The Adaptive Preconditioned Conjugate Gradient Method

Ping Chen

1 Introduction

Monteiro et al. proposed a new conjugate gradient type procedure for solving $Ax = b$ where A is a symmetric positive definite $n \times n$ matrix in [2]. Instead of having a fixed preconditioner, the preconditioner Z used in the new method can be updated by multiplying by a rank one matrix F so as to make the preconditioned matrix $Z^T AZ$ better conditioned if necessary. Therefore, the new method works better than the conjugate gradient method on extremely ill-conditioned matrices. The update matrix F not only reduces the determinant of $Z^T AZ$, but also keeps the new minimum eigenvalue larger than 1. These special properties of F make the iteration complexity bound reduced from $\mathcal{O}(\sqrt{\kappa(A)} \log \epsilon^{-1})$ to $\mathcal{O}(\sqrt{n} \log \epsilon^{-1})$ when obtaining an ϵ-solution to $Ax = b$, where $\kappa(A)$ is the condition number of A.

2 Related Work

2.1 The Conjugate Gradient (CG) Method

The CG method ([3], [1]) is a well-known iterative method for solving $Ax = b$ when A is symmetric and positive definite. Starting with an initial guess x_0 ($Ax_0 \approx b_0$) and an initial residue r_0 ($r_0 = b - Ax_0$), at the k-th iteration, the CG method finds a A-conjugate search direction p_k with $p_k^T r_0 \neq 0$, which is indeed equivalent to $p_k^T r_{k-1} \neq 0$. The initial direction p_1 is chosen as r_0. The current iterate x_k is updated using $x_{k-1} + \alpha_k p_k$ where $\alpha_k = r_{k-1}^T r_{k-1} / p_k^T A p_k$. A-conjugacy restricts $p_i^T A p_j = 0$ for all $i \neq j$ or $p_k \in span\{Ap_1, Ap_2, \ldots, Ap_{k-1}\}^\perp$. Therefore, the CG method establishes the following relationship among r_i and p_i: $span\{p_1, p_2, \ldots, p_k\} = span\{r_0, r_1, \ldots, r_{k-1}\} = span\{r_0, Ar_0, \ldots, A^{k-1} r_0\}$. In addition, the residuals r_i are mutually orthogonal. The $span\{r_0, Ar_0, \ldots, A_{k-1} r_0\}$ is called a Krylov subspace.

Unlike the steepest descent method which may suffer from slow convergence, the CG method is guaranteed to terminate within n steps in exact arithmetic. This is because the computed solution x_k minimizes $\phi_A(x) = \frac{1}{2} x^T A x - x^T b$ over the space $x_0 + span\{p_1, p_2, \ldots, p_k\}$ and all search directions are linearly independent. When A is symmetric positive definite, solving $Ax = b$ is equivalent to minimizing $\phi_A(x)$. The maximum number of iterations is equal to the number of distinct eigenvalues of A ([3]). Another advantage of the CG method is that it only requires one matrix-vector multiplication, three vector updates and two inner products per iteration ([4]) with proper implementation, which makes it an efficient technique.

2.2 The <u>P</u>reconditioned <u>C</u>onjugate <u>G</u>radient (PCG) Method

The CG method works well on well-conditioned matrices (which have relatively small condition numbers), but not on ill-conditioned matrices. The round-off error introduced in computations makes it hard to preserve the orthogonality of the residuals in practice. Preconditioning becomes necessary. The PCG method ([3]) aims to find a preconditioner M which reduces the condition number of the matrix A with acceptable extra computational costs, e.g solving a system $Mz = r$. The outline of the algorithm is given in Algorithm 1 ([1]).

Algorithm 1 The PCG Algorithm

```
1: $k = 0$
2: $r_0 = b - Ax_0$
3: while $r_k \neq 0$ do
4:   Solve $Mz_k = r_k$
5:   $k = k + 1$
6:   if $k = 1$ then
7:     $p_1 = r_0$
8:   else
9:     $\beta_k = r_{k-1}^T z_{k-1} / r_{k-2}^T z_{k-2}$
10:    $p_k = z_{k-1} + \beta_k p_{k-1}$
11:  end if
12:  $\alpha_k = r_{k-1}^T z_{k-1} / p_k^T A p_k$
13:  $x_k = x_{k-1} + \alpha_k p_k$
14:  $r_k = r_{k-1} - \alpha_k A p_k$
15: end while
16: $x = x_k$
```

The performance of the preconditioned method highly depends on the choice of preconditioners. The incomplete Cholesky preconditioner exploits the fact that there exists a lower triangular matrix L such that $A = LL^T$ if A is symmetric positive definite. It tends to find a preconditioner in the form $M = HH^T$ where H is close to L. The closer H to L the smaller the condition number of the new A. Other preconditioners include incomplete block preconditioner, those based on domain decomposition and polynomial preconditioners. Each has its own restriction. It is helpful to note that any iterative method based on the splitting $A = M - N$ (the Jacobi, Gauss-Seidel, SOR and SSOR methods) can be accelerated by the CG algorithm as long as M (the preconditioner) is symmetric and positive definite. All these methods are well explained in [1].

2.3 Other Variations

Instead of solving $Ax = b$ directly when A is unsymmetric, a normal equation approach (the Conjugate Gradient Normal Equation Residual method) is to solve the equivalent system $A^TAx = A^Tb$. Another approach (the Conjugate Gradient Normal Equation Error method) is to solve $AA^Ty = b$ first and compute $x = A^Ty$ as the final solution. The obvious drawback of these two approaches is squaring the condition number of A ([1]).

If A is symmetric positive definite, an alternative way (the Conjugate Residual method) to solve $Ax = b$ is to solve $A^{\frac{1}{2}}x = A^{-\frac{1}{2}}b$ since $A^{\frac{1}{2}}$ is also symmetric positive definite.

The Generalized Minimal Residual (GMRES) method ([5]) is a generalized CG method for solving systems with unsymmetric A. The flexible GMRES (FGMRES) method ([6]) allows variable preconditioners.

3 The Adaptive PCG (APCG) Method

Assume that we have normalized A such that its smallest eigenvalue is larger than or equal to 1, i.e., $A \succeq I$. The PCG method with preconditioner M transforms the original problem $Ax = b$ into an equivalent problem $Z^TAZx = Zb$ with $ZZ^T = M^{-1}$ with hopes of improving the condition number of the new matrix $\hat{A} = Z^TAZ$. If $\hat{A}$ is not well-conditioned, poor performance seems to be inevitable. Instead of finding another preconditioner from scratch, the APCG method attempts to make Z a good preconditioner for every iteration. A preconditioner Z is considered good if it is a ν-preconditioner at x_k satisfying $g(x_k)^TZ(Z^TAZ)Z^Tg(x_k) \leq \nu||Z^Tg(x_k)||^2$ given $\nu > n$ and $g(x_k) = Ax_k - b$. The update step, which will be discussed shortly, only occurs when Z fails to be a ν-preconditioner. The number of updates is bound by $\mathcal{O}(N_\psi)$ with $N_\psi = \frac{\log det(A)}{\psi^{-1}-1+\log\psi}$ and $\psi = \nu/n$ thanks of the above mentioned special properties of the update matrix F. The number N_ψ decreases from infinity to $\mathcal{O}(n)$ as ψ increase from 1 to $\lambda_{max}(A)/n$ where $\lambda_{max}(A)$ is the largest eigenvalue of A. In other words, the choices of ν are limited in the range $(n, \lambda_{max}(A))$. When $\nu \geq \lambda_{max}(A)$, the APCG method reduces to the PCG method with no need of updating. If Z is a ν-preconditioner and $Z^TAZ \succeq \xi I$ for some constant $\xi \leq 1$ is preserved at every iteration, significant reductions in the number of CG iterations are expected. It has been proven that an ϵ-solution x_k to $Ax = b$ with $\phi_A(x_k) \leq \epsilon\phi_A(x_0)$ can be obtained in $\mathcal{O}(N_\psi + \sqrt{n}\log\epsilon^{-1})$ steps.

