ENHANCED IMPLEMENTATION OF BICGSTAB(ℓ) FOR SOLVING LINEAR SYSTEMS OF EQUATIONS

DIEDERIK R. FOKKEMA*

Abstract. In this paper, we present a FORTRAN implementation of the BiCGstab(ℓ) algorithm. The implementation is based on the power basis variant of BiCGstab(ℓ). This variant is enhanced with a more stable way of determination of the iteration coefficients and with a more reliable update strategy for the residuals. These enhancements improve the accuracy and rate of convergence at almost no additional computational costs.

Key words. Nonsymmetric linear systems, Iterative solvers, BiCGstab(ℓ), Krylov subspace

AMS subject classifications. 65F10

1. Introduction.

The BiCGstab(ℓ) algorithm [6] is an iterative solution method for linear problems

$$Ax = b, \tag{1.1}$$

where A is some given nonsingular $(n \times n)$-matrix and b some given right hand side. Typically n is large and A is sparse. The algorithm belongs to the class of hybrid BiCG methods. The methods in this class iteratively compute, for a given initial guess $\mathbf{x}_0$, approximate solutions $\mathbf{x}_k$ for which the residual $\mathbf{r}_k = b - A\mathbf{x}_k$ can be written formally as

$$\mathbf{r}_k = q_k(A) r_k, \tag{1.2}$$

in which q_k is a polynomial of degree k with $q(0) = 1$, and r_k is the kth BiCG [2, 5] residual. In BiCGstab(ℓ) the polynomial q_k is chosen as a product of locally minimizing polynomials of degree ℓ. More precisely, for $k = m\ell$, with $\mathcal{P}_\ell^1$ the space of polynomials p of degree $\leq \ell$ with $p(0) = 1$,

$$\begin{array}{c} q_k = p_{m,\ell}^{\mathrm{MR}} q_{k-\ell}, \text{ with } p_{m,\ell}^{\mathrm{MR}} \in \mathcal{P}_\ell^1 \text{ such that } \|p(A) q_{k-\ell}(A) r_k\|_2 \\ \text{is minima with respect to } p \in \mathcal{P}_\ell^1 \text{ for } p = p_{m,\ell}^{\mathrm{MR}}. \end{array} \tag{1.3}$$

There are numerous ways to implement this procedure, see, for instance, [11, 6, 10, 4], but basically the iteration steps of a BiCGstab method can be divided into two parts, namely,

(i) *the BiCG part*, in which the BiCG residual is implicitly updated using the short recursions of BiCG

(ii) *the polynomial part*, in which the minimal residual polynomial is constructed and used to update the new residual and corresponding approximate solution.

*ISE Integrated Systems Engineering AG, Technopark Zürich, Technoparkstrasse 1, CH-8005 Zürich, Switzerland. E-mail: `fokkema@ise.ch`.

REMARK 1 In the following, our notation reflects the notation used in the algorithms ALG. 1–3: the index k of the iteration vectors is suppressed. For instance, after the BiCG part in ALG. 1, we have that $\mathbf{r}_0 = q_k(A)r_{k+\ell}$, and after the polynomial part, $\mathbf{r}_0 = q_{k+\ell}(A)r_{k+\ell}$.
Furthermore, in the algorithms, we use the MATLAB index notation.

In ALG. 1 we have pictured an efficient variant of BiCGstab(ℓ) that is suggested in [6, 10]. After the BiCG part (starting with $\mathbf{r}_0 = q_k(A)r_k$) a power basis

$$\mathbf{R} = [\mathbf{r}_0, A\mathbf{r}_0, \ldots, A^\ell \mathbf{r}_0]$$

is available, such that $\mathbf{r}_0 = q_k(A)r_{k+\ell}$. In the polynomial part, this basis $\mathbf{R}$ is then used to minimize the residual with the help of the normal equations, i.e., with $\widehat{\mathbf{R}} = [A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^\ell\mathbf{r}_0]$

$$\mathbf{r}_0 \leftarrow \mathbf{r}_0 - \widehat{\mathbf{R}}(\widehat{\mathbf{R}}^*\widehat{\mathbf{R}})^{-1}\widehat{\mathbf{R}}^*\mathbf{r}_0.$$

The associated approximate solution $\mathbf{x}\,(= \mathbf{x}_{k+\ell})$, and the search direction $\mathbf{u}_0$, needed in the BiCG part, are updated correspondingly.

However, whereas the minimal residual polynomials are optimal for reducing the residual, they are not optimal for an accurate determination of the BiCG iteration coefficients (α and β in ALG. 1) [7, 8]. Inaccurate coefficients are undesirable, because they may disturb the underlying BiCG process, and this may affect the speed of convergence.

In [7, 8] it is argued that, for accurate coefficients (at least locally) one should take a different polynomial, namely, the orthogonal polynomial $p^{\mathrm{OR}}_{m,\ell}$ that is characterized by

$$p^{\mathrm{OR}}_{m,\ell} \in \mathcal{P}^1_\ell \quad \text{for which} \quad p^{\mathrm{OR}}_{m,\ell}(A)\mathbf{r}_0 - [\mathbf{r}_0, A\mathbf{r}_0, \ldots, A^{\ell-1}\mathbf{r}_0] \tag{1.4}$$

Unfortunately, these kind of polynomials may amplify the residual, and this also may affect for the speed of convergence.

The suggestions is then to take a suitable convex combination of $p^{\mathrm{MR}}_{m,\ell}$ and $p^{\mathrm{OR}}_{m,\ell}$ as a compromise. This choice may still amplify the residual, but whereas the amplification by $p^{\mathrm{OR}}_{m,\ell}$ can be unbounded, the amplification by the convex combination is at most $\sqrt{2}$ (for $\Omega = \sqrt{2}/2 \approx 0.7$, see Section 2 below) and usually much less. If this leads to divergence, then increasing the value of ℓ may help. The implementation of this procedure is the subject of Section 2.

REMARK 2 In fact, taking a convex combination may be viewed as a cure for a breakdown possibility in the polynomial part. We do not address the breakdown possibilities in the BiCG recursions.

