IML++ v. 1.2

Iterative Methods Library
Reference Guide

May 1995

Jack Dongarra
Andrew Lumsdaine
Roldan Pozo
Karin Remington

National Institute of Standards and Technology
Oak Ridge National Laboratory
University of Notre Dame
University of Tennessee, Knoxville

Contents

1 Introduction

The Iterative Methods Library, IML++, is a collection of algorithms implemented in C++ for solving both symmetric and nonsymmetric linear systems of equations by using iterative techinques.

One motivation for this package is that high level matrix algorithms, such as those found in Barett et. al. [2], can be easily implemented with the template facilites of C++. For example, consider a preconditioned conjugate gradient algorithm, used to solve $Ax = b$, with preconditioner M. A comparison between the pseudo-code description of the algorithm and the body of a C++ routine used to implement the algorithm is shown in Figure 1. Notice that in the C++ code example, no mention of a specific matrix type (e.g. dense, sparse, distirbuted) is given. The operators such as * and + have been overloaded to work with matrix and vectors formats. Thus, the code fragment represents an *interface* and can be used with any matrix and vector classes which support these operations.

Initial $r^{(0)} = b - Ax^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence;
end

```
r = b - A * x;
for (int i = 1; i < max_iter; i++) {
    z = M.solve(r);
    rho = dot(r, z);
    if (i == 1)
       p = z;
    else {
       beta = rho1 / rho0;
       p = z + p * beta;
    }
    q = A * p;
    alpha = rho1 / dot(p, q);
    x = x + alpha * p;
    r = r - alpha * q;
    if (norm(r) / norm(b) < tol) break;
}
```

Figure 1: Comparison of an algorithm for the preconditioned conjugate gradient method in psuedocode and the corresponding IML++ routine.

The following iterative methods have been incorporated into IML++:

- Richardson Iteration
- Chebyshev Iteration
- Conjugate Gradient (CG)
- Conjugate Gradient Squared (CGS)
- BiConjugate Gradient (BiCG)
- BiConjugate Gradient Stabilized (BiCGSTAB)

- Generalized Minimum Residual (GMRES)
- Quasi-Minimal Residual Without Lookahead (QMR)

All of these methods are designed to be used in conjunction with a preconditioner.

In addition, IML++ provides SparseLib++ compatible implementations of the following preconditioners:

- Diagonal
- Incomplete LU (ILU)
- Incomplete Cholesky (IC)

The functions provided are fully templated [1] , so they can be used with any matrix and vector library that provide the required level of functionality (see Section 2), including distributed sparse and dense matrices. In effect, the details of the underlying data structure are separated from the mathematical algorithm. The result is a library of high level mathematical denotations that can run on a large variety of hardware platforms (e.g., distributed networks, multicomputers, as well as single node workstations) without modification.

The iterative methods functions and the preconditioner functions are described in detail in Section 4.

[1] There are two senses in which the word 'templates' is used in scientific computing. Templates as code exemplars is the meaning used in the context of [2]. Templates in the C++ sense indicates the more formal strategy for reusing skeletal code.

Vector	$\leftarrow$	*Matrix* **operator***(*Vector*)	"Matrix" by "Vector" product
Vector	$\leftarrow$	*Matrix*::**trans_mult**(*Vector*)	transpose-*Matrix* by *Vector* product

Figure 2: Interface requirement for tempated `Matrix` object.

2 Requirements

Here we give description of the specific functions that matrix and vector classes must provide in order to be used with IML++. In order to make IML++ as useful (and portable) as possible to other matrix and/or vector packages, we have assumed only a minimal level of functionality.

To illustrate the functionality that IML++ functions require of other packages, it is helpful to look at an example. A typical IML++ function declaration looks like this:

template < class *Matrix*, class *Vector*, class *Preconditioner*, class *Real* >
int **CG** (const Matrix& A, Vector& x, const Vector& b, const Preconditioner& M, int& *max_iter*,
Real& *tol*)

There are a few things to note about this declaration (which will be generic to most of the other IML++ functions). First, the function is fully templated. That is, the function can be called with any set of arguments that are members of classes that provide a mimimum level of functionality (described below). In fact, since the objects passed to the function are only accessed through their member functions, the classes that are substituted for Matrix, Vector, and Preconditioner do not necessarily have to be actual matrices and vectors at all (e.g. they may just be utilized in a matrix-vector product); they only need to be able to carry out required interface computations listed below.

Matrices The *Matrix* class (corresponding to A in the linear system $Ax = b$) must supply the functions listed in Figure 2.

The matrices in the linear systems $Ax = b$ are accessed only through the * operator and the **trans_mult()** member function. The return type in both cases is a *Vector* (the same type as the supplied argument). Note that not all of the IML++ functions will necessarily use **trans_mult()**.

The **GMRES()** routine in particular requires two matrices as input. The first (which will typically be a sparse matrix) corresponds to the matrix in the linear system $Ax = b$. The second (typically a smaller, dense matrix) corresponds to the upper Hessenberg matrix H that is constructed during the GMRES iterations. Since the second matrix is used in a different way than the first, its class must supply different functionality. In particular, it must have **operator()** for accessing individual elements. For this matrix class, it is important to remember that IML++ uses the C/C++ convention of 0-based indexing. That is, `A(0,0)` is the first component of the matrix `A`. Also, the type of a single matrix entry must be compatible with the type of single vector entry. That is, operations such as `A(i,j)*x(j)` must be able to be carried out. See the **GMRES()** man page for more information.

Vector()	Constructor for null (zero-length) *Vector*
Vector(unsigned int n)	Constructor for *Vector* of length n

Figure 3: Interface requirements for constructors of `Vector` class.

Vector	$\leftarrow$	**operator=**(*Vector*)	Assignment of *Vector* to *Vector*
Vector	$\leftarrow$	**operator=**(*Scalar*)	Assignment of *Scalar* to all components of *Vector*
Vector	$\leftarrow$	*Vector* **operator+**(*Vector*)	Addition
Vector	$\leftarrow$	*Vector* **operator-**(*Vector*)	Subtraction
Vector	$\leftarrow$	*Scalar* **operator***(*Vector*)	Multiplication of *Vector* by *Scalar*
Scalar	$\leftarrow$	*Vector* **operator()**(int)	Element access
Scalar	$\leftarrow$	**dot**(*Vector*, *Vector*)	*Vector* inner product
Real	$\leftarrow$	**norm**(*Vector*)	*Vector* norm

Figure 4: Interface requirements for `Vector` operations.

