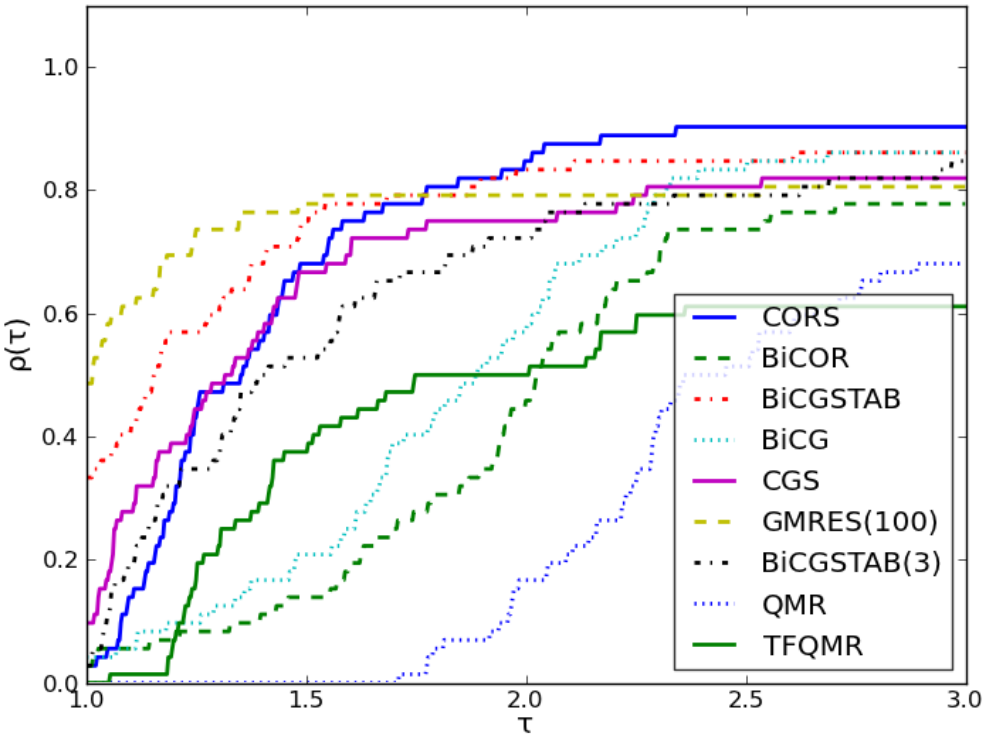
rijksuniversiteit groningen

faculteit Wiskunde en Natuurwetenschappen

Performance analysis of the CORS and BiCOR iterative methods for solving nonsymmetric sparse linear systems



Bachelor Thesis in Applied Mathematics

August 2011

Student: S. Baars

Supervisor: Dr. B. Carpentieri

Abstract

Recently, the iterative methods BiCOR and CORS for solving real nonsymmetric (or complex non-Hermitian), and possibly indefinite sparse linear systems were developed. There is not much known yet about the performance of those methods. We consider iterative methods in general, and go more into detail about CORS and BiCOR. We analyse the performance of BiCOR and CORS by comparing them to seven popular solvers on a large set of publicly available matrices coming from different areas of application. We use different qualities of preconditioners to do this. In our experiments we observe that CORS is a highly competitive solver compared to other popular solvers, like GMRES and BiCGSTAB.

Keywords: CORS, BiCOR, GMRES, BiCGSTAB; iterative methods, Krylov subspace methods; performance profiles; preconditioning.

Contents

1 Introduction

Computational simulation of scientific and engineering problems often involves solving large systems of equations of the form

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

with $A \in \mathbb{C}^{m\times n}$, $x \in \mathbb{C}^n$ and $b \in \mathbb{C}^n$. The usual way of solving small systems of linear equations of the form (1.1) is by using Gaussian elimination. Gaussian elimination, however, as well as other direct methods, has a cost of $\mathcal{O}(n^3)$[33]. This is really expensive if the order n of the matrix A is large and also unnecessarily expensive if the matrix is sparse, i.e. it contains many zero entries. If the matrix is sparse, not only the computational cost is expensive, but also the storage cost in the memory. For direct methods it is usually still needed to store n^2 entries in the memory, where one would like to only store the $\mathcal{O}(n)$ nonzero entries in the matrix. In this case it might be useful to use an iterative method, and a Krylov subspace method in particular.

Developing those methods is a continuously evolving subject of research. Recently, a new family of iterative methods were developed around the two-sided A-orthonormalisation procedure that will be introduced in this thesis. To date, very little is known about the performance of those methods. We test the performance of two of those methods, BiCOR and CORS, and compared those with some other popular iterative methods. We do this mostly using various qualities of preconditioners. To compare BiCOR and CORS to the other iterative methods, we used a FORTRAN implementation, that we also provide here.

As an introduction to the subject, we discuss iterative methods, also called Krylov methods, mostly following Van der Vorst in [40]. Then we show how the BiCOR and CORS method can be derived from the two-sided A-orthonormalisation procedure following [8], and finally, we analyse the results of our experiments.

2 Krylov subspace methods

2.1 The Krylov subspace

The general idea behind iterative methods is that we want to solve the system $Ax = b$, and at each iteration i, we have an approximate solution x_i. We can also write this as $x = x_i + \epsilon_i$ where ϵ_i is the error at step i. Multiplication by A gives us

$$A\epsilon_i = A(x - x_i) = b - Ax_i.$$

Since we do not have the real solution, we do not know the actual error either. Instead we try to solve the system

$$Mz_i = b - Ax_i$$

for z_i, with M an approximation of A that makes the system easier to solve. If we take $x_0 = 0$ for instance, the first step would be solving $Mz_0 = b$. Since M is an approximation of A, z_i is an approximation of the error. Thus solving the easier system leads to a better approximation of the solution: $x_{i+1} = x_i + z_i$. The basic iteration introduced here, now leads to

$$x_{i+1} = x_i + M^{-1}(b - Ax_i),$$

where M is called the preconditioner. One uses a preconditioner to speed up convergence. An iterative method converges fast when $M^{-1}A$ is close to identity. If $M^{-1}A$ was equal to identity,

we would have convergence in one step. We only write the inverse of M for notational purposes. In practice, M^{-1} is usually not calculated. For more information about preconditioners, see section 2.5.

If we now take $M = I$, we obtain the well known Richardson iteration [40]

$$x_{i+1} = b + (I - A)x_i = x_i + r_i,$$

with $r_i = b - Ax_i$ the residual at step i. We try to find a relation between r_{i+1} and r_i by multiplying the above relation by $-A$ and adding b to it

$$b - Ax_{i+1} = b - Ax_i - Ar_i$$

so

$$r_{i+1} = (I - A)r_i = (I - A)^{i+1}r_0.$$

It then follows that the approximate solution x_{i+1} may be written as

$$x_{i+1} = r_0 + r_1 + \ldots + r_i = \sum_{k=0}^{i}(I - A)^k r_0$$

for $x_0 = 0$. We can do this without loss of generality, because in case x_0 is nonzero, we could just shift the system by setting $Ay = b - Ax_0 = \hat{b}$ with $y_0 = 0$. We now observe that

$$x_{i+1} \in \text{Span}\left\{r_0, Ar_0, \ldots, A^i r_0\right\} \equiv \mathcal{K}_{i+1}(A; r_0).$$

The space of dimension m, spanned by a given vector v, and increasing powers of A applied to v up to the $(m-1)$th power of A is called the m-dimensional Krylov subspace generated by A and v, and is denoted as $\mathcal{K}_m(A; v)$ [13, 40].

2.2 Arnoldi's method

Assuming the matrix A has n eigenvalues $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n| \geq 0$, and linearly independent eigenvectors $\{v_1, v_2, \ldots, v_n\}$ with $Av_i = \lambda_i v_i$, we may write the solution x to the system $Ax = b$ as

$$x = \sum_{j=1}^{n} \alpha_j v_j.$$

Multiplying both sides by A^k gives

$$A^k x = \sum_{j=1}^{n} \alpha_j A^k v_j = \sum_{j=1}^{n} \alpha_j \lambda_j^k v_j.$$

If we factor out λ_1^k from the right hand side

$$A^k x = \lambda_1^k \sum_{j=1}^{n} \alpha_j \frac{\lambda_j^k}{\lambda_1^k} v_j$$

we see that, since $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n| \geq 0$, this converges to

$$A^k x = \lambda_1^k \alpha_1 v_1$$

as $k \to \infty$ and assuming that $\alpha_1 \neq 0$ [7]. This only holds because $|\lambda_1|$ has to be strictly greater than $|\lambda_2|$. Otherwise we would not able to factor out the eigenvectors belonging to λ_2 to λ_n. This is the idea behind the basic iteration for finding eigenpairs (θ_m, v_m) through the Power Method.

Something we observe, is that the obvious basis $\{r_0, Ar_0, \ldots, A^{m-1}r_0\}$ for the Krylov subspace $\mathcal{K}_m(A; r_0)$ is not very attractive. The vectors $A^j r_0$ point more and more in the direction of the dominant eigenvector for increasing j. Hence, the basis vectors will become linearly dependent in finite precision arithmetic. This is why we want to make sure that we have an orthogonal basis for our Krylov subspace [40].

A way to form an orthogonal basis for the Krylov subspace is suggested by Arnoldi [1]. Arnoldi's method often uses modified Gram-Schmidt orthogonalisation in the process of finding this basis. Modified Gram-Schmidt orthogonalisation is used, because normal Gram-Schmidt orthogonalisation can produce linearly dependent vectors due to rounding errors. We first describe this process before describing Arnoldi's method itself.

In the modified Gram-Schmidt process [31], we start with the vector

$$q_1 = \frac{1}{\|x_1\|} x_1,$$

where $x_1, \ldots, x_n$ form an ordinary basis. Now, at the beginning of the $(k+1)$-th step, the projections of the vector x_{k+1} along the vectors $q_i, \ldots, q_k$ are progressively subtracted from x_{k+1}. We can write the first step of the subtraction within the $(k+1)$-th step of the process itself as

$$x_{k+1}^{(1)} = x_{k+1} - q_1^T x_{k+1} q_1.$$

This new vector $x_{k+1}^{(1)}$ is then projected along q_2 and subtracted again from $x_{k+1}^{(1)}$, yielding

$$x_{k+1}^{(2)} = x_{k+1}^{(1)} - q_2^T x_{k+1}^{(1)} q_2.$$

We can continue this process until $x_{k+1}^{(k)}$ is computed. q_{k+1} is then given as

$$q_{k+1} = \frac{1}{\|x_{k+1}^{(k)}\|} x_{k+1}^{(k)}.$$

We can now describe the procedure of Arnoldi's method as follows. We start with $v_1 = r_0/\|r_0\|_2$. Then we compute Av_1, make it orthogonal to v_1, and normalise the result using the modified Gram Schmidt process described above. This gives us v_2. In general, we have an orthonormal basis $v_1, \ldots v_j$ for our Krylov subspace $\mathcal{K}_j(A; r_0)$. We expand this basis by calculating $t = Av_j$ and orthonormalising this vector t with respect to the basis $v_1, \ldots, v_j$. This leads to an algorithm as seen in Algorithm 1 to form a basis $v_1, \ldots, v_m$ for $\mathcal{K}_m(A; r_0)$. The orthogonalisation can be done in different ways, but the most common way is to use the modified Gram-Schmidt process [33, 40].

We clearly see here that the matrix A is only accessed through matrix-vector products, which is an advantage compares to direct methods, where the matrix is used directly. This allows us to specify our own matrix-vector product if we store the matrix for example in a sparse format.

Now let V_j denotes the matrix that has columns $v_1, \ldots, v_j$. We then see from Arnoldi's method that

$$AV_{m-1} = V_m H_{m,m-1} \tag{2.1}$$

Algorithm 1 *Arnoldi's method using modified Gram-Schmidt orthogonalisation.*

$v_1 = r_0/\|r_0\|_2$
for $j = 1, 2, \ldots, m-1$ **do**
  $t = Av_j$
  for $i = 1, \ldots, j$ **do**
    $h_{i,j} = v_i^T t$
    $t = t - h_{i,j} v_i$
  end for
  $h_{j+1,j} = \|t\|_2$
  $v_{j+1} = t/h_{j+1,j}$
end for

where $H_{m,m-1}$ is an upper Hessenberg matrix, which means that $h_{i,j} = 0$ for $i > j + 1$. The other entries of $H_{m,m-1}$ are defined by Arnoldi's method. We see that the orthogonalisation becomes increasingly expensive for increasing dimension of the subspace, since every iteration needs one extra inner product and vector update compared to the last iteration to compute a new column of $H_{m,m-1}$.

If A is symmetric however, so is $H_{m-1,m-1} = V_{m-1}^T A V_{m-1}$. Therefore, in this case, $H_{m-1,m-1}$ is tridiagonal. This means that during the orthogonalisation process, most inner products vanish, so the work does not increase. The resulting three-term recurrence relation is known as the Lanczos method [29] that has some well known methods derived from it. The constant amount of work that has to be done during every iteration in the Lanczos method is one matrix vector product, two inner products and two vector updates.

2.3 Different approaches

Methods that attempt to generate an approximate solution from the Krylov subspace, like Arnoldi's method, are usually referred to as Krylov subspace methods. There are four classes of Krylov subspace methods that can be distinguished:

- The *Ritz-Galerkin approach*: Require for x_k that the residual is orthogonal to the current subspace: $b - Ax_k \perp \mathcal{K}_k(A; r_0)$.
- The *minimum norm residual approach*: Require for x_k that the Euclidean norm $\|b - Ax_k\|_2$ is minimal over $\mathcal{K}_k(A; r_0)$.
- The *Petrov-Galerkin approach*: Require for x_k that the residual $b - Ax_k$ is orthogonal to some other suitable k-dimensional subspace.
- The *minimum norm error approach*: Require for x_k in $A^T\mathcal{K}_k(A^T; r_0)$ that the Euclidean norm $\|x_k - x\|_2$ is minimal.

The Ritz-Galerkin approach leads to methods such as the Lanczos method mentioned before and the Conjugate Gradient (CG) method [21]. The minimum norm residual approach leads to methods such as the Generalised Minimum Residual (GMRES) method [36]. The minimum norm error approach leads to some less well known methods that will not be discussed in this paper. And lastly, the Petrov-Galerkin approach leads to methods such as the Biconjugate Gradient (BiCG) method [15], the Quasi-Minimal Residual (QMR) method [18], and the Biconjugate A-Orthogonal Residual (BiCOR) [25] method we discuss later in this thesis. Other methods like the Conjugate Gradients Squared (CGS) [38], Biconjugate Gradient Stabilised (BiCGSTAB) [39] and BiCGSTAB(ℓ) [37] methods and the Conjugate A-Orthogonal Residual Squared (CORS) method [25] we discuss later, are hybrids of the different approaches.

2.4 The GMRES method

The GMRES method can be derived using the minimum residual approach. It is an optimal method, in the sense that it minimizes the 2-norm of the residual over the corresponding Krylov space. Starting from Arnoldi's method, in (2.1), we had an orthogonal basis for the Krylov subspace of dimension $i+1$, leading to

$$AV_i = V_{i+1}H_{i+1,i}.$$

We are looking for an $x_i \in \mathcal{K}_i(A; r_0)$, such that the residual, $\|b - Ax_i\|_2$, is minimal. Since $x_i \in \mathcal{K}_i(A; r_0)$, we can also write $x_i = V_i y$. The norm of the residual can be rewritten as

$$\|r_i\|_2 = \|b - Ax_i\|_2 = \|b - AV_i y\|_2 = \|\beta V_{i+1}e_1 - V_{i+1}H_{i+1,i}y\|_2,$$

with $\beta \equiv \|r_0\|_2$. Now, since the column vectors of V_{i+1} are orthonormal, we have

$$\|b - Ax_i\|_2 = \|\beta e_1 - H_{i+1,i}y\|_2,$$

which can be solved as a least squares problem. This least squares problem can be solved by making the QR-factorisation of $H_{i+1,i}$, and because of the upper Hessenberg structure, this can be done efficiently using Givens matrices. Givens matrices are elementary rotation matrices of thee form $G(i,k,\theta) = I - Y$, where I is the identity matrix and Y is a null matrix except for the elements $y_{ii} = y_{kk} = 1 - \cos(\theta)$ and $y_{ik} = -y_{ki} = -\sin(\theta)$. The Givens rotations remove the subdiagonal elements from the upper Hessenberg matrix $H_{i+1,i}$, resulting in an upper triangular matrix $R_{i,i}$:

$$H_{i+1,i} = Q_{i+1,i}R_{i,i}$$

where $Q_{i+1,i}$ is the matrix consisting of the product of successive Givens rotations. Now we can write the least squares problem as the minimisation of

$$\begin{aligned}\|\beta e_1 - H_{i+1,i}y\|_2 &= \|\beta e_1 - Q_{i+1,i}R_{i,i}y\|_2 \\ &= \|Q_{i+1,i}^T\beta e_1 - R_{i,i}y\|_2.\end{aligned}$$

This leads to the minimum norm solution

$$y = R_{i,i}^{-1}Q_{i+1,i}^T\beta e_1,$$

where the approximate solution x_i is computed as $x_i = V_i y$.

To lower the storage and computational costs of the orthogonalisation process, GMRES is usually restarted after m steps. This method is referred to as GMRES(m). By restarting GMRES, we lose the optimality property however. The non-restarted version of GMRES is also referred to as full GMRES.

2.5 Preconditioning

In general, all iterative methods we mentioned before converge rapidly if the matrix A of the problem $Ax = b$ from (1.1) is close to identity. If the matrix is equal to the identity matrix, those methods converge in one step. For most problems however, the matrix A is far from being close to identity, and therefore one can not be sure that the iterative methods will compute a good approximation of the solution in $m \ll n$ iterations. Every different method has its own drawbacks. In exact arithmetic, some methods, like full GMRES, lead to the exact solution in

at most n steps, but that might not be very practical. Other methods, like CG, only work for certain kinds of matrices. In the case of CG, the matrix must be symmetric positive definite. There are also methods, like BiCG, BiCGSTAB and the methods discussed in this thesis, CORS and BiCOR, that might suffer from breakdowns or stagnation. The rate of convergence depends in a very complicated way on the spectral properties (eigenvalue distribution, etc.) of the matrix A and in real applications, this information is not available [40].

The trick is to use a preconditioner. A preconditioner $M \approx A$ tries to get the original problem closer to identity, so the spectral properties are better. In general, one can write

$$M_1^{-1} A M_2^{-1} y = M_1^{-1} b$$

where M_1 is the left preconditioner, and M_2 is the right preconditioner with $x = M_2^{-1} y$. If we choose, for example $M_1 = A$ and $M_2 = I$, the problem is solved in one step. Note that if we precondition from the right, the residual stays the same as the residual of the original system, because

$$r = b - A\hat{x} = b - AM_2^{-1} y, \tag{2.2}$$

where $\hat{x}$ is the approximate solution.

Calculating the inverse is usually very expensive, so instead of calculating the exact inverse, one can also try to approximate it. Tools to derive those preconditioners are even more diverse than those used in the derivation of iterative methods [20], and therefore we do not discuss this here.

3 The Petrov-Galerkin projection

3.1 The basics

For nonsymmetric matrices, it is desirable to have a three-term recurrence relation similar to the one from the Lanczos method. Due to the work of Faber and Manteuffel, we know that for nonsymmetric matrices, it is not possible to find short-term recurrence relations while keeping the optimality property as for the GMRES method [14]. To reduce memory usage and computational costs, however, it is still very useful to derive non-optimal methods. Let's start with what we already know from Arnoldi's method. This suggested a basis

$$h_{i+1,i} v_{i+1} = A v_i - \sum_{j=1}^{i} h_{j,i} v_j \tag{3.1}$$

for the Krylov subspace, which could be written as

$$V_{i+1} H_{i+1,i} = A V_i \tag{3.2}$$

in matrix notation. Using V_i for the projection, we would end up making the new vector orthogonal to Krylov subspace like we would do in the Ritz-Galerkin approach, but that's not what we want here. So suppose that we have W_i for which $W_i^T V_i = D_i$ with D_i a diagonal matrix, and for which v_{i+1} is orthogonal to W_i, so $W_i^T v_{i+1} = 0$. Then

$$W_i^T A V_i = D_i H_{i,i}. \tag{3.3}$$

Our goal is to find a W_i such that $H_{i,i}$ is tridiagonal. In this case we would have a three-term recurrence relation. So from the above statement we see that $(W_i^T A V_i)^T = V_i^T A^T W_i$ should

also be tridiagonal. This looks very similar to what we have in (3.3), so this suggests we can write a relation similar to (3.1), but now with W_i.

Let's choose an arbitrary $w_1 \neq 0$ with $w_1^T v_1 \neq 0$. Then we can use (3.1) to generate v_2 and orthogonalise it with respect to w_1. So then from (3.3) we see that $h_{1,1} = w_1^T A v_1/(w_1^T v_1)$. Since

$$\begin{aligned} v_1^T h_{1,1} w_1 &= v_1^T (w_1^T A v_1) w_1/(w_1^T v_1) \\ &= (w_1^T A v_1) v_1^T w_1/(w_1^T v_1) \\ &= w_1^T A v_1 = v_1^T A^T w_1, \end{aligned}$$

we see that w_2 generated from

$$h_{2,1} w_2 = A^T w_1 - h_{1,1} w_1 \tag{3.4}$$

is also orthogonal to v_1. That is,

$$v_1^T w_2 = \frac{1}{h_{2,1}} \left(v_1^T A^T w_1 - v_1^T h_{1,1} w_1 \right) = \frac{1}{h_{2,1}} \left(v_1^T A^T w_1 - v_1^T A^T w_1 \right) = 0.$$

Relation (3.4) indeed looks similar to (3.1).

We can go on with this, and see that we can create bi-orthogonal basis sets $\{v_j\}$ and $\{w_j\}$ by making every new v_{i+1} orthogonal to $w_1, \ldots, w_i$, and then generating w_{i+1} using the same coefficients, but with A^T instead of A.

Now we have that both $W_i^T A V_i = D_i H_{i,i}$ and $V_i^T A^T W_i = D_i H_{i,i}$. This implies that $D_i H_{i,i}$ is symmetric, and hence our Hessenberg matrix $H_{i,i}$ is tridiagonal. This gives us the three-term recurrence relation we wanted with $v_1, \ldots, v_i$ a basis for $\mathcal{K}_i(A; v_1)$ and $w_1, \ldots, w_i$ a basis for $\mathcal{K}_i(A^T; w_1)$. The matrix $H_{i,i}$ is also commonly denoted as $T_{i,i}$ due to its tridiagonal form. The three-term recurrence relation we found does not only save a lot of computational power, but also requires less memory. We only have to store the last three vectors of both of the bases.