Of course, speed of convergence is one thing, but accuracy[1] of the solution is desired as well: because the approximate solution and the residual are updated with recursions, rounding errors may cause significant differences in the

[1]We say that an algorithm is *accurate* for a certain problem if the recursively computed residuals $\mathbf{r}_0$ and the true residual $b - A\mathbf{x}$ are of comparable size, i.e., $\|b - A\mathbf{x} - \mathbf{r}_0\|_2$ should be small.

```
Choose an initial guess $\mathbf{x}_0$ and some $\tilde{r}_0$
$\mathbf{r}_0 = K^{-1}(b - A\mathbf{x}_0)$, $\zeta_0 = \|\mathbf{r}_0\|_2$
$\mathbf{u}_0 = 0$, $\alpha = \rho_0 = \omega = 1$, $\zeta = \zeta_0$
while $\zeta > \epsilon\,\zeta_0$ do
    — The BiCG part —
    $\rho_0 = -\omega\rho_0$
    for $j = 0, 1, \ldots, \ell - 1$ do
        $\rho_1 = (\mathbf{r}_j, \tilde{r}_0)$,    $\beta = \alpha(\rho_1/\rho_0)$
        $\rho_0 = \rho_1$
        for $i = 0, 1, \ldots, j$ do
            $\mathbf{u}_i = \mathbf{r}_i - \beta\mathbf{u}_i$
        enddo
        $\mathbf{u}_{j+1} = K^{-1}A\mathbf{u}_j$
        $\sigma = (\mathbf{u}_{j+1}, \tilde{r}_0)$,    $\alpha = \rho_1/\sigma$
        $\mathbf{x} = \mathbf{x} + \alpha\mathbf{u}_0$
        for $i = 0, 1, \ldots, j$ do
            $\mathbf{r}_i = \mathbf{r}_i - \alpha\mathbf{u}_{i+1}$
        enddo
        $\mathbf{r}_{j+1} = K^{-1}A\mathbf{r}_j$
    enddo
    — The polynomial part —
    for $i = 1, 2, \ldots, \ell$ do
        for $j = 1, 2, \ldots, i$ do
            $Z(i,j) = \overline{Z(j,i)} = (\mathbf{r}_j, \mathbf{r}_i)$
        enddo
        $y(i) = (\mathbf{r}_0, \mathbf{r}_i)$
    enddo
    $y = Z^{-1}y$,    $\omega = y(\ell)$
    for $i = 1, 2, \ldots, \ell$ do
        $\mathbf{u}_0 = \mathbf{u}_0 - y(i)\mathbf{u}_i$
        $\mathbf{x} = \mathbf{x} + y(i)\mathbf{r}_{i-1}$
        $\mathbf{r}_0 = \mathbf{r}_0 - y(i)\mathbf{r}_i$
    enddo
    $\zeta = \|\mathbf{r}_0\|_2$
endwhile
```

ALG. 1. *Left preconditioned BiCGstab(ℓ) with power basis and normal equations. The matrix K is a preconditioner for A.*

recursively computed residual $\mathbf{r}_0$ and the *true* residual $b - A\mathbf{x}$. As an inspection of the algorithm shows, possible rounding errors in the approximation are not corrected in the update for the residual (see also [9, 3]).

Since the residual is usually involved in some kind of stopping criterion, this is may cause a premature termination of the algorithm: the true residual $b - A\mathbf{x}$

and therefore the approximate solution $\mathbf{x}$ does not satisfy the stopping criterion.

A naive strategy to overcome this problem would be to replace the computed residual by the true residual. Not only is this expensive because an extra matrix multiplication is needed in each iteration, but these true residuals do not satisfy the given BiCG recursions and this may even destroy the convergence eventually.

In [9], it is shown that an occasional replacement of the recursively computed residual by the true residual at strategic points during the iterations may lead to more accurate solutions, while maintaining the speed of convergence. However, if these replacements are performed for residuals much smaller than the initial residual, then also the update of the approximate solution requires a special treatment. This is done by accumulating groups of updates for updating the approximation. The implementation of such a strategy is the subject of Section 3.

The remainder of this paper is organized as follows. In Section 4 we present a FORTRAN implementation that incorporates these enhancements. In Section 5 we present some numerical examples. Section 6 contains our conclusions.

2. Maintaining the convergence. In [7, 8] the following convex combination for a more stable determination of the BiCG coefficients is proposed: in the polynomial part, take $\mathbf{r}_0 \leftarrow p(A)\mathbf{r}_0$ with

$$p = \frac{1-\widehat{\omega}\gamma}{1-\widehat{\omega}^2}\, p^{\mathrm{MR}}_{m,\ell} + \frac{\widehat{\omega}\gamma - \widehat{\omega}^2}{1-\widehat{\omega}^2}\, p^{\mathrm{OR}}_{m,\ell} \tag{2.1}$$

where

$$\widehat{\omega} := \frac{\|p^{\mathrm{MR}}_{m,\ell}(A)\mathbf{r}_0\|_2}{\|p^{\mathrm{OR}}_{m,\ell}(A)\mathbf{r}_0\|_2} \quad \text{and} \quad \gamma := \max\,(\widehat{\omega},\, 0.7). \tag{2.2}$$

