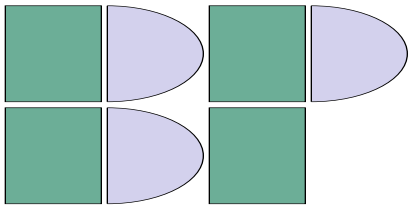


BPKIT

Block Preconditioning Toolkit

Reference Manual

Edmond Chow
Computer Science Department, and
Minnesota Supercomputer Institute
University of Minnesota

Michael A. Heroux
Scalable Computing and Algorithms Group
SGI/Cray Research, Inc.

September 20, 1996

This work was supported in part by the National Science Foundation under grant NSF/CCR-9214116 and in part by Cray Research, Inc. and the Minnesota Supercomputer Institute.

The authors wish to thank Kesheng Wu and Yousef Saad for their advice and support of this project.

BPKIT is available at `http://www.cs.umn.edu/~chow/bpkit.html`. Please send questions and comments to the authors at at `chow@cs.umn.edu` and `mamh@cray.com`.

Contents

Chapter 1

External Manual Pages

External manual pages give information that general users of BPKIT will require. Beginning users should first read the introductory manual page in this section.

Name

intro — introduction to the BPKIT toolkit for block preconditioning

Overview

BPKIT is a toolkit of block preconditioners for the iterative solution of linear systems. The most effective general purpose and black-box type of preconditioners are available, including block SSOR, block tridiagonal ILU, and the block extension of point ILU with level-of-fill. Any of these "global" preconditioners may be coupled with one of a large collection of "local" preconditioners for approximately or exactly inverting or solving with the diagonal or pivot blocks. These include a number of new approximate inverse techniques. By combining a global and local preconditioner, a wide variety of preconditionings are possible, matched to the difficulty and structure of the problem. For example, the following popular methods are possible:

- Block SSOR, using level-of-fill or threshold-based ILU to solve with the diagonal blocks
- A block version of level-based ILU using "exact" inverses for the pivot blocks
- ILU on a block tridiagonal system, using an approximate inverse technique suitable for the structure of the pivot blocks
- Matrices with no block structure may be treated as a single block, and a single local preconditioner may be used.

The blocks may be stored in dense or sparse formats, and user-defined data structures may also be used. Variable block sizes are allowed within a matrix. Operations with the blocks call the BLAS, LAPACK, or the sparse BLAS, for efficiency across many platforms. The selection of larger blocks usually gives higher performance. BPKIT supports the preconditioning of a block of vectors, to enhance efficiency when block iterative methods are used. A flexible GMRES iterative method is provided in BPKIT for users who do not have an iterative method readily available. A test program will read any linear system in Harwell-Boeing format and solve it using any combination of preconditioners and parameters specified by the user. This is important in the "experimentation" stage when trying to solve problems from new applications. BPKIT is callable from C/C++ and FORTRAN. BPKIT is written in standard C++ and FORTRAN, and runs on several types of workstations and Cray supercomputers. BPKIT is not parallel; for large applications it is suitable for the local solvers per processor/node or domain.

The most important feature of BPKIT is that it is user extensible, since an entirely black-box approach to high-performance preconditioning is currently not possible. Local and global preconditioners written in any language may be added. A few simple lines of C++ must be written in order to make the new "objects" polymorphic with the others (i.e., may be treated like other local and global preconditioners). User-defined data structures for the blocks and block matrices can also be added. The source code to BPKIT is freely provided.

In summary, BPKIT provides three functions:

1. a library of block preconditioners, callable from C/C++ or FORTRAN

2. a framework in which to add new block preconditioners
3. a testbed application program called `bp` for solving linear systems using the methods provided in the library.

Description

This is the introductory manual page for BPKIT. More specific information is given in other manual pages, which will be referenced here.

The testbed application program `bp` reads linear systems and solves them using the methods in the BPKIT library. `bp` is described under its own manual page, and may be used to test an installation of BPKIT.

BPKIT stores the matrix data in an object called a block matrix. Blocks in a block matrix may be dense or sparse. See the `BlockMat` manual page for information on how to create these objects from your data, and how to specify block partitionings.

A preconditioning for a block matrix is specified by choosing

1. a *global preconditioner* at the block matrix level (i.e., only blocks are manipulated), and
2. a *local preconditioner* to approximately or exactly invert or solve with a system of equations whose matrix is a block.

For example, to fully define the conventional block Jacobi preconditioning, one must specify the block Jacobi global preconditioner, and the LU factorization local preconditioner as a method of solving with the diagonal blocks. Sections 2 and 3 of this Reference Manual describe the global and local preconditioners, respectively.

In addition, the block size of the matrix has a role in determining the effect of the preconditioning. At one extreme, if a scalar block partitioning is used (i.e., scalar entries), then the preconditioning is entirely determined by the global preconditioner. At the other extreme, if the block matrix is treated as a single block (i.e., not partitioned), then the preconditioning is entirely determined by the local preconditioner. This has the effect of parameterizing the effect of two different methods and their cost.

BPKIT provides a flexible GMRES method for users who do not have an iterative method readily available (see `fgmres`). If a C++ iterative method is being used, BPKIT may be used very simply if certain functions are defined (see `Interface to iterative methods`). A FORTRAN interface is also provided (see `FORTRAN interface`).

Global Preconditioners

The following table lists the global preconditioners that are available, along with their arguments. For more information, see Section 2 of this Reference Manual.

	arguments
None	none
BJacobi	none
BSOR	omega, iterations
BSSOR	omega, iterations
BILUK	level
BTIF	none

Local Preconditioners

The following table lists the local preconditioners that are available, along with their arguments. For more information, see Section 3 of this Reference Manual.

	arguments	Block format	Inverse type
LP_LU	none	DENSE	implicit
LP_INVERSE	none	DENSE	explicit
LP_SVD	threshold	DENSE	explicit
LP_LU	none	CSR	implicit
LP_INVERSE	none	CSR	explicit
LP_RILUK	level, omega	CSR	implicit
LP_ILUT	lfil, threshold	CSR	implicit
LP_APINV_TRUNC	semibw	CSR	explicit
LP_APINV_BANDED	semibw	CSR	explicit
LP_APINVO	none	CSR	explicit
LP_APINVS	lfil, self-precon	CSR	explicit
LP_DIAG	none	CSR	explicit
LP_TRIDIAG	none	CSR	implicit
LP_SOR	omega, iterations	CSR	implicit
LP_SSOR	omega, iterations	CSR	implicit
LP_GMRES	restart, tolerance	CSR	implicit

Compatible Local Preconditioners

There are two compatibility rules for local preconditioners. First, the block format (DENSE or CSR) of the local preconditioner must match the block format of the block matrix. Second, the incomplete factorization global preconditioners BILUK and BTIF require the use of *explicit* preconditioners (since pivot blocks need to be explicitly inverted). The block format and inverse type are indicated in the Table above.

Notes

BPKIT was designed to be easily extensible. Users may add new block types and block matrices, and new local and global preconditioners. Please see the reference to the technical report below, and also Section 4 of this Reference Manual. Please also contact the authors for additional documentation and information.

Preconditioning is a challenging research area, and you may need to try several preconditioners and parameters before you find a combination that succeeds for

your toughest problems; please try some simple problems first to make sure that you have the software working correctly.

The choice of block size should be dependent on the block structure in the matrix and also performance considerations. Typically, larger block sizes give better performance. However, if dense blocks are used, blocks that are too large may force too many computations with zero matrix entries. The following Table gives an idea of the approximate block sizes that should be used on two different architectures.

Block Type	Sun	Cray
Dense	8	128
Sparse	16	16

Resources

Resources allow less-commonly used parameters to be transmitted to BPKIT at run-time. These parameters are stored in a resource file, whose default name is `.BpResource`. This file should reside in the directory from which you run applications compiled with BPKIT. Each line in the file has the following format:

```
parameter_name: value
```

An example `.BpResource` file is given in the `app` directory. Each manual page in this Reference Manual will describe any applicable resources. The `BpResource` manual page will be of interest if you wish to add additional resources to BPKIT.