The condition that Z be a ν-preconditioner can be achieved by multiplying Z by a rank one update matrix F whenever Z fails. The matrix F is called an Ellipsoid preconditioner, which is in the form of $\mu I + (\theta - \mu)pp^T$, where μ and θ are constants and p is a unit vector. The vector p is normal to the boundary of $E(\hat{A}) = \{z | z^T\hat{A}z \leq 1\}$ at the two points y and $-y$ with $y = w/\sqrt{w^T\hat{A}w}$, $w = \xi^{-1/2}Z^Tg(x_k)$ and x_k is the current iterate. However, the resultant preconditioner is a ν-preconditioner at $x_0, x_1, \ldots, x_{k-2}$ only and it generates the same iterate x_{k-1} as Z does. As a result, a backtrack step is required whenever Z is updated. A restart step happens when ξ gets too small. The detailed algorithm is shown as Algorithm 2.

Algorithm 2 APCG Algorithm

```
Require: $A \succeq I$, $b \in R^n$, $x_0 \in R^n$ and $\nu > n$, $\delta \in (0,1)$ and $\epsilon > 0$.
 1: Set $\phi_0 = \phi_A(x_0) = \frac{1}{2}(x_0 - x^*)^T A(x_0 - x^*)$ and $Z = I$.
 2: $k = 0$, $\xi = 1$, $g_0 = Ax_0 - b$, $d_{-1} = 0$, $\beta_0 = 0$, and $\gamma_0 = ||Z^T g_0||^2$.
 3: while $\phi_A(x_k) > \epsilon\phi_0$ do
 4:   while $g_k^T Z(Z^T AZ)Z^T g_k > \nu\gamma_k$ do
 5:     {Update Z}
 6:     $w = \xi^{-\frac{1}{2}} Z^T g_k$
 7:     $\hat{A} = \xi^{-1} Z^T AZ$
 8:     $p = \frac{\hat{A}w}{||\hat{A}w||}$
 9:     $\tau = \frac{\sqrt{w^T \hat{A}w}}{||\hat{A}w||}$
10:     $\theta = min(\tau\sqrt{n}, 1)$
11:     $\mu = \sqrt{\frac{n-\theta^2}{n-1}}$
12:     $F = \mu I + (\theta - \mu)pp^T$
13:     $Z = ZF/\mu$
14:     $\xi = \xi\mu^{-2}$
15:     if $\xi \leq \delta$ then
16:       go to Line 2 with $Z = \xi^{-\frac{1}{2}} Z$ and $x_0 = x_k$ {Restart Step}
17:     end if
18:     $k = max(k-1, 0)$ {Backtrack Step}
19:   end while
20:   if $k > 0$ then
21:     $\beta_k = \gamma_k/\gamma_{k-1}$
22:   end if
23:   $d_k = -ZZ^T g_k + \beta_k d_{k-1}$
24:   $\alpha_k = \gamma_k/(d_k^T A d_k)$
25:   $x_{k+1} = x_k + \alpha_k d_k$
26:   $g_{k+1} = g_k + \alpha_k A d_k$
27:   $\gamma_{k+1} = ||Z^T g_{k+1}||^2$
28:   $k = k + 1$
29: end while
```

4 Experimental Results

4.1 Implementation

The APCG method is implemented using Matlab (version 7.0.0.19920), a high-level technical computing language. Here, we discuss some practical issues.

4.1.1 APCG With Limited-memory

Appendix A shows the direct implementation of APCG method with some small modification on stopping criteria, which we discuss later. Due to the possibility of continuous backtracking, the entire history of computer iterates has to be stored. This could be extremely inefficient in terms of memory usage and computation costs involved in backtracking especially when n is large. More practically, we consider remembering the most recent m iterates only.

The updating procedure can be very expensive as well. Note that $Z_k = Z_0 \frac{F_0}{\mu_0} \frac{F_1}{\mu_1} \dots \frac{F_{k-1}}{\mu_{k-1}} = Z_0 F_0' F_1' \dots F_{k-1}'$ and F_j' (or F_j) is dependent on Z_j. Besides, F_j' is in the form of $I + (\frac{\theta_j}{\mu_j} - 1) p_j p_j^T$, where μ_j and θ_j are constants and p_j is a unit vector. Moreover, μ_j is determined by θ_j. Hence, F_j is determined by θ_j and p_j. Another observation is that the preconditioner Z is not used directly at the CG iterations. This suggests a way to avoid matrix-matrix multiplications by using matrix-vector multiplications instead. We only need to store the initial preconditioner Z_0, a set of θs and a set of ps. Detailed implementation can be found in Appendix B.

4.1.2 Stopping Criteria

In practice, we do not know the optimal solution x_* before hand, which makes it impossible to access $\phi_A(x_k)$ nor ϕ_0. Instead, we use the relative error in the residuals as the stopping condition. Therefore, the iteration stops whenever any of the following conditions are met at the k-th iteration.

1. $||r_k|| \leq \epsilon * ||b||$.
2. the algorithm backtracks more than m times if limited-memory used
3. the number of iterations exceeds the maximum number of iterations allowed *maxit*.

4.1.3 System Information

All the experiments are run on a machine with Intel®Pentium®4 2.8GHz CPU, 512 MB of RAM, 80G hard drive.

4.2 Parameter Descriptions

Table 1 lists different parameters used in the APCG method and its limited-memory variation.

4.3 Test Problems

4.3.1 $A_1 = tridiag(-1, 2, -1)$ With Scaling

The first type of test problems considered are matrices A_0 in the form of tridiag(-1,2,-1). The eigenvalues of A_0 are $2 - 2\cos\frac{j\pi}{n+1}$ and $j \in \{1, 2, \dots, n\}$. The minimum eigenvalue, $\lambda_{min}(A_0)$, is $2-2\cos\frac{\pi}{n+1}$; the maximum eigenvalue, $\lambda_{max}(A_0)$,

Table 1: Used Parameters

Name	description
ϵ	relative residual error tolerance
ν	preconditioner goodness factor
δ	restart control parameter
m	length of most recent history

is $2 - 2\cos\frac{n\pi}{n+1}$. It is clear that $\lambda_{min}(A_0)$ approaches 0, $\lambda_{max}(A_0)$ approaches 4, $\kappa(A_0)$ approaches ∞ as n increases. However, A_0 does not satisfy $A_0 \succeq I$. In other words, $\lambda_{min}(A_0) \not\geq 1$. This condition can be achieved by multiplying A_0 by $\lambda_{min}(A_0)^{-1} + 1$. In this way, we get a new set of problem A_1 with $\kappa(A_1) = \kappa(A_0)$. The right hand side vector b is chosen to be the product of A_1 with the all ones vector. Equivalently, $x^* = ones(n,1)$.