An equivalent formulation (cf. [7]), more suitable for implementation, is the following. With

$$\begin{aligned}
\widehat{\mathbf{R}} &= [A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^{\ell-1}\mathbf{r}_0],\\
\tilde{\mathbf{r}}_0 &= \mathbf{r}_0 - \widehat{\mathbf{R}}(\widehat{\mathbf{R}}^*\widehat{\mathbf{R}})^{-1}\widehat{\mathbf{R}}^*\mathbf{r}_0,\\
\tilde{\mathbf{r}}_\ell &= A^\ell\mathbf{r}_0 - \widehat{\mathbf{R}}(\widehat{\mathbf{R}}^*\widehat{\mathbf{R}})^{-1}\widehat{\mathbf{R}}^*A^\ell\mathbf{r}_0,
\end{aligned}$$

take

$$\mathbf{r}_0 \leftarrow \tilde{\mathbf{r}}_0 - \hat{\gamma}\frac{\|\tilde{\mathbf{r}}_0\|_2}{\|\tilde{\mathbf{r}}_\ell\|_2}\,\tilde{\mathbf{r}}_\ell, \tag{2.3}$$

where

$$\hat{\gamma} := (\varrho/|\varrho|)\ \max\,(|\varrho|,\, 0.7), \quad \text{and} \quad \varrho := \frac{(\tilde{\mathbf{r}}_0, \tilde{\mathbf{r}}_\ell)}{\|\tilde{\mathbf{r}}_0\|_2\,\|\tilde{\mathbf{r}}_\ell\|_2}. \tag{2.4}$$

This strategy can be incorporated by replacing the polynomial part in algorithm ALG. 1 by the algorithm in ALG. 2.

```
— $Z = \mathbf{R}^*\mathbf{R}$ —
for $i = 0, 1, \dots, \ell$ do
    for $j = 0, 1, \dots, i$ do
        $Z(i+1, j+1) = \overline{Z(j+1, i+1)} = (\mathbf{r}_j, \mathbf{r}_i)$
    enddo
enddo
— $\tilde{\mathbf{r}}_0$ and $\tilde{\mathbf{r}}_\ell$ —
$y_0 = (-1,\ (Z(2{:}\ell, 2{:}\ell)^{-1} Z(2{:}\ell, 1))^*,\ 0)^*$
$y_\ell = (0,\ (Z(2{:}\ell, 2{:}\ell)^{-1} Z(2{:}\ell, \ell+1))^*,\ -1)^*$
— Convex combination —
$\kappa_0 = \sqrt{y_0^* Z y_0}, \quad \kappa_\ell = \sqrt{y_\ell^* Z y_\ell}, \quad \varrho = \frac{y_\ell^* Z y_0}{\kappa_0 \kappa_\ell}$
$\hat{\gamma} = (\varrho/|\varrho|)\ \max(|\varrho|, 0.7),$
$y_0 = y_0 - \hat{\gamma} \frac{\kappa_0}{\kappa_\ell} y_\ell$
— Update —
$\omega = y_0(\ell+1)$
for $i = 1, 2, \dots, \ell$ do
    $\mathbf{u}_0 = \mathbf{u}_0 - y_0(i+1)\mathbf{u}_i$
    $\mathbf{x} = \mathbf{x} + y_0(i+1)\mathbf{r}_{i-1}$
    $\mathbf{r}_0 = \mathbf{r}_0 - y_0(i+1)\mathbf{r}_i$
enddo
$\zeta = \sqrt{y_0^* Z y_0}$
```

ALG. 2. *Convex combination of* $p^{MR}_{m,\ell}(A)\mathbf{r}_0$ *and* $p^{OR}_{m,\ell}(A)\mathbf{r}_0$.

The implementation follows from the observation that with

$$\mathbf{R} = [\mathbf{r}_0, A\mathbf{r}_0, \dots, A^\ell \mathbf{r}_0] \quad \text{and} \quad Z = \mathbf{R}^*\mathbf{R},$$

$\tilde{\mathbf{r}}_0$ is given by

$$\tilde{\mathbf{r}}_0 = \mathbf{R} y_0, \quad \text{for} \quad y_0 = (1,\ -(Z(2{:}\ell, 2{:}\ell)^{-1} Z(2{:}\ell, 1))^*,\ 0)^*$$

and, similarly, $\tilde{\mathbf{r}}_\ell$ is given by

$$\tilde{\mathbf{r}}_\ell = \mathbf{R} y_\ell, \quad \text{for} \quad y_\ell = (0,\ -(Z(2{:}\ell, 2{:}\ell)^{-1} Z(2{:}\ell, \ell+1))^*,\ 1)^*.$$

The inner product $(\tilde{\mathbf{r}}_0, \tilde{\mathbf{r}}_\ell)$ and the norms $\|\tilde{\mathbf{r}}_0\|_2$ and $\|\tilde{\mathbf{r}}_\ell\|_2$ follow from observing that:

$$(\tilde{\mathbf{r}}_0, \tilde{\mathbf{r}}_\ell) = y_\ell^* \mathbf{R}^* \mathbf{R} y_0 = y_\ell^* Z y_0,$$
$$\|\tilde{\mathbf{r}}_0\|_2 = \sqrt{y_0^* \mathbf{R}^* \mathbf{R} y_0} = \sqrt{y_0^* Z y_0},$$
$$\|\tilde{\mathbf{r}}_\ell\|_2 = \sqrt{y_\ell^* \mathbf{R}^* \mathbf{R} y_\ell} = \sqrt{y_\ell^* Z y_\ell}.$$

This implies hardly any additional costs (see also TAB. 3.1).

3. Reliable updates.

In [9] the accuracy of computed residuals is addressed. Efficient and easy to implement strategies are proposed that improve the accuracy significantly, while maintaining the speed of convergence.

Such a strategy can be incorporated as displayed in ALG. 3. For performing a group wise update of the approximate solution, we have chosen the following conditions (cf. [9]):

$$\text{if } (\|\mathbf{r}_0\|_2 < \delta\zeta_0 \;\&\; \zeta_0 \leq M(x)) \text{ then } \textit{`update\_app'} = \texttt{`true'}, \tag{3.1}$$

where ζ_0 is the norm of the initial residual, $\delta = 10^{-2}$, and $M(x)$ is the maximum of the norm of the residuals since the last group wise update of the approximation. For replacing the recursively computed residual by the true residual, we have chosen the following conditions (cf. [9]):

$$\text{if} \left\{ \begin{array}{l} (\|\mathbf{r}_0\|_2 < \delta M(r) \;\&\; \zeta_0 \leq M(r)) \\ \text{or } \textit{`update\_app'} = \texttt{`true'} \end{array} \right\} \text{then } \textit{`compute\_res'} = \texttt{`true'}, \tag{3.2}$$

where ζ_0 is the norm of the initial residual, $\delta = 10^{-2}$, and $M(r)$ is the maximum of the norm of the residuals since the last computation of the true residual. This combination of conditions for updating the solution and the replacement of the recursively computed residual by the true residual gives a compromise between costs and accuracy (cf. [9]).