Vectors The *Vector* class must supply the following constructors in figure 3, together with fundamental operations listed in figures 4.

IML++ uses the C/C++ convention that vectors use 0-based indexing. That is, `x(0)` is the first component of the vector `x`. This is in contrast to Fortran, which uses 1-based indexing. Since (presumably) all users of this package will be using it with C and/or C++ matrices and vectors, this assumption should not be limiting in any way.

Scalars We use the convention that a scalar is the same type as a single component of a vector. In particular, the **dot()** function must return a scalar type which is the same type as a single component of a vector. That is, assignments of the form `x(0) = dot(x, y)` must be made without type conversion.

Preconditioners The *Preconditioner* class can be viewed as a simple wrapper around a user-defined function. These function may, for example, perform incomplete LU factorization, diagonal scaling, or nothing at all (corresponding to unpreconditioned case). A preconditioner matrix M is typically used to compute $M^{-1}x$ or $(M^T)^{-1}x$ during the course of a basic iteration, and thus can be seen as taking some input vector and return a corresponding vector. The corresponding C++ class must therefore provide the two fundamental capabilities listed in figure 5.

Vector	$\leftarrow$	Preconditioner::**solve**(*Vector*)	solve linear system
Vector	$\leftarrow$	Preconditioner::**trans_solve**(*Vector*)	solve transpose linear system

Figure 5: Interface requirements for `Preconditioner` class.

Preconditioners are accessed only through their **solve()** and **trans_solve()** member functions. The return type in both cases is a *Vector* (the same type as the supplied argument). Note that not all all of the IML++ functions will use **trans_solve()**.

Reals At this time, all IML++ functions test the value of the residual norm against a specified tolerance to determine convergence. The type of the tolerance variable is templated so that either **float** or **double** can be used. The **norm()** function must return the *Real* type. Note that the elements of a `Vector` class can be complex or user-defined, so that *Real* type may be different than the *Scalar* type.

Summary The following is a summary of necessary functions for use with IML++.

		Vector()
		Vector(unsigned int n)
Vector	$\leftarrow$	*Matrix* **operator***(*Vector*)
Vector	$\leftarrow$	*Matrix*::**trans_mult**(*Vector*)
Vector	$\leftarrow$	**operator=**(*Vector*)
Vector	$\leftarrow$	**operator=**(*Scalar*)
Vector	$\leftarrow$	*Vector* **operator+**(*Vector*)
Vector	$\leftarrow$	*Vector* **operator-**(*Vector*)
Vector	$\leftarrow$	*Scalar* **operator***(*Vector*)
Scalar	$\leftarrow$	*Vector* **operator()**(int)
Scalar	$\leftarrow$	**dot**(*Vector*, *Vector*)
Real	$\leftarrow$	**norm**(*Vector*)
Vector	$\leftarrow$	Preconditioner::**solve**(*Vector*)
Vector	$\leftarrow$	Preconditioner::**trans_solve**(*Vector*)

Note that certain "optimized" operators such as `+=` are not used by IML++. This may cause a slight loss of performance in certain applications. However, the advantage is increased portability of IML++. In order to use operators like `+=` (if your package supplies them), you should replace occurences of operations like `x = x + y` with `x += y` in the text of the IML++ routines.

The test programs that come with IML++ use SparseLib++. When in doubt about the required functionality of other matrix and/or vector packages, refer to SparseLib++.

3 Using IML++

In order to use IML++, the user must supply matrix and vector classes (the functionality for which is described in Section 2). Typically, IML++ will be used with the matrix and vector packages together in a common program. IML++ is accessed by including the appropriate header files which provide the template declarations. The header files which provide the matrix and vector class declarations must also be included.

As an application example, the following code listing demonstrates the use of IML++ in conjuction with a publicly available matrix library, SparseLib++, to solve a linear system with **CG**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved using **CG** and a diagonal preconditioner. The modifications to this example which would be necessary to use it with different matrix and/or vector classes should be fairly obvious.

```
#include <stdlib.h>                     // System includes
#include <iostream.h>                   //

#include "compcol_double.h"             // Compressed column matrix header
#include "iohb_double.h"                // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"            // MV_Vector level 1 BLAS
#include "diagpre_double.h"             // Diagonal preconditioner

#include "cg.h"                         // IML++ CG template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                   // Convergence tolerance
  int result, maxit = 150;              // Maximum iterations

  CompCol_Mat_double A;                 // Create a matrix
  readHB_mat(argv[1], &A);              // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);    // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);              // Read rhs data

  DiagPreconditioner_double D(A);       // Create diagonal preconditioner

  result = CG(A, x, b, D, maxit, tol);  // Solve system

  cout << "CG flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

The executable for this example can be made by compiling it and linking it with the object files for the matrix and vector classes. For compilation, be sure to use appropriate compiler directives so that all header and library files can be found.

The following is an example Makefile that could be used to create an executable file (called **main**) from the code in the above example. It assumes a directory structure similar to that of the SparseLib++ distribution.

```
# Example Makefile

# The C++ compiler
CPP               = g++

# The architecture
ARCH              = sun4

# The top-level SparseLib++ directory
SPARSELIB         = /usr/local/src/SparseLib++

# The various include directories
IML_INCLUDE       = $(SPARSELIB)/mv/include
MV_INCLUDE        = $(SPARSELIB)/mv/include
SPARSELIB_INCLUDE = $(SPARSELIB)/sparselb/include

# A list of all include directives for the compiler
INCLUDES          = -I$(IML_INCLUDE) -I$(MV_INCLUDE) -I$(SPARSELIB_INCLUDE)