We set $\epsilon = 1e{-}6$; $\nu \in \{2n,\ \lfloor(2n{+}\lambda_{max}(A_1))/4\rfloor,\ \lfloor(2n{+}\lambda_{max}(A_1))/2\rfloor,\ \lfloor\lambda_{max}(A_1)\rfloor\}$; $\delta \in \{0.1,\ 0.3,\ 0.5,\ 0.7,\ 0.9\}$; $x_0 = zeros(n,1)$; $maxit = 2n$. The Experimental Results obtained by the direct APCG method on various sized matrices with different δ values and $\nu = 2n$ are summarized in Table 2. The results with a different $\nu = \lfloor(2n + \lambda_{max}(A_1))/4\rfloor$ are reported in Table 3. When we set $\nu = \lfloor(2n + \lambda_{max}(A_1))/2\rfloor$ or $\nu = \lfloor\lambda_{max}(A_1)\rfloor$, the APCG method behaves just like the CG method except slight differences on the running times. On both cases, it takes the APCG method $n/2$ CG iterations and 0 updates to converge to the optimal solution. Table 4 reports the performance by the CG method (using the pre-defined function pcg.m in Matlab). If the diagonal preconditioner with M^{-1} = the diagonal of A is used, we get the results as in Table 5.

The limited version of the APCG method without the need of explicit storage for Z may speed up the updating process when n is small. While n is large, it may often fail as it tends to backtrack more than m steps. This method is merely another efficient way of implementing the original APCG method. In addition, it lacks of theoretical convergence complexity bound when only keeping a short history. We will not discuss this variation of the APCG method in details.

4.3.2 $A_2 = A_0 - \mu I$ With Scaling And μ Is A Positive Small Shift with $0 < \mu < \lambda_{min}(A_0)$

The second type are matrices $A_2 = A_0 - \mu I$ with scaling. These matrices are harder to solve than A_1 in the sense of having even bigger condition number. The condition number of $A_2 = \kappa(A_2) = \frac{\lambda_{max}(A_0)-\mu}{\lambda_{min}(A_0)-\mu} > \frac{\lambda_{max}(A_0)}{\lambda_{min}(A_0)} (i.e., \kappa(A_1))$.

Using the similar parameter settings for matrices A_1 and $\mu = 0.9$, we obtained the results when ν is chosen as $2n$, $\lfloor(2n + \lambda_{max}(A_2))/4\rfloor$ and $\lfloor(2n + \lambda_{max}(A_2))/2\rfloor$, as shown in Table 6, 7 and 8 respectively. When letting $\nu = \lfloor\lambda_{max}(A_2)\rfloor$, the APCG method again works as the CG method, which is presented in Table 9. The performance by the PCG method with the diagonal preconditioner can be found in Table 10.

Table 2: Experimental Results of APCG Method on A_1 with various δ value and fixed $\nu = 2n$

	Parameters				Results					
n	ϵ	ν	δ	m	converge?	update	CG iter	total CG iter	time	residual [1]
10	1e-6	$2n$	0.1	-	no	5	20	20	3.125e-2	1.520e5
50	1e-6	$2n$	0.1	-	no	41	100	100	4.688e-2	5.271e8
100	1e-6	$2n$	0.1	-	no	89	200	200	5.781e-1	9.793e13
500	1e-6	$2n$	0.1	-	no	466	1000	1000	1.739e2	6.044e26
1000	1e-6	$2n$	0.1	-	no	946	2000	2000	2.416e3	2.802e30
10	1e-6	$2n$	0.3	-	no	5	20	20	0.000e0	1.520e5
50	1e-6	$2n$	0.3	-	no	41	100	100	4.688e-2	5.271e8
100	1e-6	$2n$	0.3	-	no	89	200	200	4.688e-1	9.793e13
500	1e-6	$2n$	0.3	-	no	466	1000	1000	1.737e2	6.044e26
1000	1e-6	$2n$	0.3	-	no	946	2000	2000	2.415e3	2.802e30
10	1e-6	$2n$	0.5	-	no	5	20	20	1.562e-2	1.520e5
50	1e-6	$2n$	0.5	-	no	44	54	100	4.688e-2	3.086e-2
100	1e-6	$2n$	0.5	-	no	95	130	200	4.688e-1	3.372e17
500	1e-6	$2n$	0.5	-	no	477	654	1000	1.744e2	3.416e34
1000	1e-6	$2n$	0.5	-	no	966	1308	2000	2.419e3	4.287e30
10	1e-6	$2n$	0.7	-	no	5	2	20	0.000e0	4.285e3
50	1e-6	$2n$	0.7	-	yes	46	55	100	4.688e-2	1e-6
100	1e-6	$2n$	0.7	-	no	96	128	200	3.906e-1	3.783e14
500	1e-6	$2n$	0.7	-	no	477	649	1000	1.743e2	3.302e45
1000	1e-6	$2n$	0.7	-	no	968	1296	2000	2.419e3	2.057e75
10	1e-6	$2n$	0.9	-	yes	4	1	14	1.562e-2	0.000e0
50	1e-6	$2n$	0.9	-	yes	47	35	75	3.125e-2	1e-6
100	1e-6	$2n$	0.9	-	yes [2]	98	122	200	5.313e-1	3e-5
500	1e-6	$2n$	0.9	-	no	480	612	1000	1.764e2	4.310e103
1000	1e-6	$2n$	0.9	-	no	977	1090	2000	2.424e3	6.094e89

[1]: It is the relative residual $||A * x^* - b||/||b||$ where x^* is computed solution

[2]: Although the algorithm returns $flag = 1$ since $residual \geq \epsilon$, we consider the APCG method converges as the returned solution is very close to the true solution.