METHOD	COMPUTATIONAL COSTS			MEMORY
	MVs	AXPYS	DOTS	REQUIREMENTS
BiCGstab(ℓ)	2ℓ	$\ell^2 + 5\ell$	$\ell^2/2 + 7/2\ell + 1$	$2\ell + 3$
Enhanced BiCGstab(ℓ)	2ℓ $(*)$	$\ell^2 + 5\ell$	$\ell^2/2 + 9/2\ell + 1$ $(*)$	$2\ell + 5$

The '$(*)$' indicates that additional costs are involved when a reliable update is done: 1 axpy + 1 MV, when the residual is replaced; 1 axpy + 2 copies, when the solution is updated.

TABLE 3.1. *Computational costs per iteration.*

4. Description of FORTRAN code.

Here we present a FORTRAN code for the enhanced version of BiCGstab(ℓ). In TAB. 3.1 an overview of the computational costs per iteration for the unmodified and for the enhanced version is given. As we see, the additional costs are low: 2 extra vectors for keeping track of the groups of updates for the approximate solution ($\mathbf{x}'$ and b'); ℓ dots for determining $M(x)$ and $M(r)$; and some costs for when a reliable update is performed.

The calling sequence and the parameters are explained below. The FORTRAN code itself uses subroutines and functions from LAPACK [1].

```
Choose an initial guess $\mathbf{x}_0$ and some $\tilde{r}_0$
$\mathbf{r}_0 = K^{-1}(b - A\mathbf{x}_0)$, $\zeta_0 = \|\mathbf{r}_0\|_2$
$\mathbf{u}_0 = 0$, $\alpha = \rho_0 = \omega = 1$, $\zeta = \zeta_0$
$\mathbf{x}' = \mathbf{x}_0$,   $\mathbf{x} = 0$,   $b' = \mathbf{r}_0$
$k = -\ell$
while $\zeta > \epsilon\,\zeta_0$ do
    — The BiCG part —
    $\rho_0 = -\omega\rho_0$
    for $j = 0, 1, \ldots, \ell - 1$ do
        $\rho_1 = (\mathbf{r}_j, \tilde{\mathbf{r}}_0)$,   $\beta = \alpha(\rho_1/\rho_0)$
        $\rho_0 = \rho_1$
        for $i = 0, 1, \ldots, j$ do
            $\mathbf{u}_i = \mathbf{r}_i - \beta\mathbf{u}_i$
        enddo
        $\mathbf{u}_{j+1} = K^{-1}A\mathbf{u}_j$
        $\sigma = (\mathbf{u}_{j+1}, \tilde{\mathbf{r}}_0)$,   $\alpha = \rho_1/\sigma$
        $\mathbf{x} = \mathbf{x} + \alpha\mathbf{u}_0$
        for $i = 0, 1, \ldots, j$ do
            $\mathbf{r}_i = \mathbf{r}_i - \alpha\mathbf{u}_{i+1}$
        enddo
        $\mathbf{r}_{j+1} = K^{-1}A\mathbf{r}_j$
    enddo
    — The polynomial part —
    Compute convex combination as in ALG. 2
    — The reliable update part —
    set 'update_app' (cf. (3.1)) and 'compute_res' (cf. (3.2))
    if 'compute_res' = 'true'
        $\mathbf{r}_0 = b' - A\mathbf{x}$
        if 'update_app' = 'true'
            $\mathbf{x}' = \mathbf{x}' + \mathbf{x}$,   $\mathbf{x} = 0$,   $b' = \mathbf{r}_0$
        endif
    endif
    'compute_res' = 'update_app' = 'false'
endwhile
$\mathbf{x} = \mathbf{x}' + \mathbf{x}$
```

ALG. 3. *Enhanced left preconditioned BiCGstab(ℓ). The matrix K is a preconditioner for A.*

BISTBL BISTBL — Left Preconditioned Bi-Conjugate Gradient Stabilized iterative method for solving linear systems $Ax = b$.

Declaration subroutine **bistbl** (ℓ, n, x, b, mv, solve, tol, mxmv, work, ldw, rwork, ldrw, iwork, info)

Parameters

integer $\boldsymbol{\ell}$

On entry, ℓ specifies the degree of the polynomial ($\ell > 1$). Suggested values are $\ell = 1, 2, 4, 8$. Unchanged on exit.

integer **n**

On entry, n specifies the dimension of the matrix A ($n > 1$). Unchanged on exit.

x

double precision array of size n. On entry, the array x has the value of the initial guess to the solution, e.g., $x = 0$. On exit, if $info = 0$, x is overwritten by the approximate solution.

b

double precision array of size n. On entry, the array b has the value of the right hand side of the linear problem $Ax = b$. Unchanged on exit.

external **mv**

external subroutine `mv(n, x, y)`. Must be supplied by the user and should return the vector $y = Ax$ of size n.

external **solve**

external subroutine `solve(n, x)`. Must be supplied by the user and should return $x \leftarrow K^{-1}x$, where K is a preconditioner for A.

double precision **tol**

On entry, $tol > 0$ specifies the stopping tolerance. On exit, if $info = 0$, tol has the value of the relative norm of the true residual.

integer **mxmv**

On entry, $mxmv$ specifies the maximum number of matrix multiplications. On exit, if $info = 0$, $mxmv$ has the value of the number of matrix multiplications actually performed.

work

double precision array of size ldw. Workspace for $(3+2(\ell+1))$ vectors of size n.

integer **ldw**

On entry, ldw specifies the length of $work$. $ldw \geq (3 + 2(\ell + 1))n$. Unchanged on exit.

rwork

double precision array of size *ldrw*. Workspace for three vectors of size $\ell+1$ and two matrices of size $(\ell+1)\times(\ell+1)$.

integer **ldrw**

On entry, *ldrw* specifies the length of *rwork*. $ldrw \geq (3+2(\ell+1))(\ell+1)$. Unchanged on exit.

iwork

integer array of size $\ell+1$. Workspace for one pivot array of size $\ell+1$

integer **info**

On exit, *info* defines the exit code.

info	Description
< 0	If $info = -i$, the ith argument had an illegal value.
0	The method was successful.
1	The method did not meet the specified tolerance within the specified number of matrix multiplications.
2	Breakdown; a division by zero occurred.