Example

Several examples are given in the `app` subdirectory. The following example is available in the file `simple.cc`.

```
#include "BlockMat.h"
#include "BRelax.h"
#include "HBTMat.h"
#include "FGMRES.h"

void main()
{
    HBTMat H("SHERMAN1");      // read Harwell-Boeing file from disk
    double *rhs = H.get_rhs();
    double *x = H.get_guess();

    BlockMat B(H, 100, CSR);   // block matrix with sparse 100x100 blocks

    BSSOR M;
    M.localprecon(LP_INVERSE);
    M.setup(B, 1.0, 1);        // BSSOR(omega=1, iterations=1)

    fgmres f(20, 100, 1.e-8);  // Kry_dim=20, max_iter=100, tol=1.e-8
    f.solve(B, x, rhs, M);
```

```
    f.status(cout);            // solution should be reached in 24 steps
}
```

Specifying the preconditioning is only three lines of code above. A global preconditioner `M` is specified with a very simple form of declaration. In the case of block SSOR, the declaration is

```
BSSOR M;
```

Two functions are used to specify the local preconditioner and to provide parameters to the global preconditioner:

```
M.localprecon(LP_INVERSE);
M.setup(B, 0.5, 3);        // BSSOR(omega=0.5, iterations=3)
```

See the manual pages for clarification of the parameters.

The test program `b22_test.cc` also in the `app` subdirectory illustrates how a global preconditioner can be defined that uses different local preconditioners.

Examples for the use of the FORTRAN interface are given in the `FORTRAN interface` manual page.

Availability

The BPKIT software and its documentation are available at:
`http://www.cs.umn.edu/~chow/bpkit.html`.
It is written in standard C++ and FORTRAN 77 and should be very portable to your machine if you have these standard compilers. To link with the BPKIT library, you will also need FORTRAN versions of LAPACK and the BLAS, which you can get from Netlib (`http://www.netlib.org`). A current constraint on portability is that the C++ `int` type and the FORTRAN `integer` type are the same size on your machine, which is usually the case. BPKIT has been tested on Solaris, Sun4, IRIX, and UNICOS systems. Please send bug reports and comments to `chow@cs.umn.edu`.

Reference

E. Chow and M. A. Heroux. An Object-Oriented Framework for Block Preconditioning. Technical Report UMSI 95/216 (October 1995), Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1995. Submitted to *ACM Trans. Math. Softw.*

Authors

Edmond Chow, University of Minnesota, and Michael A. Heroux, Cray Research, Inc.

Name

bp — BPKIT application program for solving linear systems

Synopsis

`bp` *HBfile* {*bsize*|*bfile*} *global_precon* [*gparams*] *local_precon* [*lparams*]

Description

`bp` is a general application program for solving linear systems using the block preconditioners from BPKIT, and the FGMRES iterative method. Matrices are read in Harwell-Boeing format, and typically, the number of steps that were required in the iterative solution is printed. This application is useful for determining the best method for solving a class of linear systems. `bp` is also useful for testing an installation of BPKIT.

Command-line options are used to specify the preconditioner and its parameters. Like any BPKIT application, any relevant BPKIT resources are also used (see the `intro` manual page). The `bp` application is found in the `app` subdirectory.

Options

`bp` with no options will print a short usage description.

HBfile
: The name of the Harwell-Boeing file containing the matrix and possible right-hand sides, initial guesses, and solutions. See the `HBTMat` manual page for information about the default right-hand side and initial guess vectors.

bsize|*bfile*
: The block size or the file that describes the block partitioning of the block matrix. If *bsize* is used, all block rows and columns have dimension *bsize*, except for the last, which may be smaller if the matrix dimension is not a multiple of *bsize*. A value of 0 may be used for *bsize*, in which case a single block which consists of the entire matrix is used. If a file *bfile* is used, its name must not begin with a numeral. *bfile* is a file that contains the scalar row numbers (starting at zero) that are the beginnings of each block row. The same partitioning is used for the block columns.

global_precon
: The name of the global preconditioner. See the `intro` manual page for the possible names. See Section 2 of this Reference Manual for a description of the global preconditioners. The names are case-sensitive.

gparams
: The parameters for the global preconditioner, if any. See the `intro` manual page for the parameters for each global preconditioner. See Section 2 of this Reference Manual for a description of the parameters.

local_precon
: The name of the local preconditioner. The possible names are given on the `intro` manual page, less the `LP_` prefix. See Section 3 of this Reference Manual for a description of the local preconditioners. The names are case-sensitive. The choice of local preconditioner must match the *compatible local preconditioner* rules described in the `intro` manual page.

lparams
: The parameters for the local preconditioner, if any. See the `intro` manual

page for the parameters for each local preconditioner. See Section 3 of this Reference Manual for a description of the parameters.

Resources

The following resources are specific to the `bp` application.

Resource	Default	Values
`bp.dense_size`	16	positive integer

The `bp.dense_size` resource determines whether dense or sparse storage will be used for the blocks in the block matrix. If the size of the first block has dimension not exceeding `bp.dense_size`, then dense storage will be used for the blocks, otherwise sparse storage will be used.

Example

The `app` subdirectory contains a script file called `test_script` which can be used to run a set of tests on the SHERMAN1 matrix, also in the `app` directory. The output can be compared to the file `test_output`. The script contains:

```
bp SHERMAN1 100 BJacobi INVERSE
bp SHERMAN1 100 BJacobi RILUK 0 0.0
bp SHERMAN1 100 BJacobi RILUK 1 0.0
bp SHERMAN1 100 BJacobi ILUT 2 0.0
bp SHERMAN1 100 BJacobi TRIDIAG
bp SHERMAN1 100 BJacobi SOR 1.0 1
bp SHERMAN1 100 BJacobi SSOR 1.0 1
bp SHERMAN1 100 BJacobi GMRES 5 0.1
bp SHERMAN1 100 BJacobi APINVS 5 1
bp SHERMAN1 100 BJacobi APINV0
bp SHERMAN1 100 BTIF INVERSE
bp SHERMAN1 100 BTIF DIAG
bp SHERMAN1 100 BTIF APINV0
bp SHERMAN1 100 BTIF APINVS 5 1
bp SHERMAN1 100 BTIF APINVS 10 1
```

If the default `.BpResource` file is used, the output should be:

```
  88      9.520099361807e-09
  93      9.281488255279e-09
  89      7.321574925991e-09
  85      8.060649575905e-09
 138      9.113356944537e-09
 600      1.057391830467e-06
 508      9.983861655495e-09
 173      9.027697796094e-09
 108      9.069159059615e-09
 171      9.971480010542e-09
   1      1.664569299480e-14
```

```
600      3.246216146858e-05
123      8.825130967892e-09
 64      7.576869182186e-09
 33      9.855480868523e-09
```

which shows the number of FGMRES steps and the final relative residual norm for each test.

Name

BlockMat — Block matrix

Synopsis

```
#include "BlockMat.h"

enum BlockType {
    DENSE,
    CSR
};

 BlockMat(const HBTMat&, int, BlockType);
 BlockMat(const HBTMat&, int, const int *, BlockType);
~BlockMat();

      LocalMat& val(unsigned int i);
const LocalMat& val(unsigned int i) const;
const int& row_ptr(unsigned int i) const;
const int& col_ind(unsigned int i) const;

int numrow() const;
int numcol() const;
int numnz()  const;

int dimrow() const;
int dimcol() const;

const int& kvst_row(unsigned int i) const;
const int& kvst_col(unsigned int i) const;

void mult(int nr, int nc, const double *u, int ldu,
    double *v, int ldv) const;
void trans_mult(int nr, int nc, const double *u, int ldu,
    double *v, int ldv) const;

ostream& operator << (ostream& os, const BlockMat& mat);
```

Description

`BlockMat` is the block-partitioned matrix class used with the global preconditioners provided by BPKIT. The blocks in `BlockMat` can be dense or sparse, determined by the `BlockType`.

`BlockMat` objects are created from Harwell-Boeing matrices (`HBTMat` objects) and a block partitioning of the matrix. Currently, two constructors are provided.

```
BlockMat(H, block_size, BlockType);
BlockMat(H, num_block_rows, partitioning, BlockType);
```

The first constructor creates a block matrix with block rows and block columns of size `block_size`, with blocks of type `BlockType`. If the order of the matrix is not a

multiple of `block_size`, then the last block row and column will be smaller. If the order of the matrix is a multiple of `block_size`, then all blocks in the block matrix will be of size `block_size` × `block_size`.