Table 3: Experimental Results of APCG Method on A_1 with various δ value and fixed $\nu = \lfloor (2n + \lambda_{max}(A_1))/4 \rfloor$

	Parameters				Results					
n	ϵ	ν	δ	m	converge?	update	CG iter	total CG iter	time	residual [1]
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.1	-	no	5	20	20	0.000e0	5.548e2
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.1	-	no	32	100	100	4.688e-2	8.898e5
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.1	-	no	65	200	200	3.906e-1	9.353e6
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.1	-	no	253	1000	1000	1.651e2	2.814e8
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.1	-	no	432	2000	2000	2.339e3	3.798e11
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.3	-	no	5	20	20	1.563e-2	5.548e2
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.3	-	no	32	100	100	4.688e-2	8.898e5
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.3	-	no	65	200	200	3.594e-1	9.353e6
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.3	-	no	253	1000	1000	1.651e2	2.814e8
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.3	-	no	432	2000	2000	2.340e3	3.798e11
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.5	-	no	5	20	20	1.563e-2	5.548e2
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.5	-	no	32	100	100	4.688e-2	8.898e5
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.5	-	no	65	200	200	3.750e-1	9.353e6
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.5	-	no	253	1000	1000	1.651e2	2.814e8
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.5	-	no	432	2000	2000	2.339e3	3.798e11
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.7	-	no	5	1	20	1.563e-2	2.487e2
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.7	-	no	37	71	100	4.688e-2	8.534e6
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.7	-	no	73	153	200	4.063e-1	1.382e11
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.7	-	no	311	818	1000	1.685e2	5.315e15
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.7	-	no	773	1280	2000	2.391e3	9.361e16
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.9	-	yes	4	1	14	0.000e0	0.000e0
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.9	-	yes	38	24	72	3.125e-2	1e-6
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.9	-	yes [2]	81	104	200	4.688e-1	1.7e-5
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.9	-	no	392	619	1000	1.710e2	9.206e22
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_1)}{4}\rfloor$	0.9	-	no	771	1253	2000	2.391e3	7.949e29

[1]: It is the relative residual $||A * x^* - b||/||b||$ where x^* is computed solution

[2]: Although the algorithm returns $flag = 1$ since $residual \geq \epsilon$, we consider the APCG method converges as the returned solution is very close to the true solution.

Table 4: Experimental Results of CG Method on A_1

n	converge?	CG iter	time	relative residual
10	yes	5	0.000e0	0.000e0
50	yes	25	0.000e0	0.000e0
100	yes	50	1.563e-2	0.000e0
500	yes	250	1.719e-1	0.000e0
1000	yes	500	3.906e-1	0.000e0

Table 5: Experimental Results of PCG Method with diagonal preconditioner on A_1

n	converge?	CG iter	time	relative residual
10	yes	5	0.000e0	0.000e0
50	yes	25	0.000e0	0.000e0
100	yes	50	0.000e0	0.000e0
500	yes	250	7.812e-2	0.000e0
1000	yes	500	2.813e-1	0.000e0

4.4 Observations On The Experimental Results

On the first type of problems A_1, both the CG method and the PCG method with the diagonal preconditioner converge to the optimal solution with only $n/2$ steps. In practice with the presence of the roundoff errors, the CG based methods sometimes take less than n steps to converge if they do converge. In this case, if we start with a different initial guess, for example, a randomized vector, it usually takes n steps to converge for both methods. For the second type of problems A_2, both methods take $n/2$ steps to converge when n is small (10,50,100) and diverge when n = 500 and n = 1000. The outcome is expected as matrices A_2 is worse conditioned compared with A_1.

On the other hand, the APCG method does not perform as well as the other two methods. First of all, the APCG method seldom converges to the optimal solution even with different ν and δ values. Moreover, if it does converge, it takes more memory storage and execution time than the above mentioned methods. Each iteration in the CG method requires one matrix-vector multiplication, three vector updates and two inner products. The PCG method with the diagonal preconditioner requires an extra matrix-vector multiplication: $z = M^{-1}r$. Each CG iteration in the APCG method, as described in Algorithm 2, requires at least three matrix-vector multiplications, three vector updates and two inner products. The concept of adaptively updating the preconditioner demands extra $\mathcal{O}(n^2)$ storage for the preconditioner. This storage requirement is replaced by computational costs if we do not store this preconditioner explicitly. Each update iteration needs at least four matrix-vector multiplications, one outer

Table 6: Experimental Results of APCG Method on A_2 with various δ value and fixed $\nu = 2n$

	Parameters				Results					
n	ϵ	ν	δ	m	converge?	update	CG iter	total CG iter	time	residual [1]
10	1e-6	$2n$	0.1	-	no	8	20	20	0.000e0	2.454e7
50	1e-6	$2n$	0.1	-	no	49	100	100	4.688e-2	1.936e12
100	1e-6	$2n$	0.1	-	no	97	200	200	5.625e-1	2.201e15
500	1e-6	$2n$	0.1	-	no	492	1000	1000	1.747e2	1.074e54
1000	1e-6	$2n$	0.1	-	no	986	2000	2000	2.420e3	5.989e65
10	1e-6	$2n$	0.3	-	no	8	20	20	1.563e-2	2.454e7
50	1e-6	$2n$	0.3	-	no	49	100	100	4.688e-2	1.936e12
100	1e-6	$2n$	0.3	-	no	97	200	200	4.688e-1	2.201e15
500	1e-6	$2n$	0.3	-	no	492	1000	1000	1.750e2	1.074e54
1000	1e-6	$2n$	0.3	-	no	986	2000	2000	2.422e3	5.898e65
10	1e-6	$2n$	0.5	-	no	11	9	20	1.562e-2	2.958e2
50	1e-6	$2n$	0.5	-	no	50	68	100	4.688e-2	3.039e12
100	1e-6	$2n$	0.5	-	no	99	134	200	4.063e-1	1.154e34
500	1e-6	$2n$	0.5	-	no	494	659	1000	1.750e2	6.232e58
1000	1e-6	$2n$	0.5	-	no	992	1313	2000	2.421e3	2.059e148
10	1e-6	$2n$	0.7	-	yes	6	7	9	0.000e0	1.000e-6
50	1e-6	$2n$	0.7	-	no	50	70	100	4.688e-2	2.743e22
100	1e-6	$2n$	0.7	-	no	98	136	200	4.063e-1	1.245e43
500	1e-6	$2n$	0.7	-	no	496	660	1000	1.751e2	5.844e127
1000	1e-6	$2n$	0.7	-	no	993	1311	2000	2.429e3	2.993e148
10	1e-6	$2n$	0.9	-	yes	5	6	6	3.125e-2	0.000e0
50	1e-6	$2n$	0.9	-	no	53	69	100	9.375e-2	2.751e19
100	1e-6	$2n$	0.9	-	yes	102	25	94	3.125e-1	1.000e-6
500	1e-6	$2n$	0.9	-	no	497	610	1000	1.842e2	3.024e95
1000	1e-6	$2n$	0.9	-	no	996	1178	2000	2.523e3	8.273e147

[1]: It is the relative residual $||A * x^* - b||/||b||$ where x^* is computed solution

Table 7: Experimental Results of APCG Method on A_2 with various δ value and fixed $\nu = \lfloor (2n + \lambda_{max}(A_2))/4 \rfloor$