The two-sided Lanczos method [30] follows from what we derived above. Since we have a tridiagonal matrix, we can write v_{i+1} and w_{i+1} at step i in the construction of the dual basis as

$$\delta_i v_{i+1} = A v_i - \alpha v_i - \beta_{i-1} v_{i-1}$$

and

$$\delta_i w_{i+1} = A^T w_i - \alpha_i w_i - \beta_{i-1} w_{i-1}.$$

Algorithm 2 *The two-sided Lanczos method.*

1: Choose a v_1 and w_1 such that $w_1^T v_1 = \gamma_1 \neq 0$
2: $\beta_0 = 0, w_0 = v_0 = 0$
3: **for** $j = 1, 2, \ldots$ **do**
4: $\quad p = A v_i - \beta_{i-1} v_{i-1}$
5: $\quad \alpha_i = w_i^T p/\gamma_i$
6: $\quad p = p - \alpha_i v_i$
7: $\quad \delta_{i+1} = \|p\|_2$
8: $\quad v_{i+1} = p/\delta_{i+1}$
9: $\quad w_{i+1} = (A^T w_i - \beta_{i-1} w_{i-1} - \alpha_i w_i)/\delta_{i+1}$
10: $\quad \gamma_{i+1} = w_{i+1}^T v_{i+1}$
11: $\quad \beta_i = \delta_{i+1} \gamma_{i+1}/\gamma_i$
12: **end for**

So here we have $\delta_i = h_{i+1,i}$, $\alpha_i = h_{i,i}$, and $\beta_i = h_{i-1,i}$ where available. The full method is given in Algorithm 2. In this algorithm we also use $\gamma_i = w_i^T v_i$. The only thing we do here is repeatedly calculating v_{i+1} and w_{i+1} using exactly what we derived above.

The method we described above can also be seen as an oblique projection of the residual onto the space orthogonal to the space spanned by W, which is exactly what our initial condition $r_k \perp \mathcal{K}_m(A^T, w_1)$ says. With an oblique projection we mean the projection of a vector, onto a space $\mathcal{K}$ orthogonal to a space $\mathcal{L}^\perp$. We also say the projection is along $\mathcal{L}$ onto $\mathcal{K}$ [35, 6]. Put in another way, oblique projections are projections that are not orthogonal. Orthogonal projections actually project onto the orthogonal space.

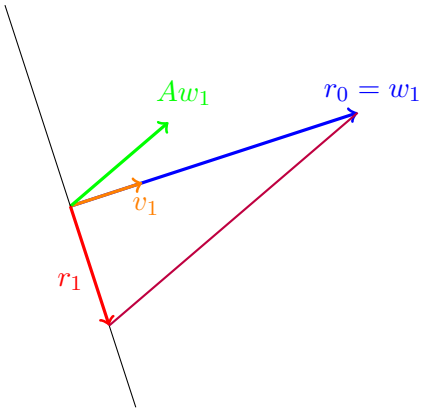


Figure 1: 2D interpretation of the Petrov-Galerkin projection

In the case of the Petrov-Galerkin projection, we are projecting the residual along AW onto the space orthogonal to the space spanned by W, as can be seen in Figure 1. In this figure, the purple line resembles the projection. The image was created for a 2×2 matrix, using the two-sides Lanczos method to construct the bases V and W. We chose $v_1 = r_0/\|r_o\|_2$ and $w_1 = r_0$. After computing V and W, we applied the prototype projection method suggested by Saad in [35]. The prototype projection method can be found in Algorithm 3

Algorithm 3 *Prototype projection method*

1: **for** $i = 1, 2, \ldots,$ until convergence **do**
2: Select a pair of subspaces $\mathcal{K}$ and $\mathcal{L}$
3: Choose bases $V = [v_1, \ldots, v_i]$ and $W = [w_1, \ldots, w_i]$ for $\mathcal{K}$ and $\mathcal{L}$
4: $r_i = b - Ax_i$
5: $y = (W^T AV)^{-1} W^T r_i$
6: $x_{i+1} = x_i + Vy$
7: **end for**

3.2 The two-sided biconjugate A-orthonormalisation method

The two-sided biconjugate A-orthonormalisation method [25] is a method similar to the two-sided Lanczos method, and can, like the two-sided Lanczos method, be used for nonsymmetric matrices. Given two vectors v_1 and w_1 for which $w_1^T A v_1 = 1$, we define two Lanczos-type vectors v_j and w_j very similar to the ones we described in the last section. We again use scalars

α_j, β_j and δ_j. The two vectors are recursively defined as

$$\delta_{j+1} v_{j+1} = A v_j - \beta_j v_{j-1} - \alpha_j v_j, \tag{3.5}$$

$$\beta_{j+1} w_{j+1} = A^T w_j - \delta_j w_{j-1} - \alpha_j w_j \tag{3.6}$$

where the scalars are chosen as

$$\alpha_j = w_j^T A^2 v_j, \quad \beta_j = w_{j-1}^T A^2 v_j, \quad \delta_j = w_j^T A^2 v_{j-1}.$$

The choice of the scalars assures that the vectors v_j and w_j form a biconjugate A-orthonormal basis. So $w_i^T A v_j = \delta_{i,j}$, with $\delta_{i,j}$ the Kronecker delta. The rest of the procedure can be derived from the two-sided Lanczos algorithm. For the sake of clarity, we show a complete version of the process in Algorithm 4.

Algorithm 4 *The biconjugate A-orthonormalisation procedure.*

1: Choose a v_1 and w_1 such that $w_1^T A v_1 = 1$
2: $\beta_0 = \delta_1 = 0, w_0 = v_0 = 0$
3: **for** $j = 1, 2, \ldots$ **do**
4: $\quad \alpha_j = w_j^T A(A v_j))$
5: $\quad \tilde{v}_{j+1} = A v_j - \alpha_j v_j - \beta_j v_{j-1}$
6: $\quad \tilde{w}_{j+1} = A^T w_j - \alpha_j w_j - \delta_j w_{j-1}$
7: $\quad \delta_{j+1} = |\tilde{w}_{j+1}^T A \tilde{v}_{j+1}|^{\frac{1}{2}}$
8: $\quad \beta_{j+1} = \dfrac{\tilde{w}_{j+1}^T A \tilde{v}_{j+1}}{\delta_{j+1}}$
9: $\quad v_{j+1} = \dfrac{\tilde{v}_{j+1}}{\delta_{j+1}}$
10: $\quad w_{j+1} = \dfrac{\tilde{w}_{j+1}}{\beta_{j+1}}$
11: **end for**

The fact that the basis sets $\{v_j\}$ and $\{w_j\}$, generated from relations (3.5) and (3.6), really form a basis of the Krylov subspaces $\mathcal{K}_i(A; v_1)$ and $\mathcal{K}_i(A^T; w_1)$ can be shown in a similar way as we did for the two-sided Lanczos method, and can be found for example in [25]. Additionally, the following relations hold

$$A V_m = V_{m+1} T_{m+1,m}, \tag{3.7}$$

$$A^T W_m = W_{m+1} T_{m,m+1}^T, \tag{3.8}$$

$$W_m^T A V_m = I_m, \tag{3.9}$$

$$W_m^T A^2 V_m = T_m, \tag{3.10}$$

with

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \delta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \delta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \delta_m & \alpha_m \end{pmatrix}$$

and $V_m = [v_1, v_2, \ldots, v_m]$, $W_m = [w_1, w_2, \ldots, w_m]$.

Due to the three-term recurrence relation like the one in the two-sided Lanczos method, we can overwrite for example w_{j-1} with w_{j+1}. After all, we see in Algorithm 4, line 6, that w_{j-1} is not used after $\tilde{w}_{j+1}$ has been computed. An advantage of the three-term recurrence relation is that storage is very limited if you compare it to Arnoldi's method.

The method can possible fail if δ_{j+1} vanishes while $\tilde{w}_{j+1}$ and $A\tilde{v}_{j+1}$ are not the zero vector. One could try to recover from such failures using so-called look-ahead strategies [32] as used in for instance the QMR implementation we use for this thesis.

3.3 The biconjugate A-orthonormalisation procedure for solving general linear systems

As one can derive for instance the Biconjugate Gradient method from the two-sided Lanczos method, one can also derive methods for solving $Ax = b$ from the the Biconjugate A-Orthonormalisation procedure by applying a Petrov-Galerkin projection. We will describe this process in three steps

Step 1 Run Algorithm 4 for $m \ll n$ steps and generate V_m, W_m and T_m as described above.

Step 2 Compute the approximate solution x_m that belongs to the Krylov subspace $x_0 + \mathcal{K}_m(A; v_1)$ by using the Petrov-Galerkin projection to project the residual orthogonally to the space $A^T\mathcal{K}_m(A^T; w_1)$, so

$$r_m = b - Ax_m \perp A^T\mathcal{K}_m(A^T; w_1). \tag{3.11}$$

Using matrix notation, we may also write

$$(A^T W_m)^T(b - Ax_m) = 0$$

and since our approximate solution is of the form

$$x_m = x_0 + V_m y_m \tag{3.12}$$

we get, using (3.10)

$$\begin{aligned}(A^T W_m)^T(b - A(x_0 + V_m y_m)) &= (A^T W_m)^T r_0 - (A^T W_m)^T V_m y_m \\ &= W_m^T A r_0 - T_m y_m = 0\end{aligned}$$

so, if we have $v_1 = r_0/\|r_0\|_2$,

$$T_m y_m = \|r_0\|_2 e_1 \tag{3.13}$$

with e_1 the first canonical unit vector.

Step 3 Compute the new residual, and terminate if it meets the stopping criterion. Otherwise, enlarge the Krylov subspace and start again.

By using this method, we not only solve the system $Ax = b$, but also implicitly the system $A^T x' = b'$. We use the notation used in [8] by denoting vectors belonging to this dual system with primed symbols. We can now say we compute the approximation x'_m that belongs to the Krylov subspace $x'_0 + \mathcal{K}_m(A^T; w_1)$, so we get a relation similar to (3.11)

$$r'_m = b' - A^T x'_m \perp A\mathcal{K}_m(A; v_1).$$

We also get similar relations as above, but now for the dual system

$$(AV_m)^T(b' - A^T x'_m) = 0, \tag{3.14}$$

$$x'_m = x'_0 + W_m y'_m, \tag{3.15}$$

$$T_m^T y'_m = \|r'_0\| e_1. \tag{3.16}$$

We can now update x_m and x'_m from x_{m-1} respectively x'_{m-1}. Assume the LU factorisation of the tridiagonal matrix T_m is

$$L_m U_m = T_m.$$

Substituting this expression in (3.12), (3.13) and (3.15), (3.16), we get

$$\begin{aligned} x_m &= x_0 + V_m(L_m U_m)^{-1}(\|r_0\|_2 e_1) \\ &= x_0 + V_m U_m^{-1} L_m^{-1}(\|r_0\|_2 e_1) \\ &= x_0 + P_m z_m \\ x'_m &= x'_0 + W_m(U_m^T L_m^T)^{-1}(\|r_0\|_2 e_1) \\ &= x'_0 + W_m(L_m^T)^{-1}(U_m^T)^{-1}(\|r_0\|_2 e_1) \\ &= x'_0 + P'_m z'_m \end{aligned}$$

where we take $P_m = V_m U_m^{-1}, z_m = L_m^{-1}(\|r_0\|_2 e_1), P'_m = W_m(L_m^T)^{-1}$ and $z'_m = (U_m^T)^{-1}(\|r'_0\|_2 e_1)$. Because of the structure of U_m, which only has a nonzero diagonal and superdiagonal, we can easily calculate the elements of P_m. We see that $u_{m-1,m}p_{m-1} + u_{m,m}p_m = v_m$, where p_m and v_m are the last column of P_m and V_m respectively, and $u_{i,j}$ is the i,jth element of U_m. Now it follows that

$$p_m = \frac{1}{u_{m,m}}(v_m - u_{m-1,m}p_{m-1})p. \tag{3.17}$$

In addition, because of the structure of L_m, which only has a nonzero diagonal (consisting of only ones) and subdiagonal, we find

$$z_m = (z_{m-1}, \zeta_m)^T$$

in which $\zeta_m = l_{m,m-1}\zeta_{m-1}$, where $l_{i,j}$ is the i,jth element of L_m. If we now substitute this back in the relation $x_m = x_0 + P_m z_m$ found above, we get

$$x_m = x_0 + [P_{m-1}, p_m]\begin{bmatrix} z_{m-1} \\ \zeta_m \end{bmatrix} = x_0 + P_{m-1}z_{m-1} + \zeta_m p_m.$$

Noting that $x_0 + P_{m-1}z_{m-1} = x_{m-1}$, we finally find

$$x_m = x_{m-1} + \zeta_m p_m. \tag{3.18}$$

If we repeat those steps for the dual system, we find a similar relation

$$x'_m = x'_{m-1} + \zeta'_m p'_m. \tag{3.19}$$

This is the same derivation as used for IOM and DIOM in [35].

We now state two propositions that we will use in the derivation in the next section.

Proposition 3.1. *The pairs of the primary and dual direction vectors p_i and p'_j form a A^2-orthonormal set, i.e. $p_i'^T A^2 p_j = \delta_{i,j}$.*

Proof.

$$\begin{aligned} (P'_m)^T A^2 P_m &= (W_m(L_m^T)^{-1})^T A^2 V_m U_m^{-1} \\ &= L_m^{-1} W_m^T A^2 V_m U_m^{-1} \\ &= L_m^{-1} T_m U_m^{-1} \qquad \text{(using (3.9))} \\ &= L_m^{-1} L_m U_m U_m^{-1} \\ &= I \end{aligned}$$

□

Proposition 3.2. *The pairs of the primary and dual residual vectors r_i and r'_j form a A-orthonormal set, i.e. $r_i'^T A r_j = 0$ for $i \neq j$.*

Proof. Combining (3.7) and (3.12)-(3.13), we get

$$\begin{aligned} r_m &= b - Ar_m \\ &= b - Ax_0 - AV_m y_m \\ &= r_0 - V_m T_m y_m - \delta_{m+1} v_{m+1} e_m^T y_m \\ &= r_0 - r_0 - \delta_{m+1} v_{m+1} e_m^T y_m \\ &= -\delta_{m+1} v_{m+1} e_m^T y_m. \end{aligned}$$

In a similar way, we get for the dual system, using (3.8) and (3.15)-(3.16),

$$r'_m = -\beta_{m+1} w_{m+1} e_m^T y'_m.$$

Combining the above two relations with (3.9), we now find that $r_i'^T A r_j = 0$ for $i \neq j$. □

4 The BiCOR method

We can now proceed in a similar way as the derivation of the Conjugate Gradient method. For the derivation of CG, see for example [40, 20]. Given an initial guess x_0, we get the coupled two-term recurrences

$$r_0 = b - Ax_0, \quad p_0 = r_0, \tag{4.1}$$

$$x_{j+1} = x_j + \alpha_j p_j, \tag{4.2}$$

$$r_{j+1} = r_j - \alpha_j A p_j, \tag{4.3}$$

$$p_{j+1} = r_{j+1} + \beta_j p_j, \qquad \text{for } j = 0, 1, \ldots \tag{4.4}$$

where $r_j = b - Ax_j$ is the residual at iteration j and p_j is the search direction vector at iteration j as in (3.17). Here the vectors p_j are multiples of the vectors p_j as seen in 3.3. It's important to note that α_j and β_j are different from the α_j and β_j used in the previous section (3.2). This is done for consistency with the notation used in the derivation of other methods. The coupled two-term recurrences for the dual system are defined in a similar way:

$$r'_{j+1} = r'_j - \alpha_j A^T p'_j, \tag{4.5}$$

$$p'_{j+1} = r'_{j+1} + \beta_j p'_j, \qquad \text{for } j = 0, 1, \ldots . \tag{4.6}$$

The parameters α_j and β_j can be determined from the orthogonality relations

$$r_{j+1} \perp \mathcal{L}_m \text{ and } Ap_{j+1} \perp \mathcal{L}_m$$

as found in section 3.2. Using propositions 3.1 and 3.2, we find the subspace $A^T\mathcal{K}_m(A^T; r'_0)$ to be suitable, where $r'_0 = P(A)r_0$ with $P(t)$ an arbitrary polynomial in variable t. A common choice is $r'_0 = r_0$, but here we will use $r'_0 = Ar_0$. If instead of $A^T\mathcal{K}_m(A^T; r'_0)$ we choose for instance $\mathcal{K}_m(A; r_0)$, we get the CG method [21]. Further derivation using (4.1)-(4.6) gives us the following expressions for α_j and β_j. For the full derivation see for example [28].

$$\alpha_j = \frac{r_j'^T A r_j}{p_j'^T A^2 p_j} \tag{4.7}$$

$$\beta_j = \frac{r_{j+1}'^T A r_{j+1}}{r_j'^T A r_j}. \tag{4.8}$$

Now, combining (4.1)-(4.8), we finally get the Biconjugate Biconjugate A-Orthogonal Residual method, or simply BiCOR [25, 28, 27, 8, 9]. The complete algorithm can be found in Algorithm 5. In the algorithm, we use the following notations: the dual vectors have a primed symbol, preconditioned vectors have a prefixed z, and a hat symbol is used for matrix-vector products.

Algorithm 5 *Left preconditioned BiCOR method.*

Compute $r_0 = b - Ax_0$ for some initial guess x_0.
Choose $r'_0 = P(A)r_0$ such that $\langle r'_0, Ar_0 \rangle \neq 0$, where $P(t)$ is a polynomial in t. (For example, $r'_0 = Ar_0$).
for $j = 1, 2, \ldots$ **do**
  solve $Mzr_{j-1} = r_{j-1}$
  if j=1 **then**
    solve $M^T zr'_0 = r'_0$
  end if
  $\widehat{zr} = Azr_{j-1}$
  $\rho_{j-1} = \langle zr'_{j-1}, \widehat{zr} \rangle$
  if $\rho_{j-1} = 0$, **method fails**
  if $j = 1$ **then**
    $p_0 = zr_0$
    $p'_0 = zr'_0$
    $q_0 = \widehat{zr}$
  else
    $\beta_{j-2} = \rho_{j-1} / \rho_{j-2}$
    $p_{j-1} = zr_{j-1} + \beta_{j-2}\, p_{j-2}$
    $p'_{j-1} = zr'_{j-1} + \beta_{j-2}\, p'_{j-2}$
    $q_{j-1} = \widehat{zr} + \beta_{j-2}\, q_{j-2}$
  end if
  $\widehat{q}\,'_{j-1} = A^T p'_{j-1}$
  solve $M^T \widehat{zq}'_{j-1} = \widehat{q}\,'_{j-1}$
  $\alpha_{j-1} = \rho_{j-1} / \langle \widehat{zq}'_{j-1}, q_{j-1} \rangle$
  $x_j = x_{j-1} + \alpha_{j-1}\, p_{j-1}$
  $r_j = r_{j-1} - \alpha_{j-1}\, q_{j-1}$
  $zr'_j = zr'_{j-1} - \alpha_{j-1}\, \widehat{zq}'_{j-1}$
  check convergence; continue if necessary
end for

5 The CORS method

Using the same strategy used when deriving the CGS method from the BiCG method, see for example [40], we can derive a transpose-free variant of the BiCOR method, the Conjugate A-Orthogonal Residual Squared method (CORS) [25, 28, 27, 8, 9].

In the previous section, we could have written the representations of the vectors r_j, p_j, r'_j, p'_j at step j as the polynomial representations

$$\begin{aligned} r_j &= \phi_j(A)r_0, & p_j &= \psi_j(A)r_0, \\ r'_j &= \phi_j(A^T)r'_0, & p'_j &= \psi_j(A^T)r'_0, \end{aligned}$$

where ϕ_j and ψ_j are Lanczos-type polynomials of degree less than or equal to j satisfying $\psi_j(0) = 1$. Substituting back in (4.7) and (4.8) gives us

$$\begin{aligned} \alpha_j &= \frac{r'^T_j A r_j}{p'^T_j A^2 p_j} = \frac{r'^T_0 A \phi_j^2(A) r_0}{r'^T_0 A^2 \psi_j^2(A) r_0} \\ \beta_j &= \frac{r'^T_{j+1} A r_{j+1}}{r'^T_j A r_j} = \frac{r'^T_0 A \phi_{j+1}^2(A) r_0}{r'^T_0 A \phi_j^2(A) r_0}. \end{aligned}$$

Also note that from (4.3) and (4.4) ϕ_j and ψ_j can be expressed recursively as

$$\phi_{j+1}(t) = \phi_j(t) - \alpha_j t\psi_j(t),$$
$$\psi_{j+1}(t) = \phi_{j+1}(t) + \beta_j\psi_j(t).$$

Using the strategy mentioned above, we now get the CORS algorithm, as described in Algorithm 6.

Algorithm 6 *Left preconditioned CORS method.*

Compute $r_0 = b - Ax_0$ for some initial guess x_0.
Choose $r'_0 = P(A)r_0$ such that $\langle r'_0, Ar_0\rangle \neq 0$, where $P(t)$ is a polynomial in t. (For example, $r'_0 = Ar_0$).
for $j = 1, 2, \ldots$ **do**
  solve $Mzr_{j-1} = r_{j-1}$
  $\widehat{zr} = Azr_{j-1}$
  $\rho_{j-1} = \langle r'_0, \widehat{zr}\rangle$
  if $\rho_{j-1} = 0$, **method fails**
  if $j = 1$ **then**
    $e_0 = r_0$
    solve $Mze_0 = e_0$
    $d_0 = \widehat{zr}$
    $q_0 = \widehat{zr}$
  else
    $\beta_{j-2} = \rho_{j-1}$ / ρ_{j-2}
    $e_{j-1} = r_{j-1} + \beta_{j-2}\ h_{j-2}$
    $ze_{j-1} = zr_{j-1} + \beta_{j-2}\ f_{j-2}$
    $d_{j-1} = \widehat{zr} + \beta_{j-2}\ g_{j-2}$
    $q_{j-1} = d_{j-1} + \beta_{j-2}\,(g_{j-2} + \beta_{j-2}q_{j-2})$
  end if
  solve $Mzq = q_{j-1}$
  $\widehat{zq} = Azq$
  $\alpha_{j-1} = \rho_{j-1}$ / $\langle r'_0, \widehat{zq}\rangle$
  $h_{j-1} = e_{j-1} - \alpha_{j-1}\ q_{j-1}$
  $f_{j-1} = ze_{j-1} - \alpha_{j-1}\ zq$
  $g_{j-1} = d_{j-1} - \alpha_{j-1}\ \widehat{zq}$
  $x_j = x_{j-1} + \alpha_{j-1}\,(2ze_{j-1} - \alpha_{j-1}zq)$
  $r_j = r_{j-1} - \alpha_{j-1}\,(2d_{j-1} - \alpha_{j-1}\ \widehat{zq})$
  check convergence; continue if necessary
end for

From our experiments, we find that CORS is highly competitive to all other popular algorithms (see section 7). However, like the CGS method, it is based on squaring the residual, which might result in a substantial buildup of rounding errors and worse approximate solutions, or possibly even overflow. This also means that CORS might in general need more time to complete a calculation than other methods, if they both succeed.