The second constructor allows a variable block partitioning to be used. `num_block_rows` is the number of block rows, and `partitioning` is a vector of scalar row numbers (starting at zero) that are the beginnings of each block row. The same partitioning is used for the block columns. `partitioning` has length `num_block_rows` + 1, its first entry is zero, and its last entry is the scalar order of the matrix. For example, `partitioning = [0, 5, 10]` specifies a 2 by 2 block matrix, with each block being size 5 by 5.

Note that `BlockMat` does not force the block row and column partitionings to be the same, only the constructor interfaces do, since only this case is useful in BPKIT. Additional interfaces may be added in the future, if necessary.

Access Functions

The block structure of a `BlockMat` matrix is stored using the compressed sparse row format (i.e., block row pointers, block column indices, pointers to `LocalMat` objects). `LocalMat` is the superclass of all block types.

The functions `row_ptr`, `col_ind` and `val` give access to the block row pointers, block column indices, and pointers to the blocks, respectively. `numrow`, `numcol` and `numnz` return the *block* row and column dimension, and the number of block nonzeros in the block matrix, respectively. `dimrow` and `dimcol` return the *scalar* row and column dimensions of the matrix, respectively.

Advanced Notes

Note that BPKIT is designed so that the user may write their own block-partitioned matrix classes. Examples may be a *symmetric* block matrix where only the lower triangular blocks are stored, or a 2 by 2 block matrix, or any block matrix whose block structure is stored with an alternative data structure. Any `LocalMat` block type and any local preconditioner may be used with the new block matrix. Global preconditioners for the new block matrix may be written with nearly the simplicity as if the block matrix were a matrix with scalar entries. (The implementation of global preconditioners must depend on the block matrix data structure, and BPKIT was designed to make writing these preconditioners easy for any block matrix).

To ensure that any block matrix that is defined conforms to the specified interface with iterative methods, block matrices are derived from the `BpMatrix` class (see the `Interface to iterative methods` manual page). This class requires that four functions be defined:

```
int dimrow() const;
int dimcol() const;
void mult(int nr, int nc, const double *u, int ldu,
    double *v, int ldv) const;
void trans_mult(int nr, int nc, const double *u, int ldu,
    double *v, int ldv) const;
```

`dimrow` and `dimcol` return the scalar row and column dimensions of the matrix, respectively. `mult` and `trans_mult` are the `BlockMat` matrix-vector product and transposed matrix-vector product operations. The interface is generalized to operate on and return a block of vectors. `nr` and `nc` are the scalar row and column dimensions of the input block of vectors, `u` and `v` are the input and output blocks of vectors, respectively, stored in FORTRAN column-major order, and `ldu` and `ldv` are the leading dimensions of these arrays, respectively. This primitive variable interface is used so that no particular vector class is required, and that it may be used easily from FORTRAN.

Debugging

A block structure of a block matrix may be printed to an output stream with the `<<` operator, as in `os << B;`, where `os` is the output stream. The output uses 1-based indexing.

Name

HBTMat — Harwell-Boeing Transposed matrix

Synopsis

```
#include "HBTMat.h"

 HBTMat(const char *filename);
 HBTMat(int n, double *a, int *ja, int *ia,
     double *rhs, double *guess, double *exact);
~HBTMat();
 FreeMat();

const double&  val(int i) const;
const int&     col_ind(int i) const;
const int&     row_ptr(int i) const;

int            dimrow() const;
int            dimcol() const;
int            numnz() const;

double        *get_rhs() const;
double        *get_guess() const;
double        *get_exact() const;
```

Description

The `HBTMat` object is the intermediary between Harwell-Boeing matrices and `BlockMat` block matrix objects. As such, it provides functionality for reading matrices in the Harwell-Boeing file format, or from Harwell-Boeing `COLPTR, ROWIND, VALUES` storage arrays. The Harwell-Boeing format optionally includes sets of vectors which are the right-hand sides, exact solutions, and initial guesses. `HBTMat` also reads these vectors if they are provided.

Two constructors are available, for the two methods that `HBTMat` accepts its data. The first constructor `HBTMat(filename)` reads the named file as a Harwell-Boeing file. Currently, only `RUA` (real, unsymmetric, assembled) matrices can be read from disk. Only the first set of vectors in the Harwell-Boeing file is returned. If the initial guess and exact solution vectors are not provided, they are set to zero. If the right-hand side is not provided, the behavior is dependent on the `HBTMat.rhs` resource (see below).

Since the matrix may require large amounts of storage, it is often convenient to release the storage associated with the matrix, but not the vectors. This is accomplished with the `FreeMat` function. Any unreleased memory, including for the vectors, is freed automatically by the destructor.

If the Harwell-Boeing storage arrays are generated, an `HBTMat` object may be created directly with the second constructor

```
HBTMat(n, val, col_ind, row_ptr, rhs, guess, exact)
```

where `n` is the number of scalar rows in the matrix, `val` is the array of numerical values, `col_ind` is the array of column indices, `row_ptr` is the array of row pointers,

and `rhs, guess` and `exact` are the arrays of right-hand sides, initial guesses, and exact solutions. Any of the last three arrays may be replaced by `NULL` pointers. `FreeMat` has no effect when this constructor is used, and the destructor does not free any arrays.

Functions are used to access the matrix and vector data. `dimrow, dimcol, numnz` return the row and column dimension and the number of nonzeros in the matrix, respectively. `val, col_ind, row_ptr` give access to the storage arrays of the matrix, and `get_rhs, get_guess, get_exact` return pointers to the vectors.

`HBTMat` actually stores the matrix in compressed sparse row (CSR) rather than compressed sparse column (CSC) format, and thus we say the matrix is transposed. The raw data that is read or provided through arrays is transposed by default, unless the `HBTMat.transpose` resource is set to 0. This latter setting is useful if matrices are actually stored or generated by rows rather than by columns. Note that this means the user data provided by the second constructor may be altered.

Harwell-Boeing matrix files are 1-based (indexing begins at 1), and are converted by the first constructor to 0-based internally. The second constructor assumes that matrices are 0-based.

Notes

The `BlockMat` object and several of the preconditioners assume that their input matrices have row or column indices that are sorted. This is usually, but not always the case for Harwell-Boeing matrices. Note that if the matrix is transposed, then a matrix with sorted indices will be created.

Early versions of `BlockMat` included functionality to read Harwell-Boeing files directly. However, since right-hand side vectors should be read at the same time, we decided to separate out this functionality into `HBTMat`. This separation also makes some kinds of intermediate processing more convenient, such as row and column scalings

The reading of Harwell-Boeing files is implemented in FORTRAN.

Resources

This object's resources are given in the following Table.

Resource	Default	Values
`HBTMat.rhs`	0	0 or 1
`HBTMat.transpose`	1	0 or 1

The value 0 for `HBTMat.rhs` specifies that a random vector (elements uniformly distributed in [-1,1]) should be generated *only if* no right-hand side is provided in the Harwell-Boeing file. A value 1 specifies that a vector of all ones should be generated instead.

The values 0 and 1 for `HBTMat.transpose` specify respectively whether or not the raw data should be transposed immediately.

Example

This example reads the matrix SHERMAN1 in the current directory, and prints the number of rows in the matrix.

```
HBTMat H("SHERMAN1");
cout << H.get_dimrow();
```

References

I. S. Duff, R. G. Grimes and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.,* 15 (1989), pp. 1–14.

I. S. Duff, R. G. Grimes and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection. TR/PA/92/86, CERFACS, Toulouse, 1992.

Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

Name

BpMatrix, BpPrecon — BPKIT interface to iterative methods

Synopsis

```
#include "BpMatrix.h"

int dimrow() const;
int dimcol() const;

void mult(int nr, int nc, const double *u, int ldu,
    double *v, int ldv) const;
void trans_mult(int nr, int nc, const double *u, int ldu,
    double *v, int ldv) const;

#include "BpPrecon.h"

void apply  (int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applyt (int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applyr (int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applyrt(int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applyl (int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applylt(int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applyc (int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applyct(int nr, int nc, const double *u, int ldu, double *v, int ldv);
```

Description

A preconditioned iterative method requires only a small number of well-defined operations from matrix and preconditioner objects. An interface to these operations are defined in the `BpMatrix` and `BpPrecon` public abstract classes and are declared pure virtual; block matrices and global preconditioners should be derived from these classes. These operations are listed in the following table. Two functions not listed in the table are matrix member functions that return the scalar row and column dimension of the matrix, which are useful for the iterative method code to help preallocate any workspace that is needed.

Matrix operations	
`mult`	matrix-vector product
`trans_mult`	transposed matrix-vector product
Preconditioner operations	
`apply`	apply preconditioner
`applyt`	apply transposed preconditioner
`applyl`	apply left part of a split preconditioner
`applylt`	above, transposed
`applyr`	apply right part of a split preconditioner
`applyrt`	above, transposed
`applyc`	apply a combined matrix-preconditioner operator
`applyct`	above, transposed

The `apply` and `applyt` operations may be used for left or right preconditionings. Split (two-sided, or symmetric) preconditionings use `applyl` and `applyr` to apply the left and right parts of the split preconditioner, respectively. To anticipate all possible functionality, the `applyc` operation defines a combined matrix-preconditioner operator to be used, for example, to implement the Eisenstat trick.

The argument lists use primitive data types so that iterative methods codes are not forced to adopt any particular "Vector" class. Block iterative methods are accommodated since the interfaces use blocks of vectors. The implementation of these operations use level 3 BLAS whenever possible. All the interfaces have the following form:

```
void mult(int nr, int nc, const double *u, int ldu,
  double* v, int ldv) const;
```

where `nr` and `nc` are the scalar row and column dimensions of the (input) blocks of vectors, `u` and `v` are arrays containing the values of the input and output vectors, respectively, and `ldu` and `ldv` are the leading dimensions of these respective arrays. Non-unit stride is not permitted; this is not likely to be very useful. The preconditioner operations are not defined as `const` functions, however, in case the preconditioner objects need to change their state as the iterations progress (and spectral information is revealed, for example).

Name

bpinitialize, blockmatrix, free_blockmatrix, preconditioner, free_preconditioner, flexgmres, matvec, apply — FORTRAN interface to BPKIT

Synopsis

```
include 'bpfort.h'

! block types
integer BP_DENSE, BP_SPARSE

! global preconditioners
integer BP_NONE, BP_BJACOBI, BP_BSOR, BP_BSSOR, BP_BILUK, BP_BTIF

! local preconditioners
integer BP_LU, BP_INVERSE, BP_SVD, BP_RILUK, BP_ILUT
integer BP_APINV_TRUNC, BP_APINV_BANDED, BP_APINVO, BP_APINVS
integer BP_DIAG, BP_TRIDIAG, BP_SOR, BP_SSOR, BP_GMRES

subroutine bpinitialize()

subroutine blockmatrix(bmat, n, a, ja, ia, nb, kvst, type)
integer bmat, n, ja(*), ia(*), nb, kvst(*), type
real*8  a(*)

subroutine free_blockmatrix(bmat)
integer bmat

subroutine preconditioner(precon, bmat, global, gparam1, gparam2,
   local, lparam1, lparam2)
integer precon, bmat, global, local
real*8  gparam1, gparam2, lparam1, lparam2

subroutine free_preconditioner(precon)
integer precon

subroutine flexgmres(bmat, x, rhs, precon, dim, max_iter, tol)
integer bmat, precon, dim, max_iter
real*8  x(*), rhs(*), tol

subroutine matvec(bmat, nr, nc, u, ldu, v, ldv)
integer bmat, nr, nc, ldu, ldv
real*8  u(*), v(*)

subroutine apply(prec, nr, nc, u, ldu, v, ldv)
integer prec, nr, nc, ldu, ldv
real*8  u(*), v(*)
```

Description

BPKIT provides an object-oriented type of FORTRAN interface, making its functionality accessible to FORTRAN users. (Currently, this interface is also the gateway for C users, until a true C interface is written.) Objects can be created, and

pointers to them are passed through functions as FORTRAN (or C) integers. Consider the following FORTRAN code excerpt.

```
call blockmatrix(bmat, n, a, ja, ia, num_block_rows, partit, BP_DENSE)
call preconditioner(precon, bmat, BP_BJACOBI, 0.d0, 0.d0, BP_LU, 0.d0, 0.d0)
call flexgmres(bmat, sol, rhs, precon, 20, 600, 1.d-8)
```

The call to `blockmatrix` above creates a block matrix with dense blocks from the compressed sparse row data structure, given a number of arguments. This 'wrapper' function is actually written in C++, but all its arguments are available to a FORTRAN program. The integer `bmat` is actually a pointer to a block matrix object in C++. The FORTRAN program is not meant to interpret this variable, but to pass it to other functions, such as `preconditioner` which defines a block preconditioner with a number of arguments, or `flexgmres` which solves a linear system using flexible GMRES. Similarly `precon` is a pointer to a preconditioner object. The constant parameters `BP_BJACOBI` and `BP_LU` are used to specify a block Jacobi preconditioner, using LU factorization to solve with the diagonal blocks.

The matrix-vector product and preconditioner operations also have 'wrapper' functions. This makes it possible to use BPKIT from an iterative solver written in FORTRAN.

BPKIT performs its own memory management. `bpinitialize` sets the memory allocation error handler, and reads the resource file (see `BpResource`). It should be called before any other FORTRAN subroutine in BPKIT.

`blockmatrix(bmat, n, a, ja, ia, nb, kvst, type)` creates a block matrix object represented by the integer `bmat` using a matrix in compressed sparse row (CSR) format. The arrays `a, ja, ia` are the input numerical values for a matrix of dimension `n`. `kvst` is a block partitioning vector. All indexing starts at 1, as is common in FORTRAN, and will be converted to 0-based. The block type `type` should be either `BP_DENSE` or `BP_CSR`.

`free_blockmatrix(bmat)` releases the storage associated with the block matrix `bmat`. It is the equivalent of the C++ destructor for this object.

`preconditioner(precon, bmat, global, gparam1, gparam2, local, lparam1, lparam2)` creates a preconditioner object represented by the integer `precon` from the block matrix `bmat`. `global` identifies the global preconditioner, and should be one of `BP_NONE, BP_BJACOBI, BP_BSOR, BP_BSSOR, BP_BILUK, BP_BTIF`. Two parameters `gparam1, gparam2` are allowed. See the `intro` manual page and Section 2 of this Reference Manual for a description of these parameters. If only one parameter is used, then the second is ignored. Note that for flexibility, both parameters are double precision values (i.e., integers should be passed as double precision variables).

Similarly, `local` identifies the local preconditioner, and should be one of `BP_LU, BP_INVERSE, BP_SVD, BP_RILUK, BP_ILUT, BP_APINV_TRUNC, BP_APINV_BANDED, BP_APINV0, BP_APINVS BP_DIAG, BP_TRIDIAG, BP_SOR, BP_SSOR, BP_GMRES`. See

the `intro` manual page and Section 3 of this Reference Manual for a description of these local preconditioners and their parameters. The `BP_` constants are defined in `bpfort.h` which should be accessible when compiling the FORTRAN program.

`free_preconditioner(precon)` releases the storage associated with the preconditioner `precon`. It is the equivalent of the C++ destructor for this object.

`flexgmres(bmat, x, rhs, precon, dim, max_iter, tol)` calls the flexible GMRES routine, for the linear system `bmat x = rhs` using the preconditioner `precon`. `max_iter` maximum iterations will be used to reduce the residual norm to `tol` times the original. The dimension of the Krylov subspace used is `dim`.

`matvec(bmat, nr, nc, u, ldu, v, ldv)` is the matrix-vector product for the block matrix `bmat` and `apply(prec, nr, nc, u, ldu, v, ldv)` is the preconditioner operation for the preconditioner `precon`. `nr` and `nc` are the scalar row and column dimensions of the input block of vectors, `u` and `v` are the input and output blocks of vectors, respectively, stored in FORTRAN column-major order, and `ldu` and `ldv` are the leading dimensions of these arrays, respectively.