# The SparseLib++ library archive file
SPARSELIB_LIB     = $(SPARSELIB)/sparselb/libs/$(ARCH)

# Libraries to be linked
LIBS              = -L$(SPARSELIB_LIB) -lsparse$(CPP) -lm

# Rule to create executable main from main.o
main: main.o
	$(CPP) $(CPPFLAGS) -o main main.o $(LIBS)

# Rule to create object main.o from main.cc
main.o: main.cc
	$(CPP) $(CPPFLAGS) $(INCLUDE_DIRS) -c main.cc

# Include dependencies
main.o: $(IML_INCLUDE)/cg.h $(SPARSELIB_INCLUDE)/diagpre.h \
	$(SPARSELIB_INCLUDE)/compcol1.h $(SPARSELIB_INCLUDE)/readhb.h \
	$(MV_INCLUDE)/vector.h $(MV_INCLUDE)/blas1.h
```

See the IML++ test directory for more examples.

4 Iterative Method Library Functions

In the following pages, we provide a detailed description of each of the iterative methods functions available in IML++. Each function is described in turn on a "man" page. For each function, we provide an example of its declaration, a detailed description of the function, its return values, an example of its usage, and cross references.

Name

BiCG — BiConjugate Gradient Iteration

Declaration

#include "bicg.h"

template < class *Matrix*, class *Vector*, class *Preconditioner*, class *Real* >
int **BiCG** (const Matrix& A, Vector& x, const Vector& b,
const Preconditioner& M, int& *max_iter*, Real& *tol*)

Description

BiCG solves the unsymmeteric linear system $Ax = b$ using the preconditioned BiConjugate Gradient method.

This is a fully templated function.

On input, A specifies the matrix, b the right-hand side, and x the initial guess for the solution of the unsymmetric linear system $Ax = b$. In addition, M specifies a preconditioner, *max_iter* specifies the maximum number of iterations that the method will take, and *tol* specifies the convergence tolerance for the method.

Convergence is achieved if the normalized residual is less than the specified tolerance, i.e., if $||r||/||b|| < tol$.

Return Values

A return value of 0 indicates convergence to the specified tolerance within the specified maximum number of iterations. A return value of 1 indicates that the method did not reach the specified convergence tolerance in the maximum number of iterations. A return value of 2 indicates that a breakdown occurred.

Upon return, the output arguments have the following values:

x	approximate solution to $Ax = b$ computed at the final iteration
tol	the value of the $||r||/||b||$ achieved after the final iteration
max_iter	the number of iterations performed before return

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **BiCG**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved using **BiCG** and a diagonal preconditioner.

```
#include <stdlib.h>                         // System includes
#include <iostream.h>                       //

#include "compcol_double.h"                 // Compressed column matrix header
#include "iohb_double.h"                    // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"                // MV_Vector level 1 BLAS
#include "diagpre_double.h"                 // Diagonal preconditioner

#include "bicg.h"                           // IML++ BiCG template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                       // Convergence tolerance
  int result, maxit = 150;                  // Maximum iterations

  CompCol_Mat_double A;                     // Create a matrix
  readHB_mat(argv[1], &A);                  // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);        // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);                  // Read rhs data

  DiagPreconditioner_double D(A);           // Create diagonal preconditioner

  result = BiCG(A, x, b, D, maxit, tol);    // Solve system

  cout << "BiCG flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

See Also

SparseLib++
DiagPreconditioner

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Numerical Analysis Dundee 1975, G. Watson, ed., Springer Verlag, Berlin, New York, 1976, pp. 73–89.

Name **BiCGSTAB** — BiConjugate Gradient Iteration Stabilized

Declaration #include "bicgstab.h"

template < class *Matrix*, class *Vector*, class *Preconditioner*, class *Real* >
int **BiCGSTAB** (const Matrix& A, Vector& x, const Vector& b,
const Preconditioner& M, int& *max_iter*, Real& *tol*)

Description **BiCGSTAB** solves the unsymmeteric linear system $Ax = b$ using the preconditioned BiConjugate Gradient Stabilized method.

This is a fully templated function.

On input, A specifies the matrix, b the right-hand side, and x the initial guess for the solution of the unsymmetric linear system $Ax = b$. In addition, M specifies a preconditioner, *max_iter* specifies the maximum number of iterations that the method will take, and *tol* specifies the convergence tolerance for the method.

Convergence is achieved if the normalized residual is less than the specified tolerance, i.e., if $||r||/||b|| < tol$.

Return Values A return value of 0 indicates convergence to the specified tolerance within the specified maximum number of iterations. A return value of 1 indicates that the method did not reach the specified convergence tolerance in the maximum number of iterations. A return value of 2 indicates that a breakdown occurred with $\rho_{i-1} = \langle \tilde{r}, r^{i-1} \rangle = 0$. A return value of 3 indicates that a breakdown occurred with $\omega_i = \langle t, s \rangle / \langle t, t \rangle = 0$.

Upon return, the output arguments have the following values:

x	approximate solution to $Ax = b$ computed at the final iteration
tol	the value of the $\|\|r\|\|/\|\|b\|\|$ achieved after the final iteration
max_iter	the number of iterations performed before return

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **BiCGSTAB**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved using **BiCGSTAB** and a diagonal preconditioner.

```
#include <stdlib.h>                       // System includes
#include <iostream.h>                     //

#include "compcol_double.h"               // Compressed column matrix header
#include "iohb_double.h"                  // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"              // MV_Vector level 1 BLAS
#include "diagpre_double.h"               // Diagonal preconditioner

#include "bicgstab.h"                     // IML++ BiCGSTAB template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                     // Convergence tolerance
  int result, maxit = 150;                // Maximum iterations

  CompCol_Mat_double A;                   // Create a matrix
  readHB_mat(argv[1], &A);                // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);      // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);                // Read rhs data

  DiagPreconditioner_double D(A);         // Create diagonal preconditioner

  result = BiCGSTAB(A, x, b, D, maxit, tol); // Solve system

  cout << "BiCGSTAB flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

See Also

SparseLib++
DiagPreconditioner

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

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

Name CG — Conjugate Gradient Iteration

Declaration #include "cg.h"

template < class *Matrix*, class *Vector*, class *Preconditioner*, class *Real* >
int **CG** (const Matrix& A, Vector& x, const Vector& b,
const Preconditioner& M, int& *max_iter*, Real& *tol*)

Description **CG** solves the symmeteric positive-definite linear system $Ax = b$ using the preconditioned Conjugate Gradient method.

This is a fully templated function.

On input, A specifies the matrix, b the right-hand side, and x the initial guess for the solution of the unsymmetric linear system $Ax = b$. In addition, M specifies a preconditioner, *max_iter* specifies the maximum number of iterations that the method will take, and *tol* specifies the convergence tolerance for the method.

Convergence is achieved if the normalized residual is less than the specified tolerance, i.e., if $||r||/||b|| < tol$.

Return Values A return value of 0 indicates convergence to the specified tolerance within the specified maximum number of iterations. A return value of 1 indicates that the method did not reach the specified convergence tolerance in the maximum number of iterations.

Upon return, the output arguments have the following values:

x	approximate solution to $Ax = b$ computed at the final iteration
tol	the value of the $\|\|r\|\|/\|\|b\|\|$ achieved after the final iteration
max_iter	the number of iterations performed before return

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **CG**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved using **CG** and a diagonal preconditioner.