6 Computational aspects

6.1 Preconditioning

In our experiments, we use preconditioners constructed by ILUPACK [5]. The algorithms of ILUPACK compute an incomplete LU-factorisation $A = LDU + E$. Here L is a lower triangular matrix with unit diagonal, D is a diagonal matrix and U is an upper triangular matrix with unit diagonal. LDU is an approximation of the standard LU-factorisation that can be used as a preconditioner. Furthermore $\|E\| < \tau$ where τ is the drop tolerance. The matrices L, D and

U are easily implicitly computed. In the case of ILUPACK, the diagonal matrix is not a real diagonal matrix. ILUPACK computes

$$\tilde{P}^T A \tilde{Q} = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L_B & 0 \\ L_E & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & S_C \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix}$$

and then uses the inverse

$$(\tilde{P}^T A \tilde{Q})^{-1} = \begin{pmatrix} B & F \\ E & C \end{pmatrix}^{-1} \approx \begin{pmatrix} \tilde{B}^{-1} & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} \tilde{B}^{-1} F \\ I \end{pmatrix} S_C^{-1} \begin{pmatrix} -E\tilde{B}^{-1} & I \end{pmatrix}$$

where $\tilde{B} = L_B D_B U_B$ [4].

6.2 Stopping criteria

An iterative method will never provide an exact solution with a zero residual, $r = b - Ax = 0$, unless of course b is equal to zero. For this reason, we have to choose a good stopping criterion that we can use in all the different solvers we use for testing. A really small relative error $\|x - y\|/\|x\|$ with respect to the approximate solution y is usually enough, but this can not always be achieved. Also, since we do not have the actual solution, we can not explicitly calculate the relative error. Therefore, our stopping criterion will be based on the backward error analysis introduced by Wilkinson [41].

A calculated solution $\hat{x}$ of a system $Ax = b$ can be seen as the (exact) solution of the perturbed problem

$$(A + \delta A)\hat{x} = (b + \delta b).$$

The so called backward error measures the distance between the data of the original system and the perturbed system. The uncertainties in the data can either be due to measurements, or due to accumulation or propagation of roundoff errors [22]. If the backward error is not larger than those uncertainties, we may assume that the approximation is accurate. Componentwise perturbations and normwise perturbations can be used to calculate backward error. These lead to explicit formulas to calculate the backward error. It is generally accepted that for iterative methods, the use of normwise perturbations is appropriate [17]. We use this strategy to stop our solvers.

At iteration j of an iterative method, we compute an approximation x_j of the actual solution $x = A^{-1}b$. We can see x_j as the solution of the perturbed problem $(A+\delta A)x_j = (b+\delta b)$. We introduce

$$\begin{aligned} \eta_j &= \min\{\epsilon > 0 : (A + \delta A)x_j = (b + \delta b), \|\delta A\|_2 \leq \epsilon\alpha, \|\delta b\|_2 \leq \epsilon\beta\} \\ &= \frac{\|b - Ax_j\|_2}{\alpha\|x_j\|_2 + \beta} \end{aligned}$$

as the normwise backward error [22]. When the machine precision has been reached by our method, the method does not converge any further, so at best, the backward error is as small as the machine precision. In the testing application, we stop when $\eta \leq 10^{-10}$.

Common choices for α and β are, respectively, $\|A\|_2$ and $\|b\|_2$. In this case, η_j is called the normwise relative backward error [22]. For the sake of simplicity, however, we have chosen to use $\alpha = 0$ and $\beta = \|b\|_2$ in the testing application.

Value	Meaning
-1	An error occurred
1	Convergence has been achieved or the user may check for convergence
2	The user must perform a matrix-vector product
3	The user must perform the preconditioning operation

Table 2: Return values of `IACT` for our CORS implementation

6.3 Implementational aspects

Implementations of iterative methods basically require vector updates, scalar products and matrix-vector products. The first two are standard routines that are implemented in the BLAS library, but for the matrix-vector products, the user might want to provide their own implementations. This is mainly because matrices can be stored in various ways. Sometimes matrices are not even stored explicitly, but only as a subroutines. The same holds for preconditioners. ILUPACK for example does not provide an explicit matrix to use for preconditioning.

We could just let the user implement matrix-vector products in the code themselves, but that is not very user-friendly. For this reason, we allow the user to specify their own matrix-vector products and preconditioning operations, using reverse communication [12]. Reverse communication is commonly used in FORTRAN implementations of iterative methods, for example in the Harwell Subroutine Library (HSL) [23]. Here we explain how it works.

In the call to the iterative method, several variables are provided. One of those is the reverse communication variable. Once you call the function for the first time, it has to have a certain value, so the method knows it's the first time you call it. In our case the variable is `IACT` and this default value is zero. Other values of the reverse communication variable tell you to perform for instance a matrix-vector product, a preconditioning operation, or they tell you that an error occurred or convergence has been achieved. See for an example of the values of `IACT` Table 2.

One of the other variables is an array of vectors the method will use during the process. Once the user is told to perform for example a matrix-vector product, other variables are used to tell the user which vectors in the array to use. In our case those variables are `LOCY` and `LOCZ`. Meaning the location of y and z coming from the assignment $y = Az$. So one reads from the `LOCZ`-th vector and writes to the `LOCY`-th vector. Once the user performed the operation he is supposed to perform, the same subroutine is called again with the same argument. This process is repeated until convergence is observed by either the user or the algorithm itself, depending on whether the user wants to check or not.

The reverse communication method is overall very fast, because no memory has to be allocated during any of the operations. The user only has to perform a certain operation. It's also very user friendly, because the user can use any implementation of a matrix-vector product, preconditioning operation, or convergence check.

A different way to implement this in FORTRAN would be allowing the user to pass a function or subroutine to the subroutine that is then called by the subroutine itself, but this limits the user to passing only subroutines or functions that require a set amount of variables, whereas the user probably wants to pass more variables. A way to do this in object oriented languages, like C++, is by overloading operators.

7 Numerical experiments

7.1 Information about the experiments

In our experiments, we consider a collection of various matrices available from the University of Florida Sparse Matrix Collection from Tim Davis [10]. The matrices we used are a reasonable representation of all nonsymmetric and real matrices available in the collection, covering every field of research in the collection. To analyse the performance of BiCOR and CORS, we compared them to the popular methods BiCG, BiCGSTAB, CGS, GMRES, BiCGSTAB(ℓ), QMR and TFQMR. For GMRES we used a value of restart equal to 100. This reduced the memory needed to run the solver on the largest problems. We chose the value $\ell = 3$ for BiCGSTAB(ℓ), because this yielded the best results.

We implemented the BICOR and CORS methods in FORTRAN 77 by ourselves. For BiCG, BiCGSTAB, CGS and GMRES we used implementations from the Harwell Subroutine Library (HSL) [23]. The implementation of BiCGSTAB(ℓ) was obtained from Van der Vorst's website, [16] and the implementations of QMR and TFQMR came from QMRPACK [19].

The tests were run on a PC equipped with an AMD Athlon^{TM} 7850 Dual-Core Processor running at 2,8 GHz and 4GB 800 MHz DDR2-RAM. Our code was compiled with the GNU FORTRAN compiler (gfortran) version 4.5.2 that came with Ubuntu 11.04. The implementations of all the solvers we tested were in FORTRAN 77, the testing application calling those implementations was in FORTRAN 2003 to allow us to keep running the application without having to restart for every other preconditioner or matrix. This needed memory allocation and making sure the results were saved on the local disk required flushing the file after every solver completed.

To read the matrices from our hard-drive, we downloaded the matrices in MatrixMarket format [3]. To store our matrices in the main memory, we used the compressed sparse row format. We used SPARSKIT [34] to convert the matrices from the coordinate format used in the MatrixMarket script to the compressed sparse row format. The matrix-vector product and transpose matrix-vector product were performed by `AMUX` and `ATMUX` in SPARSKIT. The preconditioning operation was performed by ILUPACK. Those libraries, as well as the implementations of all the different solvers, needed a BLAS implementation, for which we used the ACML library optimised for AMD processors.

The data we gathered from running the different solvers included the amount of time it took to solve a problem, the amount of matrix-vector products, and, if a solver did not complete, the reason why. We ran every solver six times if it completed the first time to be able to get rid of any flaws caused by processes running in the background. To minimise this effect, no applications were started other than the default startup applications, excluding Ubuntu One, and including Dropbox to make sure results were not lost, a terminal, and nautilus. During the process, the CPU was monitored to make sure nothing interfered with the testing application. As a result, the testing application ran at 99%-100% of one core essentially all of the time.

To make a better selection of the matrices to test, we excluded matrices that completed faster than 0.05 seconds, because the results we got from the `CPU_TIME` routine were only accurate to up to 2 decimals. Additionally, we excluded those matrices of which the problems took too long to solve, i.e. took more than 20000 matrix-vector operations, for none of the iterative solvers using the best preconditioner. The reason we stopped at 20000 matrix-vector products is that for the bigger problems it would take five days or more to get up to n iterations for any solver. This would mean that we would be done maybe two years from now. The reason we excluded the problems that failed to complete for every solver was so we did not have to

rerun them for worse preconditioners.

We also checked if a result of a given solver on a given problem was much different from the average of the other runs with the same solver on the same problem. If it was more than 10% off, it was excluded from the results. This rarely ever happened.

7.2 Data analysis

To analyse the data we gathered from running the testing application, we make use of performance profiles of the computation time and the amount of matrix-vector products as suggested by Dolan and Moré in [11]. In the performance profiles, we can see what solver is most likely to solve a certain problem after a certain amount of time or with a certain amount of matrix-vector products compared to the other solvers. The best solver for every matrix gets a value of one associated with them, and the other solvers get a value greater than one, that is the ratio between this solver and the best solver. So the performance ratio of a solver s on a problem p is given by

$$r_{p,s} = \frac{t_{p,s}}{\min_s t_{p,s}}$$

and the cumulative distribution function for the performance ratio is given by

$$\rho_s(\tau) = \frac{1}{n_p}\text{size}\{p : r_{p,s} \leq \tau\}$$

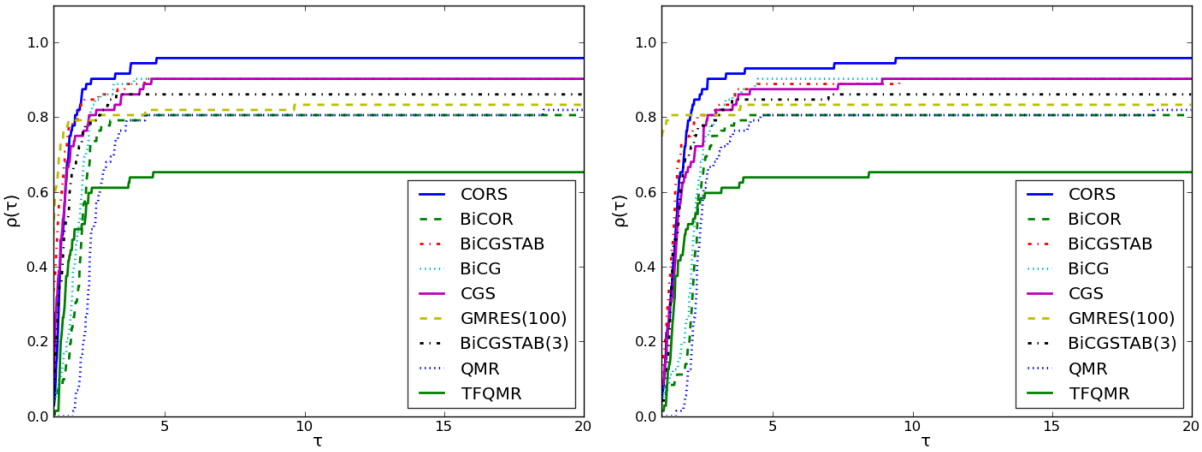
where n_p is the total amount of problems we tested. So at $\tau = \tau_1$, a certain solver s has a probability $\rho_s(\tau_1)$ of solving a problem at a ratio τ_1 worse than the fastest solver. $\rho(1)$, is of particular interest, because we can see there how many times a solver was the best.

If a solver does not solve a problem, the ratio r_M is assigned. This ratio should be higher than the highest ratio found for any solver on any problem that did not fail. In this way, the solver will still have a value assigned for the certain problem where it failed, but we simply will not plot for $\tau \geq r_M$. So we will see the solver that solved the most problems overall on top when we look at the far right of the plot.

7.3 Results

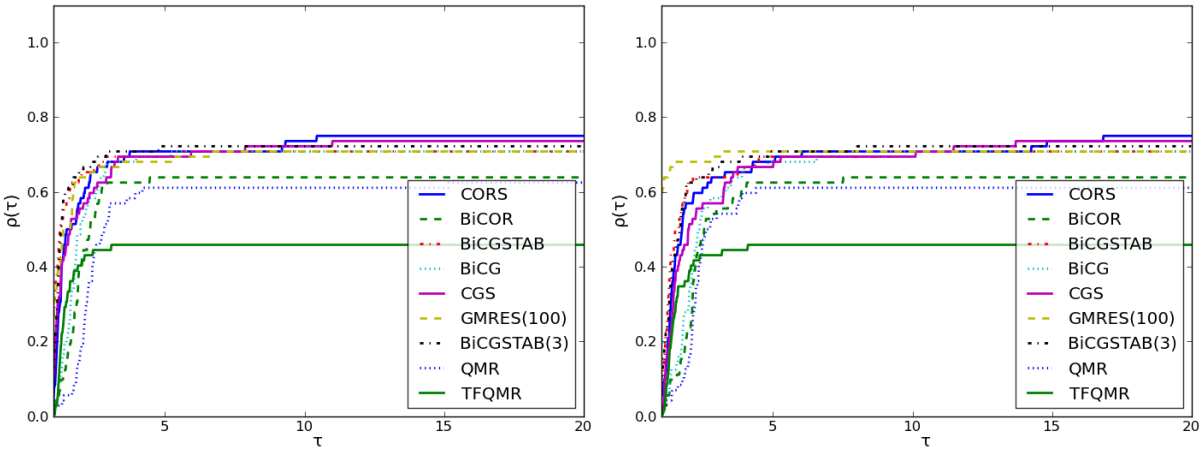
We ran our preliminary tests with three different preconditioners constructed with drop tolerances of 0.1, 1.0 and 10.0 on a total of more than 100 matrices, but ended up with only 72 matrices that satisfied our criteria. We solved the linear system using preconditioning from the right. This means that we solved the system $AM_2^{-1}y = b$ with our solution $x = M_2^{-1}y$. From (2.2), we see that we did not have to adjust our stopping criterion to work on the preconditioned system. If we would have used preconditioning from the left, in a real implementation one would have had to adjust every solver to have a stopping criterion based on the preconditioned system. We could, however, in reverse communication just replace the matrix-vector products with a preconditioning operation and a matrix-vector product, and the preconditioning operation with a vector copy.

The time it took to complete the experiments with 9 solvers on 72 matrices was over 120 hours. We analyse the results in the next sections.



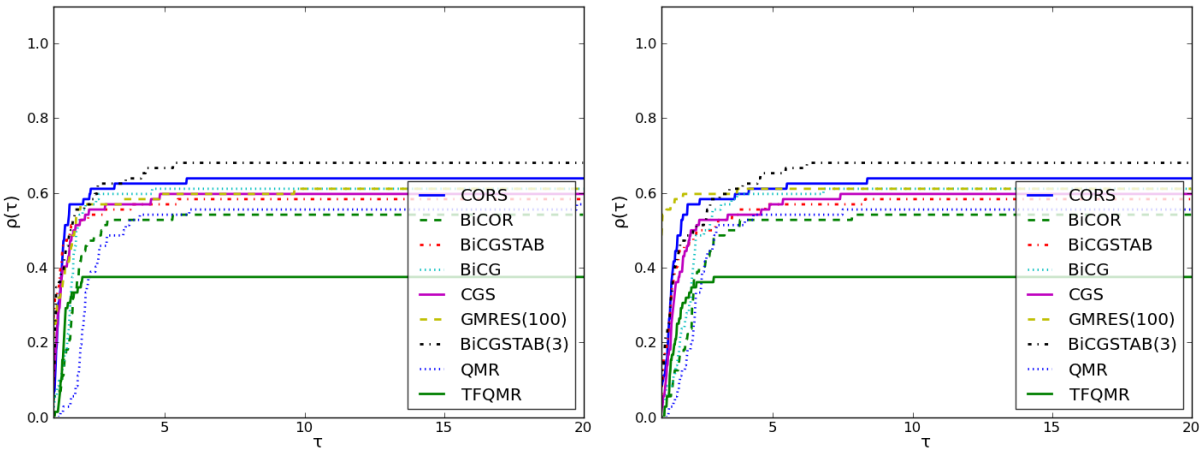
(a) Ratio of CPU time (b) Ratio of matrix-vector products

Figure 2: Performance profiles with a tolerance of 0.1



(a) Ratio of CPU time (b) Ratio of matrix-vector products

Figure 3: Performance profiles with a tolerance of 1.0



(a) Ratio of CPU time (b) Ratio of matrix-vector products

Figure 4: Performance profiles with a tolerance of 10.0

7.3.1 Speed

As one can see in Figures 2-4, our tests revealed that GMRES and BiCGSTAB were in general the fastest solvers. In terms of matrix-vector products, GMRES was of course the fastest, because it uses only one matrix-vector product per iteration, where the other methods use two. In terms of time however, GMRES became better compared to the other solvers for better preconditioners. For the preconditioner constructed with a tolerance of 0.1, GMRES had 35 wins where BiCGSTAB had 24, with a tolerance of 1.0, they both had 19 wins, and with a tolerance of 10, BiCGSTAB had 19 wins and GMRES 15. Because GMRES got worse for sparser preconditioners. This also meant that the other solvers got relatively more wins.

Here it must be noted that we used a restart value of 100 for GMRES, but for really large problems, where memory use is an issue, we would not be able to use such a high value for the restart. In such a case, it would be more fair to have a value of restart that makes the memory use of GMRES similar to that of the other methods. We tried this, but this gave such bad results (worse than TFQMR), that we decided to use a value of 100.

If we do not only look at the winners, but at a slightly bigger region of interest, say $\tau \leq 2$, we see that CORS, BiCGSTAB, GMRES and CGS are the most competitive solvers. For the best preconditioner, the one with a drop tolerance of 0.1, CORS is even on top after $\tau \approx 1.7$ (see Figure 5).

We also see that CORS is considerably faster in terms of matrix-vector products (see Figures 2-4). It performs a bit worse in terms of time is mainly due to the amount of scalar times a vector plus a different vector operations, or simply Scalar A X Plus Y (SAXPY) operations. Those are the most expensive operations done in the algorithms themselves. CORS uses 12 of such per iteration where BiCOR for example only uses 6.

We now classify the solvers according to the information we gathered about the different problems, see Appendix A.2. In this case, we excluded solvers like QMR and TFQMR from this analysis, because they performed very badly. We also only name the solvers where we saw something notable. First, we see that the bigger the problem, the better CORS and BiCGSTAB perform and the worse GMRES performs. This is mainly of interest, because one tends to use iterative methods only for bigger problems. For smaller problems one could as well just use direct methods.

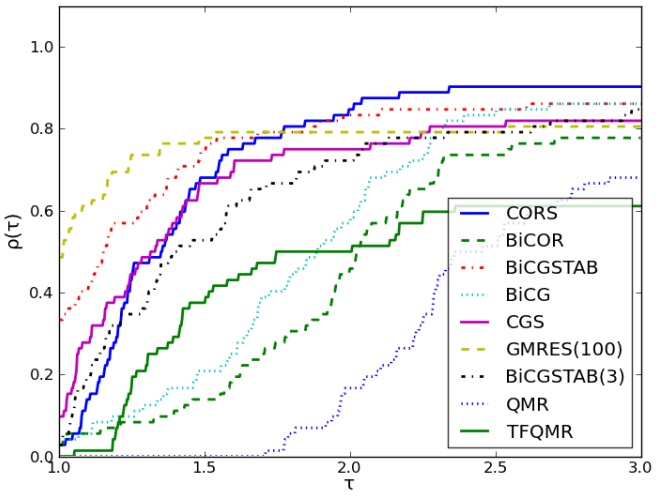


Figure 5: Performance profile with a tolerance of 0.1 on a smaller region: Ratio of CPU time

Solver	breakdown	iterations	NAN
CORS	1	2	0
BiCOR	11	3	0
BiCGSTAB	7	0	0
BiCG	5	2	0
CGS	0	7	0
GMRES(100)	0	12	0
BiCGSTAB(3)	0	8	2
QMR	0	13	0
TFQMR	0	25	0

Table 4: Failures with a perconditioner tolerance of 0.1

We also investigated if the percentage of pattern symmetry and value symmetry mattered. Here we see that GMRES performs better for the highly nonsymmetric problems, and CORS and BiCGSTAB for the more symmetric problems. We also noted that for diagonal dominant matrices, BiCGSTAB was better and GMRES worse.

Now we get to the kind of problems. Here we see that GMRES performed better for economic, and semiconductor device problems. BiCGSTAB performed better for circuit simulation, computational fluid dynamics, and semiconductor device problems. We also saw that CORS performed considerably well on circuit simulation problems, and that BiCOR and BiCG were very efficient for the electromagnetics problems.

7.3.2 Reliability

If we look at a region of large values of τ in the performance profiles in Figure 2, we find that CORS ends up solving 4 more problems that the two next best solvers, BiCG and CGS. For this preconditioner, CORS only fails to solve three problems. CORS also ends up being on top for the preconditioner constructed with a tolerance of 1.0. For the preconditioner constructed with a tolerance of 10, we see that BiCGSTAB(ℓ) ends up on top. CORS is the second best solver, and it's performance comes very close to BiCGSTAB(ℓ) if we look at values of τ greater than 20, but this is not shown in the performance profiles. If we look at the other preconditioner tolerances, we see that BiCGSTAB(ℓ) does not even come close to the performance of CORS.

We're now interested in what happens when CORS fails, since it's the solver that has the least amount of failures. We find that for the preconditioner constructed with a tolerance of 0.1, CORS exceeds the maximum amount of iterations twice, and breaks down once. A breakdown in the implementations of CORS, BiCOR, BiCGSTAB, CGS and BiCG means that $|\rho_{j-1}| < un$ and $|\rho_{j-1}| < u\|r_{j-1}\|_2\|r'_{j-1}\|_2$ where n is the size of the problem, u is the machine precision and the other variables as in Algorithm 5. This is as it is adopted in the HSL. The GMRES method, being an optimal method, can not break down, so for GMRES, we do not see any breakdowns. BiCGSTAB(ℓ) returned quite a lot of not-a-number answers, which might be due to the breakdown implementation which is different from that the HSL.

