Freund and Nachtigal [9], Freund [10]) is conceptually derived from the CGS method. The residual is minimized over the space of the residual vectors generated by the CGS iterations under the simplifying assumption that residuals are almost orthogonal. In practice, this is not the case, but theoretical analysis has proved the validity of the method. This has the effect of remedying the rather irregular convergence behaviour with wild oscillations in the residual norm that can degrade the numerical performance and robustness of the CGS method. In general, the TFQMR method can be expected to converge at least as fast as the CGS method, in terms of number of iterations, although each iteration involves a higher operation count. When the CGS method exhibits irregular convergence, the TFQMR method can produce much smoother, almost monotonic convergence curves. However, the close relationship between the CGS and TFQMR method implies that the *overall* speed of convergence is similar for both methods. In some cases, the TFQMR method may converge faster than the CGS method.

1.5 Preconditioning

Faster convergence can often be achieved using a preconditioner (see the module `nag_sparse_prec`), where (1) is replaced by the modified system

$$\bar{A}\bar{x} = \bar{b}. \tag{2}$$

An unsuitable preconditioner or no preconditioning at all may result in a very slow rate or lack of convergence. However, preconditioning involves a trade-off between the reduction in the number of iterations required for convergence and the additional computational costs per iteration. A *left* preconditioner M^{-1} can be used by the GMRES(m), CGS and TFQMR methods, such that $\bar{A} = M^{-1}A \sim I_n$ in (2), where I_n is the identity matrix of order n; a *right* preconditioner M^{-1} can be used by the Bi-CGSTAB (ℓ) method, such that $\bar{A} = AM^{-1} \sim I_n$. These are formal definitions, used only in the design of the algorithms; in practice, only the means to compute the matrix–vector products $v = Au$ and $v = A^H u$ (the latter only being required when an estimate of $\|A\|_1$ or $\|A\|_\infty$ is computed internally), and to solve the preconditioning equations $Mv = u$ are required.

2 Termination Criteria

The procedures provide a choice of termination criteria and the norms used in them. They allow monitoring of the approximate solution and can return estimates of the norm of A and the largest singular value of the preconditioned matrix $\bar{A}$.

For each method, a sequence of solution iterates $\{x_i\}$ is generated such that, hopefully, the sequence of the residual norms $\{\|r_i\|\}$ converges to a required tolerance. Note that, in general, convergence, when it occurs, is not monotonic.

The first termination criterion

$$\|r_k\|_p \leq \tau \left(\|b\|_p + \|A\|_p \, \|x_k\|_p\right) \tag{3}$$

is available for all four methods. In (3), $p = 1$, ∞ or 2 and τ denotes a user-specified tolerance subject to $\max(10, \sqrt{n})\epsilon \leq \tau < 1$, where ϵ is the machine precision. Facilities are provided for the estimation of the norm of the matrix, $\|A\|_1$ or $\|A\|_\infty$, when this is not known in advance, by applying Higham's method (see Higham [5]). Note that $\|A\|_2$ cannot be estimated internally. This criterion uses an error bound derived from backward error analysis to ensure that the computed solution is the exact solution of a problem as close to the original as the termination tolerance requires. Termination criteria employing bounds derived from forward error analysis are not used because any such criteria would require information about the condition number $\kappa(A)$, which is not easily obtainable.

The second termination criterion

$$\|\bar{r}_k\|_2 \leq \tau \left(\|\bar{r}_0\|_2 + \sigma_1(\bar{A}) \, \|\Delta\bar{x}_k\|_2\right) \tag{4}$$

is available for all methods except TFQMR. In (4), $\sigma_1(\bar{A}) = \|\bar{A}\|_2$ is the largest singular value of the (preconditioned) iteration matrix $\bar{A}$. This termination criterion monitors the progress of the solution of the preconditioned system of equations and is less expensive to apply than criterion (3) for the Bi-CGSTAB (ℓ) method with $\ell > 1$. Only the GMRES(m) method provides facilities to estimate $\sigma_1(\bar{A})$ internally, when this is not supplied.

Termination criterion (3) is the recommended choice, despite its additional costs per iteration when using the Bi-CGSTAB (ℓ) method with $\ell > 1$. Also, if the norm of the initial estimate is much larger than the norm of the solution, that is, if $\|x_0\| \gg \|x\|$, a dramatic loss of significant digits could result in complete lack of convergence. The use of criterion (3) will enable the detection of such a situation, and the iteration will be restarted at a suitable point. No such restart facilities are provided for criterion (4).