```
#include <stdlib.h>                     // System includes
#include <iostream.h>                   //

#include "compcol_double.h"             // Compressed column matrix header
#include "iohb_double.h"                // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"            // MV_Vector level 1 BLAS
#include "icpre_double.h"               // Diagonal preconditioner

#include "cg.h"                         // IML++ CG template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                   // Convergence tolerance
  int result, maxit = 150;              // Maximum iterations

  CompCol_Mat_double A;                 // Create a matrix
  readHB_mat(argv[1], &A);              // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);    // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);              // Read rhs data

  ICPreconditioner_double M(A);         // Create IC preconditioner

  result = CG(A, x, b, M, maxit, tol);  // Solve system

  cout << "CG flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

See Also

SparseLib++
ICPreconditioner

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

G. H. Golub and C. F. Van Loan, *Matrix Computations*, The John Hopkins University Press, Baltimore, Maryland, 1983.

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

Name CGS — Conjugate Gradient Squared Iteration

Declaration #include "cgs.h"

template < class *Matrix*, class *Vector*, class *Preconditioner*, class *Real* >
int **CGS** (const Matrix& A, Vector& x, const Vector& b,
const Preconditioner& M, int& *max_iter*, Real& *tol*)

Description **CGS** solves the unsymmeteric linear system $Ax = b$ using the preconditioned Conjugate Gradient Squared method.

This is a fully templated function.

On input, A specifies the matrix, b the right-hand side, and x the initial guess for the solution of the unsymmetric linear system $Ax = b$. In addition, M specifies a preconditioner, *max_iter* specifies the maximum number of iterations that the method will take, and *tol* specifies the convergence tolerance for the method.

Convergence is achieved if the normalized residual is less than the specified tolerance, i.e., if $||r||/||b|| < tol$.

Return Values A return value of 0 indicates convergence to the specified tolerance within the specified maximum number of iterations. A return value of 1 indicates that the method did not reach the specified convergence tolerance in the maximum number of iterations. A return value of 2 indicates that a breakdown occurred.

Upon return, the output arguments have the following values:

x	approximate solution to $Ax = b$ computed at the final iteration
tol	the value of the $\|\|r\|\|/\|\|b\|\|$ achieved after the final iteration
max_iter	the number of iterations performed before return

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **CGS**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved using **CGS** and a diagonal preconditioner.

```
#include <stdlib.h>                     // System includes
#include <iostream.h>                   //

#include "compcol_double.h"             // Compressed column matrix header
#include "iohb_double.h"                // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"            // MV_Vector level 1 BLAS
#include "diagpre_double.h"             // Diagonal preconditioner

#include "cgs.h"                        // IML++ CGS template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                   // Convergence tolerance
  int result, maxit = 150;              // Maximum iterations

  CompCol_Mat_double A;                 // Create a matrix
  readHB_mat(argv[1], &A);              // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);    // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);              // Read rhs data

  DiagPreconditioner_double D(A);       // Create diagonal preconditioner

  result = CGS(A, x, b, D, maxit, tol); // Solve system

  cout << "CGS flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

See Also

SparseLib++
DiagPreconditioner

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

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

Name

CHEBY — Chebyshev Iteration

Declaration

#include "cheby.h"

template < class *Matrix*, class *Vector*, class *Preconditioner*, class *Real*,
class *Type* >
int **CHEBY** (const Matrix& A, Vector& x, const Vector& b,
const Preconditioner& M, int& *max_iter*, Real& *tol*,
Type *eigmin*, Type *eigmax*)

Description

CHEBY solves the unsymmeteric linear system $Ax = b$ using the preconditioned Chebyshev iteration.

This is a fully templated function.

On input, A specifies the matrix, b the right-hand side, and x the initial guess for the solution of the unsymmetric linear system $Ax = b$. In addition, M specifies a preconditioner, *max_iter* specifies the maximum number of iterations that the method will take, and *tol* specifies the convergence tolerance for the method. Finally, the parameters *eigmin* and *eigmax* are parameters provided to estimate an ellipse containing the spectrum of A. In the case of positive-definite A, these parameters are real and correspond to estimates of the minimal and maximal eigenvalues of A, respectively. Note that poor estimates for these values can cause poor convergence behavior (including divergence).

Convergence is achieved if the normalized residual is less than the specified tolerance, i.e., if $||r||/||b|| < tol$.

Return Values

A return value of 0 indicates convergence to the specified tolerance within the specified maximum number of iterations. A return value of 1 indicates that the method did not reach the specified convergence tolerance in the maximum number of iterations.

Upon return, the output arguments have the following values:

x	approximate solution to $Ax = b$ computed at the final iteration
tol	the value of the $\|\|r\|\|/\|\|b\|\|$ achieved after the final iteration
max_iter	the number of iterations performed before return

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **CHEBY**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved using **CHEBY** and a diagonal preconditioner. The parameters *eigmin* and *eigmax* for this example are chosen based on an 8×8 discretization of the two-dimensional Poisson problem on the unit square.

```
#include <stdlib.h>                     // System includes
#include <iostream.h>                   //

#include "compcol_double.h"             // Compressed column matrix header
#include "iohb_double.h"                // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"            // MV_Vector level 1 BLAS
#include "diagpre_double.h"             // Diagonal preconditioner

#include "cheby.h"                      // IML++ Cheby template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                   // Convergence tolerance
  int result, maxit = 300;              // Maximum iterations
  double mineig = .01;                  // eigenvalue information
  double maxeig = 3;                    // (this info for la2d8 example)

  CompCol_Mat_double A;                 // Create a matrix
  readHB_mat(argv[1], &A);              // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);    // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);              // Read rhs data

  DiagPreconditioner_double D(A);       // Create diagonal preconditioner

  result = CHEBY(A, x, b, D, maxit, tol, mineig, maxeig);  // Solve system

  cout << "cheby flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

See Also

SparseLib++
DiagPreconditioner

S. Ashby, *CHEBYCODE: A Fortran implementation of Manteuffel's adaptive Chebyshev algorithm*, Tech. Report UIUCDCS-R-85-1203, University of Illinois, 1985.