So let's look at the only case where CORS broke down for the best tolerance we tested. This was on the `torso1` matrix. For this matrix, we find that not only CORS, but also BiCGSTAB broke down, BiCOR and BiCG converged, and the others exceeded the maximum amount of allowed iterations. If we look at the convergence history of the 2-norm of the residual of CORS, BiCOR, BiCGSTAB, BiCG, CGS and GMRES, we see that for BiCOR and BiCG the residual

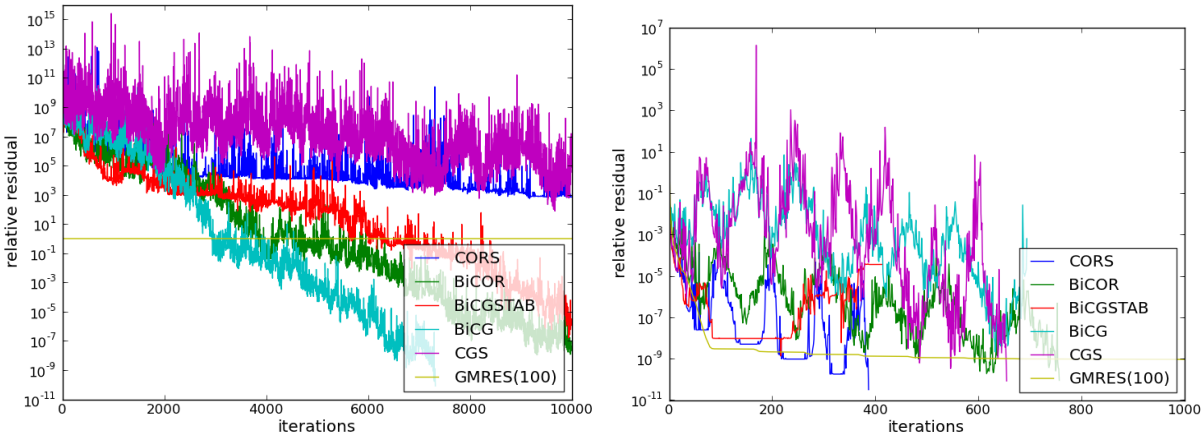
started reducing at a nice rate after about 200 iterations. For GMRES, the residual stayed constant after about 80 iterations, and for CGS the residual heavily fluctuated somewhere above 10^5. The residual of BICGSTAB suddenly increased after about 300 iterations and the method broke down after doing a few more steps. CORS at first showed about the same behaviour as CGS, then fluctuated less heavily, but did still not converge, and after that, CORS broke down.

The two matrices where CORS took too many iterations were the `cryg10000` and `invextr1_new` matrices. For the first one, CORS did not seem to converge at all, but for the last one, CORS converged at steady rate, but unfortunately not fast enough to complete within the set maximum amount of iterations, as can be seen in Figure 6(a). What we can also see in this figure is the relatively wild behaviour of CGS due to the squaring of the residual, and the behaviour of GMRES, which usually converges steadily, but in this case not at all. What we can conclude, is that CORS is able to solve most problems, and therefore is the most robust method for this preconditioner.

In Figure 6(b), we again see that here CGS is a lot wilder than CORS, which seems to be the usual behaviour. This is probably the reason why CORS is more stable than CGS, while they are based on the same ideas. In this figure we also see the breakdown of BiCGSTAB. The method first behaves like other solvers, but then sees a sudden increase in the residual after which the residual stays the same and then the method breaks down. This is the standard behaviour we observed for this method and other methods.

The default drop tolerance used by ILUPACK is 0.01 instead of the 0.1 we used as lowest tolerance. We did this mainly to make sure the solvers did not complete too quickly, and so we could find out which solver was most robust. The construction of the preconditioner for the bigger problems only took a small amount of time compared to solving the problem itself, so it would be reasonable to use a better preconditioner. So we ran some more tests. We tried several better preconditioners on the matrices that failed for all methods for the preconditioner with a drop tolerance of 0.1. In those tests, CORS again turned out to be the most reliable method in every test. Some other solvers were able to compete with CORS in some tests, but CORS was the only one that turned out to be the most reliable in every single test.

If we look at BiCOR we see that a lot of breakdowns occur. We checked that in all cases the value of ρ was indeed smaller than the machine precision. Two new methods, BiCOR Stabilised (BiCORSTAB) [28] and Composite Step BiCOR (CSBiCOR) [26], have already been developed to prevent those failures. We however did not have a chance to test those methods.



(a) invextr1_new (b) para-6

Figure 6: Convergence history of the relative residual on two problems

The last question we tried to answer was how the distribution of the eigenvalues of the preconditioned matrix affected convergence. If the preconditioner is good, then AM^{-1} is close to identity, so we may expect that most of the eigenvalues are close to one. We calculated the eigenvalues of the 20 smallest matrices, but it was not possible to draw any conclusions.

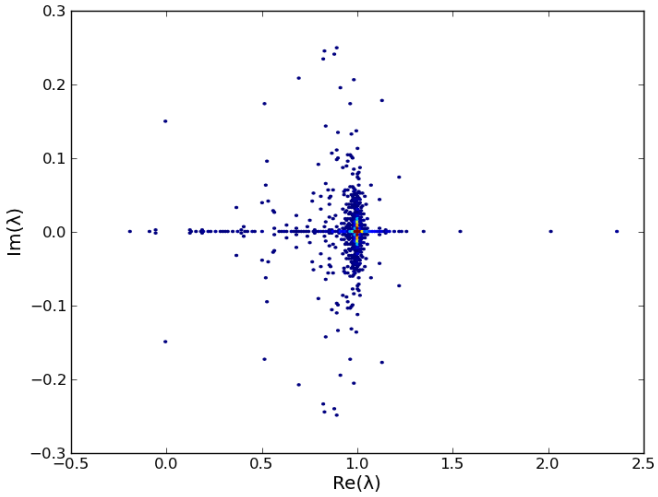


Figure 7: Typical distribution of eigenvalues, in this case of the `powersim` matrix.

8 Conclusion

When we started comparing CORS and BiCOR to other iterative methods, we had no idea whether they were competitive or not. In our experiments we found that BiCOR broke down many times, and therefore is not very attractive for solving realistic applications. We also found that the BiCGSTAB method and GMRES method with sufficiently large restart, are the most popular methods in use today for a reason: they turned out to be the fastest methods. In terms of stability however, CORS proves to be the best. It might not be as fast, mostly due to the larger amount of SAXPY operations, but reliability comes with a cost. We also see this when we compare for example BiCGSTAB(ℓ) to normal BiCGSTAB, which is generally a lot faster than BiCGSTAB(ℓ).

The most interesting case when using iterative methods is a large problem with a good preconditioner. Bigger problems are more vulnerable to the performance of the methods, simply because they take longer to solve. Also, because ILUPACK constructs the preconditioners quite fast, with an amount of nonzeros of the order of the problem itself, one most likely wants to use a better preconditioner. We found CORS to excel in both cases: it was better for better preconditioners, and also faster for bigger problems in comparison with other solvers.

We conclude that the CORS method turns out to be a valuable addition to the long list of iterative methods already available.

9 Acknowledgments

I would really like to thank Bruno Carpentieri for all his support and help during every step of the process and for developing BiCOR and CORS in the first place to make this research possible. My thanks also go to Matthias Bollhöfer (Institute of “Computational Mathematics”,

Technische Universität Braunschweig, Germany) for his support in the usage of the ILUPACK software. And finally, I would like to thank Sjoerd Meesters, Paulus Meessen and Jeroen Lanting for thoroughly reading the manuscript and giving advice on how to improve it, and Diederik Perdok, who read the manuscript with no knowledge of numerical mathematics in general, but still gave really good comments.

References

[1] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9:17–29, 1951.

[2] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1993. Obtainable from research.att.com:/netlib/linalg using ftp.

[3] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra. Matrix market: A web resource for test matrix collections. In *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman & Hall, 1997.

[4] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse–based ILUs. 27(5):1627–1650, 2006.

[5] M. Bollhöfer, Y. Saad, and O. Schenk. ILUPACK — preconditioning software package, June 2011. http://ilupack.tu-bs.de/. Release 2.4.

[6] C. Brezinski and L. Wuytack. *Projection methods for systems of equations*. North Holland, 1997.

[7] R. L. Burden and J. D. Faires. *Numerical Analysis*. Thompson, 8 edition, 2005.

[8] B. Carpentieri, Y.-F. Jing, and T.-Z. Huang. The BiCOR and CORS algorithms for solving nonsymmetric linear systems. *SIAM J. Scientific Computing*, 2011. In press.

[9] B. Carpentieri, Y.-F. Jing, T.-Z. Huang, W.-C. Pi, and X.-Q. Sheng. A novel family of iterative solvers for Method of Moments discretizations of Maxwells equations. In L. Gürel, editor, *CEM'11 Computational Electromagnetics*, pages 85–90. Bilkent University, Computational Electromagnetics Research Center, August 2011.

[10] T. A. Davis. University of florida sparse matrix collection. Technical report, 1994.

[11] E. D. Dolan and J. J. More. Benchmarking optimization software with performance profiles. *Math. Programming, Ser. A*, 91:201–212, 2002.

[12] J. Dongarra, V. Eijkhout, and A. Kalhan. Reverse communication interface for linear algebra templates for iterative methods. Technical Report UT-CS-95-291, May 1995.

[13] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical linear algebra for high-performance computers*, volume 7 of *Software, Environments, and Tools*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998.

[14] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Numerical Analysis*, 21(2):352–362, Apr. 1984.

[15] R. Fletcher. *Conjugate gradient methods for indefinite systems*, volume 506 of *Lecture Notes Math.*, pages 73–89. Springer-Verlag, Berlin, 1976.

[16] D. R. Fokkema. Bicgstab(ell), full version. http://www.staff.science.uu.nl/ vorst102/software.html.

[17] V. Frayssé, L. Giraud, S. Gratton, and J. Langou. A set of GMRES routines for real and complex arithmetics on high performance computers. *ACM Trans. Math. Softw.*, 31(2):228–238, 2005.

[18] R. W. Freund and N. M. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–340, 1991.

[19] R. W. Freund and N. M. Nachtigal. QMRPACK: A package of QMR algorithms. *ACM Trans. Math. Softw*, 22(1):46–77, 1996.

[20] A. Greenbaum. *Iterative Methods for Solving Linear Systems.* SIAM, Philadelphia, 1997.

[21] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[22] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[23] HSL(2011). A collection of fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk/.

[24] I. C. F. Ipsen and C. D. Meyer. The idea behind Krylov methods. *The American Mathematical Monthly*, 105(10):889–899, 1998.

[25] Y.-F. Jing, B. Carpentieri, and T.-Z. Huang. Experiments with Lanczos biconjugate A-orthonormalization methods for MoM discretizations of Maxwell's equations. *Progress In Electromagnetics Research, PIER 99*, pages 427–451, 2009.

[26] Y.-F. Jing, T.-Z. Huang, B. Carpentieri, and Y. Duan. Investigating the composite step biconjugate A-orthogonal residual method for non-hermitian linear systems in Electromagnetics. In L. Gürel, editor, *CEM'11 Computational Electromagnetics*, pages 80–84. Bilkent University, Computational Electromagnetics Research Center, August 2011.

[27] Y.-F. Jing, T.-Z. Huang, Y. Duan, and B. Carpentieri. A comparative study of iterative solutions to linear systems arising in quantum mechanics. *Journal of Computational Physics*, 229:8511–8520, November 2010.

[28] Y.-F. Jing, T.-Z. Huang, Y. Zhang, L. Li, G.-H. Cheng, Z.-G. Ren, Y. Duan, T. Sogabe, and B. Carpentieri. Lanczos-type variants of the COCR method for complex nonsymmetric linear systems. *Journal of Computational Physics*, 228(17):6376–6394, 2009.

[29] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bur. Standards*, 45:255–282, 1950.

[30] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.

[31] S. J. Leon. *Linear algebra with applications.* Prentice-Hall, pub-PH:adr, seventh edition, 2006.

[32] B. N. Parlett, D. R. Taylor, and Z. A. Liu. A look-ahead Lanczos algorithm for unsymmetric matrices. *Math. Comp.*, 44:105–124, 1985.

[33] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics.* Springer, New York, 2000.

[34] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[35] Y. Saad. *Iterative methods for sparse linear systems.* SIAM, 2003.

[36] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 7:856–869, 1986.

[37] G. L. G. Sleijpen and D. R. Fokkema. BiCGstab(L) for linear equations involving unsymmetric matrices with complex spectrum. *Elect. Trans. Numer. Anal.*, 1:11–32, 1993.

[38] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 10:36–52, 1989.

[39] 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. Scientific and Statistical Computing*, 13:631–644, 1992.

[40] H. A. van der Vorst. *Iterative Krylov methods for large linear systems*, volume 13 of *Cambridge Monographs on Applied and Computational Mathematics.* Cambridge University Press, Cambridge, UK, 2003.

[41] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Notes on Applied Science No. 32, Her Majesty's Stationery Office, London, 1963. Also published by Prentice-Hall, Englewood Cliffs, NJ, USA. Reprinted by Dover, New York, 1994.

Appendices

In the appendices, one can find a description of all problems used, the implementation and documentation of the CORS and BiCOR methods, and the two main programs used in the analysis of the results. In addition to this, two Python modules were written, 5 more Python scripts, and 4 more Fortran applications. Since the appendix would be a lot longer if those were also included, and does not really add any valuable content to the thesis, those were left out.

A Problems

A.1 Problem types

List taken from `http://www.cise.ufl.edu/research/sparse/matrices/kind.html`

A.1.1 Problems with 2D/3D geometry

- 2D/3D problem
- acoustics problem
- computational fluid dynamics problem
- computer graphics/vision problem
- electromagnetics problem
- materials problem
- model reduction problem
- robotics problem
- semiconductor device problem
- structural problem
- thermal problem

A.1.2 Problems that normally do not have 2D/3D geometry

- chemical process simulation problem
- circuit simulation problem
- counter-example problem
- economic problem
- frequency-domain circuit simulation problem
- least squares problem
- linear programming problem
- optimization problem
- power network problem
- statistical/mathematical problem
- theoretical/quantum chemistry problem
- combinatorial problem
- graph problems

A.2 Problem list

matrix name	number of rows	nonzeros	nonzero pattern symmetry	numeric value symmetry	row diagonal dominance	column diagonal dominance	kind
torso1	116,158	8,516,500	42%	0%	0.00%	0.15%	2D/3D problem
shermanACb	18,510	145,149	15%	3%	38.10%	37.36%	2D/3D problem
av41092	41,092	1,683,902	0%	0%	0.00%	0.00%	2D/3D problem
Baumann	112,211	748,331	100%	0%	16.58%	97.05%	2D/3D problem
heart3	2,339	680,341	100%	0%	0.00%	0.00%	2D/3D problem
chem_master1	40,401	201,201	100%	0%	1.98%	100.00%	2D/3D problem
e40r0100	17,281	553,562	31%	0%	0.14%	0.14%	2D/3D problem
Zd_Jac3	22,835	1,915,726	0%	0%	0.00%	0.00%	chemical process simulation problem
std1_Jac2_db	21,982	498,771	33%	0%	66.96%	54.41%	chemical process simulation problem
memplus	17,758	99,147	100%	50%	75.96%	87.85%	circuit simulation problem
ASIC_320k	321,821	1,931,828	100%	36%	71.29%	71.96%	circuit simulation problem
hcircuit	105,676	513,072	100%	20%	83.19%	84.66%	circuit simulation problem
scircuit	170,998	958,936	100%	80%	97.64%	97.39%	circuit simulation problem
ASIC_680k	682,862	2,638,997	100%	0%	85.46%	94.34%	circuit simulation problem
circuit_3	12,127	48,137	77%	30%	61.21%	63.71%	circuit simulation problem
transient	178,866	961,368	100%	24%	90.18%	90.74%	circuit simulation problem
trans4	116,835	749,800	85%	30%	57.54%	52.20%	circuit simulation problem sequence
lung2	109,460	492,564	57%	0%	49.61%	49.61%	computational fluid dynamics problem
airfoil_2d	14,214	259,688	98%	0%	4.48%	18.01%	computational fluid dynamics problem
atmosmodl	1,489,752	10,319,760	100%	67%	100.00%	100.00%	computational fluid dynamics problem
Ill_Stokes	20,896	191,368	99%	33%	17.67%	17.58%	computational fluid dynamics problem
atmosmodd	1,270,432	8,814,880	100%	67%	100.00%	100.00%	computational fluid dynamics problem
goodwin	7,320	324,772	64%	0%	4.09%	0.17%	computational fluid dynamics problem
poisson3Db	85,623	2,374,949	100%	0%	2.04%	10.50%	computational fluid dynamics problem
invextr1_new	30,412	1,793,881	97%	72%	2.98%	5.29%	computational fluid dynamics problem
GT01R	7,980	430,909	88%	0%	3.72%	4.11%	computational fluid dynamics problem
raefsky1	3,242	293,409	100%	9%	25.87%	25.87%	computational fluid dynamics problem sequence
cage11	39,082	559,722	100%	18%	97.40%	92.67%	directed weighted graph
language	399,130	1,216,334	6%	0%	81.25%	78.62%	directed weighted graph
psmigr_2	3,140	540,022	48%	0%	0.00%	0.00%	economic problem
g7jac160	47,430	564,952	3%	0%	25.94%	28.87%	economic problem
g7jac060	17,730	183,325	4%	0%	25.87%	29.20%	economic problem
mark3jac040	18,289	106,803	7%	1%	23.17%	27.14%	economic problem
jan99jac040	13,694	72,734	0%	0%	29.89%	65.81%	economic problem
mark3jac080sc	36,609	214,643	7%	1%	17.44%	29.55%	economic problem

g7jac140	41,490	488,633	3%	0%	25.95%	28.91%	economic problem
fp	7,548	834,222	76%	0%	3.11%	3.11%	electromagnetics problem
dw8192	8,192	41,746	96%	92%	10.64%	10.64%	electromagnetics problem
utm5940	5,940	83,842	53%	0%	12.82%	15.57%	electromagnetics problem
tmt_unsym	917,825	4,584,801	100%	0%	49.95%	49.96%	electromagnetics problem
viscoplastic2	32,769	381,326	57%	0%	43.94%	8.07%	materials problem
cryg10000	10,000	49,699	100%	0%	54.72%	1.97%	materials problem
inlet	11,730	328,323	61%	0%	3.30%	1.50%	model reduction problem
flowmeter5	9,669	67,391	100%	6%	73.72%	73.62%	model reduction problem
chipcool1	20,082	281,150	100%	9%	4.17%	4.95%	model reduction problem
crashbasis	160,000	1,750,416	55%	0%	49.73%	95.75%	optimization problem
hvdc1	24,842	158,426	98%	10%	5.41%	4.58%	power network problem
powersim	15,838	64,424	59%	53%	36.49%	49.09%	power network problem
TSOPF_RS_b39_c19	38,098	684,206	6%	0%	0.49%	0.49%	power network problem
nmos3	18,588	237,130	100%	17%	18.15%	19.91%	semiconductor device problem
matrix_9	103,430	1,205,518	100%	17%	10.99%	34.87%	semiconductor device problem
matrix-new_3	125,329	893,984	99%	28%	57.78%	62.60%	semiconductor device problem
igbt3	10,938	130,500	100%	17%	25.60%	3.74%	semiconductor device problem
2D_27628_bjtcai	27,628	206,670	100%	22%	43.23%	49.79%	semiconductor device problem
ohne2	181,343	6,869,939	100%	9%	23.86%	2.09%	semiconductor device problem
3D_51448_3D	51,448	537,038	99%	19%	34.34%	44.88%	semiconductor device problem
3D_28984_Tetra	28,984	285,092	99%	36%	47.07%	49.30%	semiconductor device problem
2D_54019_highK	54,019	486,129	100%	19%	31.67%	43.35%	semiconductor device problem
ibm_matrix_2	51,448	537,038	99%	19%	35.72%	44.57%	semiconductor device problem
wang3	26,064	177,168	100%	98%	84.12%	85.09%	semiconductor device problem
sme3Db	29,067	2,081,063	100%	44%	0.00%	0.00%	structural problem
sme3Da	12,504	874,887	100%	44%	0.00%	0.00%	structural problem
t2d_q4	9,801	87,025	100%	69%	74.97%	75.16%	structural problem sequence
venkat50	62,424	1,717,777	100%	6%	0.00%	0.00%	subsequent computational fluid dynamics problem
barrier2-10	115,625	2,158,759	100%	20%	27.31%	4.96%	subsequent semiconductor device problem
para-6	155,924	2,094,873	100%	37%	27.60%	4.02%	subsequent semiconductor device problem
barrier2-4	113,076	2,129,496	100%	19%	25.31%	5.07%	subsequent semiconductor device problem
para-9	155,924	2,094,873	100%	18%	27.71%	4.02%	subsequent semiconductor device problem
epb1	14,734	95,053	73%	0%	52.65%	60.66%	thermal problem
thermomech_dK	204,316	2,846,228	100%	67%	0.00%	0.00%	thermal problem
ted_A	10,605	424,587	57%	11%	0.00%	0.00%	thermal problem
FEM_3D_thermal1	17,880	430,740	100%	95%	0.00%	0.00%	thermal problem

B Implementation of BiCOR

B.1 User documentation

The implementation of the algorithm is based on the methods used in the Harwell Subroutine Library (HSL) [23]. Since the implementation is quite similar, the documentation is also quite similar to the documentation for the various subroutines in the HSL.

B.1.1 Argument lists and calling sequence

B.1.1.1 Initialization of the control parameters

The following subroutines have to be called before using the algorithm with `BICORA(D)`. For single precision we have

```
CALL BICORI(ICNTL, CNTL, ISAVE, RSAVE)
```

and for double precision we have

```
CALL BICORID(ICNTL, CNTL, ISAVE, RSAVE)
```

where

`ICNTL`	is an `INTEGER` array of length 8 that does not have to be set by the user. On return, it contains the default values as described in section B.1.2.
`CNTL`	is a `REAL` (`DOUBLE PRECISION` in the D version) array of length 5 that does not have to be set by the user. On return, it contains the default values and described in section B.1.2.
`ISAVE`	is an `INTEGER` array of length 17 that must not be altered by the user.
`RSAVE`	is a `REAL` (`DOUBLE PRECISION` in the D version) array of length 9 that must not be altered by the user.

B.1.1.2 Solving Ax=b

Here we will actually solve $Ax = b$. For single precision we have

```
 CALL BICORA(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL, INFO,
+            ISAVE, RSAVE)
```

and for double precision we have

```
 CALL BICORAD(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL, INFO,
+             ISAVE, RSAVE)
```

where

IACT is an INTEGER that indicate the action the user has to preform on every return of the BICORA/AD routines. Prior to the first call to BICORA/AD, IACT should be set to 0. Possible values are as follows:

-1 An error occurred, and the user must terminate the computation. The reason for the error is in INFO(1). See section B.1.3 for more information.

1 If ICNTL(4)=0 (the default value), convergence has been achieved, and the user should terminate the computation. If ICNTL(4) is nonzero, the user may test for convergence. If convergence has not been achieved, BICORA/AD should be called again, without changes to its arguments.