Note The implementation does not handle Lanczos and pivot breakdowns.

The FORTRAN code

```
      subroutine bistbl (l, n, x, b, mv, solve, tol,
     $     mxmv, work, ldw, rwork, ldrw, iwork, info)
c
c subroutine bistbl v1.0 1995
c
c Copyright (c) 1995 by D.R. Fokkema.
c Permission to copy all or part of this work is granted,
c provided that the copies are not made or distributed
c for resale, and that the copyright notice and this
c notice are retained.
c
c THIS WORK IS PROVIDED ON AN "AS IS" BASIS.  THE AUTHOR
c PROVIDES NO WARRANTY WHATSOEVER, EITHER EXPRESSED OR IMPLIED,
c REGARDING THE WORK, INCLUDING WARRANTIES WITH RESPECT TO ITS
c MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE.
c
      implicit none
```

```
c
c     .. Parameters ..
c
      integer l, n, mxmv, ldw, ldrw, iwork(l+1), info
      double precision x(n), b(n), tol
      double precision work(n,3+2*(l+1)), rwork(l+1,3+2*(l+1))
c
c     .. Matrix ..
c
      external mv
      external solve
c
c     .. Local ..
c
      logical rcmp, xpdt
      integer i, j, k, nmv
      double precision alpha, beta, omega, rho0, rho1, sigma
      double precision varrho, hatgamma
      double precision rnrm0, rnrm
      double precision mxnrmx, mxnrmr, kappa0, kappal
c
c     .. Work Aliases ..
c
      integer z, zz, y0, yl, y
      integer rr, r, u, xp, bp
c
c     .. Constants ..
c
      double precision  zero, one, delta
      parameter (zero = 0d0, one = 1d0, delta = 1d-2)
c
c     .. BLAS and LAPACK ..
c
c     subroutine daxpy
c     subroutine dcopy
c     subroutine dgemv
c     subroutine dgetrf
c     subroutine dgetrs
c     subroutine dlacpy
c     subroutine dlaset
c     subroutine dsymv
c     function ddot
c     function dnrm2
c
      double precision dnrm2, ddot
c
c     .. Intrinsic ..
c
      intrinsic abs, max, sqrt
c
c     ===========================
c     .. Executable Statements ..
```

```
c     ===========================
c
      info = 0

      if (l.lt.1) info = -1
      if (n.lt.1) info = -2
      if (tol.le.zero) info = -7
      if (mxmv.lt.0) info = -8

      rr = 1
      r = rr+1
      u = r+(l+1)
      xp = u+(l+1)
      bp = xp+1
      if (bp*n.gt.ldw) info = -10

      z = 1
      zz = z+(l+1)
      y0 = zz+(l+1)
      yl = y0+1
      y = yl+1
      if (y*(l+1).gt.ldrw) info = -12

      if (info.ne.0) return
c
c     --- Initialize first residual
c
      call mv (n, x, work(1,r))
      do i=1,n
         work(i,r) = b(i) - work(i,r)
      enddo
      call solve (n, work(1,r))
c
c     --- Initialize iteration loop
c
      nmv = 0

      call dcopy (n, work(1,r), 1, work(1,rr), 1)
      call dcopy (n, work(1,r), 1, work(1,bp), 1)
      call dcopy (n, x, 1, work(1,xp), 1)
      call dlaset ('n', n, 1, zero, zero, x, 1)
      rnrm0 = dnrm2 (n, work(1,r), 1)
      rnrm = rnrm0

      mxnrmx = rnrm0
      mxnrmr = rnrm0
      rcmp = .false.
      xpdt = .false.

      alpha = zero
      omega = one
      sigma = one
      rho0 = one
c
c     --- Iterate
c
      do while (rnrm.gt.tol*rnrm0 .and. nmv.lt.mxmv)
c
c     ====================
c     --- The BiCG part ---
c     ====================
c
         rho0 = -omega*rho0
         do k=1,l
            rho1 = ddot (n, work(1,rr), 1, work(1,r+k-1), 1)
            if (rho0.eq.zero) then
               info = 2
               return
            endif
            beta = alpha*(rho1/rho0)
            rho0 = rho1
            do j=0,k-1
               do i=1,n
                  work(i,u+j) = work(i,r+j) - beta*work(i,u+j)
               enddo
            enddo
            call mv (n, work(1,u+k-1), work(1,u+k))
            call solve (n, work(1,u+k))
            nmv = nmv+1
            sigma = ddot (n, work(1,rr), 1, work(1,u+k), 1)
            if (sigma.eq.zero) then
               info = 2
               return
            endif
            alpha = rho1/sigma
            call daxpy (n, alpha, work(1,u), 1, x, 1)
            do j=0,k-1
               call daxpy (n, (-alpha), work(1,u+j+1), 1,
     $              work(1,r+j), 1)
            enddo
            call mv (n, work(1,r+k-1), work(1,r+k))
            call solve (n, work(1,r+k))
            nmv = nmv+1
            rnrm = dnrm2 (n, work(1,r), 1)
            mxnrmx = max (mxnrmx, rnrm)
            mxnrmr = max (mxnrmr, rnrm)
         enddo
c
c     ==================================
c     --- The convex polynomial part ---
c     ==================================
c
c        --- Z = R'R
c
```

```
            do i=1,l+1
               call dgemv ('t', n, l+1-(i-1), one, work(1,r+i-1),
     $              n, work(1,r+i-1), 1, zero, rwork(i,z+i-1), 1)
               call dcopy (l-(i-1), rwork(i+1,z+i-1), 1,
     $              rwork(i,z+i), l+1)
            enddo
            call dlacpy ('a', l+1, l+1, rwork(1,z), l+1,
     $           rwork(1,zz), l+1)
            call dgetrf (l-1, l-1, rwork(2,zz+1), l+1,
     $           iwork, info)
c
c           --- tilde r0 and tilde rl (small vectors)
c
            rwork(1,y0) = -one
            call dcopy (l-1, rwork(2,z), 1, rwork(2,y0), 1)
            call dgetrs ('n', l-1, 1, rwork(2,zz+1), l+1, iwork,
     $           rwork(2,y0), l+1, info)
            rwork(l+1,y0) = zero

            rwork(1,yl) = zero
            call dcopy (l-1, rwork(2,z+l), 1, rwork(2,yl), 1)
            call dgetrs ('n', l-1, 1, rwork(2,zz+1), l+1, iwork,
     $           rwork(2,yl), l+1, info)
            rwork(l+1,yl) = -one
c
c           --- Convex combination
c
            call dsymv ('u', l+1, one, rwork(1,z), l+1,
     $           rwork(1,y0), 1, zero, rwork(1,y), 1)
            kappa0 = sqrt(ddot (l+1, rwork(1,y0), 1,
     $           rwork(1,y), 1))

            call dsymv ('u', l+1, one, rwork(1,z), l+1,
     $           rwork(1,yl), 1, zero, rwork(1,y), 1)
            kappal = sqrt(ddot (l+1, rwork(1,yl), 1,
     $           rwork(1,y), 1))

            call dsymv ('u', l+1, one, rwork(1,z), l+1,
     $           rwork(1,y0), 1, zero, rwork(1,y), 1)
            varrho =
     $           ddot (l+1, rwork(1,yl), 1, rwork(1,y), 1)
     $           / (kappa0*kappal)

            hatgamma =
     $           varrho/abs(varrho)*max(abs(varrho),7d-1)
     $           * (kappa0/kappal)

            call daxpy (l+1, (-hatgamma), rwork(1,yl), 1,
     $           rwork(1,y0), 1)
c
c           --- Update
c
```

```
         omega = rwork(l+1,y0)

         call dgemv ('n', n, l, (-one), work(1,u+1), n,
     $        rwork(2,y0), 1, one, work(1,u), 1)
         call dgemv ('n', n, l, one, work(1,r), n,
     $        rwork(2,y0), 1, one, x, 1)
         call dgemv ('n', n, l, (-one), work(1,r+1), n,
     $        rwork(2,y0), 1, one, work(1,r), 1)

         call dsymv ('u', l+1, one, rwork(1,z), l+1,
     $        rwork(1,y0), 1, zero, rwork(1,y), 1)
         rnrm = sqrt (ddot (l+1, rwork(1,y0), 1,
     $        rwork(1,y), 1))
c
c     ================================
c     --- The reliable update part ---
c     ================================
c
         mxnrmx = max (mxnrmx, rnrm)
         mxnrmr = max (mxnrmr, rnrm)
         xpdt = (rnrm.lt.delta*rnrm0.and.rnrm0.lt.mxnrmx)
         rcmp = ((rnrm.lt.delta*mxnrmr.and.rnrm0.lt.mxnrmr)
     $        .or.xpdt)
         if (rcmp) then
            call mv (n, x, work(1,r))
            call solve (n, work(1,r))
            do i=1,n
               work(i,r) =  work(i,bp) - work(i,r)
            enddo
            mxnrmr = rnrm
            if (xpdt) then
               call daxpy (n, one, x, 1, work(1,xp), 1)
               call dlaset ('n', n, 1, zero, zero, x, 1)
               call dcopy (n, work(1,r), 1, work(1,bp), 1)
               mxnrmx = rnrm
            endif
         endif
      enddo
c
c     ========================
c     --- End of iterations ---
c     ========================
c
      call daxpy (n, one, work(1,xp), 1, x, 1)
c
c     --- Check stopping criterion
c
      call mv (n, x, work(1,r))
      do i=1,n
         work(i,r) = b(i) - work(i,r)
      enddo
      call solve (n, work(1,r))
      rnrm = dnrm2 (n, work(1,r), 1)
      if (rnrm.gt.tol*rnrm0) info = 1
c
c     --- Return
c
      tol = rnrm/rnrm0
      mxmv = nmv

      return
      end
```