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

G. H. Golub and C. F. Van Loan, *Matrix Computations*, The John Hopkins University Press, Baltimore, Maryland, 1983.

T. Manteuffel, *The Tchebychev iteration for nonsymmetric linear systems*, Numer. Math., 28 (1977), pp. 307–327.

Name

GMRES — Generalized Minimum Residual Iteration

Declaration

#include "gmres.h"

template < class *SMatrix*, class *Vector*, class *Preconditioner*, class *DMatrix*, class *Real* >
int **GMRES** (const SMatrix& A, Vector& x, const Vector& b,
const Preconditioner& M, DMatrix& H, int& m,
int& *max_iter*, Real& *tol*)

Description

GMRES solves the unsymmeteric linear system $Ax = b$ using the preconditioned Generalized Minimum Residual method.

This is a fully templated function.

On input, A specifies the matrix, b the right-hand side, and x the initial guess for the solution of the unsymmetric linear system $Ax = b$. In addition, M specifies a preconditioner, H specifies a matrix to hold the coefficients of the upper Hessenberg matrix constructed by the GMRES iterations, m specifies the number of iterations for each restart, *max_iter* specifies the maximum number of iterations that the method will take, and *tol* specifies the convergence tolerance for the method. Note that the size of H must be at least $m \times m$.

GMRES() requires two matrices as input, A and H. The matrix A (which will typically be a sparse matrix) corresponds to the matrix in the linear system $Ax = b$. The matrix H (which will typically be a dense matrix) corresponds to the upper Hessenberg matrix H that is constructed during the GMRES iterations. Within GMRES, H is used in a different way than A, soits class must supply different functionality. That is, A is only accessed through its matrix-vector and transpose-matrix-vector multiplication functions. On the other hand, GMRES solves a (dense upper triangular) linear system of equations based on H. Therefore, the class to which H belongs (DMatrix) must provide **operator()** for element access. For this matrix class, it is important to remember that IML++ uses the C/C++ convention that matrices use 0-based indexing. That is, `A(0,0)` is the first component of the matrix `A`. Also, the type of a single matrix entry must be compatible with the type of single vector entry. That is, operations such as `A(i,j)*x(j)` must be able to be carried out.

Convergence is achieved if the normalized residual is less than the specified tolerance, i.e., if $||r||/||b|| < tol$.

Return Values

A return value of 0 indicates convergence to the specified tolerance within the specified maximum number of iterations. A return value of 1 indicates that the method did not reach the specified convergence tolerance in the maximum number of iterations.

Upon return, the output arguments have the following values:

x	approximate solution to $Ax = b$ computed at the final iteration
tol	the value of the $\|\|r\|\|/\|\|b\|\|$ achieved after the final iteration
max_iter	the number of iterations performed before return
H	the upper triangular factor of the upper Hessenberg matrix constructed by GMRES

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **GMRES**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved using **GMRES** and a diagonal preconditioner. A restart value of 32 is used. The matrix `A` is a sparse matrix and `H` is a dense matrix.

```
#include <stdlib.h>                         // System includes
#include <iostream.h>                       //

#include "compcol_double.h"                 // Compressed column matrix header
#include "iohb_double.h"                    // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"                // MV_Vector level 1 BLAS
#include "ilupre_double.h"                  // Diagonal preconditioner

#include MATRIX_H                           // MV_Matrix dense matrix header
#include "gmres.h"                          // IML++ GMRES template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                       // Convergence tolerance
  int result, maxit = 150, restart = 32;    // Maximum, restart iterations

  CompCol_Mat_double A;                     // Create a matrix
  readHB_mat(argv[1], &A);                  // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);        // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);                  // Read rhs data

  MATRIX_double H(restart+1, restart, 0.0); // storage for upper Hessenberg H

  CompCol_ILUPreconditioner_double M(A);    // Create ILU preconditioner

  result = GMRES(A, x, b, M, H, restart, maxit, tol); // Solve system

  cout << "GMRES flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

See Also

SparseLib++
ILUPreconditioner

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

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

Name

IR — Richardson Iteration

Declaration

#include "ir.h"

template < class *Matrix*, class *Vector*, class *Preconditioner*, class *Real* >
int **IR** (const Matrix& A, Vector& x, const Vector& b,
const Preconditioner& M, int& *max_iter*, Real& *tol*)

Description

IR solves the unsymmeteric linear system $Ax = b$ using the preconditioned Richardson method.

This is a fully templated function.

On input, A specifies the matrix, b the right-hand side, and x the initial guess for the solution of the unsymmetric linear system $Ax = b$. In addition, M specifies a preconditioner, *max_iter* specifies the maximum number of iterations that the method will take, and *tol* specifies the convergence tolerance for the method.

Convergence is achieved if the normalized residual is less than the specified tolerance, i.e., if $||r||/||b|| < tol$.

The iterative refinement algorithm is realized by taking A to be the preconditioner (assuming class to which A belongs has a **solve()** member function).

Return Values

A return value of 0 indicates convergence to the specified tolerance within the specified maximum number of iterations. A return value of 1 indicates that the method did not reach the specified convergence tolerance in the maximum number of iterations.

Upon return, the output arguments have the following values:

x	approximate solution to $Ax = b$ computed at the final iteration
tol	the value of the $||r||/||b||$ achieved after the final iteration
max_iter	the number of iterations performed before return

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **IR**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved with **IR** and a diagonal preconditioner, i.e., with the Gauss-Jacobi iteration.

```
#include <stdlib.h>                     // System includes
#include <iostream.h>                   //

#include "compcol_double.h"             // Compressed column matrix header
#include "iohb_double.h"                // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"            // MV_Vector level 1 BLAS
#include "diagpre_double.h"             // Diagonal preconditioner

#include "ir.h"                         // IML++ IR template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                   // Convergence tolerance
  int result, maxit = 300;              // Maximum iterations

  CompCol_Mat_double A;                 // Create a matrix
  readHB_mat(argv[1], &A);              // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);    // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);              // Read rhs data

  DiagPreconditioner_double D(A);       // Create diagonal preconditioner

  result = IR(A, x, b, D, maxit, tol);  // Solve system

  cout << "IR flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

See Also

SparseLib++
DiagPreconditioner

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

R. S. Varga, *Matrix Iterative Analysis*, Automatic Computation Series, Prentice-Hall Inc, Englewood Cliffs, New Jersey, 1962.