Optionally, a vector w of user-specified weights can be used in the computation of the vector norms in termination criterion (3), i.e., $\|v\|_p^{(w)} = \|v^{(w)}\|_p$, where $(v^{(w)})_i = w_i \, v_i$, for $i = 1, 2, \ldots, n$. Note that the use of weights will increase the computational costs.

3 Choice of Iterative Method

In general, it is not possible to recommend one method in preference to another. GMRES(m) is often used in the solution of systems arising from PDEs. On the other hand, it can easily stagnate when the size, m, of the orthogonal basis is too small, or the preconditioner is not good enough. CGS can be the fastest method, but the computed residuals can exhibit instability which may greatly affect the convergence and quality of the solution. Bi-CGSTAB (ℓ) seems robust and reliable, but it can be slower than the other methods: if a preconditioner is used and $\ell > 1$, Bi-CGSTAB (ℓ) computes the solution of the preconditioned system $\bar{x}_k = Mx_k$ and the preconditioning equations must be solved to obtain the required solution. The algorithm employed limits to 10% or less, when no intermediate monitoring is requested, the number of times the preconditioner has to be thus applied compared with the total number of applications of the preconditioner. TFQMR can be viewed as a more robust variant of the CGS method: it shares the CGS method speed but avoids the CGS fluctuations in the residual, which may give rise to instability. Also, when the termination criterion (3) is used, the CGS, Bi-CGSTAB (ℓ) and TFQMR methods will restart the iteration automatically, when necessary, in order to solve the given problem.

Procedure: nag_sparse_gen_lin_sol

1 Description

nag_sparse_gen_lin_sol is a generic procedure which uses an iterative method to compute the solution x of $Ax = b$, where A is a real non-symmetric or complex non-Hermitian n by n sparse matrix and b is a given right-hand-side vector. This procedure can be used in two ways. Either A is supplied explicitly as a structure of derived type nag_sparse_mat_real_*wp*/nag_sparse_mat_cmplx_*wp* or A is supplied implicitly via the mandatory function argument mat_vec, which returns Au or $A^T u$ for a given vector u.

2 Usage

```
USE nag_sparse_lin_sys
```

EITHER
`CALL nag_sparse_gen_lin_sol(a,b,x [, optional arguments])`
(when the matrix A is supplied explicitly)

OR
`CALL nag_sparse_gen_lin_sol(mat_vec,b,x [, optional arguments])`
(when the matrix A is supplied implicitly)

2.1 Interfaces

Distinct interfaces are provided for each of the four combinations of the following cases.

Real / complex data

Real data: the arguments b and x are of type real(kind=*wp*), a and p (if present) are of type nag_sparse_mat_real_*wp*.

Complex data: the arguments b and x are of type complex(kind=*wp*), a and p (if present) are of type nag_sparse_mat_cmplx_*wp*.

Explicit / implicit sparse matrix

Explicit: a is supplied explicitly.

Implicit: a is not supplied, but mat_vec is.

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$n \geq 1$ — the order of the matrix A

3.1 Mandatory Arguments

One only of the arguments a or mat_vec must be supplied.

EITHER
supply the sparse matrix explicitly by using

a — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(in)

Input: a structure containing details of the representation of the sparse matrix A.

Constraints: `mat_vec` must not be supplied if `a` is supplied. `a` must be as output from a call to one of the procedures `nag_sparse_mat_init_coo`, `nag_sparse_mat_init_csc`, `nag_sparse_mat_init_csr` or `nag_sparse_mat_init_dia` (see module `nag_sparse_mat`).

OR
supply the sparse matrix implicitly by using

mat_vec — function

The function `mat_vec` is used to perform the matrix vector multiplications, `mat_vec`= Au or `mat_vec`= A^Tu, for the iterative method. The optional arguments `i_mat_comm` and `a_mat_comm` may be used to supply information about the sparse matrix to this function.