5. Numerical experiments. In this section we compare the performance of the unmodified BiCGstab(ℓ) version (ALG. 1) and the enhanced version (ALG. 3). We consider two of the linear problems that are also used in [9, 7]. In both cases no preconditioning is used. The computations where done in double precision (≈ 15 digits) on a Sun workstation.

5.1. Example 1. With this example we show that taking a convex combination of the MR polynomial and the OR polynomial may indeed cure stagnation.

The linear systems stems from a 65×65 finite volume discretization on the unit square of the partial differential equation

$$-\Delta u + 100(xu_x + yu_y) - 200u = f,$$

where f is such that $u(x,y) \equiv 1$ is the solution.

In FIG. 5.1 and 5.1 we have plotted the number of matrix multiplications versus the $\log_{10}$ of the *true* residual norm for the unmodified and for the enhanced version of BiCGstab(ℓ) for $\ell = 1, 2, 4, 8$, respectively. FIG. 5.1 and 5.1 display plots for the number of flops versus the true residual norm. The maximum number of matrix multiplications was set to 300.

From these figures we see that the unmodified version of BiCGstab(1) stagnates, whereas the enhanced version does converge. Apparently, taking the convex combination of the MR polynomial and the OR polynomial of degree 1 cures the stagnation. For larger values of ℓ the unmodified version converges increasingly better, but still they converge not as fast as their enhanced counterparts.

In number of matrix multiplications, the enhanced version converges comparable for all considered values of ℓ. Looking at the number of flops we see that the enhanced BiCGstab(1) algorithm is the most efficient.