	Parameters				Results					
n	ϵ	ν	δ	m	converge?	update	CG iter	total CG iter	time	residual [1]
10	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.1	-	no	6	20	20	0.000e0	7.392e4
50	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.1	-	no	33	100	100	4.688e-2	1.789e6
100	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.1	-	no	64	200	200	4.531e-1	7.403e6
500	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.1	-	no	271	1000	1000	1.656e2	1.641e8
1000	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.1	-	no	444	2000	2000	2.340e3	2.594e9
10	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.3	-	no	6	20	20	1.563e-2	7.392e4
50	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.3	-	no	33	100	100	4.688e-2	1.789e6
100	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.3	-	no	64	200	200	4.063e-1	7.403e6
500	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.3	-	no	271	1000	1000	1.658e2	1.641e8
1000	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.3	-	no	444	2000	2000	2.339e3	2.594e9
10	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.5	-	no	6	20	20	0.000e0	7.392e4
50	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.5	-	no	33	100	100	4.688e-2	1.789e6
100	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.5	-	no	64	200	200	3.906e-1	7.403e6
500	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.5	-	no	271	1000	1000	1.659e2	1.641e8
1000	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.5	-	no	444	2000	2000	2.339e3	2.594e9
10	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.7	-	no	7	7	20	1.563e-2	3.780e2
50	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.7	-	yes [2]	38	50	100	4.688e-2	2.000e-6
100	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.7	-	no	80	112	200	3.906e-1	1.136e3
500	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.7	-	no	339	820	1000	1.686e2	6.541e9
1000	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.7	-	no	770	1283	2000	2.390e3	2.175e14
10	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.9	-	no	7	3	20	0.000e0	4.876e0
50	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.9	-	yes [2]	39	59	100	1.250e-1	4.300e-4
100	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.9	-	no	81	94	200	4.375e-1	5.925e-2
500	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.9	-	no	394	614	1000	1.725e2	2.338e20
1000	1e-6	$\lfloor \frac{2n+\lambda_{max}(A_2)}{4} \rfloor$	0.9	-	no	781	1253	2000	2.388e3	7.184e20

[1]: It is the relative residual $||A * x^* - b||/||b||$ where x^* is computed solution

[2]: Although the algorithm returns $flag = 1$ since $residual \geq \epsilon$, we consider the APCG method converges as the returned solution is very close to the true solution.

Table 8: Experimental Results of APCG Method on A_2 with various δ value and fixed $\nu = \lfloor(2n + \lambda_{max}(A_2))/2\rfloor$

	Parameters				Results					
n	ϵ	ν	δ	m	converge?	update	CG iter	total CG iter	time	residual [1]
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.1	-	no	6	20	20	0.000e0	7.392e4
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.1	-	no	9	100	100	3.125e-2	1.368e0
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.1	-	no	10	200	200	2.969e-1	1.831e-1
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.1	-	no	10	1000	1000	1.197e2	7.018e-3
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.1	-	no	8	2000	2000	1.738e3	4.450e-3
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.3	-	no	6	20	20	3.125e-2	7.392e4
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.3	-	no	9	100	100	4.688e-2	1.368e0
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.3	-	no	10	200	200	2.969e-1	1.831e-1
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.3	-	no	10	1000	1000	1.199e2	7.018e-3
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.3	-	no	8	2000	2000	1.739e3	4.450e-3
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.5	-	no	6	20	20	0.000e0	7.392e4
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.5	-	no	9	100	100	6.250e-2	1.368e0
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.5	-	no	10	200	200	2.969e-1	1.831e-1
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.5	-	no	10	1000	1000	1.199e2	7.018e-3
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.5	-	no	8	2000	2000	1.750e3	4.450e-3
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.7	-	no	6	7	20	0.000e0	6.618e2
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.7	-	no	9	100	100	1.094e-1	1.368e0
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.7	-	no	10	200	200	3.125e-1	1.831e-1
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.7	-	no	10	1000	1000	1.200e2	7.018e-3
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.7	-	no	8	2000	2000	1.739e3	4.450e-3
10	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.9	-	yes	2	5	6	0.000e0	0.000e0
50	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.9	-	no	27	13	100	6.250e-2	2.953e-2
100	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.9	-	no	10	200	200	3.281e-1	1.831e-1
500	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.9	-	no	10	1000	1000	1.200e2	7.018e-3
1000	1e-6	$\lfloor\frac{2n+\lambda_{max}(A_2)}{2}\rfloor$	0.9	-	no	8	2000	2000	1.740e3	4.450e-3

[1]: It is the relative residual $||A * x^* - b||/||b||$ where x^* is computed solution

[2]: Although the algorithm returns $flag = 1$ since $residual \geq \epsilon$, we consider the APCG method converges as the returned solution is very close to the true solution.

Table 9: Experimental Results of CG Method on A_2

n	converge?	CG iterations	time	residual
10	yes	5	3.125e-2	0.000e0
50	yes	25	1.563e-2	0.000e0
100	yes	50	0.000e0	0.000e0
500	no	-	-	-
1000	no	-	-	-

Table 10: Experimental Results of PCG Method with diagonal preconditioner on A_2

n	converge?	CG iterations	time	residual
10	yes	5	1.563e-2	0.000e0
50	yes	25	0.000e0	0.000e0
100	yes	50	0.000e0	0.000e0
500	no	-	-	-
1000	no	-	-	-

product and two inner products. The updating procedure makes the APCG method backtrack or restart if necessary. This in turn takes more CG iterations than necessary, as suggested by the difference between *CG iter* and *total CG iter*.

The APCG method prefers large δ values. The performances with lower δ values (0.1, 0.3, 0.5) do not vary a lot on both types of problems. We see a slightly better performance when δ value is increased to 0.9. In other words, it benefits from frequent restarts. It is not necessary for ν to be as large as $\lambda_{max}(A)$ so that the APCG method acts like the CG method. For instance, $\nu = \lfloor (2n + \lambda_{max}(A_1))/2 \rfloor$ is sufficient for the APCG method to work like the CG method for matrices A_1.

5 Conclusion and Future Work

Though the idea of adaptively updating the preconditioner sounds encouraging, we find out that it does not out perform other CG based procedures. In the future, we may consider deriving a different update matrix F, which does not require any backtracking. In this way, we may avoid intensive storage requirement and reduce computational costs involved on backtracking. Another possibility is to replace Z with a different preconditioner without violating the constraint $Z^T AZ \succeq I$ whenever the method restarts, which gives us a flexible APCG method.

References

[1] Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, The John Hopkins University Press, 3rd ed., 1996.

[2] Renato D.C. Monteiro, Jerome W. O'Neal and Arkadi Nuemirovski, *A New Conjugate Gradient Algorithm Incorporating Adaptive Ellipsoid Preconditioning*, Technical Report, Georgia Institute of Technology, 2004.

[3] Stephen G. Nash and Ariela Safer, *Linear and Nonlinear Programming*, McGraw-Hill, 1996.

[4] Wolfram Research, *MathWorld*, http://mathworld.wolfram.com/ConjugateGradientMethod.html.

[5] Youcef Saad and Martin H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm For Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comput., Vol. 7, No. 3, July 1986, 856-869.

[6] Youcef Saad, *A Flexible Inner-Outer Preconditioned Algorithm*, SIAM J. Sci. Stat. Comput., Vol. 14, No. 2, March 1993, 461-469.