Its specification is:

```
function mat_vec(trans,u,i_mat_comm,a_mat_comm)

logical, intent(in) ::  trans
```

Input: specifies whether Au or A^Tu is to be performed.
If `trans` = `.false.`, the matrix-vector multiplication Au is performed;
if `trans` = `.true.`, the transpose matrix-vector multiplication A^Tu is performed.

```
real(kind=wp)/complex(kind=wp), intent(in) ::  u(:)
```

Shape: `u` has shape (n).
Input: the vector u to be pre-multiplied by the sparse matrix.

```
integer, intent(in), optional ::  i_mat_comm(:)
real(kind=wp)/complex(kind=wp), intent(in), optional ::  a_mat_comm(:)
```

Input: you are free to use these arrays to supply information to this procedure.
Constraints: `a_mat_comm` must be of the same type as `u`.

```
real(kind=wp)/complex(kind=wp) ::  mat_vec(SIZE(u))
```

Result: `mat_vec`(i) must contain either $\sum_{j=1}^{n} a_{ij}u_j$ when `trans` = `.false.` or $\sum_{j=1}^{n} a_{ji}u_j$ when `trans` = `.true.`, where a_{ij} is the entry in row i and column j of A.
Constraints: `mat_vec` must be of the same type as `u`.

Constraints: `a` must not be supplied if `mat_vec` is supplied; `u`, `a_mat_comm` and `mat_vec` must be of the same type as `b`.

b(n) — real(kind=*wp*)/complex(kind=*wp*), intent(in)

Input: right-hand-side vector, b.

x(n) — real(kind=*wp*)/complex(kind=*wp*), intent(inout)

Input: an initial approximation to the solution vector, x.
Output: the solution vector.
Constraints: `x` must be of the same type as `b`.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

method — character(len=1), intent(in), optional

Input: indicates the iterative method to be used.

If `method` = `'G'` or `'g'`, restarted Generalized Minimum Residual method (GMRES(m));

if `method` = `'C'` or `'c'`, Conjugate Gradient Squared method (CGS);

if `method` = `'B'` or `'b'`, Stabilized Biconjugate Gradient method of order ℓ (Bi-CGSTAB(ℓ));

if `method` = `'T'` or `'t'`, Transpose-Free Quasi-Minimal Residual method (TFQMR).

Default: `method` = `'G'`.

Constraints: `method` = `'g'`, `'G'`, `'c'`, `'C'`, `'b'`, `'B'`, `'t'` or `'T'`.

i_mat_comm(:) — integer, intent(in), optional

a_mat_comm(:) — real(kind=*wp*)/complex(kind=*wp*), intent(in), optional

Input: you are free to use these arrays to supply information to the procedure `mat_vec`.

Constraints: `i_mat_comm` and `a_mat_comm` must not be present if `a` is supplied; `a_mat_comm` must be of the same type as `b`.

p — type(nag_sparse_mat_real_*wp*)/type(nag_sparse_mat_cmplx_*wp*), intent(in), optional

Input: a structure containing details of the representation of the sparse preconditioning matrix M.

Constraints: `psolve` must not be present if `p` is present; `p` must be of the same type as `a` and `p` must be as output from a call to one of the procedures `nag_sparse_prec_init_jac`, `nag_sparse_prec_init_ssor` or `nag_sparse_prec_init_ilu` (see the module `nag_sparse_prec`).

i_prec_comm(:) — integer, intent(in), optional

a_prec_comm(:) — real(kind=*wp*)/complex(kind=*wp*), intent(in), optional

Input: you are free to use these arrays to supply information to the procedure `psolve`.

Constraints: `i_prec_comm` and `a_prec_comm` must not be supplied unless `psolve` is present; `a_prec_comm` must be of the same type as `b`.

psolve — subroutine, optional

The procedure `psolve` is used to return the solution of the preconditioned system $Mz = r$ or $M^T z = r$, when `p` is not supplied. The optional array arguments `i_prec_comm` and `a_prec_comm` may be used to supply information about the preconditioned system.

Its specification is:

```
subroutine psolve(trans,r,z,i_prec_comm,a_prec_comm)

logical, intent(in) ::  trans
```
Input: specifies whether the preconditioned system or its transpose is to be solved.
If `trans` = `.false.`, the preconditioned system $Mz = r$ is solved;
if `trans` = `.true.`, the transposed preconditioned system $M^T z = r$ is solved.

```
real(kind=wp)/complex(kind=wp), intent(in) ::  r(:)
```
Shape: `r` has shape (n).
Input: the right-hand-side vector r of the preconditioned system.

```
real(kind=wp)/complex(kind=wp), intent(out) ::  z(:)
```
Shape: `z` has shape (n).
Output: the solution vector z of the preconditioned system.
Constraints: `z` must be of the same type as `r`.

```
integer, intent(in), optional ::  i_prec_comm(:)
real(kind=wp)/complex(kind=wp), intent(in), optional ::  a_prec_comm(:)
```
Input: you are free to use these arrays to supply information to this procedure.
Constraints: `a_prec_comm` must be of the same type as `r`.