5.2. Example 2. With this example we show that taking the convex combination does not always cure stagnation. In this case increasing the value of ℓ helps. Moreover, the reliable update strategy in the enhanced version results in a more accurate solution.

The linear systems stems from a 65×65 finite volume discretization on the unit square of the partial differential equation

$$-\Delta u + 1000(xu_x + yu_y) + 10u = f,$$

where f is such that $u(x,y) \equiv 1$ is the solution.

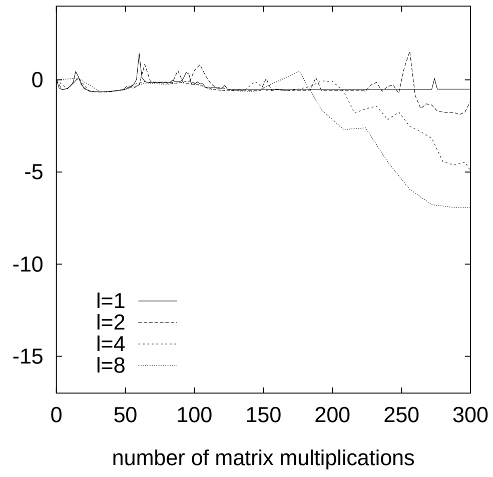

FIG. 5.1. *Number of MVs versus* $\log_{10}$ *of the true residual norm for BiCGstab(ℓ) for example 1.*

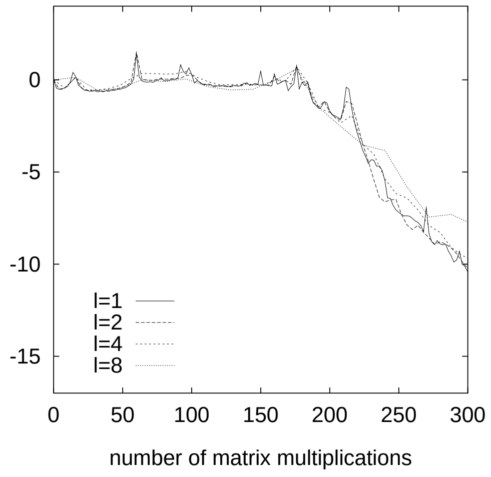

FIG. 5.2. *Number of MVs versus* $\log_{10}$ *of the true residual norm for Enhanced BiCGstab(ℓ) for example 1.*

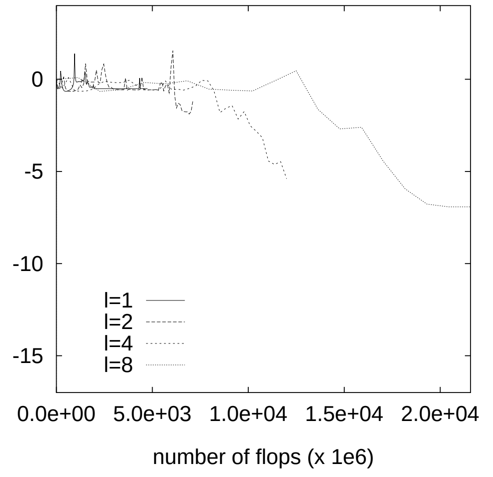

FIG. 5.3. *Number of flops versus* $\log_{10}$ *of the true residual norm for Enhanced BiCGstab(ℓ) for example 1.*

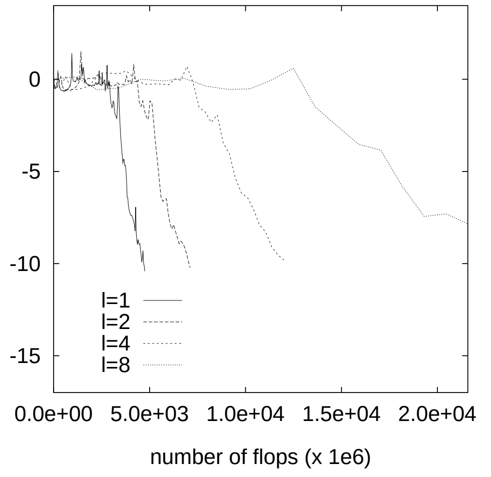

FIG. 5.4. *Number of flops versus* $\log_{10}$ *of the true residual norm for Enhanced BiCGstab(ℓ) for example 1.*

In FIG. 5.2 and 5.2 we have plotted the number of matrix multiplications versus the $\log_{10}$ of the *true* residual norm for the unmodified and for the enhanced version of BiCGstab(ℓ) for $\ell = 1, 2, 4, 8$, respectively. FIG. 5.2 and 5.2 display plots for the number of flops versus the $\log_{10}$ of the true residual norm. The maximum number of matrix multiplications was set to 1000.

From these figures we see that taking the convex combination of the MR polynomial and the OR polynomial of degree 1 does not cure the stagnation and leads to divergence. Increasing ℓ however, results in increasingly better convergence (in terms of MVs) for the unmodified version. The convergence for the enhanced version is for $\ell = 2, 4, 8$ comparable. Apparently, the BiCG

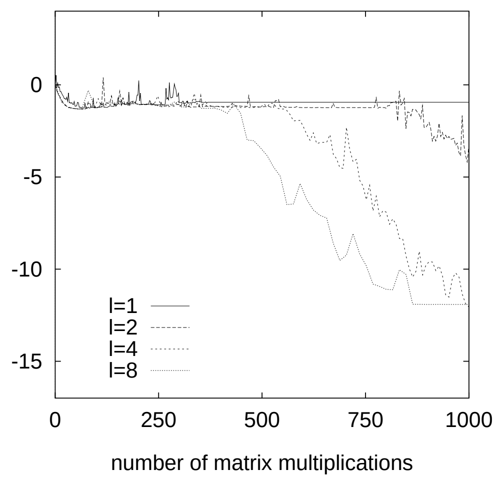

FIG. 5.5. *Number of MVs versus* $\log_{10}$ *of the true residual norm for BiCGstab(ℓ) for example 2.*