Examples

Two examples on using the FORTRAN interface are given in the `app` subdirectory. A third example of using this interface with a C program is also given.

The following FORTRAN excerpt solves a linear system with a matrix stored in compressed sparse row format (`a, ja, ia` format). This example is available in the file `f1example.cc`.

```
integer   n, nnz
integer   ia(n+1), ja(nnz), kvst(n+1)
real*8    a(nnz), rhs(n), sol(n)
integer   bmat, precon

call bpinitialize()
call blockmatrix(bmat, n, a, ja, ia, 10, kvst, BP_DENSE)
call preconditioner(precon, bmat, BP_BJACOBI, 0.d0, 0.d0,
         BP_LU, 0.d0, 0.d0)
call flexgmres(bmat, sol, rhs, precon, 20, 600, 1.d-8)
```

This next example demonstrates how BPKIT block matrices and preconditioners may be used with an iterative solver written in FORTRAN. In this example the iterative solver has a reverse communicate interface, so that the calls to the matrix-vector product and preconditioning operations are well-separated. It is assumed that the block matrix `bmat` and block preconditioner `precon` have already been created using the `blockmatrix` and `preconditioner` subroutines. This example is available in the file `f2example.cc`.

```
 10   call user_iterative_method(n,rhs,sol,ipar,fpar,wk)
      if (ipar(1).eq.1) then
         call matvec(bmat, n, 1, wk(ipar(8)), n, wk(ipar(9)), n)
```

```
      goto 10
   else if (ipar(1).eq.5) then
      call apply(precon, n, 1, wk(ipar(8)), n, wk(ipar(9)), n)
      goto 10
   endif
```

An example of the use of this interface from a C language program is given in the file `c_example.c`.

Name

fgmres — flexible GMRES for solving linear systems

Synopsis

```
#include "fgmres.h"

fgmres();
fgmres(int dim, int max_iter, double tol);
~fgmres();

void   solve(const BpMatrix &A, double *x, const double *b,
           BpPrecon &M);

int    get_iter();
double get_rel_resid_norm();

void   status(ostream& os);
```

Description

`fgmres` is an object that solves linear systems using a flexible GMRES iterative method. `fgmres` is provided as a convenience in BPKIT for users who do not have an iterative solver readily available. It is likely to be sufficient if no particular iterative method is required.

FGMRES is a 'flexible' version of GMRES, meaning that it accommodates the situation where the preconditioning operation is not a constant operator. This arises in BPKIT, for example, when a relaxation method is used with more than one step.

There are two constructors. `fgmres()` with no arguments reads its parameters from the resource file, or uses default values if any resources are not set. `fgmres(dim, max_iter, tol)` creates an fgmres object that will attempt to solve a linear system using a Krylov subspace dimension of `dim`, `max_iter` iterations, and to a *relative* residual norm reduction of `tol`. The default values are 20, 300, and 1.e-6, respectively. (The *relative* residual norm reduction is the current residual norm divided by the initial residual norm.) The object resources are read when the object is created. Resources for parameters that are explicitly set (the second constructor above) are not read.

The method `solve(A, x, b, M)` is the work-horse of the class. This solves the linear system `A x = b` with a preconditioner `M`, where `A` is a matrix derived from the `BpMatrix` class, and `M` is a preconditioner derived from the `BpPrecon` class. An initial guess `x` may be provided. The solution `x` is returned.

The access functions `get_iter()` and `get_rel_resid_norm()` may be called at any time after `solve` has been called to return the number of iterations that have been consumed, and the relative residual norm reduction that has been achieved. In particular, preconditioner objects may use these functions to determine the progress of the iterative method. Functions to return information about the spectrum of the iteration matrix are planned, but not yet implemented.

The information function `status(os)` prints some information about the `fgmres` object to the `os` output stream. Currently, the number of iterations and the relative residual norm reduction is printed.

Implementation Notes

The `fgmres` object is implemented with the most common options. The Krylov basis is constructed with the Arnoldi Modified Gram-Schmidt process with *no* reorthogonalization. Restarting is used after a number of steps specified by the user. Right-preconditioning is used. Flexible GMRES is only available with right-preconditioning; this preconditioning option also directly minimizes the actual residual norm, rather than a preconditioned one. This is a single vector version of FGMRES. (BPKIT, however, supports block iterative methods in its interface, and preconditions blocks of vectors efficiently.)

`fgmres` is implemented as a C++ class, primarily so that it may interact with preconditioner objects. These objects may request convergence and spectral information from fgmres and change their behavior accordingly. These features allow linear systems to be solved in a more robust fashion.

The implementation uses the iterative methods interface required by BPKIT (see the `Interface to iterative methods` manual page). In particular, the polymorphic `mult` and `apply` functions are called, invoking the correct instance of the matrix-vector product and preconditioning operations, respectively.

Stopping Criteria

The stopping criterion is based on the residual norm estimate within the FGMRES algorithm. The algorithm is stopped when this residual norm is reduced to below the tolerance `tol` times the initial residual norm, or when the maximum number of iterations `max_it` has been reached. There is no absolute tolerance.

Resources

The resource file `.BpResource` is read when the `fgmres` object is created, allowing certain parameters to be changed without recompilation.

Resource	Default	Values
`fgmres.dim`	20	positive integer
`fgmres.max_iter`	300	positive integer
`fgmres.tol`	1.e-6	positive real
`fgmres.history`	0	0=don't print, 1=print

The first three resources in the above table control the Krylov subspace dimension, maximum number of iterations, and relative residual norm tolerance for the `fgmres` object. The `fgmres.history` resource controls whether or not printing of the relative residual norm estimate should occur at each iteration.

Example

The following example creates an `fgmres` object `f`, solves a linear system, and prints the final number of iterations and the residual norm reduction to the standard output stream.

```
fgmres f(50, 500, 1.e-8);
f.solve(A, x, b, M);
f.status(cout);
```

Reference

Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.,* 14 (1993), pp. 461–469.

Chapter 2

Global Preconditioners

A summary of the global preconditioners that are currently available is given in the `intro` manual page.

The global preconditioners all have a `localprecon` function to set the local preconditioner to be used, and a set of `apply` functions that execute the preconditioning operation. In addition, a `setup` function performs the work of setting up the global preconditioner.

`localprecon` must be called before `setup`. `setup` can usually be called many times, with different parameters. `localprecon` cannot be called again, however.

The `localprecon` function has no set prototype. Its arguments depend on the local preconditioner chosen. See Section 3 of this Reference manual for the arguments for each local preconditioner.

Usually only the `apply` function has been implemented, rather all variants `applyt`, `applyr`, etc.

Name

None — identity global preconditioner

Synopsis

```
#include "BRelax.h"

 None();
~None();

void apply(int nr, int nc, const double *u, int ldu, double *v, int ldv);
```

Description

`None` is the identity preconditioner, equivalent no preconditioning. The preconditioning operation `apply` simply copies its input to its output. This object is provided in case a preconditioner must be specified. In many iterative solvers, it is possible to turn off preconditioning so that no preconditioner needs to be specified, and no `apply` function ever needs to be called.

Note that `None` is defined in the `BRelax.h` header file, for lack of a better place.

Name

BJacobi — Block Jacobi global preconditioner

Synopsis

```
#include "BRelax.h"

 BJacobi();
~BJacobi();

void localprecon(LocalPreconName, ...);
void setup(const BlockMat& A);
void apply(int nr, int nc, const double *u, int ldu, double *v, int ldv);
```

Description

`BJacobi` is a Block Jacobi global preconditioner. Only one step of the method is taken, so *block diagonal preconditioner* is perhaps a better name.

Name

BSOR — Block Successive Over Relaxation global preconditioner

Synopsis

```
#include "BRelax.h"

 BSOR();
~BSOR();

double& omega();
int& iterations();

void localprecon(LocalPreconName, ...);
void setup(const BlockMat& A, double omega = 1.0, int iterations = 1);
void apply(int nr, int nc, const double *u, int ldu, double *v, int ldv);
```

Description

`BSOR` is a Block Successive Over Relaxation global preconditioner. The relaxation factor `omega` is used, and `iterations` steps are taken. The default values for both these parameters are 1, as shown in the synopsis above.