Constraints: `p` must not be present if `psolve` is present; `r`, `z` and `a_prec_comm` must be of the same type as `b`.

term — integer, intent(in), optional

Input: selects the termination criterion to be used, as defined in the Introduction of this module document.

If `term` = 1, termination criterion (3) is used;
if `term` = 2, termination criterion (4) is used.

Default: `term` = 1.

Constraints: `term` = 1 or 2; `term` = 2 cannot be used when `method` = `'T'` or `'t'`.

norm — character(len=1), intent(in), optional

Input: specifies the matrix and vector norm to be used in the termination criterion when `term` = 1. Ignored if `term` = 2.

If `norm` = `'1'`, `'o'` or `'O'`, the 1-norm, $\|A\|_1$ is used;
if `norm` = `'i'` or `'I'`, the ∞-norm, $\|A\|_\infty$ is used;
if `norm` = `'2'`, `'t'` or `'T'`, the 2-norm, $\|A\|_2$ is used.

Default: `norm` = `'i'` or `'I'`, the infinity norm.

Constraints: `norm` = `'1'`, `'o'`, `'O'`, `'i'`, `'I'`, `'2'`, `'t'` or `'T'`; `a_norm` must be present when `norm` = `'2'`, `'t'` or `'T'`.

wt(n) — real(kind=*wp*), intent(in), optional

Input: a vector of user-supplied weights used in the computation of vector norms for the termination criterion when `term` = 1. Ignored if `term` = 2.

Default: `wt`$(i) = 1.0$, for $i = 1, 2, \ldots, n$.

a_norm — real(kind=*wp*), intent(in), optional

Input: the value of $\|A\|_p$ to be used in the termination criterion when `term` = 1. Ignored if `term` = 2.

Default: the corresponding norm, $\| A \|_1$ or $\| A \|_\infty$, is estimated internally when `norm` = `'i'`, `'I'`, `'1'`, `'o'` or `'O'`.

Constraints: `a_norm`> 0.0; `a_norm` must be present if `term`= 1 and `norm` = `'2'`, `'t'` or `'T'`.

a_norm_out — real(kind=*wp*), intent(out), optional

Output: the internal estimate of $\|A\|_1$ or $\|A\|_\infty$ used in the termination criterion when `term` = 1.

Constraints: `a_norm_out` must not be present when either `a_norm` is present, `term` = 2 or (`term` = 1 and `norm` = `'2'`, `'t'` or `'T'`).

sigma — real(kind=*wp*), intent(in), optional

Input: the value of the largest singular value to be used in the termination criterion when `term` = 2. Ignored if `term` = 1.

Default: this is estimated internally when `method` = `'G'` or `'g'` and `term` = 2.

Constraints: `sigma` > 0.0.

If `term`= 2 and `method`=`'C'`, `'c'`, `'B'` or `'b'`, `a_norm` must be present;

if `term`= 2 and `method`=`'T'` or `'t'`, an error will be raised (see `term`).

sigma_out — real(kind=*wp*), intent(out), optional

Output: the internal estimate of the largest singular value used in the termination criterion when `method`=`'G'` or `'g'` and `term` = 2.

Constraints: `sigma_out` must not be present unless `sigma` is not present, `term` = 2 and `method`=`'G'` or `'g'`.

restart — integer, intent(in), optional

Input: the dimension of the restart subspace when the method is GMRES(m).

Default: `restart` = 20.

Constraints: `restart`> 0. Only used when `method`=`'G'` or `'g'`, otherwise it will be ignored.

l_order — integer, intent(in), optional

Input: the order, ℓ, of the polynomial Bi-CGSTAB(ℓ) method.

Default: `l_order` = 4.

Constraints: $0 <$ `l_order` $\leq \min(n, 10)$. Only used when `method` = `'B'` or `'b'`, otherwise it will be ignored.

tol — real(kind=*wp*), intent(in), optional

Input: determines the tolerance, τ, used for the termination criterion, with $\tau = \max(\texttt{tol}, 10\epsilon, \sqrt{n}\epsilon)$ where ϵ is `EPSILON(1.0_`*wp*`)`.

Default: $\tau = \max(\sqrt{\epsilon}, \sqrt{n}\epsilon)$.

Constraints: $0.0 <$ `tol` < 1.0.

max_iter — integer, intent(in), optional

Input: the maximum number of iterations allowed.