Name

QMR — Quasi Minimal Residual Iteration (Without Look Ahead)

Declaration

#include "qmr.h"

template < class *Matrix*, class *Vector*, class *Preconditioner1*,
class *Preconditioner2*, class *Real* >
int **QMR** (const Matrix& A, Vector& x, const Vector& b,
const Preconditioner& *M1*, const Preconditioner& *M2*,
int& *max_iter*, Real& *tol*)

Description

QMR solves the unsymmeteric linear system $Ax = b$ using the preconditioned Quasi-Minimal Residual method.

This is a fully templated function.

On input, A specifies the matrix, b the right-hand side, and x the initial guess for the solution of the unsymmetric linear system $Ax = b$. In addition, *M1* and *M2* specify preconditioners, *max_iter* specifies the maximum number of iterations that the method will take, and *tol* specifies the convergence tolerance for the method.

Convergence is achieved if the normalized residual is less than the specified tolerance, i.e., if $||r||/||b|| < tol$.

Return Values

A return value of 0 indicates convergence to the specified tolerance within the specified maximum number of iterations. A return value of 1 indicates that the method did not reach the specified convergence tolerance in the maximum number of iterations. A return value of 2 indicates that a breakdown associated with ρ occurred. A return value of 3 indicates that a breakdown associated with β occurred. A return value of 4 indicates that a breakdown associated with γ occurred. A return value of 5 indicates that a breakdown associated with δ occurred. A return value of 6 indicates that a breakdown associated with ϵ occurred. A return value of 7 indicates that a breakdown associated with ξ occurred.

Upon return, the output arguments have the following values:

x	approximate solution to $Ax = b$ computed at the final iteration
tol	the value of the $\|\|r\|\|/\|\|b\|\|$ achieved after the final iteration
max_iter	the number of iterations performed before return

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **QMR**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file specified in `argv[1]`. An initial guess of 0 is made for the solution and the system is solved using **QMR** and diagonal preconditioners.

```
#include <stdlib.h>                          // System includes
#include <iostream.h>                        //

#include "compcol_double.h"                  // Compressed column matrix header
#include "iohb_double.h"                     // Harwell-Boeing matrix I/O header
#include "mv_blas1_double.h"                 // MV_Vector level 1 BLAS
#include "diagpre_double.h"                  // Diagonal preconditioner

#include "qmr.h"                             // IML++ QMR template

int
main(int argc, char * argv[])
{
  if (argc < 2) {
    cerr << "Usage: " << argv[0] << " HBfile " << endl;
    exit(-1);
  }

  double tol = 1.e-6;                        // Convergence tolerance
  int result, maxit = 150;                   // Maximum iterations

  CompCol_Mat_double A;                      // Create a matrix
  readHB_mat(argv[1], &A);                   // Read matrix data
  VECTOR_double b, x(A.dim(1), 0.0);         // Create rhs, solution vectors
  readHB_rhs(argv[1], &b);                   // Read rhs data

  DiagPreconditioner_double D(A);            // Create diagonal preconditioner

  result = QMR(A, x, b, D, D, maxit, tol);   // Solve system

  cout << "QMR flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;

  return result;
}
```

See Also

SparseLib++
DiagPreconditioner

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

R. W. Freund and N. M. Nachtigal, *A quasi-minimal residual method for non-Hermition linear systems*, Numer. Math., 60 (91), pp. 315–339.

5 References

[1] S. Ashby, *CHEBYCODE: A Fortran implementation of Manteuffel's adaptive Chebyshev algorithm*, Tech. Report UIUCDCS-R-85-1203, University of Illinois, 1985.

[2] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

[3] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Numerical Analysis Dundee 1975, G. Watson, ed., Springer Verlag, Berlin, New York, 1976, pp. 73–89.

[4] R. W. Freund and N. M. Nachtigal, *A quasi-minimal residual method for non-Hermition linear systems*, Numer. Math., 60 (91), pp. 315–339.

[5] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The John Hopkins University Press, Baltimore, Maryland, 1983.

[6] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Res. Nat. Bur. Standards, 49 (1952), pp. 409–436.

[7] T. Manteuffel, *The Tchebychev iteration for nonsymmetric linear systems*, Numer. Math., 28 (1977), pp. 307–327.

[8] J. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.

[9] Y. Saad and M. Schultz, *GMRES: A generalized minimum residual algorithm for solving non-symmetric linear systems*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856–869.

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

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

[12] R. S. Varga, *Matrix Iterative Analysis*, Automatic Computation Series, Prentice-Hall Inc, Englewood Cliffs, New Jersey, 1962.

A Code Listings

To demonstrate the elegance and power of using C++ for scientific computing, we include the complete text of each iterative method function in the following appendix. Note that in most cases, each function is completely specified on a single page.

Name

bicg.h — BiConjugate Gradient Iteration template header file

Code

```
template < class Matrix, class Vector, class Preconditioner, class Real >
int BiCG(const Matrix &A, Vector &x, const Vector &b,
         const Preconditioner &M, int &max_iter, Real &tol)
{
  Real resid;
  Vector rho_1(1), rho_2(1), alpha(1), beta(1);
  Vector z, ztilde, p, ptilde, q, qtilde;
  Real normb = norm(b);
  Vector r = b - A * x;
  Vector rtilde = r;
  if (normb == 0.0)
    normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  for (int i = 1; i <= max_iter; i++) {
    z = M.solve(r);
    ztilde = M.trans_solve(rtilde);
    rho_1(0) = dot(z, rtilde);
    if (rho_1(0) == 0) {
      tol = norm(r) / normb;
      max_iter = i;
      return 2;
    }
    if (i == 1) {
      p = z;
      ptilde = ztilde;
    } else {
      beta(0) = rho_1(0) / rho_2(0);
      p = z + beta(0) * p;
      ptilde = ztilde + beta(0) * ptilde;
    }
    q = A * p;
    qtilde = A.trans_mult(ptilde);
    alpha(0) = rho_1(0) / dot(ptilde, q);
    x += alpha(0) * p;
    r -= alpha(0) * q;
    rtilde -= alpha(0) * qtilde;
    rho_2(0) = rho_1(0);
    if ((resid = norm(r) / normb) < tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }
  }
  tol = resid;
  return 1;
}
```