Two additional functions, `omega` and `iterations`, allow these parameters to be returned or changed during the linear iterations.

Name

BSSOR — Block Symmetric Successive Over Relaxation global preconditioner

Synopsis

```
#include "BRelax.h"

 BSSOR();
~BSSOR();

double& omega();
int& iterations();

void localprecon(LocalPreconName, ...);
void setup(const BlockMat& A, double omega = 1.0, int iterations = 1);
void apply(int nr, int nc, const double *u, int ldu, double *v, int ldv);
```

Description

`BSSOR` is a Block Symmetric Successive Over Relaxation global preconditioner. The relaxation factor `omega` is used, and `iterations` steps are taken. The default values for both these parameters are 1, as shown in the synopsis above.

Two additional functions, `omega` and `iterations`, allow these parameters to be returned or changed during the linear iterations.

Name

BILUK — Block Incomplete LU factorization global preconditioner, with level-of-fill

Synopsis

```
#include "BILUK.h"

 BILUK();
~BILUK();

void localprecon(LocalPreconName, ...);
void setup(const BlockMat& A, int level);
void apply (int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applyr(int nr, int nc, const double *u, int ldu, double *v, int ldv);
void applyl(int nr, int nc, const double *u, int ldu, double *v, int ldv);

void multiply(int nr, int nc, const double *u, int ldu, double *v, int ldv);
```

Description

`BILUK` is a Block incomplete factorization global preconditioner based on level-of-fill `level`. It is the block extension of point ILU. A block unit lower triangular matrix L and block upper triangular matrix U are produced. Typically, values of level-of-fill up to 3 are practical. If `level` is less than zero, then the pattern of the pre-existing factorization is used (it assumes `setup`) has been called at least once before.

Fill-in is always allowed onto the diagonal block, which differs from most implementations.

The `multiply` function is an additional function that multiplies a vector by the preconditioner.

Explicit local inverses must be used with this global preconditioner, since pivot blocks need to be inverted. Note that BILUK can become very expensive if large sparse blocks are used and the factorization becomes denser and denser as it progresses.

Resources

The `BILUK.growth` resource controls the estimate for the storage required for the symbolic incomplete factorization. Let nnz be the number of block nonzeros in the lower triangular part of A. Then the nonzero storage provided for L is

$$(\texttt{level} + \texttt{growth}) \times nnz.$$

A default value of 2 is used for `growth`. If `level` $= 0$, the storage provided is simply nnz; if `level` $= n$, the order of the matrix, the storage is $n(n+1)/2$. The same estimate is used for U, in which case nnz is the number of nonzeros in the upper triangular part of A. If you get a failure message due to insufficient space for the factorization, then this parameter should be increased.

Resource	Default	Values
`BILUK.growth`	2	positive integer

Name

BTIF — Block Tridiagonal Incomplete factorization global preconditioner

Synopsis

```
#include "BTIF.h"

 BTIF();
~BTIF();

void localprecon(LocalPreconName, ...);
void setup(const BlockMat& A);
void apply(int nr, int nc, const double *u, int ldu, double *v, int ldv);
```

Description

`BTIF` is a Block tridiagonal incomplete factorization global preconditioner. The effect is exactly the same as `BILUK` for block tridiagonal matrices, but the algorithm is optimized for this type of matrix (only a series of block diagonal elements need to be computed). Explicit local inverses must be used with this global preconditioner, since pivot blocks need to be inverted.

The code for this preconditioner is very short, and is a good example of how to add a global preconditioner to BPKIT.

Chapter 3

Local Preconditioners

A global preconditioner requires the inverse or solves with diagonal or pivot blocks. Local preconditioners approximate these blocks or their inverses for this purpose. This Section describes each local preconditioner and its use.

The synopsis of each local preconditioner includes a table that summarizes its compatibility with block types and global preconditioners The block type (DENSE or CSR) must match the type of blocks in the block matrix, and explicit local preconditioners must be used with incomplete factorization global preconditioners. The Table also gives the FORTRAN name that is used in the FORTRAN interface (see `FORTRAN interface`).

Most of the implicit local preconditioners have not yet been coded so that they handle blocks of vectors. Thus if these implicit local preconditioners are used, then the global preconditioners will also not be able to handle blocks of vectors.

Name

LP_LU — LU factorization of a local matrix

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_LU);
```

Block formats	Inverse type	Fortran name
DENSE and CSR	Implicit	LU

Description

`LP_LU` is the LU factorization of a local matrix, and is used to provide 'exact' solutions to local linear systems. The general term 'preconditioner' is a misnomer in this case.

For DENSE blocks, the LU factorization is calculated with LAPACK, and thus pivoting is used.

The CSR version is useful when diagonal or pivot blocks are large and banded. Currently, the CSR version simply calls `RILUK` (an incomplete factorization) with no dropping to simulate the full factorization without pivoting. Users should replace this with their own sparse factorization routine for which they have a license.

Name

LP_INVERSE — inverse of a local matrix

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_INVERSE);
```

Block formats	Inverse type	Fortran name
DENSE and CSR	Explicit	INVERSE

Description

`LP_INVERSE` is the inverse of a local matrix, and is used to provide 'exact' inverses or solutions to local linear systems. The general term 'preconditioner' is a misnomer in this case.

For DENSE blocks, the inverse is computed from an LU factorization with pivoting. For CSR blocks, the matrix is converted to a full matrix, and the algorithm for DENSE blocks is used. The CSR version is not likely to be useful in practice due to its cost. However, it is useful when trying to determine how well a global preconditioner will perform when exact local inverses are used.

Name

LP_SVD — stabilized inverse of a local matrix using singular value decomposition

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_SVD, double rthresh, double athresh);
```

Block formats	Inverse type	Fortran name
DENSE	Explicit	SVD

Description

`LP_SVD` is a perturbed inverse to a local matrix. The inverse is perturbed to reduce its condition number, using the singular value decomposition (SVD). This produces a 'stable' inverse which can help stabilize block factorizations.

Given the SVD of a block $A = U\Sigma V^T$, a dense approximate inverse $M = V\bar{\Sigma}^{-1}U^T$ is produced, where $\bar{\Sigma}$ is Σ with its singular values thresholded by a function of the largest singular value. More precisely, if

$$\Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix}$$

with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$, then

$$\bar{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \eta & \\ & & & \eta \end{bmatrix}$$

where all the singular values less than η have been replaced by η, and η is assigned to be

$$\eta = (\texttt{rthresh})\sigma_1 + (\texttt{athresh}).$$

The thresholds are parameters input to this local preconditioner.

Small values of η are normally used to protect against singular blocks A (e.g., `rthresh` = machine epsilon and `athresh` = 0). A stabilization effect occurs for large η (as large as `rthresh` = 0.1) when this local preconditioner is used with a block incomplete factorization, e.g., `BILUK`. This essentially makes the matrix more diagonally dominant. However, too large a perturbation will make the factorization inaccurate.

The `LU_SVD` preconditioner is only available for DENSE blocks. When CSR blocks are used in the block matrix, a similar stabilization may occur with local preconditioners that attempt to approximate the inverse directly, rather than with a factorization.

Reference

E. Chow and Y. Saad. Stabilized block ILU preconditioners. *In preparation.*

Name

LP_RILUK — relaxed incomplete LU factorization, using level-of-fill and stabilization (local preconditioner)

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_RILUK, int level, double omega);
```

Block formats	Inverse type	Fortran name
CSR	Implicit	RILUK

Description

`LP_RILUK` is a relaxed incomplete LU factorization of a local matrix. The parameter `omega` parameterizes between an unmodified and a modified factorization. Consider the incomplete factorization

$$A \approx LU$$

where L is unit lower triangular and U is upper triangular. In a *modified* factorization the diagonal entries of U are modified during the factorization such that the row-sums of A match the row-sums of LU. This may be thought of as adding back the dropped entries to the diagonal of A. In a *relaxed* factorization, a fraction `omega` of these dropped quantities is added back. A value of `omega` of 0 corresponds to an unmodified factorization, while a value of 1 gives the modified factorization. A value of `omega` of 0.95 has been suggested by Van der Vorst for discretizations of second order elliptic partial differential equations. This value is a compromise that gives better control of the highest eigenvalues of the preconditioned system, while not deteriorating too much the lower eigenvalues.