Default: `max_iter` = $\max(500, \sqrt{n})$.

Constraints: `max_iter` ≥ 1.

num_iter — integer, intent(out), optional

Output: the number of iterations performed.

resid_norm — real(kind=*wp*), intent(out), optional

Output: the value of the residual norm on termination.

stop_rhs — real(kind=*wp*), intent(out), optional

Output: the final value of the right-hand-side of the chosen termination criterion.

monit — integer, intent(in), optional

Input: the frequency at which a monitoring step is executed: if `method` = `'CGS'` or `'TFQMR'`, a monitoring step is executed every `monit` iterations; otherwise a monitoring step is executed every `monit` super-iterations (groups of up to m or ℓ super-iterations for GMRES(m) or Bi-CGSTAB(ℓ) respectively). Note that there are some additional computational costs involved in monitoring the solution and residual vectors when the Bi-CGSTAB(ℓ) method is used with $\ell > 1$.

Default: `monit`=0 (i.e., no monitoring is performed).

Constraints: $0 \leq$ `monit` $\leq$ `max_iter`.

unit — integer, intent(in), optional

Input: specifies the Fortran unit number to which all output produced by `nag_sparse_gen_lin_sol` is sent.

Default: `unit`= the default output unit number for the implementation.

Constraints: `unit` ≥ 0. Only used when `monit` is present and > 0.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	The iterative procedure has failed to converge within the maximum number of iterations. The last available iterates are returned.
202	Algorithmic breakdown. The last available iterates are returned, but it is likely that these are completely inaccurate.

Warnings (error%level = 1):

error%code	Description
101	An argument (`restart`) has been supplied when it is not required.
	Execution continues, but this argument is not referenced.
102	An argument (`l_order`) has been supplied when it is not required.
	Execution continues, but this argument is not referenced.
103	An argument (`norm`) has been supplied when it is not required.
	Execution continues, but this argument is not referenced.
104	An argument (`sigma`) has been supplied when it is not required.
	Execution continues, but this argument is not referenced.
105	An argument (`a_norm`) has been supplied when it is not required.
	Execution continues, but this argument is not referenced.
106	An argument (`wt`) has been supplied when it is not required.
	Execution continues, but this argument is not referenced.
107	The required accuracy determined by `tol` has not been achieved.
	However, an acceptable accuracy may have been obtained.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 and 2 of this module document.

6 Description of Printed Output

This section describes the intermediate printout produced by `nag_sparse_gen_lin_sol`. The frequency of printed output is controlled by `monit`.

When `monit` is present and > 0, the following output is produced every `monit` iterations:

`Monitoring at iteration no.`	the iteration count.
`Residual norm`	the current value of the residual norm.
`Solution vector`	the current value of the solution $\mathtt{x}_i$, $i = 1, 2, \ldots, n$.
`Residual vector`	the current value of the residual r_i, $i = 1, 2, \ldots, n$.

After the solution has converged, the final results are printed.

`Final Results`	
`Number of iterations for convergence`	the final iteration count.
`Residual norm`	the final residual norm.
`Right hand side of termination criterion`	final value of the right-hand-side.
`1-norm of matrix A`	the estimate of the 1-norm of A.
`Solution vector`	the solution $\mathtt{x}_i$, $i = 1, 2, \ldots, n$.
`Residual vector`	the residual r_i, $i = 1, 2, \ldots, n$.

7 Further Comments

7.1 Algorithmic Detail

GMRES(m) can estimate internally the maximum singular value σ_1 of the iteration matrix, using $\sigma_1 \sim \|T\|_1$ where T is the upper triangular matrix obtained by QR factorization of the upper Hessenberg matrix generated by the Arnoldi process. The costs of this computation are negligible when compared to the overall costs.

Loss of orthogonality in the GMRES(m) method, or of bi-orthogonality in the Bi-CGSTAB (ℓ) method may degrade the solution and speed of convergence. For both methods, the algorithms employed include checks on the basis vectors so that the number of basis vectors used for a given super-iteration may be less than the value specified in the input parameter `restart` or `l_order`. Also, the CGS, Bi-CGSTAB (ℓ) and TFQMR methods will automatically restart the computation from the last available iterates when the stability of the solution process requires this.