2 The user must preform the matrix-vector product

$$y = Az$$

and recall BICORA/AD. The vectors y and z are in columns LOCY and LOCZ of array W respectively. The user should not change z.

3 The user must preform the preconditioning operation

$$y = Mz$$

where M is the preconditioner. The vectors y and z are in columns LOCY and LOCZ of array W respectively. The user should not change z. Preconditioning is only used when ICNTL(3) is nonzero.

4 The user must preform the transpose matrix-vector product

$$y = A^T z$$

and recall BICORA/AD. The vectors y and z are in columns LOCY and LOCZ of array W respectively. The user should not change z.

5 The user must preform the transpose preconditioning operation

$$y = M^T z$$

where M is the preconditioner. The vectors y and z are in columns LOCY and LOCZ of array W respectively. The user should not change z. Preconditioning is only used when ICNTL(3) is nonzero.

N is an INTEGER variable that must be set by the user to the order of the matrix A. The variable must be preserved between calls to BICORA/AD. This argument is not altered by the routine.

W is a REAL (DOUBLE PRECISION in the D version) two-dimensional array with dimensions (LWD, 8). Prior to the first call to BICORA/AD, the first column must hold the right hand side b. IF ICNTL(5) is nonzero, the second column must contain the initial estimate of the solution x. On exit, the first column holds the residual $r = b - A\hat{x}$ and the second column holds the estimate of the solution x. Other than the vector contained in the LOCYth column of W, W should remain unchanged between calls to BICORA/AD.

LWD is an INTEGER variable that must be set by the user to the first dimension of W. This argument is not altered by the routine. It should be greater than N.

LOCY, LOCZ are INTEGER variables that need not be set by the user. On return with IACT > 1, they indicate which columns of W should be used to preform the operations as specified under IACT (see above). These arguments must not be altered by the user between calls to BICORA/AD.

RESID is a REAL (DOUBLE PRECISION in the D version) variable that need not be set by the user. On return with IACT=1, it contains the 2-norm of the residual vector $\|b - A\hat{x}\|_2$, where $\hat{x}$ is the current estimate of the solution.

ICNTL is an INTEGER array of length 8 that has to be set by the user. The default values are set by a call to BICORA/AD as described in section B.1.1.1. Details of the control parameters are given in section B.1.2. This argument is not altered by the routine.

CNTL is a REAL (DOUBLE PRECISION in the D version) array of length 5 that does not have to be set by the user. that has to be set by the user. The default values are set by a call to BICORA/AD as described in section B.1.1.1. Details of the control parameters are given in section B.1.2. This argument is not altered by the routine.

INFO is an INTEGER array of length 4 that need not be set by the user. It is used to store information about the subroutine. On return of BICORA/AD, INFO(1) tells if the subroutine was successful (value 0) or if an error occurred (non-zero values). More information about this is in section B.1.3. INFO(2) holds the amount of iterations preformed by the subroutine. INFO(3) and INFO(4) are unused.

ISAVE is an INTEGER array of length 17 that must not be altered by the user.

RSAVE is a REAL (DOUBLE PRECISION in the D version) array of length 9 that must not be altered by the user.

B.1.2 Control parameters

ICNTL and CNTL contain the control parameters of BICORA/AD. ICNTL controls the actions BICORA/AD takes, and CNTL controls the tolerances used by BICORA/AD. The default values are set by BICORI/ID.

ICNTL(1) is the stream number for error messages and has default value 6. Printing of error messages is suppressed if ICNTL(1) ≤ 0.

ICNTL(2) is the stream number for warning messages and has default value 6. Printing of warning messages is suppressed if ICNTL(1) ≤ 0.

ICNTL(3) controls whether the user wishes to use preconditioning. It has default value 0 and in this case no preconditioning is used. If ICNTL(3) is non-zero, the user will be expected to preform preconditioning when IACT $= 3$.

ICNTL(4) controls whether the convergence test offered by BICORA/AD is used. It has default value 0 and in this case the computed solution $\hat{x}$ is accepted if the 2-norm of the residual ($\|b - A\hat{x}\|_2$) is less or equal to max(CNTL(1)*(CNTL(2)+$\|x\|_2$*CNTL(3)),CNTL(4)), If the user does not want to use this test, ICNTL(4) should be non-zero. In this case, the user will be expected to test for convergence when IACT $= 1$.

ICNTL(5)	controls whether the user wishes to supply an initial estimate of the solution x. It has default value 0 and in this case the initial estimate is set to the zero vector. If the user wishes to supply an initial estimate, ICNTL(5) should be non-zero. In this case, the initial estimate should be put in the second column of W prior to the first call to BICORA/AD.
ICNTL(6)	determines the maximum number of iterations allowed. It has default value -1, and in this case the maximum number of iterations is equal to the order of the matrix A (N). If the user wishes to use a different maximum number of iterations, ICNTL(6) should be set to this number. In case of a negative number, the default will be used.
ICNTL(7), ICNTL(8)	have default value 0 and are unused by BICORA/AD
CNTL(1)	is one of the two convergence tolerances, as described under ICNTL(4). CNTL(1) has default value $\sqrt{u}$, where u is the relative machine precision. If ICNTL(4) is non-zero, this will not be used. See section B.2 for more information.
CNTL(2)	is the first variable used in the normwise backward error, and has a default value $\|b\|_2$. The default value is set in BICORA/AD, not in BICORI/ID. See section B.2 for more information.
CNTL(3)	is the second variable used in the normwise backward error, and has a default value of zero. If this is left zero, the norm of x will also not be calculated. See section B.2 for more information.
CNTL(4)	is one of the two convergence tolerances, as described under ICNTL(4). CNTL(2) has default value 0. If ICNTL(4) is non-zero, this will not be used. See section B.2 for more information.
CNTL(5)	is the breakdown tolerance. It has default value u, where u is the relative machine precision. If ρ is close enough to zero according to this tolerance, the method has broken down. See section B.2 for more information.
CNTL(4), CNTL(5)	have default value 0 and are unused by BICORA/AD

B.1.3 Error values

Upon the return of BICORA/AD, negative values for INFO(1) indicate an error and positive values indicate a warning. If everything went well, the value should be zero. Error messages are written to ICNTL(1) and warnings to ICNTL(2). Possible non-zero values for INFO(1) are:

-1 The value of N is out of range (< 1). There is an immediate return without any input parameters changed.

-2 The value of LWD is out of range ($< N$). There is an immediate return without any input parameters changed.

-3 The algorithm has broken down.

-4 The maximum amount of iterations determined by ICNTL(6) if it is not the default or N if ICNTL(6) is the default has been exceeded.

1 The convergence tolerance specified by the user in CNTL(1) lies outside the interval $(u, 1.0)$ where u is the machine precision. CNTL(1) is reset to the default value $\sqrt{u}$.

B.1.4 General information

Files needed to run the algorithm:

`bicor.f, ddeps.f`

Routines called:

BLAS `SNRM2/DNRM2, SCOPY/DCOPY, SAXPY/DAXPY, SSCAL/DSCAL, SDOT/DDOT`

HSL `FD15A/AD`

Restriction:

`LWD` $\geq$ `N` ≥ 1

B.2 Implementation

```
      SUBROUTINE BICORID(ICNTL,CNTL,ISAVE,RSAVE)
C Variables passed to the subroutine
      IMPLICIT NONE
      DOUBLE PRECISION CNTL(5)
      INTEGER ICNTL(8)
      INTEGER ISAVE(17)
      DOUBLE PRECISION RSAVE(9)
C Local variables
      INTEGER I
      DOUBLE PRECISION ZERO
      PARAMETER (ZERO=0.0D+0)
      DOUBLE PRECISION FD15AD
      EXTERNAL FD15AD
      INTRINSIC SQRT
      ICNTL(1) = 6
      ICNTL(2) = 6
      ICNTL(3) = 0
      ICNTL(4) = 0
      ICNTL(5) = 0
      ICNTL(6) = -1
      ICNTL(7) = 0
      ICNTL(8) = 0
      CNTL(1) = SQRT(FD15AD('E'))
      CNTL(2) = ZERO
      CNTL(3) = ZERO
      CNTL(4) = ZERO
      CNTL(5) = FD15AD('E')
      DO 10 I = 1, 15
      ISAVE(I) = 0
   10 CONTINUE
      DO 20 I = 1, 9
      RSAVE(I) = 0.0
   20 CONTINUE
      RETURN
      END
      SUBROUTINE BICORAD(IACT,N,W,LDW,LOCY,LOCZ,RESID,ICNTL,CNTL,INFO,
     +                   ISAVE,RSAVE)
C Variables passed to the subroutine
      IMPLICIT NONE
      DOUBLE PRECISION RESID
      INTEGER IACT,LDW,LOCY,LOCZ,N
      DOUBLE PRECISION CNTL(5),W(LDW,9)
      INTEGER ICNTL(8),INFO(4)
      INTEGER ISAVE(17)
      DOUBLE PRECISION RSAVE(9)
C Local variables
      DOUBLE PRECISION ONE, ZERO
      PARAMETER (ONE=1.0D+0,ZERO=0.0D+0)
      DOUBLE PRECISION BNRM2, RNRM2, RTNRM2, ALPHA, BETA, RHO, RHO1,
     +                 XNRM2
      INTEGER B, R, X, RPRM, ZR, ZRPRM, ZRHAT, P, PPRM, Q, QPRM,
     +        QPRMHAT, ZQPRMHAT, ITMAX, IPOS, I
      DOUBLE PRECISION DDOT, DNRM2, FD15AD
      EXTERNAL DDOT, DNRM2, FD15AD
      INTRINSIC ABS,MAX,SQRT
      EXTERNAL DAXPY,DCOPY,DSCAL
C Code
C Load all the local variables as they were on the last run
      IPOS     = ISAVE(1)
      ITMAX    = ISAVE(2)
      B        = ISAVE(3)
      X        = ISAVE(4)
      R        = ISAVE(5)
      RPRM     = ISAVE(6)
      ZR       = ISAVE(7)
      ZRPRM    = ISAVE(8)
      ZRHAT    = ISAVE(9)
      P        = ISAVE(10)
      PPRM     = ISAVE(11)
      Q        = ISAVE(12)
      QPRM     = ISAVE(13)
      QPRMHAT  = ISAVE(14)
      ZQPRMHAT = ISAVE(15)
      BNRM2    = RSAVE(1)
      ALPHA    = RSAVE(2)
      BETA     = RSAVE(3)
      RHO      = RSAVE(4)
      RHO1     = RSAVE(5)
      XNRM2    = RSAVE(6)
      IF (IACT.EQ.0) GO TO 10
      IF (IACT.LT.0) GO TO 1000
      IF (IACT.EQ.1 .AND. ICNTL(4).EQ.0) GO TO 1000
      IF (IACT.EQ.1 .AND. BNRM2.EQ.ZERO) GO TO 1000
      IF (IPOS.EQ.1) GO TO 40
      IF (IPOS.EQ.2) GO TO 60
      IF (IPOS.EQ.3) GO TO 70
      IF (IPOS.EQ.4) GO TO 80
      IF (IPOS.EQ.5) GO TO 90
      IF (IPOS.EQ.6) GO TO 100
      IF (IPOS.EQ.7) GO TO 110
      IF (IPOS.EQ.8) GO TO 120
   10 CONTINUE
      INFO(1) = 0
C No negative order possible
      IF (N.LE.0) THEN
         INFO(1) = -1
C W can't be larger than the order
      ELSE IF (LDW.LT.MAX(1,N)) THEN
         INFO(1) = -2
      END IF
C Something went wrong, return an error
      IF (INFO(1).LT.0) THEN
         IACT = -1
         IF (ICNTL(1).GT.0) WRITE (ICNTL(1),FMT=9000) INFO(1)
         GO TO 1000
      END IF
      B = 1
      X = 2
      R = 1
      RPRM = 3
      ZR = 4
      ZRPRM = 5
      ZRHAT = 6
      P = 7
      PPRM = 8
      Q = 9
      QPRM = 3
      QPRMHAT = 6
      ZQPRMHAT = 10
      INFO(2) = 0
```

```
C Max amount of iterations is N
      ITMAX = N
C or ICNTL(6) if specified
      IF (ICNTL(6).GT.0) ITMAX = ICNTL(6)
C If the 2 norm of b is zero, that means that b is zero, so the solution
C is zero, the residual is zero, everything is zero
      BNRM2 = DNRM2(N,W(1,B),1)
      IF (BNRM2.EQ.ZERO) THEN
         IACT = 1
         DO 20 I = 1,N
            W(I,X) = ZERO
            W(I,B) = ZERO
   20    CONTINUE
         RESID = ZERO
         GO TO 1000
      END IF
C In this case, the user may test for convergence when IACT = 1 is
C returned.
      IF (ICNTL(4).EQ.0) THEN
         IF (CNTL(1).LT.FD15AD('E') .OR. CNTL(1).GT.ONE) THEN
            INFO(1) = 1
            IF (ICNTL(2).GT.0) THEN
               WRITE (ICNTL(2),FMT=9010) INFO(1)
               WRITE (ICNTL(2),FMT=9020)
            END IF
            CNTL(1) = SQRT(FD15AD('E'))
         END IF
         IF (CNTL(2).EQ.ZERO) THEN
            CNTL(2) = BNRM2
         END IF
      END IF
C Initial estimate for x is the 0 vector
      IF (ICNTL(5).EQ.0) THEN
         DO 30 I = 1,N
            W(I,X) = ZERO
   30    CONTINUE
         GO TO 50
      ELSE
C or if ICNTL(5) is not 0, you need to have specified W(1,X)
         IF (DNRM2(N,W(1,X),1).EQ.ZERO) GO TO 50
         IPOS = 1
         IACT = 2
         LOCY = P
         LOCZ = X
         GO TO 1000
      END IF
C We have x and b, so r = -p (is Ax) + r (is b), so b-Ax
C We don't need b any more
   40 CALL DAXPY(N,-ONE,W(1,P),1,W(1,R),1)
   50 CONTINUE
C Set r prime as Ar
      IPOS = 2
      IACT = 2
      LOCY = RPRM
      LOCZ = R
      GO TO 1000
   60 CONTINUE
C Calculate zr prime on the first run
      IF (ICNTL(3).NE.0) THEN
         IPOS = 3
         IACT = 5
         LOCY = ZRPRM
         LOCZ = RPRM
         GO TO 1000
      ELSE
         CALL DCOPY(N,W(1,RPRM),1,W(1,ZRPRM),1)
      END IF
   70 CONTINUE
      INFO(2) = INFO(2) + 1
C Check maximum number of iterations has not been exceeded.
      IF (INFO(2).GT.ITMAX) THEN
         INFO(1) = -4
         IACT = -1
         IF (ICNTL(1).GT.0) THEN
            WRITE (ICNTL(1),FMT=9000) INFO(1)
            WRITE (ICNTL(1),FMT=9030) ITMAX
         END IF
         GO TO 1000
      END IF
C Perform the preconditioning operation
      IF (ICNTL(3).NE.0) THEN
         IPOS = 4
         IACT = 3
         LOCY = ZR
         LOCZ = R
         GO TO 1000
      ELSE
         CALL DCOPY(N,W(1,R),1,W(1,ZR),1)
      END IF
   80 CONTINUE
C Calculate zr hat
      IPOS = 5
      IACT = 2
      LOCY = ZRHAT
      LOCZ = ZR
      GO TO 1000
   90 CONTINUE
C See if the algorithm broke down. Otherwise, we can use rho in the
C remaining part of the algorithm
      RHO = DDOT(N, W(1,ZRPRM), 1, W(1,ZRHAT), 1)
      IF (ABS(RHO).LT.CNTL(5)*N) THEN
         RNRM2 = DNRM2(N,W(1,R),1)
         RTNRM2 = DNRM2(N,W(1,RPRM),1)
         IF (ABS(RHO).LT.CNTL(5)*RNRM2*RTNRM2) THEN
            INFO(1) = -3
            IACT = -1
            IF (ICNTL(1).GT.0) WRITE (ICNTL(1),FMT=9000) INFO(1)
            GO TO 1000
         END IF
      END IF
      IF (INFO(2).GT.1) THEN
         BETA = RHO/RHO1
         CALL DSCAL(N,BETA,W(1,P),1)
         CALL DAXPY(N,ONE,W(1,ZR),1,W(1,P),1)
         CALL DSCAL(N,BETA,W(1,PPRM),1)
         CALL DAXPY(N,ONE,W(1,ZRPRM),1,W(1,PPRM),1)
         CALL DSCAL(N,BETA,W(1,Q),1)
         CALL DAXPY(N,ONE,W(1,ZRHAT),1,W(1,Q),1)
      ELSE
         CALL DCOPY(N,W(1,ZR),1,W(1,P),1)
```

```
241          CALL DCOPY(N,W(1,ZRPRM),1,W(1,PPRM),1)
242          CALL DCOPY(N,W(1,ZRHAT),1,W(1,Q),1)
243       END IF
244       IPOS = 6
245       IACT = 4
246       LOCY = QPRMHAT
247       LOCZ = PPRM
248       GO TO 1000
249   100 CONTINUE
250 C Perform preconditioning
251       IF (ICNTL(3).NE.0) THEN
252          IPOS = 7
253          IACT = 5
254          LOCY = ZQPRMHAT
255          LOCZ = QPRMHAT
256          GO TO 1000
257       ELSE
258          CALL DCOPY(N,W(1,QPRMHAT),1,W(1,ZQPRMHAT),1)
259       END IF
260   110 CONTINUE
261       ALPHA = RHO/DDOT(N, W(1, ZQPRMHAT), 1, W(1,Q), 1)
262       CALL DAXPY(N,ALPHA,W(1,P),1,W(1,X),1)
263       CALL DAXPY(N,-ALPHA,W(1,Q),1,W(1,R),1)
264       CALL DAXPY(N,-ALPHA,W(1,ZQPRMHAT),1,W(1,ZRPRM),1)
265       RESID = DNRM2(N,W(1,R),1)
266       IPOS = 8
267 C The user can check the error if ICNTL(4) is non-zero at IACT.EQ.1
268       IF (ICNTL(4).NE.0) THEN
269          IACT = 1
270          GO TO 1000
271       ELSE
272          IF (CNTL(3).NE.ZERO) THEN
273             XNRM2 = DNRM2(N,W(1,X),1)
274          END IF
275          IF (RESID.LE.MAX(CNTL(1)*(CNTL(2)+XNRM2*CNTL(3)),CNTL(4))) THEN
276             IACT = 1
277             GO TO 1000
278          END IF
279       END IF
280   120 CONTINUE
281       RHO1 = RHO
282       GO TO 70
283  1000 CONTINUE
284 C Save all the local variables to use on the next run
285       ISAVE(1)  = IPOS
286       ISAVE(2)  = ITMAX
287       ISAVE(3)  = B
288       ISAVE(4)  = X
289       ISAVE(5)  = R
290       ISAVE(6)  = RPRM
291       ISAVE(7)  = ZR
292       ISAVE(8)  = ZRPRM
293       ISAVE(9)  = ZRHAT
294       ISAVE(10) = P
295       ISAVE(11) = PPRM
296       ISAVE(12) = Q
297       ISAVE(13) = QPRM
298       ISAVE(14) = QPRMHAT
299       ISAVE(15) = ZQPRMHAT
300       RSAVE(1)  = BNRM2
301       RSAVE(2)  = ALPHA
302       RSAVE(3)  = BETA
303       RSAVE(4)  = RHO
304       RSAVE(5)  = RHO1
305       RSAVE(6)  = XNRM2
306       RETURN
307  9000 FORMAT (/' Error message from BICOR. INFO(1) = ',I4)
308  9010 FORMAT (/' Warning message from BICOR. INFO(1) = ',I4)
309  9020 FORMAT (' Convergence tolerance out of range.')
310  9030 FORMAT (' Number of iterations required exceeds the maximum of ',
311     +       I8,/' allowed by ICNTL(6)')
312       END
```

C Implementation of CORS

C.1 User documentation

The implementation of the algorithm is based on the methods used in the Harwell Subroutine Library (HSL) [23]. Since the implementation is quite similar, the documentation is also quite similar to the documentation for the various subroutines in the HSL.

C.1.1 Argument lists and calling sequence

C.1.1.1 Initialization of the control parameters

The following subroutines have to be called before using the algorithm with `CORSA(D)`. For single precision we have

```
CALL CORSI(ICNTL, CNTL, ISAVE, RSAVE)
```

and for double precision we have

```
CALL CORSID(ICNTL, CNTL, ISAVE, RSAVE)
```

where

`ICNTL`	is an `INTEGER` array of length 8 that does not have to be set by the user. On return, it contains the default values as described in section C.1.2.
`CNTL`	is a `REAL` (`DOUBLE PRECISION` in the D version) array of length 5 that does not have to be set by the user. On return, it contains the default values and described in section C.1.2.
`ISAVE`	is an `INTEGER` array of length 19 that must not be altered by the user.
`RSAVE`	is a `REAL` (`DOUBLE PRECISION` in the D version) array of length 9 that must not be altered by the user.

C.1.1.2 Solving Ax=b

Here we will actually solve $Ax = b$. For single precision we have

```
 CALL CORSA(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL, INFO,
+           ISAVE, RSAVE)
```

and for double precision we have

```
 CALL CORSAD(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL, INFO,
+            ISAVE, RSAVE)
```

where

IACT is an INTEGER that indicate the action the user has to preform on every return of the CORSA/AD routines. Prior to the first call to CORSA/AD, IACT should be set to 0. Possible values are as follows:

-1 An error occurred, and the user must terminate the computation. The reason for the error is in INFO(1). See section C.1.3 for more information.

1 If ICNTL(4)=0 (the default value), convergence has been achieved, and the user should terminate the computation. If ICNTL(4) is nonzero, the user may test for convergence. If convergence has not been achieved, CORSA/AD should be called again, without changes to its arguments.

2 The user must preform the matrix-vector product

$$y = Az$$

and recall CORSA/AD. The vectors y and z are in columns LOCY and LOCZ of array W respectively. The user should not change z.

3 The user must preform the preconditioning operation

$$y = Mz$$

where M is the preconditioner. The vectors y and z are in columns LOCY and LOCZ of array W respectively. The user should not change z. Preconditioning is only used when ICNTL(3) is nonzero.

N is an INTEGER variable that must be set by the user to the order of the matrix A. The variable must be preserved between calls to CORSA/AD. This argument is not altered by the routine.

W is a REAL (DOUBLE PRECISION in the D version) two-dimensional array with dimensions (LWD, 13). Prior to the first call to CORSA/AD, the first column must hold the right hand side b. IF ICNTL(5) is nonzero, the second column must contain the initial estimate of the solution x. On exit, the first column holds the residual $r = b - A\hat{x}$ and the second column holds the estimate of the solution x. Other than the vector contained in the LOCYth column of W, W should remain unchanged between calls to CORSA/AD.

LWD is an INTEGER variable that must be set by the user to the first dimension of W. This argument is not altered by the routine. It should be greater than N.