Name

bicgsta.h — BiConjugate Gradient Stabilized Iteration template header file

Code

```
template < class Matrix, class Vector, class Preconditioner, class Real >
int BiCGSTAB(const Matrix &A, Vector &x, const Vector &b,
             const Preconditioner &M, int &max_iter, Real &tol)
{
  Real resid;
  Vector rho_1(1), rho_2(1), alpha(1), beta(1), omega(1);
  Vector p, phat, s, shat, t, v;
  Real normb = norm(b);
  Vector r = b - A * x;
  Vector rtilde = r;
  if (normb == 0.0)
    normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  for (int i = 1; i <= max_iter; i++) {
    rho_1(0) = dot(rtilde, r);
    if (rho_1(0) == 0) {
      tol = norm(r) / normb;
      return 2;
    }
    if (i == 1)
      p = r;
    else {
      beta(0) = (rho_1(0)/rho_2(0)) * (alpha(0)/omega(0));
      p = r + beta(0) * (p - omega(0) * v);
    }
    phat = M.solve(p);
    v = A * phat;
    alpha(0) = rho_1(0) / dot(rtilde, v);
    s = r - alpha(0) * v;
    if ((resid = norm(s)/normb) < tol) {
      x += alpha(0) * phat;
      tol = resid;
      return 0;
    }
    shat = M.solve(s);
    t = A * shat;
    omega = dot(t,s) / dot(t,t);
    x += alpha(0) * phat + omega(0) * shat;
    r = s - omega(0) * t;
    rho_2(0) = rho_1(0);
    if ((resid = norm(r) / normb) < tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }
```

```
    if (omega(0) == 0) {
      tol = norm(r) / normb;
      return 3;
    }
  }
  tol = resid;
  return 1;
}
```

Name

cg.h — Conjugate Gradient Iteration template header file

Code

```
template < class Matrix, class Vector, class Preconditioner, class Real >
int CG(const Matrix &A, Vector &x, const Vector &b,
       const Preconditioner &M, int &max_iter, Real &tol)
{
  Real resid;
  Vector p, z, q;
  Vector alpha(1), beta(1), rho(1), rho_1(1);
  Real normb = norm(b);
  Vector r = b - A*x;
  if (normb == 0.0)
    normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  for (int i = 1; i <= max_iter; i++) {
    z = M.solve(r);
    rho(0) = dot(r, z);
    if (i == 1)
      p = z;
    else {
      beta(0) = rho(0) / rho_1(0);
      p = z + beta(0) * p;
    }
    q = A*p;
    alpha(0) = rho(0) / dot(p, q);
    x += alpha(0) * p;
    r -= alpha(0) * q;
    if ((resid = norm(r) / normb) <= tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }
    rho_1(0) = rho(0);
  }
  tol = resid;
  return 1;
}
```

Name

cgs.h — Conjugate Gradient Squared Iteration template header file

Code

```
template < class Matrix, class Vector, class Preconditioner, class Real >
int CGS(const Matrix &A, Vector &x, const Vector &b,
        const Preconditioner &M, int &max_iter, Real &tol)
{
  Real resid;
  Vector rho_1(1), rho_2(1), alpha(1), beta(1);
  Vector p, phat, q, qhat, vhat, u, uhat;
  Real normb = norm(b);
  Vector r = b - A*x;
  Vector rtilde = r;
  if (normb == 0.0)
    normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  for (int i = 1; i <= max_iter; i++) {
    rho_1(0) = dot(rtilde, r);
    if (rho_1(0) == 0) {
      tol = norm(r) / normb;
      return 2;
    }
    if (i == 1) {
      u = r;
      p = u;
    } else {
      beta(0) = rho_1(0) / rho_2(0);
      u = r + beta(0) * q;
      p = u + beta(0) * (q + beta(0) * p);
    }
    phat = M.solve(p);
    vhat = A*phat;
    alpha(0) = rho_1(0) / dot(rtilde, vhat);
    q = u - alpha(0) * vhat;
    uhat = M.solve(u + q);
    x += alpha(0) * uhat;
    qhat = A * uhat;
    r -= alpha(0) * qhat;
    rho_2(0) = rho_1(0);
    if ((resid = norm(r) / normb) < tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }
  }
  tol = resid;
  return 1;
}
```

Name

cheby.h — Chebyshev Iteration template header file

Code

```
template < class Matrix, class Vector, class Preconditioner, class Real,
           class Type >
int CHEBY(const Matrix &A, Vector &x, const Vector &b,
          const Preconditioner &M, int &max_iter, Real &tol,
          Type eigmin, Type eigmax)
{
  Real resid;
  Type alpha, beta, c, d;
  Vector p, q, z;
  Real normb = norm(b);
  Vector r = b - A * x;
  if (normb == 0.0)
    normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  c = (eigmax - eigmin) / 2.0;
  d = (eigmax + eigmin) / 2.0;
  for (int i = 1; i <= max_iter; i++) {
    z = M.solve(r);                    // apply preconditioner
    if (i == 1) {
      p = z;
      alpha = 2.0 / d;
    } else {
      beta = c * alpha / 2.0;          // calculate new beta
      beta = beta * beta;
      alpha = 1.0 / (d - beta);        // calculate new alpha
      p = z + beta * p;                // update search direction
    }
    q = A * p;
    x += alpha * p;                 // update approximation vector
    r -= alpha * q;                 // compute residual
    if ((resid = norm(r) / normb) <= tol) {
      tol = resid;
      max_iter = i;
      return 0;                        // convergence
    }
  }
  tol = resid;
  return 1;                            // no convergence
}
```