When termination criterion (3) is available, it involves only the residual (or norm of the residual) produced directly by the iteration process. This may differ from the norm of the true residual $\tilde{r}_k = b - Ax_k$, particularly when the norm of the residual is very small. Also, if the norm of the initial estimate of the solution is much larger than the norm of the exact solution, convergence can be achieved despite very large errors in the solution. On the other hand, termination criterion (4) is cheaper to use and inspects the progress of the actual iteration. Termination criterion (3) should be preferred in most cases, despite its slightly larger costs.

Additional matrix-vector products are necessary for the computation of $\|A\|_1$ or $\|A\|_\infty$, when this is required by the termination criterion employed and has not been supplied.

7.2 Timing

The number of operations performed for each iteration is likely to be principally determined by the computation of the matrix-vector products $v = Au$ and by the solution of the preconditioning equation $Mz = r$ in the calling program. Each of these operations is performed once every iteration.

The number of the remaining operations for each iteration is approximately proportional to n.

7.3 Accuracy

On successful completion, the termination criterion is satisfied to within the user-specified tolerance. The number of iterations required to achieve a prescribed accuracy cannot easily be determined at the outset, as it can depend dramatically on the conditioning and spectrum of the preconditioned matrix of the coefficients $\bar{A} = M^{-1}A$ (GMRES(m), CGS and TFQMR methods) or $\bar{A} = AM^{-1}$ (Bi-CGSTAB (ℓ) method).

If the termination criterion $\|r_k\|_p \leq \tau\left(\|b\|_p + \|A\|_p \|x_k\|_p\right)$ is used and $\|x_0\| \gg \|x_k\|$, then the required accuracy cannot be obtained due to loss of significant digits. The iteration is restarted automatically at some suitable point, $x_0 = x_k$ and the computation begins again. For particularly badly scaled problems, more than one restart may be necessary. This does not apply to the GMRES(m) method which, self-restarts every super-iteration. Naturally, restarting adds to computational costs: it is recommended that the iteration should start from a value x_0 which is as close to the true solution $\tilde{x}$ as can be estimated. Otherwise, the iteration should start from $x_0 = 0$.

Example 1: Iterative Solution of a Real, Non-symmetric Sparse System of Linear Equations Using GMRES(m)

Solve a real, non-symmetric sparse system of linear equations $Ax = b$. This example calls the procedures nag_sparse_mat_init_coo and nag_sparse_gen_lin_sol.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_sparse_lin_sys_ex01

  ! Example Program Text for nag_sparse_lin_sys
  ! NAG f190, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_sparse_mat, ONLY : nag_sparse_mat_init_coo, &
   nag_sparse_mat_real_wp => nag_sparse_mat_real_dp, nag_deallocate
  USE nag_sparse_lin_sys, ONLY : nag_sparse_gen_lin_sol
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n, nnz, num_iter
  REAL (wp) :: resid_norm
  TYPE (nag_sparse_mat_real_wp) :: a
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: col(:), row(:)
  REAL (wp), ALLOCATABLE :: b(:), value(:), x(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) &
   'Example Program Results for nag_sparse_lin_sys_ex01'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n, nnz

  ALLOCATE (row(nnz),col(nnz),value(nnz),b(n),x(n))

  DO i = 1, nnz
    READ (nag_std_in,*) value(i), row(i), col(i)
  END DO
  READ (nag_std_in,*) b

  CALL nag_sparse_mat_init_coo(a,n,value,row,col)

  WRITE (nag_std_out,*)
  WRITE (nag_std_out,*) 'Method: GMRES'
  WRITE (nag_std_out,*)

  x = 0.0_wp

  CALL nag_sparse_gen_lin_sol(a,b,x,resid_norm=resid_norm, &
   num_iter=num_iter)

  WRITE (nag_std_out,*) ' Solution'
  WRITE (nag_std_out,'(10F7.1)') x
  WRITE (nag_std_out,'(2x,''residual norm . . .  ='',1PE9.1)') resid_norm
```

```
      WRITE (nag_std_out,'(2x,''number of iterations ='',I4)') num_iter

      CALL nag_deallocate(a)
      DEALLOCATE (row,col,value,b,x)

    END PROGRAM nag_sparse_lin_sys_ex01
```

2 Program Data

```
Example Program Data for nag_sparse_lin_sys_ex01
  8   24         : n, nnz
 -3.   5    7    : value(1), row(1), col(1)
 -4.   4    3
  4.   2    1
 -1.   7    5
  2.   2    5
 -7.   3    3
  2.   8    6
  2.   3    6
  3.   4    1
  3.   8    8
  5.   4    4
 -1.   5    2
  2.   1    1
  8.   5    5
  5.   6    3
  1.   1    8
  2.   6    6
 -5.   7    3
  5.   4    7
 -3.   2    2
  6.   7    7
 -1.   8    2
 -1.   1    4
 -6.   6    1    : value(nnz), row(nnz), col(nnz)
  6.   8.  -9.  46. 17.  21.  22.  34.  : b(1:n)
```

3 Program Results