LOCY, LOCZ are INTEGER variables that need not be set by the user. On return with IACT > 1, they indicate which columns of W should be used to preform the operations as specified under IACT (see above). These arguments must not be altered by the user between calls to CORSA/AD.

RESID is a REAL (DOUBLE PRECISION in the D version) variable that need not be set by the user. On return with IACT=1, it contains the 2-norm of the residual vector $\|b - A\hat{x}\|_2$, where $\hat{x}$ is the current estimate of the solution.

ICNTL is an INTEGER array of length 8 that has to be set by the user. The default values are set by a call to CORSA/AD as described in section C.1.1.1. Details of the control parameters are given in section C.1.2. This argument is not altered by the routine.

CNTL — is a REAL (DOUBLE PRECISION in the D version) array of length 5 that does not have to be set by the user. that has to be set by the user. The default values are set by a call to CORSA/AD as described in section C.1.1.1. Details of the control parameters are given in section C.1.2. This argument is not altered by the routine.

INFO — is an INTEGER array of length 4 that need not be set by the user. It is used to store information about the subroutine. On return of CORSA/AD, INFO(1) tells if the subroutine was successful (value 0) or if an error occurred (non-zero values). More information about this is in section C.1.3. INFO(2) holds the amount of iterations preformed by the subroutine. INFO(3) and INFO(4) are unused.

ISAVE — is an INTEGER array of length 17 that must not be altered by the user.

RSAVE — is a REAL (DOUBLE PRECISION in the D version) array of length 9 that must not be altered by the user.

C.1.2 Control parameters

ICNTL and CNTL contain the control parameters of CORSA/AD. ICNTL controls the actions CORSA/AD takes, and CNTL controls the tolerances used by CORSA/AD. The default values are set by CORSI/ID.

ICNTL(1) — is the stream number for error messages and has default value 6. Printing of error messages is suppressed if ICNTL(1) ≤ 0.

ICNTL(2) — is the stream number for warning messages and has default value 6. Printing of warning messages is suppressed if ICNTL(1) ≤ 0.

ICNTL(3) — controls whether the user wishes to use preconditioning. It has default value 0 and in this case no preconditioning is used. If ICNTL(3) is non-zero, the user will be expected to preform preconditioning when IACT $= 3$.

ICNTL(4) — controls whether the convergence test offered by CORSA/AD is used. It has default value 0 and in this case the computed solution $\hat{x}$ is accepted if the 2-norm of the residual ($\|b - A\hat{x}\|_2$) is less or equal to max(CNTL(1)*(CNTL(2)+$\|x\|_2$*CNTL(3)),CNTL(4)), If the user does not want to use this test, ICNTL(4) should be non-zero. In this case, the user will be expected to test for convergence when IACT $= 1$.

ICNTL(5) — controls whether the user wishes to supply an initial estimate of the solution x. It has default value 0 and in this case the initial estimate is set to the zero vector. If the user wishes to supply an initial estimate, ICNTL(5) should be non-zero. In this case, the initial estimate should be put in the second column of W prior to the first call to CORSA/AD.

ICNTL(6) — determines the maximum number of iterations allowed. It has default value -1, and in this case the maximum number of iterations is equal to the order of the matrix A (N). If the user wishes to use a different maximum number of iterations, ICNTL(6) should be set to this number. In case of a negative number, the default will be used.

ICNTL(7), ICNTL(8) — have default value 0 and are unused by CORSA/AD

CNTL(1) — is one of the two convergence tolerances, as described under ICNTL(4). CNTL(1) has default value $\sqrt{u}$, where u is the relative machine precision. If ICNTL(4) is non-zero, this will not be used. See section C.2 for more information.

CNTL(2) is the first variable used in the normwise backward error, and has a default value $\|b\|_2$. The default value is set in CORSA/AD, not in CORSI/ID. See section C.2 for more information.

CNTL(3) is the second variable used in the normwise backward error, and has a default value of zero. If this is left zero, the norm of x will also not not be calculated. See section C.2 for more information.

CNTL(4) is one of the two convergence tolerances, as described under ICNTL(4). CNTL(2) has default value 0. If ICNTL(4) is non-zero, this will not be used. See section C.2 for more information.

CNTL(5) is the breakdown tolerance. It has default value u, where u is the relative machine precision. If ρ is close enough to zero according to this tolerance, the method has broken down. See section C.2 for more information.

C.1.3 Error values

Upon the return of CORSA/AD, negative values for INFO(1) indicate an error and positive values indicate a warning. If everything went well, the value should be zero. Error messages are written to ICNTL(1) and warnings to ICNTL(2). Possible non-zero values for INFO(1) are:

-1 The value of N is out of range (< 1). There is an immediate return without any input parameters changed.

-2 The value of LWD is out of range ($< N$). There is an immediate return without any input parameters changed.

-3 The algorithm has broken down.

-4 The maximum amount of iterations determined by ICNTL(6) if it is not the default or N if ICNTL(6) is the default has been exceeded.

1 The convergence tolerance specified by the user in CNTL(1) lies outside the interval $(u, 1.0)$ where u is the machine precision. CNTL(1) is reset to the default value $\sqrt{u}$.

C.1.4 General information

Files needed to run the algorithm:

cors.f, ddeps.f

Routines called:

BLAS SNRM2/DNRM2, SCOPY/DCOPY, SAXPY/DAXPY, SSCAL/DSCAL, SDOT/DDOT

HSL FD15A/AD

Restriction:

LWD $\geq$ N ≥ 1

C.2 Implementation

```
      SUBROUTINE CORSID(ICNTL,CNTL,ISAVE,RSAVE)
C Variables passed to the subroutine
      IMPLICIT NONE
      DOUBLE PRECISION CNTL(5)
      INTEGER ICNTL(8)
      INTEGER ISAVE(19)
      DOUBLE PRECISION RSAVE(9)
C Local variables
      INTEGER I
      DOUBLE PRECISION ZERO
      PARAMETER (ZERO=0.0D+0)
      DOUBLE PRECISION FD15AD
      EXTERNAL FD15AD
      INTRINSIC SQRT
      ICNTL(1) = 6
      ICNTL(2) = 6
      ICNTL(3) = 0
      ICNTL(4) = 0
      ICNTL(5) = 0
      ICNTL(6) = -1
      ICNTL(7) = 0
      ICNTL(8) = 0
      CNTL(1) = SQRT(FD15AD('E'))
      CNTL(2) = ZERO
      CNTL(3) = ZERO
      CNTL(4) = ZERO
      CNTL(5) = FD15AD('E')
      DO 10 I = 1, 19
      ISAVE(I) = 0
   10 CONTINUE
      DO 20 I = 1, 9
      RSAVE(I) = 0.0
   20 CONTINUE
      RETURN
      END
      SUBROUTINE CORSAD(IACT,N,W,LDW,LOCY,LOCZ,RESID,ICNTL,CNTL,INFO,
     +                  ISAVE,RSAVE)
C Variables passed to the subroutine
      IMPLICIT NONE
      DOUBLE PRECISION RESID
      INTEGER IACT,LDW,LOCY,LOCZ,N
      DOUBLE PRECISION CNTL(5),W(LDW,13)
      INTEGER ICNTL(8),INFO(4)
      INTEGER ISAVE(19)
      DOUBLE PRECISION RSAVE(9)
C Local variables
      DOUBLE PRECISION TWO, ONE, ZERO
      PARAMETER (TWO=2.0D+0,ONE=1.0D+0,ZERO=0.0D+0)
      DOUBLE PRECISION BNRM2, RNRM2, RTNRM2, ALPHA, BETA, RHO, RHO1,
     +                 XNRM2
      INTEGER B, R, X, RPRM, ZR, ZRHAT, E, ZE, D, Q, ZQ, ZQHAT, H, F, G,
     +        ITMAX, IPOS, I
      DOUBLE PRECISION DDOT, DNRM2, FD15AD
      EXTERNAL DDOT, DNRM2, FD15AD
      INTRINSIC ABS,MAX,SQRT
      EXTERNAL DAXPY,DCOPY,DSCAL
C Code
C Load all the local variables as they were on the last run
      IPOS     = ISAVE(1)
      ITMAX    = ISAVE(2)
      B        = ISAVE(3)
      X        = ISAVE(4)
      R        = ISAVE(5)
      RPRM     = ISAVE(6)
      ZR       = ISAVE(7)
      ZRHAT    = ISAVE(8)
      E        = ISAVE(9)
      ZE       = ISAVE(10)
      D        = ISAVE(11)
      Q        = ISAVE(12)
      ZQ       = ISAVE(13)
      ZQHAT    = ISAVE(14)
      H        = ISAVE(15)
      F        = ISAVE(16)
      G        = ISAVE(17)
      BNRM2    = RSAVE(1)
      ALPHA    = RSAVE(2)
      BETA     = RSAVE(3)
      RHO      = RSAVE(4)
      RHO1     = RSAVE(5)
      XNRM2    = RSAVE(6)
      IF (IACT.EQ.0) GO TO 10
      IF (IACT.LT.0) GO TO 1000
      IF (IACT.EQ.1 .AND. ICNTL(4).EQ.0) GO TO 1000
      IF (IACT.EQ.1 .AND. BNRM2.EQ.ZERO) GO TO 1000
      IF (IPOS.EQ.1) GO TO 40
      IF (IPOS.EQ.2) GO TO 60
      IF (IPOS.EQ.3) GO TO 70
      IF (IPOS.EQ.4) GO TO 80
      IF (IPOS.EQ.5) GO TO 90
      IF (IPOS.EQ.6) GO TO 100
      IF (IPOS.EQ.7) GO TO 110
      IF (IPOS.EQ.8) GO TO 120
   10 CONTINUE
      INFO(1) = 0
C No negative order possible
      IF (N.LE.0) THEN
         INFO(1) = -1
C W can't be larger than the order
      ELSE IF (LDW.LT.MAX(1,N)) THEN
         INFO(1) = -2
      END IF
C Something went wrong, return an error
      IF (INFO(1).LT.0) THEN
         IACT = -1
         IF (ICNTL(1).GT.0) WRITE (ICNTL(1),FMT=9000) INFO(1)
         GO TO 1000
      END IF
      B = 1
      X = 2
      R = 1
      RPRM = 3
      ZR = 4
      ZRHAT = 5
      E = 6
      ZE = 7
      D = 8
      Q = 9
      ZQ = 5
      ZQHAT = 10
```

```
      H = 11
      F = 12
      G = 13
      INFO(2) = 0
C Max amount of iterations is N
      ITMAX = N
C or ICNTL(6) if specified
      IF (ICNTL(6).GT.0) ITMAX = ICNTL(6)
C If the 2 norm of b is zero, that means that b is zero, so the solution
C is zero, the residual is zero, everything is zero
      BNRM2 = DNRM2(N,W(1,B),1)
      IF (BNRM2.EQ.ZERO) THEN
         IACT = 1
         DO 20 I = 1,N
            W(I,X) = ZERO
            W(I,B) = ZERO
   20    CONTINUE
         RESID = ZERO
         GO TO 1000
      END IF
C In this case, the user may test for convergence when IACT = 1 is
C returned.
      IF (ICNTL(4).EQ.0) THEN
         IF (CNTL(1).LT.FD15AD('E') .OR. CNTL(1).GT.ONE) THEN
            INFO(1) = 1
            IF (ICNTL(2).GT.0) THEN
               WRITE (ICNTL(2),FMT=9010) INFO(1)
               WRITE (ICNTL(2),FMT=9020)
            END IF
            CNTL(1) = SQRT(FD15AD('E'))
         END IF
         IF (CNTL(2).EQ.ZERO) THEN
            CNTL(2) = BNRM2
         END IF
      END IF
C Initial estimate for x is the 0 vector
      IF (ICNTL(5).EQ.0) THEN
         DO 30 I = 1,N
            W(I,X) = ZERO
   30    CONTINUE
         GO TO 50
      ELSE
C or if ICNTL(5) is not 0, you need to have specified W(1,X)
         IF (DNRM2(N,W(1,X),1).EQ.ZERO) GO TO 50
         IPOS = 1
         IACT = 2
         LOCY = Q
         LOCZ = X
         GO TO 1000
      END IF
C We have x and b, so r = -q (is Ax) + r (is b), so b-Ax
C We don't need b any more
   40 CALL DAXPY(N,-ONE,W(1,Q),1,W(1,R),1)
   50 CONTINUE
C Set r prime as Ar
      IPOS = 2
      IACT = 2
      LOCY = RPRM
      LOCZ = R
      GO TO 1000
   60 CONTINUE
      INFO(2) = INFO(2) + 1
C Check maximum number of iterations has not been exceeded.
      IF (INFO(2).GT.ITMAX) THEN
         INFO(1) = -4
         IACT = -1
         IF (ICNTL(1).GT.0) THEN
            WRITE (ICNTL(1),FMT=9000) INFO(1)
            WRITE (ICNTL(1),FMT=9030) ITMAX
         END IF
         GO TO 1000
      END IF
C Perform the preconditioning operation
      IF (ICNTL(3).NE.0) THEN
         IPOS = 3
         IACT = 3
         LOCY = ZR
         LOCZ = R
         GO TO 1000
      ELSE
         CALL DCOPY(N,W(1,R),1,W(1,ZR),1)
      END IF
   70 CONTINUE
C Calculate zr hat
      IPOS = 4
      IACT = 2
      LOCY = ZRHAT
      LOCZ = ZR
      GO TO 1000
   80 CONTINUE
C See if the algorithm broke down. Otherwise, we can use rho in the
C remaining part of the algorithm
      RHO = DDOT(N, W(1,RPRM), 1, W(1,ZRHAT), 1)
      IF (ABS(RHO).LT.CNTL(5)*N) THEN
         RNRM2 = DNRM2(N,W(1,R),1)
         RTNRM2 = DNRM2(N,W(1,RPRM),1)
         IF (ABS(RHO).LT.CNTL(5)*RNRM2*RTNRM2) THEN
            INFO(1) = -3
            IACT = -1
            IF (ICNTL(1).GT.0) WRITE (ICNTL(1),FMT=9000) INFO(1)
            GO TO 1000
         END IF
      END IF
      IF (INFO(2).GT.1) THEN
         BETA = RHO/RHO1
C e = r + beta*h
         CALL DCOPY(N,W(1,R),1,W(1,E),1)
         CALL DAXPY(N,BETA,W(1,H),1,W(1,E),1)
C ze = zr + beta*f
         CALL DCOPY(N,W(1,ZR),1,W(1,ZE),1)
         CALL DAXPY(N,BETA,W(1,F),1,W(1,ZE),1)
C d = zrhat + beta*g
         CALL DCOPY(N,W(1,ZRHAT),1,W(1,D),1)
         CALL DAXPY(N,BETA,W(1,G),1,W(1,D),1)
C q = d+beta*(g+beta*q)
         CALL DSCAL(N,BETA,W(1,Q),1)
         CALL DAXPY(N,ONE,W(1,G),1,W(1,Q),1)
         CALL DSCAL(N,BETA,W(1,Q),1)
         CALL DAXPY(N,ONE,W(1,D),1,W(1,Q),1)
      ELSE
```

```
241          CALL DCOPY(N,W(1,R),1,W(1,E),1)
242          CALL DCOPY(N,W(1,ZRHAT),1,W(1,D),1)
243          CALL DCOPY(N,W(1,ZRHAT),1,W(1,Q),1)
244 C Calculate ze on the first run by a preconditioning operation
245          IF (ICNTL(3).NE.0) THEN
246             IPOS = 5
247             IACT = 3
248             LOCY = ZE
249             LOCZ = E
250             GO TO 1000
251          ELSE
252             CALL DCOPY(N,W(1,E),1,W(1,ZE),1)
253          END IF
254       END IF
255    90 CONTINUE
256 C Perform preconditioning
257       IF (ICNTL(3).NE.0) THEN
258          IPOS = 6
259          IACT = 3
260          LOCY = ZQ
261          LOCZ = Q
262          GO TO 1000
263       ELSE
264          CALL DCOPY(N,W(1,Q),1,W(1,ZQ),1)
265       END IF
266   100 CONTINUE
267       IPOS = 7
268       IACT = 2
269       LOCY = ZQHAT
270       LOCZ = ZQ
271       GO TO 1000
272   110 CONTINUE
273       ALPHA = RHO/DDOT(N, W(1, RPRM), 1, W(1,ZQHAT), 1)
274 C h=e-alpha*q
275       CALL DCOPY(N,W(1,E),1,W(1,H),1)
276       CALL DAXPY(N,-ALPHA,W(1,Q),1,W(1,E),1)
277 C f=ze-alpha*zq
278       CALL DSCAL(N,-ALPHA,W(1,ZQ),1)
279       CALL DCOPY(N,W(1,ZQ),1,W(1,F),1)
280       CALL DAXPY(N,ONE,W(1,ZE),1,W(1,F),1)
281 C g=d-alpha*zqhat
282       CALL DSCAL(N,-ALPHA,W(1,ZQHAT),1)
283       CALL DCOPY(N,W(1,ZQHAT),1,W(1,G),1)
284       CALL DAXPY(N,ONE,W(1,D),1,W(1,G),1)
285 C x=x+alpha*(2*ze-alpha*zq), -alpha*zq is already stored in zq
286       CALL DAXPY(N,TWO,W(1,ZE),1,W(1,ZQ),1)
287       CALL DAXPY(N,ALPHA,W(1,ZQ),1,W(1,X),1)
288 C r=r-alpha*(2*d-alpha*zqhat), -alpha*zqhat is already stored in zqhat
289       CALL DAXPY(N,TWO,W(1,D),1,W(1,ZQHAT),1)
290       CALL DAXPY(N,-ALPHA,W(1,ZQHAT),1,W(1,R),1)
291       RESID = DNRM2(N,W(1,R),1)
292       IPOS = 8
293 C The user can check the error if ICNTL(4) is non-zero at IACT.EQ.1
294       IF (ICNTL(4).NE.0) THEN
295          IACT = 1
296          GO TO 1000
297       ELSE
298          IF (CNTL(3).NE.ZERO) THEN
299             XNRM2 = DNRM2(N,W(1,X),1)
300          END IF
301          IF (RESID.LE.MAX(CNTL(1)*(CNTL(2)+XNRM2*CNTL(3)),CNTL(4))) THEN
302             IACT = 1
303             GO TO 1000
304          END IF
305       END IF
306   120 CONTINUE
307       RHO1 = RHO
308       GO TO 60
309  1000 CONTINUE
310 C Save all the local variables to use on the next run
311       ISAVE(1)  = IPOS
312       ISAVE(2)  = ITMAX
313       ISAVE(3)  = B
314       ISAVE(4)  = X
315       ISAVE(5)  = R
316       ISAVE(6)  = RPRM
317       ISAVE(7)  = ZR
318       ISAVE(8)  = ZRHAT
319       ISAVE(9)  = E
320       ISAVE(10) = ZE
321       ISAVE(11) = D
322       ISAVE(12) = Q
323       ISAVE(13) = ZQ
324       ISAVE(14) = ZQHAT
325       ISAVE(15) = H
326       ISAVE(16) = F
327       ISAVE(17) = G
328       RSAVE(1)  = BNRM2
329       RSAVE(2)  = ALPHA
330       RSAVE(3)  = BETA
331       RSAVE(4)  = RHO
332       RSAVE(5)  = RHO1
333       RSAVE(6)  = XNRM2
334       RETURN
335  9000 FORMAT (/' Error message from CORS. INFO(1) = ',I4)
336  9010 FORMAT (/' Warning message from CORS. INFO(1) = ',I4)
337  9020 FORMAT (' Convergence tolerance out of range.')
338  9030 FORMAT (' Number of iterations required exceeds the maximum of ',
339     +       I8,/' allowed by ICNTL(6)')
340       END
```