Name **gmres.h** — GMRES Iteration template header file

Code

```
#include <math.h>
template < class Operator, class Vector, class Preconditioner,
           class Matrix, class Real >
int GMRES(const Operator &A, Vector &x, const Vector &b,
          const Preconditioner &M, Matrix &H, int &m, int &max_iter,
          Real &tol)
{
  Real resid;
  int i, j = 1, k;
  Vector s(m+1), cs(m+1), sn(m+1), w;
  Real normb = norm(M.solve(b));
  Vector r = M.solve(b - A * x);
  Real beta = norm(r);
  if (normb == 0.0)
    normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  Vector *v = new Vector[m+1];
  while (j <= max_iter) {
    v[0] = r * (1.0 / beta);
    s = 0.0;
    s(0) = beta;
    for (i = 0; i < m && j <= max_iter; i++, j++) {
      w = M.solve(A * v[i]);
      for (k = 0; k <= i; k++) {
        H(k, i) = dot(w, v[k]);
        w -= H(k, i) * v[k];
      }
      H(i+1, i) = norm(w);
      v[i+1] = w * (1.0 / H(i+1, i));
      for (k = 0; k < i; k++)
        ApplyPlaneRotation(H(k,i), H(k+1,i), cs(k), sn(k));
      GeneratePlaneRotation(H(i,i), H(i+1,i), cs(i), sn(i));
      ApplyPlaneRotation(H(i,i), H(i+1,i), cs(i), sn(i));
      ApplyPlaneRotation(s(i), s(i+1), cs(i), sn(i));
      if ((resid = abs(s(i+1)) / normb) < tol) {
        Update(x, i, H, s, v);
        tol = resid;
        max_iter = j;
        delete [] v;
        return 0;
      }
    }
```

```
    Update(x, m - 1, H, s, v);
    r = M.solve(b - A * x);
    beta = norm(r);
    if ((resid = beta / normb) < tol) {
      tol = resid;
      max_iter = j;
      delete [] v;
      return 0;
    }
  }
  tol = resid;
  delete [] v;
  return 1;
}
template < class Matrix, class Vector >
void
Update(Vector &x, int k, Matrix &h, Vector &s, Vector v[])
{
  Vector y(s);

  for (int i = k; i >= 0; i--) {
    y(i) /= h(i,i);
    for (int j = i - 1; j >= 0; j--)
      y(j) -= h(j,i) * y(i);
  }

  for (int j = 0; j <= k; j++)
    x += v[j] * y(j);
}
template < class Real >
Real
abs(Real x)
{
  return (x > 0 ? x : -x);
}
template<class Real>
void GeneratePlaneRotation(Real &dx, Real &dy, Real &cs, Real &sn)
{
  if (dy == 0.0) {
    cs = 1.0;
    sn = 0.0;
  } else if (abs(dy) > abs(dx)) {
    Real temp = dx / dy;
    sn = 1.0 / sqrt( 1.0 + temp*temp );
    cs = temp * sn;
  } else {
    Real temp = dy / dx;
    cs = 1.0 / sqrt( 1.0 + temp*temp );
    sn = temp * cs;
  }
}
```

```
template<class Real>
void ApplyPlaneRotation(Real &dx, Real &dy, Real &cs, Real &sn)
{
  Real temp  =  cs * dx + sn * dy;
  dy = -sn * dx + cs * dy;
  dx = temp;
}
```

Name

ir.h — Richardson Iteration template header file

Code

```
template < class Matrix, class Vector, class Preconditioner, class Real >
int IR(const Matrix &A, Vector &x, const Vector &b,
       const Preconditioner &M, int &max_iter, Real &tol)
{
  Real resid;
  Vector z;
  Real normb = norm(b);
  Vector r = b - A*x;
  if (normb == 0.0)
    normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  for (int i = 1; i <= max_iter; i++) {
    z = M.solve(r);
    x += z;
    r = b - A * x;
    if ((resid = norm(r) / normb) <= tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }
  }
  tol = resid;
  return 1;
}
```

Name

qmr.h — Quasi-Minimal Residual Iteration template header file

Code

```
#include <math.h>
template < class Matrix, class Vector, class Preconditioner1,
           class Preconditioner2, class Real >
int
QMR(const Matrix &A, Vector &x, const Vector &b, const Preconditioner1 &M1,
    const Preconditioner2 &M2, int &max_iter, Real &tol)
{
  Real resid;
  Vector rho(1), rho_1(1), xi(1), gamma(1), gamma_1(1), theta(1), theta_1(1);
  Vector eta(1), delta(1), ep(1), beta(1);
  Vector r, v_tld, y, w_tld, z;
  Vector v, w, y_tld, z_tld;
  Vector p, q, p_tld, d, s;
  Real normb = norm(b);
  r = b - A * x;
  if (normb == 0.0)
    normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  v_tld = r;
  y = M1.solve(v_tld);
  rho(0) = norm(y);
  w_tld = r;
  z = M2.trans_solve(w_tld);
  xi(0) = norm(z);
  gamma(0) = 1.0;
  eta(0) = -1.0;
  theta(0) = 0.0;
  for (int i = 1; i <= max_iter; i++) {
    if (rho(0) == 0.0)
      return 2;                        // return on breakdown
    if (xi(0) == 0.0)
      return 7;                        // return on breakdown
    v = (1. / rho(0)) * v_tld;
    y = (1. / rho(0)) * y;
    w = (1. / xi(0)) * w_tld;
    z = (1. / xi(0)) * z;
    delta(0) = dot(z, y);
    if (delta(0) == 0.0)
      return 5;                        // return on breakdown
    y_tld = M2.solve(y);               // apply preconditioners
    z_tld = M1.trans_solve(z);
```

```
    if (i > 1) {
      p = y_tld - (xi(0) * delta(0) / ep(0)) * p;
      q = z_tld - (rho(0) * delta(0) / ep(0)) * q;
    } else {
      p = y_tld;
      q = z_tld;
    }
    p_tld = A * p;
    ep(0) = dot(q, p_tld);
    if (ep(0) == 0.0)
      return 6;                        // return on breakdown
    beta(0) = ep(0) / delta(0);
    if (beta(0) == 0.0)
      return 3;                        // return on breakdown
    v_tld = p_tld - beta(0) * v;
    y = M1.solve(v_tld);
    rho_1(0) = rho(0);
    rho(0) = norm(y);
    w_tld = A.trans_mult(q) - beta(0) * w;
    z = M2.trans_solve(w_tld);
    xi(0) = norm(z);
    gamma_1(0) = gamma(0);
    theta_1(0) = theta(0);
    theta(0) = rho(0) / (gamma_1(0) * beta(0));
    gamma(0) = 1.0 / sqrt(1.0 + theta(0) * theta(0));
    if (gamma(0) == 0.0)
      return 4;                        // return on breakdown
    eta(0) = -eta(0) * rho_1(0) * gamma(0) * gamma(0) /
      (beta(0) * gamma_1(0) * gamma_1(0));
    if (i > 1) {
      d = eta(0) * p + (theta_1(0) * theta_1(0) * gamma(0) * gamma(0)) * d;
      s = eta(0) * p_tld + (theta_1(0) * theta_1(0) * gamma(0) * gamma(0)) * s;
    } else {
      d = eta(0) * p;
      s = eta(0) * p_tld;
    }
    x += d;                         // update approximation vector
    r -= s;                         // compute residual
    if ((resid = norm(r) / normb) <= tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }
  }
  tol = resid;
  return 1;                            // no convergence
}
```