```
 Example Program Results for nag_sparse_lin_sys_ex01

 Method: GMRES

  Solution
    1.0    2.0    3.0    4.0    5.0    6.0    7.0    8.0
  residual norm . . .  =  2.1E-14
  number of iterations =   8
```

Example 2: Iterative Solution of a Complex Non-Hermitian Sparse System of Linear Equations Using CGS

Solve a complex non-Hermitian sparse system of linear equations $Ax = b$ with a supplied function to perform the matrix-vector multiply. This example calls the single procedure `nag_sparse_gen_lin_sol`.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE sparse_lin_sys_ex02_mod
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  COMPLEX (wp), PARAMETER :: zero = (0.0_wp,0.0_wp)

CONTAINS

  FUNCTION mat_vec(trans,u,i_mat_comm,a_mat_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC CONJG, DOT_PRODUCT, SIZE
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (IN) :: trans
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_mat_comm(:)
    COMPLEX (wp), OPTIONAL, INTENT (IN) :: a_mat_comm(:)
    COMPLEX (wp), INTENT (IN) :: u(:)
    ! .. Function Return Value ..
    COMPLEX (wp) :: mat_vec(SIZE(u))
    ! .. Local Scalars ..
    INTEGER :: i, k1, k2, n, nnz
    ! .. Executable Statements ..
    nnz = SIZE(a_mat_comm)
    n = SIZE(u)
    ! Compute matrix vector product.

    IF ( .NOT. trans) THEN

      DO i = 1, n
        k1 = i_mat_comm(i+nnz)
        k2 = i_mat_comm(i+nnz+1) - 1
        mat_vec(i) = DOT_PRODUCT(CONJG(a_mat_comm(k1:k2)),u(i_mat_comm( &
         k1:k2)))
      END DO

    ELSE

      mat_vec = zero
      DO i = 1, n
        k1 = i_mat_comm(i+nnz)
        k2 = i_mat_comm(i+1+nnz) - 1
        mat_vec(i_mat_comm(k1:k2)) = mat_vec(i_mat_comm(k1:k2)) + &
         u(i)*a_mat_comm(k1:k2)
      END DO
```

```
      END IF

    END FUNCTION mat_vec

  END MODULE sparse_lin_sys_ex02_mod
  PROGRAM nag_sparse_lin_sys_ex02

    ! Example Program Text for nag_sparse_lin_sys
    ! NAG f190, Release 4. NAG Copyright 2000.

    ! .. Use Statements ..
    USE sparse_lin_sys_ex02_mod, ONLY : mat_vec, wp, zero
    USE nag_examples_io, ONLY : nag_std_in, nag_std_out
    USE nag_sparse_lin_sys, ONLY : nag_sparse_gen_lin_sol
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Local Scalars ..
    INTEGER :: i, n, nnz
    ! .. Local Arrays ..
    INTEGER, ALLOCATABLE :: i_mat_comm(:)
    COMPLEX (wp), ALLOCATABLE :: b(:), value(:), x(:)
    ! .. Executable Statements ..
    WRITE (nag_std_out,*) &
     'Example Program Results for nag_sparse_lin_sys_ex02'

    READ (nag_std_in,*)          ! Skip heading in data file
    READ (nag_std_in,*) n, nnz

    ALLOCATE (value(nnz),b(n),x(n),i_mat_comm(nnz+n+1))

    ! The sparse matrix is given row by row in increasing
    ! row order (Compressed Sparse Row format).
    ! i_mat_comm and value are used to store the sparse matrix
    ! information to be passed to the mat_vec function as follows:
    ! .  SIZE(value) = nnz
    ! .  SIZE(i_mat_comm) = nnz+n+1
    ! value(i) and i_mat_comm(i), for i = 1,..,nnz contain the
    ! .   value and column index for entry i
    ! i_mat_comm (nnz+i) j=1,..,n contains the index of first entry of row j
    ! i_mat_comm (nnz+n+1) = nnz+1

    DO i = 1, nnz
      READ (nag_std_in,*) value(i), i_mat_comm(i)
    END DO
    READ (nag_std_in,*) i_mat_comm(nnz+1:)
    READ (nag_std_in,*) b

    WRITE (nag_std_out,*)
    WRITE (nag_std_out,*) 'Method CGS with user supplied mat_vec'
    WRITE (nag_std_out,*)

    x = zero

    CALL nag_sparse_gen_lin_sol(mat_vec,b,x,method='C', &
     i_mat_comm=i_mat_comm,a_mat_comm=value)

    ! Output results

    WRITE (nag_std_out,*) ' Solution'
    WRITE (nag_std_out,'(3X,''('',F4.1,'','',F4.1,'')'')') x
```

```
      DEALLOCATE (b,x,value,i_mat_comm)

    END PROGRAM nag_sparse_lin_sys_ex02
```