D Implementation of the testing application

```
      PROGRAM TEST

C !!!!!!!!!!!!!!!!!! WARNING !!!!!!!!!!!!!!!!!!!!!!!
C IF ILUPACK RETURNS AN ERROR, THERE WILL APPEAR A
C *** glibc detected *** double free or corruption
C ERROR. TO PREVENT THIS, RUN THE PROGRAM WITH
C MALLOC_CHECK_=0


C  Solve the linear system A x = b

C     .. Parameters ..
      IMPLICIT NONE
      INTEGER N, LDW
C     ..
C     .. Local Scalars ..
      DOUBLE PRECISION RESID
      INTEGER I, IACT, ROWS, COLS, NNZ, RNNZ
      INTEGER LOCY, LOCZ
      CHARACTER REP*10
      CHARACTER FIELD*7
      CHARACTER SYMM*19
      DOUBLE PRECISION BNRM2
C     ..
C     .. Local Arrays ..
      INTEGER LOCY2(2), LOCZ2(2)
      DOUBLE PRECISION CNTL(5)
      DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: W
      INTEGER ICNTL(8), INFO(4)
      INTEGER ISAVE(19)
      DOUBLE PRECISION RSAVE(9)
      COMPLEX, DIMENSION(:), ALLOCATABLE :: CVAL
      DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: RVAL
      INTEGER, DIMENSION(:), ALLOCATABLE :: IVAL
      INTEGER, DIMENSION(:), ALLOCATABLE :: INDX
      INTEGER, DIMENSION(:), ALLOCATABLE :: JNDX
      DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: RVALO
      INTEGER, DIMENSION(:), ALLOCATABLE :: INDXO
      INTEGER, DIMENSION(:), ALLOCATABLE :: JNDXO
      DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: SOLUTION
      DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: RHS
      DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: TEMP
C     ..
C     .. Local Variables needed for ILUPACK ..
      DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: ILUA
      INTEGER, DIMENSION(:), ALLOCATABLE :: ILUIA
      INTEGER, DIMENSION(:), ALLOCATABLE :: ILUJA
      INTEGER, DIMENSION(:), ALLOCATABLE :: ILUIND
C     ..
C     .. ILUPACK external parameters
      INTEGER ILUMATCHING, ILUMAXIT, ILULFIL, ILULFILS, ILUNRESTART,
     +        ILUIERR, ILUMIXEDPRECISION
      CHARACTER ILUORDERING*20
      DOUBLE PRECISION ILUDROPTOL, ILUDROPTOLS, ILUCONDEST, ILURESTOL,
     +                 ILUELBOW
      INTEGER*8 ILUPARAM, ILUPREC
      INTEGER  DGNLAMGFACTOR, DGNLAMGNNZ
      EXTERNAL DGNLAMGINIT, DGNLAMGSOL, DGNLAMGFACTOR, DGNLAMGTSOL,
     +         DGNLAMGNNZ, DGNLAMGDELETE
C     ..
C     .. Local Variables needed for BICGSTAB1 ..
      INTEGER L, LDRW, MXMV, LDWB
      DOUBLE PRECISION TOL
      DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: WORK
      DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: RWORK
      INTEGER, DIMENSION(:), ALLOCATABLE :: IWORK
      INTEGER, DIMENSION(:,:), ALLOCATABLE ::  IWORK2, IWORK3
      EXTERNAL BISTBL, PRECSOLVE, MV
C     ..
C     .. Variables for GMRES
      INTEGER M
      LOGICAL LSAVE(4)
      DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: H
      INTEGER LDH
C     ..
C     .. Variables for QMR
      INTEGER MAXPQ, MAXVW, MVEC
C     ..
C     .. Local Variables needed for testing ..
      REAL TARRAY(2)
      CHARACTER MNAME*50, TOL_STRING*5
      CHARACTER MFNAME*70
      CHARACTER FNAME*70
      INTEGER MVOPP, MAXMVP
      INTEGER FSTAT
      INTEGER NPRT, PREC, CALLS, RUNS
      PARAMETER (NPRT=20)
      DOUBLE PRECISION PREC_TOL, PREC_UPPER_TOL
C     ..
C     .. External Subroutines ..
      EXTERNAL MMREAD, MMINFO
      EXTERNAL CORSAD, CORSID
      EXTERNAL BICORAD, BICORID
      EXTERNAL MI26AD, MI26ID
      EXTERNAL MI25AD, MI25ID
      EXTERNAL MI24AD, MI24ID
      EXTERNAL MI23AD, MI23ID
      EXTERNAL AMUX, ATMUX
      DOUBLE PRECISION FD15AD, DNRM2
      EXTERNAL DCOPY, DNRM2, FD15AD
      EXTERNAL CLEARALL
      DOUBLE PRECISION ETA

C Some parameters
      PARAMETER (ETA = 1.0D-10)
      PARAMETER (MAXMVP = 20000)

C Open the list with matrices
      OPEN(11, FILE='matrices.txt', STATUS='OLD')
    5 CONTINUE
C     ..
C     .. Load our matrix and convert to the sparse matrix format ..
      READ(11, *, END=2000) MNAME
      WRITE(6, 4010) TRIM(MNAME)
      MFNAME = '/home/sven/matrices/' // TRIM(MNAME) // '.mtx'
      OPEN(1, FILE=MFNAME, STATUS='OLD')
      WRITE(6, 4020)
      CALL MMINFO(1, REP, FIELD, SYMM, ROWS, COLS, NNZ)
      IF (ROWS.NE.COLS) THEN
         WRITE(6,9990)
```

```
121          GO TO 5
122       END IF
123
124       N = ROWS
125       LDW = N
126
127       ALLOCATE(TEMP(N))
128       ALLOCATE(RVAL(NNZ))
129       ALLOCATE(CVAL(NNZ))
130       ALLOCATE(IVAL(NNZ))
131       ALLOCATE(INDX(NNZ))
132       ALLOCATE(JNDX(NNZ))
133       ALLOCATE(RVALO(NNZ))
134       ALLOCATE(JNDXO(NNZ))
135       ALLOCATE(INDXO(N+1))
136
137       CALL MMREAD(1, REP, FIELD, SYMM, ROWS, COLS, NNZ, NNZ,
138      +                 INDX, JNDX, IVAL, RVAL, CVAL)
139       CLOSE(1)
140 C     .. column indices and row pointers
141       WRITE(6, 4030)
142       CALL COOCSR(ROWS, NNZ, RVAL, INDX, JNDX, RVALO, JNDXO, INDXO)
143
144       DEALLOCATE(RVAL)
145       DEALLOCATE(CVAL)
146       DEALLOCATE(IVAL)
147       DEALLOCATE(INDX)
148       DEALLOCATE(JNDX)
149
150 C     ..
151 C Preconditioning is required
152       PREC = 0
153       ICNTL(3) = PREC
154 C  Set right hand side, b
155       WRITE(6, 4035)
156
157       ALLOCATE(SOLUTION(N))
158       ALLOCATE(RHS(N))
159
160       OPEN(13, FILE='/home/sven/matrices/' // TRIM(MNAME)
161      +          // '_rhs1.mtx', STATUS='OLD', IOSTAT=FSTAT)
162       IF (FSTAT.NE.0) THEN
163          OPEN(13, FILE='/home/sven/matrices/' // TRIM(MNAME)
164      +           // '_b.mtx', STATUS='OLD', IOSTAT=FSTAT)
165          IF (FSTAT.NE.0) THEN
166             WRITE(6, 4036)
167             SOLUTION = 1.0D+0
168             CALL AMUX(N, SOLUTION, RHS, RVALO, JNDXO, INDXO)
169          ELSE
170             CALL MMREAD(13, REP, FIELD, SYMM, ROWS, COLS, RNNZ, N,
171      +                  SOLUTION, SOLUTION, SOLUTION, RHS, SOLUTION)
172          END IF
173       ELSE
174          CALL MMREAD(13, REP, FIELD, SYMM, ROWS, COLS, RNNZ, N,
175      +                  SOLUTION, SOLUTION, SOLUTION, RHS, SOLUTION)
176       END IF
177       CLOSE(13)
178       BNRM2 = DNRM2(N, RHS, 1)
179 C  Preconditioner switch
180       PREC_TOL = 0.01
181       PREC_UPPER_TOL = 5.0
182    7  CONTINUE
183       RUNS = 0
184       PREC_TOL = PREC_TOL * 10.0
185       WRITE(TOL_STRING,'(F5.2)') PREC_TOL
186       TOL_STRING = REPEAT('0',5 - LEN_TRIM(ADJUSTL(TOL_STRING)))
187      +   // ADJUSTL(TOL_STRING)
188       FNAME = 'output/' // TRIM(MNAME) // '_' // TOL_STRING // '.txt'
189       OPEN(10, FILE=FNAME, STATUS='NEW', IOSTAT=FSTAT)
190       IF (FSTAT.NE.0) THEN
191          WRITE(6,FMT=4037)
192          RUNS = 100
193          GO TO 1010
194       END IF
195       WRITE(10, FMT=3010) TRIM(MNAME), N, NNZ
196
197       ALLOCATE(W(N,13))
198
199 C  Initialize data for the preconditioner
200 C
201       WRITE(6, FMT=4040)
202
203       ALLOCATE(ILUJA(NNZ))
204       ALLOCATE(ILUIA(N+1))
205       ALLOCATE(ILUA(NNZ))
206       ALLOCATE(ILUIND(N))
207
208       ILUJA = JNDXO
209       ILUIA = INDXO
210       CALL DCOPY(NNZ, RVALO, 1, ILUA, 1)
211
212       CALL DGNLAMGINIT(N, ILUIA, ILUJA, ILUA, ILUMATCHING,
213      +                    ILUORDERING, ILUDROPTOL, ILUDROPTOLS,
214      +                    ILUCONDEST, ILURESTOL, ILUMAXIT, ILUELBOW,
215      +                    ILULFIL, ILULFILS, ILUNRESTART,
216      +                    ILUMIXEDPRECISION, ILUIND)
217
218 c     threshold for ILU, default: 1e-2
219       ILUDROPTOL=PREC_TOL
220 c
221 c     threshold for the approximate Schur complements, default: 0.1*droptol
222       ILUDROPTOLS=0.1*ILUDROPTOL
223
224       ILUMIXEDPRECISION = 0
225       CALL CPU_TIME(TARRAY(1))
226
227       ILUIERR=DGNLAMGFACTOR(ILUPARAM, ILUPREC, N, ILUIA, ILUJA,
228      +          ILUA, ILUMATCHING, ILUORDERING, ILUDROPTOL, ILUDROPTOLS,
229      +          ILUCONDEST, ILURESTOL, ILUMAXIT, ILUELBOW, ILULFIL,
230      +          ILULFILS, ILUNRESTART, ILUMIXEDPRECISION, ILUIND)
231
232       CALL CPU_TIME(TARRAY(2))
233
234       IF (ILUIERR.EQ.-1) THEN
235          WRITE (6,'(A)') 'Error. input matrix may be wrong.'
236       ELSEIF (ILUIERR.EQ.-2) THEN
237          WRITE (6,'(A)')
238      +       'matrix L overflow, increase elbow and retry'
239       ELSEIF (ILUIERR.EQ.-3) THEN
240          WRITE (6,'(A)')
```

```
     +       'matrix U overflow, increase elbow and retry'
      ELSEIF (ILUIERR.EQ.-4) THEN
         WRITE (6,'(A)') 'Illegal value for lfil'
      ELSEIF (ILUIERR.EQ.-5) THEN
         WRITE (6,'(A)') 'zero row encountered'
      ELSEIF (ILUIERR.EQ.-6) THEN
         WRITE (6,'(A)') 'zero column encountered'
      ELSEIF (ILUIERR.EQ.-7) THEN
         WRITE (6,'(A)') 'buffers are too small'
      ELSEIF (ILUIERR.NE.0) THEN
         WRITE (6,'(A,I3)')
     +      'zero pivot encountered at step number', ILUIERR
      ENDIF
      IF (ILUIERR.NE.0) THEN
         WRITE(10, FMT=3032) ILUIERR
         GO TO 1000
      END IF

      WRITE(10, FMT=3020) TARRAY(2)-TARRAY(1), N,
     +                  DGNLAMGNNZ(ILUPARAM, ILUPREC)
      CALL FLUSH(10)
      WRITE(6, FMT=4050)
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCC        CORS        CCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      WRITE(10, FMT=3030) 'CORS'

      CALLS = 0
   10 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1
C Clear everything

      CALL CORSID(ICNTL, CNTL, ISAVE, RSAVE)
      CALL CLEARALL(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

C  Perform an iteration of the CORS method

      CALL CPU_TIME(TARRAY(1))
   20 CONTINUE
      CALL CORSAD(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE)

      IF (MVOPP.GE.MAXMVP) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) -88
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, -88)
         GO TO 30
      END IF

      IF (IACT.LT.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 30
      END IF

      IF (IACT.EQ.2) THEN
C  Perform the matrix-vector product
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,LOCZ), TEMP, N)
         CALL AMUX(N, TEMP, W(1, LOCY), RVALO, JNDXO, INDXO)
         MVOPP = MVOPP + 1
         GO TO 20
      END IF

      IF (IACT.EQ.3) THEN
C  Perform the preconditioning operation
         CALL DCOPY(N, W(1,LOCZ), 1, W(1,LOCY), 1)
         GO TO 20
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,2), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 30
      END IF

      CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) INFO(2), (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 10
      END IF

   30 CONTINUE

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCC         BICOR        CCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      WRITE(10,3030) 'BiCOR'

      CALLS = 0
  110 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1
C Clear everything

      CALL BICORID(ICNTL, CNTL, ISAVE, RSAVE)
      CALL CLEARALL(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

C  Perform an iteration of the BICOR method

      CALL CPU_TIME(TARRAY(1))
  120 CONTINUE
      CALL BICORAD(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE)

      IF (MVOPP.GE.MAXMVP) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) -88
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, -88)
```

```
         GO TO 130
      END IF

      IF (IACT.LT.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 130
      END IF

      IF (IACT.EQ.2) THEN
C  Perform the matrix-vector product
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,LOCZ), TEMP, N)
         CALL AMUX(N, TEMP, W(1, LOCY), RVALO, JNDXO, INDXO)
         MVOPP = MVOPP + 1
         GO TO 120
      END IF

      IF (IACT.EQ.3) THEN
C  Perform the preconditioning operation
         CALL DCOPY(N, W(1,LOCZ), 1, W(1,LOCY), 1)
         GO TO 120
      END IF

      IF (IACT.EQ.4) THEN
C  Perform the matrix-vector product
         CALL ATMUX(N, W(1, LOCZ), TEMP, RVALO, JNDXO, INDXO)
         CALL DGNLAMGTSOL(ILUPARAM, ILUPREC, TEMP, W(1,LOCY), N)
         MVOPP = MVOPP + 1
         GO TO 120
      END IF

      IF (IACT.EQ.5) THEN
C  Perform the preconditioning operation
         CALL DCOPY(N, W(1,LOCZ), 1, W(1,LOCY), 1)
         GO TO 120
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,2), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 130
      END IF

      CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) INFO(2), (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 110
      END IF

  130 CONTINUE

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCC        BICG-STAB     CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      WRITE(10,3030) 'BiCG-stab'

      CALLS = 0
  210 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1
C Clear everything

      CALL MI26ID(ICNTL, CNTL, ISAVE, RSAVE)
      CALL CLEARALL(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

C  Perform an iteration of the BICG-STAB method

      CALL CPU_TIME(TARRAY(1))
  220 CONTINUE
      CALL MI26AD(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE)

      IF (MVOPP.GE.MAXMVP) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) -88
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, -88)
         GO TO 230
      END IF

      IF (IACT.LT.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 230
      END IF

      IF (IACT.EQ.2) THEN
C  Perform the matrix-vector product
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,LOCZ), TEMP, N)
         CALL AMUX(N, TEMP, W(1, LOCY), RVALO, JNDXO, INDXO)
         MVOPP = MVOPP + 1
         GO TO 220
      END IF

      IF (IACT.EQ.3) THEN
C  Perform the preconditioning operation
         CALL DCOPY(N, W(1,LOCZ), 1, W(1,LOCY), 1)
         GO TO 220
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,2), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 230
      END IF
```

```
      CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) INFO(2), (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 210
      END IF

  230 CONTINUE

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCccccc         BICG       CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      WRITE(10,3030) 'BiCG'

      CALLS = 0
  310 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1
C Clear everything

      CALL MI25ID(ICNTL, CNTL, ISAVE, RSAVE)
      CALL CLEARALL(IACT, N, W, LDW, LOCY2, LOCZ2, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

C  Perform an iteration of the method

      CALL CPU_TIME(TARRAY(1))
  320 CONTINUE
      CALL MI25AD(IACT, N, W, LDW, LOCY2, LOCZ2, RESID, ICNTL, CNTL,
     +             INFO, ISAVE, RSAVE)

      IF (MVOPP.GE.MAXMVP) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) -88
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, -88)
         GO TO 330
      END IF

      IF (IACT.LT.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 330
      END IF

      IF (IACT.EQ.2) THEN
C  Perform the matrix-vector products
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,LOCZ2(1)),
     +                   TEMP, N)
         CALL AMUX(N, TEMP, W(1, LOCY2(1)), RVALO, JNDXO,
     +             INDXO)
         CALL ATMUX(N, W(1,LOCZ2(2)), TEMP, RVALO, JNDXO,
     +              INDXO)
         CALL DGNLAMGTSOL(ILUPARAM, ILUPREC, TEMP,
     +                   W(1, LOCY2(2)), N)
         MVOPP = MVOPP + 2
         GO TO 320
      END IF

      IF (IACT.EQ.3) THEN
C  Perform the preconditioning operations
         CALL DCOPY(N, W(1,LOCZ2(1)), 1, W(1,LOCY2(1)), 1)
         CALL DCOPY(N, W(1,LOCZ2(2)), 1, W(1,LOCY2(2)), 1)
         GO TO 320
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,2), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 330
      END IF

      CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) INFO(2), (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 310
      END IF

  330 CONTINUE

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCC        CGS        CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      WRITE(10,3030) 'CGS'

      CALLS = 0
  410 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1
C Clear everything

      CALL MI23ID(ICNTL, CNTL, ISAVE, RSAVE)
      CALL CLEARALL(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

C  Perform an iteration of the BICG-STAB method

      CALL CPU_TIME(TARRAY(1))
  420 CONTINUE
      CALL MI23AD(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +             INFO, ISAVE, RSAVE)

      IF (MVOPP.GE.MAXMVP) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) -88
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, -88)
         GO TO 430
      END IF
```

```
      IF (IACT.LT.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 430
      END IF

      IF (IACT.EQ.2) THEN
C  Perform the matrix-vector product
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,LOCZ), TEMP, N)
         CALL AMUX(N, TEMP, W(1, LOCY), RVALO, JNDXO, INDXO)
         MVOPP = MVOPP + 1
         GO TO 420
      END IF

      IF (IACT.EQ.3) THEN
C  Perform the preconditioning operation
         CALL DCOPY(N, W(1,LOCZ), 1, W(1,LOCY), 1)
         GO TO 420
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,2), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 430
      END IF

      CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) INFO(2), (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 410
      END IF

  430 CONTINUE

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCcccccc        GMRES       CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      WRITE(10,3030) 'GMRES'

      CALLS = 0
      M = 100
      LDH = M+1
      ALLOCATE(H(LDH,M+2))
      DEALLOCATE(W)
      ALLOCATE(W(LDW,M+7))
  510 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1
C Clear everything

      CALL MI24ID(ICNTL, CNTL, ISAVE, RSAVE, LSAVE)
      CALL CLEARALL(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

C  Perform an iteration of the method

      CALL CPU_TIME(TARRAY(1))
  520 CONTINUE
      CALL MI24AD(IACT, N, M, W, LDW, LOCY, LOCZ, H, LDH,  RESID,
     +              ICNTL, CNTL, INFO, ISAVE, RSAVE, LSAVE)

      IF (MVOPP.GE.MAXMVP) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) -88
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, -88)
         GO TO 530
      END IF

      IF (IACT.LT.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 530
      END IF

      IF (IACT.EQ.2) THEN
C  Perform the matrix-vector products
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,LOCZ), TEMP, N)
         CALL AMUX(N, TEMP, W(1, LOCY), RVALO, JNDXO, INDXO)
         MVOPP = MVOPP + 1
         GO TO 520
      END IF

      IF (IACT.EQ.3) THEN
C  Perform the preconditioning operations
         CALL DCOPY(N, W(1,LOCZ), 1, W(1,LOCY), 1)
         GO TO 520
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,2), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))
         GO TO 530
      END IF

      CALL SAVERESULTS(TARRAY, INFO(2), MVOPP, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) INFO(2), (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 510
      END IF

  530 CONTINUE
```

```
      DEALLOCATE(H)

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCccc      BICGSTABl       CCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      L = 1
  600 CONTINUE
      L = L + 1
      WRITE(10,3031) L
      CALLS = 0
      ALLOCATE(WORK(N, 3+2*(L+1)))
      LDRW = (L+1)*(3+2*(L+1))
      LDWB = N*(3+2*(L+1))
      ALLOCATE(RWORK(L+1, 3+2*(L+1)))
      ALLOCATE(IWORK(L+1))
  610 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1

      MXMV = MIN(2 * N, MAXMVP)
      TOL = ETA
C Clear everything
      CALL CLEARALL(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

C  Perform an iteration of the method

      CALL CPU_TIME(TARRAY(1))
      CALL BISTBL(L, N, W(1,2), W(1,1), MV, PRECSOLVE, TOL,
     +            MXMV, WORK, LDWB, RWORK, LDRW, IWORK, INFO, NNZ,
     +            RVALO, JNDXO, INDXO, ILUPARAM, ILUPREC, TEMP, BNRM2)

      IF (INFO(1).NE.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, L * INFO(2), MXMV, INFO(1))
         GO TO 630
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,2), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, L * INFO(2), MXMV, INFO(1))
         GO TO 630
      END IF

      CALL SAVERESULTS(TARRAY, L * INFO(2), MXMV, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) INFO(2), (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 610
      END IF

  630 CONTINUE

      DEALLOCATE(WORK)
      DEALLOCATE(RWORK)
      DEALLOCATE(IWORK)

      IF (L.LT.3) THEN
         GO TO 600
      END IF

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCccccc         QMR        CCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      WRITE(10,3030) 'QMR'

      CALLS = 0
      MAXVW = 1
      MAXPQ = 1
      MVEC = MAXPQ + MAXVW
      M = MAXPQ + MAXVW + 2
      L = N

      ALLOCATE(WORK(M, 8*M+18))
      ALLOCATE(IWORK2(6, L+2))
      ALLOCATE(IWORK3(M, 13))
      DEALLOCATE(W)
      ALLOCATE(W(LDW, 5*MVEC+3))

  710 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1
C Clear everything
      CALL CLEARALL(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

      CALL DCOPY(N, RHS, 1, W(1,2), 1)
      RESID = 1.0
      TOL = ETA
      MAXVW = 1
      MAXPQ = 1
      MVEC = MAXPQ + MAXVW
      M = MAXPQ + MAXVW + 2
      L = N

C  Perform an iteration of the method

      CALL CPU_TIME(TARRAY(1))
  720 CONTINUE
      CALL DUCPL(LDW, N, L, MAXPQ, MAXVW, M, MVEC, RESID, WORK, IWORK2,
     +                   IWORK3, W, TOL, INFO)

      IF (MVOPP.GE.MAXMVP) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) -88
         CALL SAVERESULTS(TARRAY, L, MVOPP, -88)
         INFO(2) = -1
         CALL DUCPL(LDW, N, L, MAXPQ, MAXVW, M, MVEC, RESID, WORK,
     +                    IWORK2, IWORK3, W, TOL, INFO)
         GO TO 730
      END IF