The fill-in in `LP_RILUK` is controlled using the level-of-fill parameter `level`. A level of 0 corresponds to no additional fill-in over the original nonzeros in A. Higher levels usually improve the accuracy of the factorization, but levels beyond 3 are typically too expensive to be useful. Fill-in is always allowed onto the diagonal block, which differs from most implementations.

The estimate of the storage required by the symbolic factorization is controlled by the resource parameter `growth`. Let nnz be the number of nonzeros in the lower triangular part of A. Then the nonzero storage provided for L is

$$(\texttt{level} + \texttt{growth}) \times nnz.$$

A default value of 2 is used for `growth`. If `level` $= 0$, the storage provided is simply nnz; if `level` $= n$, the order of the matrix, the storage is $n(n+1)/2$. The same estimate is used for U, in which case nnz is the number of nonzeros in the upper triangular part of A. If you get a failure message due to insufficient space for the factorization, then this parameter should be increased.

`LP_RILUK` is also capable of producing a stabilized factorization, where the diagonal entries of U are further modified if they are too small. The following algorithm is used. Let the diagonal entry be ρ. If $|\rho| <$ `thresh`, then ρ is replaced by

$$\texttt{rel}\rho \pm \texttt{abs}$$

where the sign of `abs` is taken to be the same as the sign of ρ. This will artificially increase the size of ρ and improve the stability of the factorization. The parameters `thresh`, `rel` and `abs` are set via resources. Their default values are given below.

Resources

The use of the following resources is described above.

Resource	Default	Values
`RILUK.growth`	2	positive integer
`RILUK.thresh`	0.0	real ≥ 0
`RILUK.rel`	1.0	real ≥ 1
`RILUK.abs`	0.0	real ≥ 0

References

O. Axelsson and G. Lindskog. On the eigenvalue distribution of a class of preconditioning methods. *Num. Math.*, 48:479–498, 1986.

O. Axelsson and G. Lindskog. On the rate of convergence of the preconditioned conjugate gradient method. *Num. Math.*, 48:499–523, 1986.

H. A. Van der Vorst. High performance preconditioning. *SIAM J. Sci. Comput.*, 10 (1989), pp. 1174–1185.

Name

LP_ILUT — incomplete LU factorization, based on threshold and maximum nonzeros per row (local preconditioner)

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_ILUT, int lfil, double thresh);
```

Block formats	Inverse type	Fortran name
CSR	Implicit	ILUT

Description

`LP_ILUT` is a threshold-based incomplete LU factorization of a local matrix. The relative threshold parameter `thresh` is used to drop elements with small magnitude compared to the 2-norm of the original row. First, elements are dropped immediately (i.e., not used in the factorization) if they are small after being divided by their pivot elements. After the rows in L and U are computed, elements are also dropped if they are small.

An additional parameter `lfil` controls the maximum number of nonzeros in each row of L and U, i.e., if there are more than `lfil` elements in L or U and thus the maximum storage of the preconditioner is known beforehand.

No pivoting in the factorization is used in this version.

Notes

This local preconditioner is implemented with a FORTRAN code from SPARSKIT (see references). The source file (CSRilut.f) is a good example of how to add FORTRAN code to BPKIT.

References

Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Num. Lin. Alg. Appl.,* 1 (1994), pp. 387–402.

Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

Name

LP_APINV_TRUNC — truncated inverse of a local matrix

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_APINV_TRUNC, int semibw);
```

Block formats	Inverse type	Fortran name
CSR	Explicit	APINV_TRUNC

Description

`LP_APINV_TRUNC` is the truncated inverse of a local matrix. The exact inverse is computed, but only the diagonal band with semi-bandwidth `semibw` is retained. This is a good approximate inverse if the inverse has decay of its entries from the diagonal. Although the exact inverse is calculated, this local preconditioner may be faster to compute if the local matrices are small, but not too small so that the entire inverse should be retained.

Name

LP_APINV_BANDED — banded approximate inverse of a local matrix

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_APINV_BANDED, int semibw);
```

Block formats	Inverse type	Fortran name
CSR	Explicit	APINV_BANDED

Description

`LP_APINV_BANDED` is a banded approximate inverse of a local matrix. The approximation has semi-bandwidth `semibw`. The approximation M to the inverse of A is the solution of the problem

$$\min \|I - AM\|_F.$$

The algorithm that is used is the same as that for `LP_APINV0`; see its manual page for more information.

Resources

The approximate right-inverse is computed by default. The approximate left-inverse is found by minimizing

$$\min \|I - MA\|_F$$

and is computed if the `APINV0.right` resource is set to 0.

Resource	Default	Values
`APINV_BANDED.right`	1	0=left, 1=right

Name

LP_APINV0 — approximate inverse of a local matrix using least-squares minimization

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_APINV0);
```

Block formats	Inverse type	Fortran name
CSR	Explicit	APINV0

Description

`LP_APINV0` is an approximate inverse of a local matrix. For a local matrix A, the `LP_APINV0` approximate inverse M has the same sparsity pattern as A. More precisely, it has the same structure as A, including explicit zeros that may be stored in A.

If the sparsity pattern of M is fixed, then M is the solution to the minimization problem

$$\min \|I - AM\|_F = \sum_{j=1}^{n} \|e_j - Am_j\|_2^2$$

where $\|\cdot\|_F$ denotes the Frobenius norm, and e_j and m_j are the j-th columns of the identity matrix and of the matrix M, respectively. Note that the problem reduces to n least-squares problems. This approximate inverse technique is very simple to use because it requires no parameters.

`LP_APINV0` may encounter some difficulties for very irregularly structured matrices. Also, it is possible to dynamically select the sparsity pattern of M by testing candidate positions, but this has not been implemented.

Resources

The approximate right-inverse is computed by default. The approximate left-inverse is found by minimizing

$$\min \|I - MA\|_F$$

and is computed if the `APINV0.right` resource is set to 0.

Resource	Default	Values
`APINV0.right`	1	0=left, 1=right

References

J. D. F. Cosgrove, J. C. Díaz, and A. Griewank. Approximate inverse preconditioning for sparse linear systems. *Intl. J. Comp. Math.,* 44 (1992), pp.91–110.

M. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.,* (to appear).

Name

LP_APINVS — approximate inverse local preconditioner, using minimization and sparsity pattern selection by an iterative method

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_APINVS, int lfil, int self_precon);
```

Block formats	Inverse type	Fortran name
CSR	Explicit	APINVS

Description

`LP_APINVS` is an approximate inverse of a local matrix. For a local matrix A, the `LP_APINVS` approximate inverse M is the solution to the minimization problem

$$\min \|I - AM\|_F = \sum_{j=1}^{n} \|e_j - Am_j\|_2^2$$

where $\|\cdot\|_F$ denotes the Frobenius norm, and e_j and m_j are the j-th columns of the identity matrix and of the matrix M, respectively.

`LP_APINVS` differs from `LP_APINVS` in that an iterative method is used to perform the minimization, and that the sparsity pattern of the approximate inverse M is chosen dynamically. Technically, the minimal residual method is used, and entries in the columns of M are dropped if they are small. A maximum of `lfil` entries are allowed in each column of M. If `self_precon` is nonzero, then the current approximate inverse is used to precondition the minimal residual iterations.

Resources

Resources are used to set the options required by `LP_APINVS`. Two minimization methods are provided. The original method is selected by setting the `APINVS.method` resource to 0, while the new method is selected by a value of 1. (See the reference for a description of these methods).

The original method takes `APINVS.ninner` steps on each column before moving on to the next column. `APINVS.nouter` sweeps such as these are made over the matrix. If self-preconditioning is used, then setting `APINVS.ninner` to 1 will give greatest effect. The original method is somewhat more efficient than the new method, but does not guarantee the monotone reduction of the residual matrix norm; it may *spoil* if too many iterations are taken.

The new method always takes `lfil` steps, introducing a new fill-in at each step, unless the residual norm of a column becomes less than `APINVS.epsilon`. The norm of the residual matrix is guaranteed to be reduced monotonely.

Resource	Default	Values
`APINVS.method`	1	0=orig, 1=new
`APINVS.guess`	0	0=ident, 1=transp
`APINVS.nouter`	5	positive integer
`APINVS.ninner`	1	positive integer
`APINVS.right`	1	0=left, 1=right
`APINVS.epsilon`	0.0	real ≥ 0

Reference

E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iteration. *SIAM J. Sci. Comput.*, to appear.

Name

LP_DIAG — diagonal approximation to the inverse of a local matrix

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_DIAG);
```

Block formats	Inverse type	Fortran name
CSR	Explicit	DIAG

Description

`LP_DIAG` is the inverse of the diagonal of a local matrix. It is a very inexpensive approximate inverse that can be effective for very diagonally-dominant local matrices. Also, it is the specialized implementation of an exact inverse if the diagonal or pivot blocks of a block matrix are diagonal.