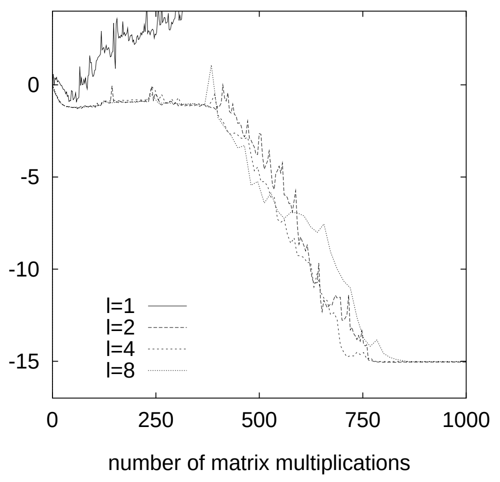

FIG. 5.6. *Number of MVs versus* $\log_{10}$ *of the true residual norm for Enhanced BiCGstab(ℓ) for example 2.*

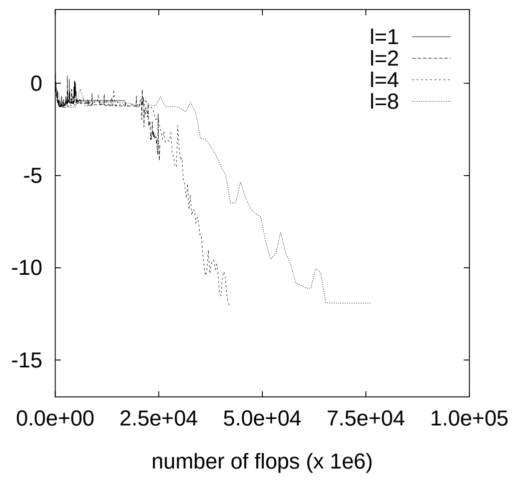

FIG. 5.7. *Number of flops versus* $\log_{10}$ *of the true residual norm for Enhanced BiCGstab(ℓ) for example 2.*

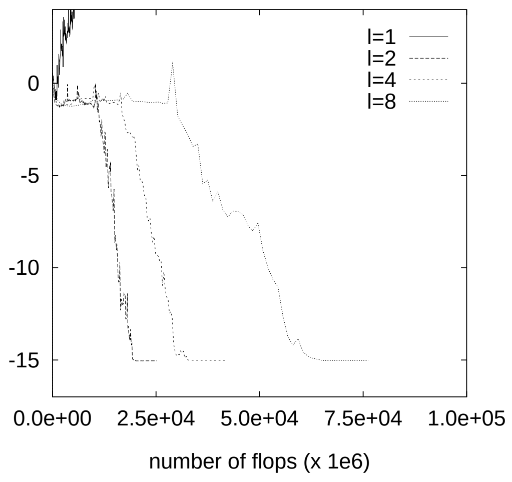

FIG. 5.8. *Number of flops versus* $\log_{10}$ *of the true residual norm for Enhanced BiCGstab(ℓ) for example 2.*

iteration coefficients of the enhanced version are already accurate for $\ell = 2$ and increasing ℓ does not lead to better convergence. Notice that with the reliable update strategy we obtain almost full precision.

6. Conclusions. We have presented an enhanced FORTRAN implementation of the BiCGstab(ℓ) algorithm. The enhancements consist of two parts: (1) the accuracy of the BiCG iteration coefficients (and thereby the convergence behavior) is improved by taking a suitable combination of minimal residual (MR-) and orthogonal (OR-) polynomials in the polynomial part of the BiCGstab(ℓ) algorithm, and (2) the accuracy of the approximate solution is improved by incorporating a "reliable update" strategy. The additional computational costs involved are small and easily compensated for by the much better overall performance.

Taking a suitable convex combination of MR- and OR-polynomials may cure stagnation (as is sometimes observed for BiCGstab(1), for instance). It may also lead to divergence, but then increasing the value of ℓ usually helps.

The reliable update strategy occasionally replaces the recursively computed residual by the the true residual at strategic chosen points, thereby improving the accuracy of the approximate solution without affecting the speed of convergence. A stopping criterion that involves the residual is therefore much more reliable.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[2] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Numerical Analysis Dundee 1975, Lecture Notes in Mathematics 506, G. A. Watson, ed., Berlin, Heidelberg, New York, 1976, Springer-Verlag, pp. 73–89.

[3] A. Greenbaum, *Estimating the attainable accuracy of recursively computed residual methods*, preprint, Courant Institute of Math. Sc., 1995.

[4] M. H. Gutknecht, *Variants of BiCGStab for matrices with complex spectrum*, SIAM J. Sci. Comput., 14 (1993), pp. 1020–1033.

[5] C. Lanczos, *Solution of systems of linear equations by minimized iteration*, J. Res. Nat. Bur. Standards, 49 (1952), pp. 33–53.

[6] G. L. G. Sleijpen and D. R. Fokkema, *BiCGstab(ℓ) for linear equations involving matrices with complex spectrum*, Electronic Transactions on Numerical Analysis, 1 (1993), pp. 11–32.

[7] G. L. G. Sleijpen and H. A. Van der Vorst, *Maintaining convergence properties of BiCGstab methods in finite precision arithmetic*, Numerical Algorithms, 10 (1995), pp. 203–223.

[8] ———, *An overview of approaches for the stable computation of hybrid BiCG methods*, Preprint 908, Department of Mathematics, Utrecht University, Utrecht, The Netherlands, March 1995. To appear in Appl. Numer. Math.

[9] ———, *Reliable updated residuals in hybrid Bi-CG methods*, Computing, 56 (1996), pp. 141–163.

[10] G. L. G. Sleijpen, H. A. Van der Vorst, and D. R. Fokkema, *BiCGstab(ℓ) and other hybrid Bi-CG methods*, Numerical Algorithms, 7 (1994), pp. 75–109.

[11] H. A. Van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 631–644.