```

```
      IF (INFO(2).EQ.1) THEN
C  Perform the matrix-vector products
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,INFO(3)), TEMP, N)
         CALL AMUX(N, TEMP, W(1, INFO(4)), RVALO, JNDXO, INDXO)
         MVOPP = MVOPP + 1
         GO TO 720
      END IF

      IF (INFO(2).EQ.2) THEN
C  Perform the transpose matrix-vector products
         CALL ATMUX(N, W(1,INFO(3)), TEMP, RVALO, JNDXO, INDXO)
         CALL DGNLAMGTSOL(ILUPARAM, ILUPREC, TEMP, W(1, INFO(4)), N)
         MVOPP = MVOPP + 1
         GO TO 720
      END IF

      IF (INFO(1).NE.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, L, MVOPP, INFO(1))
         GO TO 730
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,1), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, L, MVOPP, INFO(1))
         GO TO 730
      END IF

      CALL SAVERESULTS(TARRAY, L, MVOPP, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) L, (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 710
      END IF

  730 CONTINUE

      DEALLOCATE(WORK)
      DEALLOCATE(IWORK2)
      DEALLOCATE(IWORK3)

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCcccccc        TFQMR        CCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      WRITE(10,3030) 'TFQMR'

      CALLS = 0

      DEALLOCATE(W)
      ALLOCATE(W(LDW, 13))

  810 CONTINUE
      CALLS = CALLS + 1
      RUNS = RUNS + 1
C Clear everything

      CALL CLEARALL(IACT, N, W, LDW, LOCY, LOCZ, RESID, ICNTL, CNTL,
     +              INFO, ISAVE, RSAVE, RHS, PREC, TEMP, ETA)

      CALL DCOPY(N, RHS, 1, W(1,2), 1)
      L = N
      TOL = ETA

C  Perform an iteration of the method

      CALL CPU_TIME(TARRAY(1))
  820 CONTINUE
      CALL DUTFX (LDW, N, L, W, TOL, INFO)

      IF (MVOPP.GE.MAXMVP) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) -88
         CALL SAVERESULTS(TARRAY, L, MVOPP, -88)
         INFO(2) = -1
         CALL DUTFX (LDW, N, L, W, TOL, INFO)
         GO TO 830
      END IF

      IF (INFO(2).EQ.1) THEN
C  Perform the matrix-vector products
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,INFO(3)), TEMP, N)
         CALL AMUX(N, TEMP, W(1, INFO(4)), RVALO, JNDXO, INDXO)
         MVOPP = MVOPP + 1
         GO TO 820
      END IF

      IF (INFO(1).NE.0) THEN
         CALL CPU_TIME(TARRAY(2))
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, L, MVOPP, INFO(1))
         GO TO 830
      END IF

      CALL CPU_TIME(TARRAY(2))

      CALL DGNLAMGSOL(ILUPARAM, ILUPREC, W(1,1), TEMP, N)
      IF (ISNAN(TEMP(1))) THEN
         INFO(1) = -99
         WRITE (6, FMT=9020) INFO(1)
         CALL SAVERESULTS(TARRAY, L, MVOPP, INFO(1))
         GO TO 830
      END IF

      CALL SAVERESULTS(TARRAY, L, MVOPP, INFO(1))

C  Solution found
      WRITE (6, FMT=9000) L, (TEMP(I), I=1, NPRT)
      IF (INFO(1).GT.0) WRITE (6, FMT=9010) INFO(1)

      IF (CALLS.LE.5) THEN
         GO TO 810
      END IF
```

```
 830 CONTINUE

1000 CONTINUE
     CLOSE(10)
C  Deallocate everything
C  Deallocate data for the preconditioner
     IF (ILUIERR.EQ.0) THEN
        CALL DGNLAMGDELETE(ILUPARAM, ILUPREC)
     END IF

     DEALLOCATE(ILUIA)
     DEALLOCATE(ILUJA)
     DEALLOCATE(ILUA)
     DEALLOCATE(ILUIND)

     DEALLOCATE(W)

1010 CONTINUE

     IF (PREC_TOL.LT.PREC_UPPER_TOL) THEN
        IF (RUNS.GT.10) THEN
           GO TO 7
        END IF
     END IF

     DEALLOCATE(TEMP)
     DEALLOCATE(SOLUTION)
     DEALLOCATE(RHS)
     DEALLOCATE(RVALO)
     DEALLOCATE(JNDXO)
     DEALLOCATE(INDXO)

     GO TO 5
2000 CONTINUE
     CLOSE(11)
     STOP
3010 FORMAT ( '% Matrix:     name,        N,        NNZ', / A18,
    +        ', ', I8, ', ', I10 )
3020 FORMAT ( '% Prec:       time,        N,        NNZ', / F18.3,
    +        ', ', I8,
    +         ', ', I10,
    +         / '% Results:    time,     iter,   m-v prod, error')
3030 FORMAT ( '% ' A)
3031 FORMAT ( '% BICGSTAB(', I1, ')')
3032 FORMAT ( '% ' I4)
4010 FORMAT (' New matrix: ', A)
4020 FORMAT (' Reading matrix ' )
4030 FORMAT (' Converting to CSR ' )
4035 FORMAT (' Forming RHS ' )
4036 FORMAT (' No RHS file found ' )
4037 FORMAT (' Already done ' )
4040 FORMAT (' Building preconditioner ')
4050 FORMAT (' Starting solvers ')
9000 FORMAT (/ ' Solution found', / I6, ' iterations required ',
    +        //' Solution = ', / (1P, 5D10.2))
9010 FORMAT (' Warning: INFO( 1 ) = ', I3, ' on exit ')
9020 FORMAT (' Error return: INFO( 1 ) = ', I3, ' on exit ')
9990 FORMAT (' MATRIX NOT SQUARE ')
     END PROGRAM TEST
```

```

C Subroutines used by BiCGSTAB(ell)
      SUBROUTINE MV(N, X, Y, NNZ, RVALO, JNDXO,
     +     INDXO, ILUPARAM, ILUPREC, TEMP)
      INTEGER N
      DOUBLE PRECISION X(N), Y(N)
      DOUBLE PRECISION RVALO(NNZ), TEMP(N)
      INTEGER INDXO(N+1), JNDXO(NNZ)
      INTEGER*8 ILUPARAM, ILUPREC
      EXTERNAL AMUX, DGNLAMGSOL
         CALL DGNLAMGSOL(ILUPARAM, ILUPREC, X, TEMP, N)
      CALL AMUX(N, TEMP, Y, RVALO, JNDXO, INDXO)
      END SUBROUTINE MV

      SUBROUTINE PRECSOLVE(N, X)
      INTEGER N
      DOUBLE PRECISION X(N)
C     Right precontitioning. Do nothing
      END SUBROUTINE PRECSOLVE

C Subroutine to save all results
      SUBROUTINE SAVERESULTS(TARRAY, ITER, MVOPP, ERROR)
      REAL TARRAY(2)
      INTEGER ITER, MVOPP, ERROR
      WRITE(10,100) TARRAY(2)-TARRAY(1), ITER, MVOPP, ERROR
      CALL FLUSH(10)
      MVOPP = 0
      RETURN
  100 FORMAT (F18.3, ', ', I8, ', ', I10,
     +        ', ', I3)
      END SUBROUTINE SAVERESULTS
```

E Implementation of the data analysis tool

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

import os
import sys
import scipy
import matplotlib.pyplot as pyplot
from optparse import OptionParser
import shutil

import TableToLatex
import MatrixFileFetcher

class Parser(object):
    ''' The parser class that parses the matrix files of a certain
    preconditioner tolerance'''

    TIMETOL = 0.05
    OFFTOL = 0.1
    UPPERTOL = 150.0
    MATRIXMODE = 4
    COLUMNS = 4
    MAX_MVPS = 20000

    def __init__(self, prec_tol, prec_tols):
        ''' constructor '''
        self.mode = 0
        self.solver = 0
        self.options = self.set_options()
        self.solver_dict = ['CORS', 'BiCOR', 'BiCGSTAB', 'BiCG',
            'CGS', 'GMRES(100)', 'BiCGSTAB(2)', 'BiCGSTAB(3)', 'QMR', 'TFQMR']
        self.solver_used_amount = len(self.solver_dict)
        self.exclude = sorted(self.options.exclude, reverse=True)
        for i in self.exclude:
            self.solver_dict.pop(i)
        self.solver_amount = len(self.solver_dict)
        self.prec_tol = prec_tol
        self.prec_tols = prec_tols
        self.prefix = './test'
        self.matrix_file = ''
        self.matrix_parser = MatrixParser(Parser.COLUMNS)
        self.matrix_name = ''
        self.matrix_number = 0
        self.data = SolverData(self.solver_amount)
        self.current = scipy.zeros([self.solver_used_amount,
                                    Parser.COLUMNS + 1])
        self.plotting_pattern = ['b-', 'g--', 'r-.', 'c:', 'm-', 'y--',
                                 'k-.', 'b:', 'g-', 'r--']
        self.skip = False
        self.rerun = ''
        self.time = 0

        self.latex_list = []
        self.current_latex_list = []
        self.first = True

    def run(self):
        ''' run the parser '''
        filename_list = os.listdir(self.prefix)
        filename_list = sorted(filename_list, key=str.lower)
        try:
            matrix_file_list, dummy = MatrixFileFetcher.fetch()
            matrix_file_list = matrix_file_list \
            if (self.options.n <= 0 or self.options.n > len(matrix_file_list)) \
            else matrix_file_list[0:self.options.n]
        except IOError:
            matrix_file_list = []
        for filename in filename_list:
            matrix, sep, prec_tol = filename.rpartition('_')
            try:
                prec_tol = float(prec_tol.rpartition('.')[0])
            except ValueError:
                print prec_tol
                self.quit_on_error()
            if prec_tol != self.prec_tol or (matrix_file_list and matrix not in
                matrix_file_list):
                if matrix_file_list and matrix not in matrix_file_list:
                    print 'Matrix not in matrix file list: ', matrix
                continue
            self.current = scipy.zeros([self.solver_used_amount,
                                        Parser.COLUMNS + 1])
            self.mode = 0
            self.solver = 0
            self.skip = False
            self.matrix_file = self.prefix+'/'+filename
            matrix_file = open(self.matrix_file)
            lines = matrix_file.readlines()
            matrix_file.close()
            for line in lines:
                if line.startswith('%'):
                    # line is a comment
                    self.mode += 1
                    if self.mode > Parser.MATRIXMODE:
                        # read the data of the previous solver into self.current
                        self.read()
                        self.solver += 1
                    self.matrix_parser.clear()
                    continue
                elif self.mode == 1:
                    # line with the matrix name
                    spline = self.split(line)
                    self.matrix_name = spline[0]
                    self.current_latex_list = [self.matrix_name]
                    if self.options.verbose:
                        print 'Read: {}: {}'.format(self.matrix_number + 1,
                                                    self.matrix_name)
                elif self.mode >= Parser.MATRIXMODE:
                    # line contains solver data
                    spline = self.split(line)
                    self.matrix_parser.add(spline)
                if self.skip:
                    break
            if self.skip:
                continue
            self.read()
            self.write_time_to_file()
            self.add()
        self.draw_plot()
        self.draw_latex_tables()
        self.draw_table()
```

```
120
121     def split(self, line):
122         ''' separate the data on each line '''
123         spline = line.split(',')
124         for i in range(len(spline)):
125             spline[i] = spline[i].strip()
126         return spline
127
128     def read(self):
129         ''' read the data for one parser '''
130         if self.solver in self.exclude:
131             return
132
133         data = self.matrix_parser.read()
134
135         times = data[:, 0]
136         self.time += sum(times)
137         length = len(times)
138         if length == 0:
139             print 'Length is 0: {}'.format(self.matrix_name)
140             self.skip = True
141             return
142
143         average = float(sum(times)) / length
144
145         if self.matrix_parser.fail:
146             excl = len([item for item in self.exclude if item < self.solver])
147             err = int(data[0, 3])
148             # 1 is breakdown, 2 is iterations, 3 is NAN
149             error_type = 2 if (err < -3 and err > -99) \
150                 or ((self.solver == 6 or self.solver == 7) and err == 1) \
151                 or ((self.solver == 8 or self.solver == 9) and err == 4) \
152                 else (3 if err == -99 else 1)
153             self.data.solver[self.solver - excl].error(error_type)
154             self.current[self.solver, 3] = data[0, 3]
155             self.write_to_current(average, data, error_type)
156             return
157
158         # check the time values for validity
159         for value in times:
160             if value < Parser.TIMETOL:
161                 print 'Time too short: {}, {}, {}'.format(value,
162                     average, self.matrix_name)
163                 if not self.skip:
164                     self.skip = True
165                     self._move_matrix('small')
166                     self._move_results('small')
167             if value > (1.0 + Parser.OFFTOL) * average:
168                 print 'Result not accurate enough: {}, {}, {}'\
169                 .format(value, average, self.matrix_name)
170                 length -= 1
171                 average = float(average * (length + 1)) / length
172
173         self.write_to_current(average, data)
174
175     def write_to_current(self, average, data, error=None):
176         ''' write the data to self.current '''
177         self.current[self.solver, 0] = average
178         self.current[self.solver, 1] = data[0, 1]
179         self.current[self.solver, 2] = data[0, 2]
180         self.current[self.solver, 3] = data[0, 3]
181         if error is None:
182             self.current_latex_list.append('')
183         elif self.solver not in self.exclude:
184             self.current_latex_list.append(
185                 '\\colorbox{yellow}{iterations}' if error == 2 else \
186                 ('\\colorbox{magenta}{NAN}' if error == 3 else \
187                 '\\colorbox{red}{breakdown}'))
188
189     def add(self):
190         ''' add the data for the current matrix to self.data usning the
191             ratio method '''
192         if self.options.restriction is not None and not \
193                 eval(str(self.current[int(self.options.restriction[0]), 3]) \
194                 + self.options.restriction[1]):
195             return
196         for i in range(3):
197             self.ratio(i)
198         if not self.skip:
199             self.matrix_number += 1
200             self.latex_list.append(self.current_latex_list)
201             print 'Added: {}: {}'.format(self.matrix_number,
202                                          self.matrix_name)
203
204     def ratio(self, column):
205         ''' calculate the ratio and add it to self.data '''
206         invalid_runs = sorted([item for item in range(self.solver_used_amount) \
207                         if self.current[item, 3] != 0 or item in self.exclude],
208                         reverse = True)
209         stat = self.current[:, column]
210         valid_runs = list(stat)
211         for i in invalid_runs:
212             valid_runs.pop(i)
213         if valid_runs == []:
214             if column == 1:
215                 print 'No valid runs at all: {}'.format(self.matrix_name)
216                 if self.prec_tol == self.prec_tols[0]:
217                     self._move_matrix('bad')
218                     self._move_results('bad')
219                     self.skip = True
220             winner = 1
221         else:
222             winner = min(valid_runs)
223         j = 0
224         latex_list = {}
225         # calculate the ratios of all used solvers and add them where needed
226         for i in range(self.solver_used_amount):
227             if i in self.exclude:
228                 continue
229             if self.current[i, 3] != 0:
230                 ratio = Parser.UPPERTOL
231                 if column == 0 and self.options.verbose:
232                     print 'Failed method: {}, {}'.format(self.matrix_name,
233                         i)
234                 elif column == 2 and (self.current[i, 3] == -88 or \
235                         (self.current[i, 3] == 1 and (i == 6 or i == 7))) and \
236                         self.current[i, column] < 20 * winner and \
237                         self.current[i, column] >= Parser.MAX_MVPS:
238                     if self.options.verbose:
239                         print 'Max runs not enough: {}, {}, {}, {}'.format(\
```

```
                            self.matrix_name, i, winner,
                            self.current[i, column])
                    self.rerun += '{!s}\n{:d}\n{:05.2f}\n{:d}\n'.format(\
                        self.matrix_name, i, self.prec_tol, int(winner))
            else:
                ratio = float(stat[i]) / winner
                if ratio >= Parser.UPPERTOL:
                    print 'Ratio too high: {}, {}, {}, {}, {}'.format(stat[i],
                        winner, self.matrix_name, i, self.prec_tol)
                    print stat
                    self.quit_on_error()
                if column == 2 and stat[i] > Parser.MAX_MVPS:
                    print 'Solver converged with more than {} mvps: {}, \
                        {}'.format(Parser.MAX_MVPS, stat[i], self.matrix_name)
            self.data.solver[j].append(ratio, column)
            latex_list[j] = ratio
            j += 1
        if column == 0:
            sorted_latex_list = sorted(latex_list, key=latex_list.get)
            for i, item in enumerate(sorted_latex_list):
                if latex_list[item] != Parser.UPPERTOL:
                    self.current_latex_list[item+1] = ('\\colorbox{green!'+\
                        str(int(100/len(latex_list)*(len(latex_list)-i)))+\
                        '!black}{'+str(i+1)+'}')
                    #print self.current_latex_list[item]

    def draw_latex_tables(self):
        ''' draw a latex table of all the errors '''
        tex_parser = TableToLatex.TableToLatex()
        header_list = list(self.solver_dict)
        header_list.insert(0, 'matrix name')
        tex_parser.set_header(header_list,
            '|p{100px}|'+ 'p{65px}|'*(len(header_list)-1))
        tex_parser.add_package('xcolor')
        tex_parser.set_table(self.latex_list)
        tex_parser.make('table' + str(self.prec_tol))
        tex_parser = TableToLatex.TableToLatex()
        header_list = ['Solver', 'breakdown', 'iterations', 'NAN']
        tex_parser.set_header(header_list)
        table = []
        for i in range(self.solver_amount):
            table.append([self.solver_dict[i],
                          str(self.data.solver[i].errors),
                          str(self.data.solver[i].iter_errors),
                          str(self.data.solver[i].nan_errors)])
        tex_parser.use_separator(False)
        tex_parser.set_table(table)
        tex_parser.set_caption('Failures with a perconditioner tolerance of '\
            + str(self.prec_tol), str(self.prec_tol))
        tex_parser.make('failure_table' + str(self.prec_tol))

    def draw_table(self):
        ''' draw a table containing some userful numbers '''
        print 'Errors'
        for i in range(self.solver_amount):
            print '{:12}: {}'.format(self.solver_dict[i],
                                     self.data.solver[i].errors)
        print 'More than max iterations'
        for i in range(self.solver_amount):
            print '{:12}: {}'.format(self.solver_dict[i],
                                     self.data.solver[i].iter_errors)
        print 'Winner in terms of time'
        for i in range(self.solver_amount):
            print '{:12}: {}'.format(self.solver_dict[i],
                                     self.data.solver[i].timer.count(1.0))
        print 'Winner in terms of matrix-vector products'
        for i in range(self.solver_amount):
            print '{:12}: {}'.format(self.solver_dict[i],
                                     self.data.solver[i].mvps.count(1.0))

    def draw_plot(self):
        ''' draw the plots '''
        time_plot = self.make_figure(1)
        mvp_plot = self.make_figure(2)
        iter_plot = self.make_figure(3)
        time_plot.savefig(str(self.prec_tol) + '_time.png', bbox_inches='tight')
        mvp_plot.savefig(str(self.prec_tol) + '_mvp.png', bbox_inches='tight')
        iter_plot.savefig(str(self.prec_tol) + '_iter.png', bbox_inches='tight')
        #pyplot.show()

    def make_figure(self, number):
        ''' make a figure '''
        figure = pyplot.figure()
        #pyplot.subplots_adjust(left=0.1, right=0.1, top=0.1, bottom=0.1)
        plot = figure.add_subplot(111)
        for i in range(self.solver_amount):
            if number == 1:
                data = self.data.solver[i].timer
            elif number == 2:
                data = self.data.solver[i].mvps
            elif number == 3:
                data = self.data.solver[i].iterations
            self._plot(data, i, plot)
        leg = plot.legend(self.solver_dict, loc=4)
        frame  = leg.get_frame()
        frame.set_alpha(0.8)
        plot.set_xlabel(u'  ', size='large')
        plot.set_ylabel(u'  (  )', size='large')
        return figure

    def _plot(self, data, index, plot):
        ''' specify what the plot looks like '''
        xlim = self.options.xlim if self.options.xlim > 1 else 10
        x = scipy.arange(1, xlim, xlim/1000.0)
        y = []
        for xi in x:
            y.append(1.0 / self.matrix_number * \
                len([item for item in data if item <= xi]))
        plot.plot(x, y, self.plotting_pattern[index], linewidth=2)
        plot.set_xlim([1, xlim])
        plot.set_ylim([0, 1.1])
        plot.hold(True)

    def write_time_to_file(self):
        if self.first:
            time_file = open('time.txt', 'w')
            self.first = False
        else:
            time_file = open('time.txt', 'a')
        time_file.write(self.matrix_name+' ('+str(round(sum(self.current[:,0])))+')
```

```
            \n')
        time_file.close()

    def _move_matrix(self, dest):
        ''' move a bad matrix '''
        for pref in ['', '_b', '_rhs1', '_x']:
            name = self.matrix_name + pref + '.mtx'
            try:
                os.rename('/home/sven/matrices/' + name,
                    '/home/sven/matrices-' + dest + '/' + name)
            except OSError, err:
                if pref == '':
                    print 'Moving matrix failed: {}, {}'.format(name, err)

    def _move_results(self, dest):
        ''' move bad test results '''
        if dest == 'small':
            pref = './test-small/'
        else:
            pref = '/home/sven/matrices-bad/'
        for tol in self.prec_tols:
            try:
                name = self.matrix_name + '_%05.2f.txt' % tol
                os.rename(self.prefix + '/' + name, pref + name)
            except OSError, (errno, err):
                if errno == 18:
                    try:
                        shutil.move(self.prefix + '/' + name, pref + name)
                    except IOError:
                        if dest == 'small' or tol == self.prec_tols[0]:
                            print 'Moving results failed: {}, {}'\
                                .format(name, err)
                        else:
                            pass
                elif dest == 'small' or tol == self.prec_tols[0]:
                    print 'Moving results failed: {}, {}'.format(name, err)
                else:
                    pass
        return

    def quit_on_error(self):
        ''' error '''
        print 'Aborting: an error occured'
        sys.exit(1)

    def set_options(self):
        ''' set all options '''
        option_parser = OptionParser()
        option_parser.add_option('-x', '--xlim', dest='xlim', default=10,
                type='float',
                help='the maximum ratio on the x axis of the plots')
        option_parser.add_option('-l', dest='l', default=3,
                type='int',
                help='the value of l to use in BiCGSTAB(l), 0 means both')
        option_parser.add_option('-e', '--exclude', dest='exclude',
                type='int', action='append',
                help='solver to exclude')
        option_parser.add_option('-r', '--restriction', dest='restriction',
                type='string', nargs=2,
                help='only use the matrices where solver has an error \
                      conform to the supplied test, i.e. 5 \'<-4\' \
                      means solver 5 has an error value smaller than -4')
        option_parser.add_option('-v', '--verbose', dest='verbose',
                action='store_true', default=False,
                help='print more output (failed methods)')
        option_parser.add_option('-n', '--filelim', dest='n', default=0,
                type='int',
                help='the amount of matrices to parse')
        (options, args) = option_parser.parse_args()
        # add bicgstab(l) to the exclude list
        if options.exclude is None:
            options.exclude = []
        if options.l == 2:
            options.exclude.append(options.l + 5)
        elif options.l == 3:
            options.exclude.append(options.l + 3)
        return options

class MatrixParser(object):
    def __init__(self, columns):
        self.runs = 6
        self.columns = columns
        self.clear()

    def clear(self):
        self.length = 0
        self.data = scipy.zeros([self.runs, self.columns])
        self.fail = False

    def add(self, line):
        for i, value in enumerate(line):
            self.data[self.length, i] = \
                float(value) if '.' in value else int(value)
        if int(line[3]) != 0:
            self.fail = True
        self.length += 1

    def read(self):
        return self.data[0:self.length, :]

class SolverData(object):
    ''' class that contains the data of all solvers using multiple
        Solver objects'''
    def __init__(self, amount):
        self.solver = []
        self.amount = amount
        for i in range(self.amount):
            self.solver.append(Solver())

class Solver(object):
    ''' class that contain the data of one solver '''
    def __init__(self):
        self.timer = []
        self.iterations = []
        self.mvps = []
        self.errors = 0
        self.iter_errors = 0
        self.nan_errors = 0

    def error(self, error_type):
```

```
        if error_type == 1:
            self.errors += 1
        elif error_type == 2:
            self.iter_errors += 1
        elif error_type == 3:
            self.nan_errors += 1

    def append(self, ratio, number):
        if number == 0:
            self.timer.append(ratio)
        elif number == 1:
            self.iterations.append(ratio)
        elif number == 2:
            self.mvps.append(ratio)

def main():
    ''' main method '''
    def write_to_file(text):
        '''write the results to a file'''
        f = open('rerun.txt', 'w')
        f.write(text)
        f.close()

    tolerances = [0.1, 1.0, 10.0]
    rerun = ''
    time = 0
    for i, tol in enumerate(tolerances):
        parser = Parser(tol, tolerances)
        parser.first = (i == 0)
        parser.run()
        rerun += parser.rerun
        time += parser.time
        parser = None
    print 'Total time: {}'.format(time)
    write_to_file(rerun)

if __name__ == "__main__":
    main()
```