A Direct Implementation

```
function [xstar,flag,relres,update,cg_iter,resvec,total_iter] = ...
    APCG(A,b,nu,delta,x0,tol,maxit,Z);
% APCG    Adaptive Preconditioned Conjugate Gradient Method
%
% Reference:
% ----------
% Renato D.C. Monteiro, Jerome W. O'Neal and Arkadi Nuemirovski,
% A New Conjugate Gradient Algorithm Incoporating Adaptive Ellipoid
% Preconditioning
% Technical Report, Georgia Institute of Technology, 2004.
%
% This function attempts to solve the system of linear equations A*x=b
% Input and Parameter:
% ---------------------
% A   - The n-by-n coefficient matrix A must be symmetric and positive
%       definite. Also, A's minimum eigenvalue has to be larger than or
%       equal to 1.
% b   - The right hand side column vector b must have size n-by-1.
% nu - Specifies the constant used to test if Z is a nu-preconditioner at
%      certain iterate. If it is not, updating Z is required. It must be
%      between n and the maximum eigenvalue of A. See the paper.
% delta - Specifies the constant used to control restart. 0 < delta < 1.
% x0  - Specifies the initial guess. If it is [], default value (all zero
%       vector) will be used.
% tol - Specifies the tolerance of the method. If it is [], default value
%       1e-6 is used.
% maxit - specifies the maximum number of iterations. If it is [], default
%         value 2*n is used.
% Z - Specifies the n-by-n initial preconditioner with default value
%     speye(n) if it is [].
%
% Output:
% ---------
% xstar - computed solution to A*x = b
% flag  - a) if 0, APCG converged to the desired tolerance TOL within
%             MAXIT iterations.
%         b) if 1, APCG iterated MAXIT times but did not converge, and the
%            latest computed iterate returned as xstar
%         c) if 2, preconditioner Z was ill-conditioned, as it fails to
%            satisfy the minimum eigenvalue of Z'AZ is larger than or equal
%            to 1. The latest computed iterate is returned as xstar.
% relres - returns the relative residual norm(b-A*x)/norm(b). If flag is
%          0, then, relres <= tol.
% update - returns the number of iterations of updating the preconditioner
% cg_iter - returns the number of CG iterations within one restart round,
%           at which the returned xstar computed.
% resvec - returns a vector of the residual norms at each iteration
%          including NORM(b-A*X0).
% total_iter - returns the total number of CG iterations within multiple
%              restarts
%
% Note: This comment document follows the format of pcg method by MatLab.
%       The error checking and setting default value code are adapted from
%       pcg method.

format long;

if (nargin < 4)
   error('MATLAB:APCG:NotEnoughInputs', 'Not enough input arguments.');
end

% Check matrix and right hand side vector inputs have appropriate sizes
[m,n] = size(A);
if (m ~= n)
    error('MATLAB:APCG:NonSquareMatrix', 'Matrix must be square.');
end
if ~isequal(size(b),[m,1])
    es = sprintf(['Right hand side must be a column vector of' ...
            ' length %d to match the coefficient matrix.'],m);
```

```
        error('MATLAB:APCG:RHSsizeNotMatchCoeffMatrix', es)
end

m = size(b,1);
n = m;
if (size(b,2) ~= 1)
    error('MATLAB:APCG:RHSnotColumnVec',...
        'Right hand side must be a column vector.');
end

% Assign default values to unspecified parameters
if (nargin < 6) || isempty(tol)
   tol = 1e-6;
end
if (nargin < 7) || isempty(maxit)
    maxit = 2*n;
end
if (nargin < 8) || isempty(Z)
    Z = speye(n);
end

% Check for all zero right hand side vector => all zero solution
n2b = norm(b);                      % Norm of rhs vector, b
if (n2b == 0)                       % if    rhs vector is all zeros
   xstar = zeros(n,1);              % then  solution is all zeros
   flag = 0;                        % a valid solution has been obtained
   relres = 0;                      % the relative residual is actually 0/0
   iter = 0;                        % no iterations need be performed
   resvec = 0;                      % resvec(1) = norm(b-A*x) = norm(0)
   return;
end

if ((nargin >= 5) && ~isempty(x0))
   if ~isequal(size(x0),[n,1])
      es = sprintf(['Initial guess must be a column vector of' ...
            ' length %d to match the problem size.'],n);
      error('MATLAB:APCG:WrongInitialGuessSize', es);
   end
else
   x0 = zeros(n,1);
end

if ((nargin >= 8) && ~isempty(Z))
    if ~isequal(size(Z),[m,m])
        es = sprintf(['Preconditioner must be a square matrix' ...
            ' of size %d to match the problem size.'],m);
        error('MATLAB:APCG:WrongPreconditionerSize', es);
    else
        vec = eig(Z'*A*Z);
        if vec(n) < 1
            flag = 2;
            es = sprintf(['Preconditioner has to make the minimum' ...
                ' eigenvalue of the preconditioned A larger than' ...
                ' or equal to 1']);
            error('MATLAB:APCG:IllConditionedPreconditioner',es);
            return;
        end
    end
end

% Set up for the method
total_iter = 0;           %total iteration including restart
cg_iter = 0;              %CG iteration counter

k=1;                     % start index of iterates
xi = 1;                  % constant used in updating iterations
d_previous = zeros(n,1); % initial search direction
g{1} = A*x0 - b;         % -r0, following the paper's notation
beta = zeros(maxit+1,1);
beta(1) = 0;             % initial beta value used in CG iterations
gamma = zeros(maxit+1,1);
gamma(1) = norm(Z'*g{1},2)^2;  %norm of preconditioned g0
X{1} = x0;               % initial guess as the first iterate
update = 0;             % update iteration counter
r_flag = 0;             % flag indicating a restart
r0 = -g{1};             % initial residual
flag = 1;
normr = norm(r0);
tolb = tol * n2b;

if (normr <= tolb)                 % Initial guess is a good enough solution
   xstar = x0;
   flag = 0;
   relres = normr/n2b;
   total_iter = 0;
   cg_iter = 0;
   resvec = normr;
   return;
end

resvec = zeros(maxit+1,1);         % Preallocate vector for norm of residuals
resvec(1) = normr;                 % resvec(1) = norm(b-A*x0)
```

```
% loop over maxit iterations (unless convergence or failure)
while resvec(k) > tolb && total_iter < maxit

    while g{k}'*(Z*(Z'*(A*(Z*(Z'*g{k}))))) > nu*gamma(k)

        w = xi^(-1/2)*(Z'*g{k});
        A_hat = xi^(-1)*Z'*A*Z;
        p = A_hat*w;
        tau = sqrt(w'*p)/norm(p,2);
        p = p/norm(p,2);
        theta = min(tau*sqrt(n),1);
        mu = sqrt((n - theta^2)/(n-1));
        Z = Z + (theta/mu -1)*Z*p*p';
        xi = xi*(mu^(-2));

        if xi > delta
            if k==1
                ;
            else
                k = max(k-1,1);
            end
        else
            Z = xi^(-1/2)*Z;
            X{1} = X{k};

            cg_iter = 0;
            k=1;
            xi = 1;
            d_previous = zeros(n,1);
            g{1}=A*X{1} - b;
            beta(1) = 0;
            gamma(1) = norm(Z'*g{1})^2;
            resvec(1) = norm(g{1});

            r_flag = 1;
        end

        update = update +1;

        if r_flag == 1
            break;
        else
            ;
        end
    end
    if r_flag ==1
        r_flag = 0;
        continue;
    end

    if k == 1
        d{k} = -Z*Z'*g{1} + beta(1)*d_previous;
    else
        beta(k) = gamma(k)/gamma(k-1);
        d{k} = -Z*Z'*g{k} + beta(k)*d{k-1};
    end

    alpha = gamma(k)/((d{k}'*A)*d{k});

    X{k+1} = X{k} + alpha*d{k};
    g{k+1} = g{k} + alpha*(A*d{k});
    gamma(k+1) = norm(Z'*g{k+1})^2;
    resvec(k+1) = norm(g{k+1});

    k = k+1;

    cg_iter = cg_iter + 1;
    total_iter = total_iter + 1;
end

xstar=X{k};
normr = resvec(k);
relres = normr/n2b;
if(normr <= tolb)
    flag = 0;
    resvec = resvec(1:k);
else
    resvec = resvec(1:k+1);
end
```