2 Program Data

```
Example Program Data for nag_sparse_lin_sys_ex02
   7   18        : n, nnz
 ( 2. ,-1. )  2  : value(1), i_mat_comm(1) containing col(1)
 (-1. , 4. )  4
 ( 5. , 1. )  5
 ( 1. ,-2. )  1
 (-3. , 2. )  3
 ( 2. ,-6. )  6
 ( 7. , 2. )  7
 (-3. ,-2. )  2
 ( 6. ,-3. )  6
 (-1. , 4. )  1
 ( 4. ,-3. )  3
 ( 5. , 4. )  4
 (-5. ,-6. )  6
 ( 6. , 3. )  3
 ( 4. ,-6. )  4
 ( 7. , 6. )  7
 (-1. , 7. )  1
 ( 6. , 7. )  6  : value(nnz), i_mat_comm(nnz) containing col(nnz)
  1   4   8  10
 12  14  17  19  : i_mat_comm(nnz+1:nnz+n+1) containing pntrb(1:n+1)
 ( 12.0,-42.0) ( 43.0, 54.0) (-19.0, 16.0) ( 8.0,-22.0)
 ( -8.0, 45.0) (114.0, 55.0) (-23.0,-26.0)   : b(1:n)
```

3 Program Results

```
 Example Program Results for nag_sparse_lin_sys_ex02

 Method CGS with user supplied mat_vec

  Solution
   ( 1.0,-3.0)
   (-2.0, 4.0)
   ( 3.0,-5.0)
   (-4.0, 6.0)
   ( 5.0,-7.0)
   (-6.0, 1.0)
   ( 7.0,-2.0)
```

Additional Examples

Not all example programs supplied with NAG *fl*90 appear in full in this module document. The following additional examples, associated with this module, are available.

`nag_sparse_lin_sys_ex03`

Iterative solution of a complex, non-Hermitian sparse system of linear equations using incomplete LU preconditioned Bi-CGSTAB(ℓ).

`nag_sparse_lin_sys_ex04`

Iterative solution of a real, non-symmetric sparse system of linear equations using TFQMR with a supplied matrix-vector multiply function.

References

[1] Barrett R, Berry M, Chan T F, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C and van der Vorst H (1994) *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* SIAM, Philadelphia

[2] Saad Y and Schultz M (1986) GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **7** 856–869

[3] Dias da Cunha R and Hopkins T (1994) PIM 1.1 — the the parallel iterative method package for systems of linear equations user's guide — Fortran 77 version *Technical Report* Computing Laboratory, University of Kent at Canterbury, Kent CT2 7NZ, UK

[4] Arnoldi W (1951) The principle of minimized iterations in the solution of the matrix eigenvalue problem *Quart. Appl. Math.* **9** 17–29

[5] Higham N J (1988) FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation *ACM Trans. Math. Software* **14** 381–396

[6] Sonneveld P (1989) CGS, a fast Lanczos-type solver for nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **10** 36–52

[7] van der Vorst H (1989) Bi-CGSTAB, A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems *SIAM J. Sci. Statist. Comput.* **13** 631–644

[8] Sleijpen G L G and Fokkema D R (1993) BiCGSTAB(ℓ) for linear equations involving matrices with complex spectrum *ETNA* **1** 11–32

[9] Freund R W and Nachtigal N (1991) QMR: a Quasi-Minimal Residual Method for Non-Hermitian Linear Systems *Numer.Math.* **60** 315–339

[10] Freund R W (1993) A Transpose-Free Quasi-Mimimal Residual Algorithm for Non-Hermitian Linear Sytems *SIAM J.Sci.Comput.* **14** 470–482