B Efficient Implementation

```
function [xstar,flag,relres,update,cg_iter,resvec,total_iter] = ...
    APCG_implicit(A,b,nu,delta,m,x0,tol,maxit,Z);
% APCG_implicit   Limited Memory version of Adaptive Preconditioned
%                 Conjugate Gradient Method
%
% This method differs from the APCG method in the way that only m latest
% iterates are kept. Hence, if the method needs to backtrack more than m
% times, the procedure halts with a failure. Also, we don't store matrices
% Z, A_hat, F explicitly to make it memory efficient.
%
% Reference:
% ----------
% Renato D.C. Monteiro, Jerome W. O'Neal and Arkadi Nuemirovski,
% A New Conjugate Gradient Algorithm Incoporating Adaptive Ellipoid
% Preconditioning
% Technical Report, Georgia Institute of Technology, 2004.
%
% This function attempts to solve the system of linear equations A*x=b
% Input and Parameter:
% ---------------------
% A   - The n-by-n coefficient matrix A must be symmetric and positive
%       definite. Also, A's minimum eigenvalue has to be larger than or
%       equal to 1.
% b   - The right hand side column vector b must have size n-by-1.
% nu - Specifies the constant used to test if Z is a nu-preconditioner at
%      certain iterate. If it is not, updating Z is required. It must be
%      between n and the maximum eigenvalue of A. See the paper.
% delta - Specifies the constant used to control restart. 0 < delta < 1.
% m   - Specifies the length of recent iterates. If it is [], 20 is used as
%       the default value.
% x0  - Specifies the initial guess. If it is [], default value (all zero
%       vector) will be used.
% tol - Specifies the tolerance of the method. If it is [], default value
%       1e-6 is used.
% maxit - specifies the maximum number of iterations. If it is [], default
%         value 2*n is used.
% Z - Specifies the n-by-n initial preconditioner with default value
%     speye(n) if it is [].
%
% Output:
% ---------
% xstar - computed solution to A*x = b
% flag  - a) if 0, APCG converged to the desired tolerance TOL within
%             MAXIT iterations.
%         b) if 1, APCG iterated MAXIT times but did not converge, and the
%            latest computed iterate returned as xstar
%         c) if 2, preconditioner Z was ill-conditioned, as it fails to
%            satisfy the minimum eigenvalue of Z'AZ is larger than or equal
%            to 1. The latest computed iterate is returned as xstar.
%         d) if 3, the method tries to backtrack more than m times. The
%            current iterate will be returned as xstar.
% relres - returns the relative residual norm(b-A*x)/norm(b). If flag is
%          0, then, relres <= tol.
% update - returns the number of iterations of updating the preconditioner
% cg_iter - returns the number of CG iterations within one restart round,
%           at which the returned xstar computed.
% resvec - returns a vector of the residual norms at each iteration
%          including NORM(b-A*X0).
% total_iter - returns the total number of CG iterations within multiple
%              restarts
%
% Note: This comment document follows the format of pcg method by MatLab.
%       The error checking and setting default value code are adapted from
%       pcg method.

format long;

if (nargin < 4)
   error('MATLAB:APCG_IMPLICIT:NotEnoughInputs', 'Not enough input arguments.');
end

% Check matrix and right hand side vector inputs have appropriate sizes
[s,n] = size(A);
if (s ~= n)
    error('MATLAB:APCG_IMPLICIT:NonSquareMatrix', 'Matrix must be square.');
end
if ~isequal(size(b),[s,1])
    es = sprintf(['Right hand side must be a column vector of' ...
            ' length %d to match the coefficient matrix.'],s);
        error('MATLAB:APCG_IMPLICIT:RHSsizeNotMatchCoeffMatrix', es)
end

s = size(b,1);
n = s;
if (size(b,2) ~= 1)
    error('MATLAB:APCG_IMPLICIT:RHSnotColumnVec',...
        'Right hand side must be a column vector.');
end

% Assign default values to unspecified parameters
if (nargin < 5) || isempty(m)
    m = 20;
end
if (nargin < 7) || isempty(tol)
   tol = 1e-6;
end
if (nargin < 8) || isempty(maxit)
    maxit = 2*n;
end
```

```
if (nargin < 9) || isempty(Z)
    Z = speye(n);
end

% Check for all zero right hand side vector => all zero solution
n2b = norm(b);                      % Norm of rhs vector, b
if (n2b == 0)                       % if    rhs vector is all zeros
   xstar = zeros(n,1);              % then  solution is all zeros
   flag = 0;                        % a valid solution has been obtained
   relres = 0;                      % the relative residual is actually 0/0
   iter = 0;                        % no iterations need be performed
   resvec = 0;                      % resvec(1) = norm(b-A*x) = norm(0)
   return;
end

if ((nargin >= 6) && ~isempty(x0))
   if ~isequal(size(x0),[n,1])
      es = sprintf(['Initial guess must be a column vector of' ...
            ' length %d to match the problem size.'],n);
      error('MATLAB:APCG_IMPLICIT:WrongInitialGuessSize', es);
   end
else
   x0 = zeros(n,1);
end

if ((nargin >= 9) && ~isempty(Z))
    if ~isequal(size(Z),[s,s])
        es = sprintf(['Preconditioner must be a square matrix' ...
            ' of size %d to match the problem size.'],s);
        error('MATLAB:APCG_IMPLICIT:WrongPreconditionerSize', es);
    else
        vec = eig(Z'*A*Z);
        if vec(n) < 1
            flag = 2;
            es = sprintf(['Preconditioner has to make the minimum' ...
                ' eigenvalue of the preconditioned A larger than' ...
                ' or equal to 1']);
            error('MATLAB:APCG_IMPLICIT:IllConditionedPreconditioner',es);
            return;
        end
    end
end

% Set up for the method
total_iter = 0;          %total iteration including restart
cg_iter = 0;             %CG iteration counter
```

```
Z0 = Z;                  % change notation to follow the convention of Z
                         % as the most recent preconditioner
k=1;                     % start index of iterates
xi = 1;                  % constant used in updating iterations
d = zeros(n,m);          % preallocate vector of storing search directions
d_previous = zeros(n,1); % initial search direction
g = zeros(n,m);          % preallocate space of computed gradients
g(:,1) = A*x0 - b;         % -r0, following the paper's notation
beta = zeros(m,1);
beta(1) = 0;             % initial beta value used in CG iterations
gamma = zeros(m,1);
X(:,1) = x0;               % initial guess as the first iterate
update = 0;             % update iteration counter
r_flag = 0;             % flag indicating a restart
r0 = -g(:,1);             % initial residual
flag = 1;
normr = norm(r0);
tolb = tol * n2b;

if (normr <= tolb)              % Initial guess is a good enough solution
   xstar = x0;
   flag = 0;
   relres = normr/n2b;
   total_iter = 0;
   cg_iter = 0;
   resvec = normr;
   return;
end

resvec = zeros(maxit+1,1);         % Preallocate vector for norm of residuals
resvec(1) = normr;                 % resvec(1) = norm(b-A*x0)

bkcounts = 0;                   % Count number of consecutive backtracking

ind = mod(k,m);      %current index in the memory list
if ind == 0
    ind = m;
end

bkcounts = 0;

%initialization of implicit storage of Z
counter = 1;            % indicates how many non-I F compute
p_vec(:,1) = zeros(n,1); % p_vec stores all p vectors
theta_vec(1) = 0;        % theta_vec stores all theta values
```

```
% loop over maxit iterations (unless convergence or failure)
while resvec(k) > tolb && total_iter < maxit

    % now compute g_k'Z(Z'AZ)Z'g_k needed to test if Z is nu-preconditioner
    % the objective to choose a proper intermediate representation is to
    % simplify computation process
    %
    % let v1 = g_k'*Z = g_k'*Z0*F0*F2*...*Fj with j = k-1
    v1 = g(:,ind)'*Z0;
    for i = 1:counter-1,
        ttmp = theta_vec(i);
        utmp = sqrt((n-ttmp^2)/(n-1));
        v1 = v1 + (ttmp/utmp-1)*(v1*p_vec(:,i))*p_vec(:,i)';
    end

    gamma(ind) = norm(v1')^2;

    % let v2 = g_k'*Z*Z' = v1*Z'
    v2 = v1;
    for i = counter-1:-1:1,
        ttmp = theta_vec(i);
        utmp = sqrt((n-ttmp^2)/(n-1));
        v2 = v2 + (ttmp/utmp-1)*(v2*p_vec(:,i))*p_vec(:,i)';
    end
    v2 = v2*Z0';

    val = v2*(A*v2');

    while val > nu*gamma(ind)
        % Update is necessary, i.e, compute theta and vector p
        % By the formula
        % w = xi^(-1/2)*(Z'*g_k) = xi^(-1/2)*v1'
        % A_hat = xi^(-1)*Z'*A*Z
        % p = A_hat*w = xi^(-3/2)*Z'*A*Z*Z'*g_k = xi^(-3/2)*Z'*A*v2';
        p = xi^(-3/2)*(Z0'*(A*v2'));
        for i = 1:counter-1,
            ttmp = theta_vec(i);
            utmp = sqrt((n-ttmp^2)/(n-1));
            p = p + (ttmp/utmp-1)*p_vec(:,i)*(p_vec(:,i)'*p);
        end
        normp = norm(p);
        % tau = sqrt(w'*A_hat*w)/normp = sqrt(w'*p)/normp
        %     = sqrt(xi^(-1/2)*(v1*p))/normp
        tau = sqrt(xi^(-1/2)*(v1*p))/normp;
        p = p/normp;
```

```
theta = tau*sqrt(n);

if theta < 1
    mu = sqrt((n-theta^2)/(n-1));
    theta_vec(counter) = theta;
    p_vec(:,counter) = p;
    counter = counter + 1;
else
    mu = 1;        % F = I, no need to store
end
xi = xi*(mu^(-2));

if xi > delta
    if k == 1
        bkcounts = 0;
    else
        bkcounts = bkcounts + 1;
        if bkcounts >= m
            xstar = X(:,ind);
            resvec = resvec(1:k);
            relres = resvec(k)/n2b;
            disp('The APCG_implicit method reached maximum number of backtrack');
            flag = 3;
            return;
        end
        k = max(k-1,1);
        ind = mod(k,m);
        if ind == 0
            ind = m;
        end
    end
else
    Z0 = xi^(-1/2)*Z0;
    X(:,1) = X(:,ind);

    total_iter = 0;
    cg_iter = 0;
    k = 1;
    xi = 1;
    d_previous = zeros(n,1);
    g(:,1) = A*X(:,1) - b;
    resvec(1) = norm(g(:,1));
    beta(1) = 0;
    r_flag = 1;

    bkcounts = 0;
```

```
        ind = mod(k,m);
        if ind == 0
            ind = m;
        end

        r_flag = 1;
    end

    update = update +1;

    if r_flag == 1
        break;
    end


    % re-evaluate value due to the following reasons
    % 1) Z may be updated by a non-identity F
    % 2) g_k changes as k changes due to backtrack (if restart, it
    %    jumps out early and won't reach here)
    %
    % The value of g_k'Z(Z'AZ)Z'g_k is required to test inner while
    % condition
    v1 = g(:,ind)'*Z0;
    for i = 1:counter-1,
        ttmp = theta_vec(i);
        utmp = sqrt((n-ttmp^2)/(n-1));
        v1 = v1 + (ttmp/utmp-1)*(v1*p_vec(:,i))*p_vec(:,i)';
    end

    % let v2 = g_k'*Z*Z' = v1*Z'
    v2 = v1;
    for i = counter-1:-1:1,
        ttmp = theta_vec(i);
        utmp = sqrt((n-ttmp^2)/(n-1));
        v2 = v2 + (ttmp/utmp-1)*(v2*p_vec(:,i))*p_vec(:,i)';
    end
    v2 = v2*Z0';

    val = v2*(A*v2');
end

if r_flag ==1
    r_flag = 0;
    continue;
end
```

```
    if k == 1
        d(:,ind) = -v2' + beta(ind)*d_previous;
    else
        pre_ind = mod(k-1,m);
        if pre_ind == 0
            pre_ind = m;
        end
        beta(ind) = gamma(ind)/gamma(pre_ind);
        d(:,ind) = -v2' + beta(ind)*d(:,pre_ind);
    end

    alpha = gamma(ind)/(d(:,ind)'*(A*d(:,ind)));

    next_ind = mod(k+1,m);
    if next_ind == 0
        next_ind = m;
    end

    X(:,next_ind) = X(:,ind) + alpha*d(:,ind);
    g(:,next_ind) = g(:,ind) + alpha*(A*d(:,ind));

    k = k+1;
    ind = next_ind;

    resvec(k) = norm(g(:,ind));

    cg_iter = cg_iter + 1;
    total_iter = total_iter + 1;
end

xstar=X(:,ind);
normr = resvec(k);
relres = normr/n2b;
if(normr <= tolb)
    flag = 0;
    resvec = resvec(1:k);
else
    resvec = resvec(1:k+1);
end
```