Name

LP_TRIDIAG — tridiagonal approximation to a local matrix

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_TRIDIAG);
```

Block formats	Inverse type	Fortran name
CSR	Implicit	TRIDIAG

Description

`LP_TRIDIAG` uses the tridiagonal part as an approximation to a local matrix. It is a fast implicit approximate inverse, using the tridiagonal solution algorithm. The approximation is good for diagonally-dominant local matrices. This is the ideal local preconditioner if the diagonal blocks or pivot blocks of a block matrix are (or are forced to be) tridiagonal, in which case the solutions are exact.

Name

LP_SOR — Successive Over Relaxation local preconditioner

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_SOR, double omega, int iterations);
```

Block formats	Inverse type	Fortran name
CSR	Implicit	SOR

Description

`LP_SOR` uses successive over relaxation to solve linear systems involving a local matrix. A relaxation factor of `omega` is used, and `iterations` steps are taken. The common case of `omega` $= 1$ and `iterations` $= 1$ corresponds to a solve with the lower-triangular part of the local matrix A.

Name

LP_SSOR — Symmetric Successive Over Relaxation local preconditioner

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_SSOR, double omega, int iterations);
```

Block formats	Inverse type	Fortran name
CSR	Implicit	SSOR

Description

`LP_SSOR` uses symmetric successive over relaxation to solve linear systems involving a local matrix. A relaxation factor of `omega` is used, and `iterations` steps are taken.

Name

LP_GMRES — GMRES iterative method applied to a local matrix

Synopsis

```
GlobalPrecon M;
M.localprecon(LP_GMRES, int iterations, double tolerance);
```

Block formats	Inverse type	Fortran name
CSR	Implicit	GMRES

Description

`LP_GMRES` is a generalized minimal residual iterative solver for local linear systems. It is a type of inner iterative process. This inner iterative process is not preconditioned itself. Note that a flexible iterative solver must be used in the outer iteration. `LP_GMRES` takes a maximum of `iterations` steps to reduce the residual norm to `tolerance` times the original. No restarting is used.

Chapter 4

Internal Manual Pages

BPKIT was designed to be easily extensible. Users may add new block types and block matrices, and new local and global preconditioners. This Section gives two manual pages that are useful for users who wish to extend or modify BPKIT. Please read the reference below, and contact the authors for additional documentation and information.

Reference

E. Chow and M. A. Heroux. An Object-Oriented Framework for Block Preconditioning. Technical Report UMSI 95/216 (October 1995), Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1995. Submitted to *ACM Trans. Math. Softw.*

Name

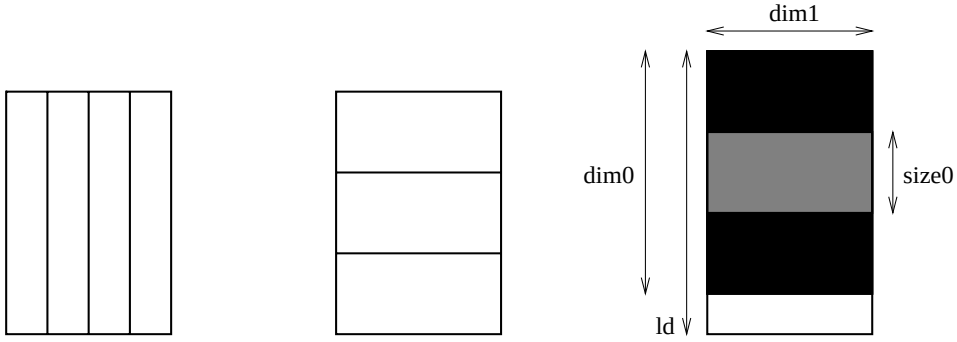
BlockVec — block of vectors; also, block within a block of vectors

Synopsis

```
#include "BlockVec.h"

const double *v;
int dim0;
int dim1;
int ld;
int size0;

BlockVec& operator()(int i);

BlockVec(int nr, int nc, const double *a, int lda, const int *partitioning)
BlockVec(const BlockVec& A);
BlockVec(const BlockVec& A, int i);

~BlockVec();

void VecCopy(const BlockVec& A);
void VecSetToZero();

void BlockCopy(const BlockVec& A);
void BlockSetToZero();
```

Description

BlockVec is actually two different types of objects, combined into one for efficiency. The figures below will aid the explanation.



The first object is a **block of vectors**, i.e., a set of column vectors of the same length, shown by figure (a). A block of vectors is partitioned into blocks, e.g., three blocks as shown in figure (b). The second object is a single **block** within a block of vectors. The two objects are combined because they share the same numerical data, and it is not desirable to create separate block objects due to the overhead involved. (We have avoided the use of the term 'block vector' due to its ambiguity.)

Within the `BlockVec` class, the name `Vec` is used to refer to a block of vectors, and the name `Block` is used to refer to a block within a block of vectors. `Vec` objects

are used primary in block matrices and global preconditioners; `Block` objects are used in local matrices and local preconditioners.

Suppose `V` is a `Vec` object. The primary functionality of the `BlockVec` class is to allow expressions such as `V(i)` to refer to the `i`th block in V. Thus `V(i)` is a `Block` object.

The other functions of the class allow `Vec` and `Block` objects to have their values set or initialized to zero. `VecCopy` and `VecSetToZero` perform these two functions for `Vec` objects, and `BlockCopy` and `BlockSetToZero` perform these two functions for `Block` objects. Note that the new data is *injected*, and no new memory is allocated.

Constructors

Unfortunately, we cannot use names to distinguish between constructors for the `Vec` and `Block` objects. Instead, we use overloading to partially solve the problem.

`Vec` objects are constructed using

```
BlockVec(nr, nc, double *a, lda, int *partitioning)
```

where `nr` and `nc` are the row and column dimension of the block of vectors, `a` points to the beginning of the data, stored in FORTRAN column-major order, `lda` is the leading dimension of the data array `a`, and `partitioning` is the block partitioning of the vectors. Thus `Vec` objects greatly simply the use of blocks of vectors, by collecting together all their parameters.

`Vec` objects may also be created from an existing `Vec` object, when a copy of a block of vectors is required. If `U` is a `Vec` object, the declaration `BlockVec V(U);` creates `V` with a *copy* of the data of `U`.

`Block` objects not usually physically constructed, since the expression `V(i)` turns a `Vec` object into a `Block` object. However, it is sometimes necessary to make a copy of a block, and this is accomplished with the declaration

```
BlockVec V(U, i);
```

which creates `V` with a copy of the data of `U(i)`. There is one special case, in which `U` itself is already a `Block` object. This case should be identified to the constructor with a negative index `i`.

Notes

The `BlockVec` destructor automatically deallocates any memory allocated by the constructors.

Figure (c) above defines the public variables that may be accessed in the class.

Debugging

This is is a complicated class. Run-time checking of the use of this class may be turned on by compiling with `-DDEBUG`. Since this used so often, run-time checking should normally be left off.

Name

BpResource — functions to read parameters from resource file

Synopsis

```
#include "BpResource.h"

int    BpGetInt(char *resource_name, int val);
double BpGetDouble(char *resource_name, double val);
char  *BpGetStr(char *resource_name, char *val);
```

Description

BpResource functions return parameters stored in a resource file, This allows the characteristics of BPKIT objects and functions to be customized without recompilation. For instance, the `fgmres` object may be customized to print or not print the convergence history while it is running. This option may be changed from run to run without needing to change the program and recompiling.

The functions `BpGetInt(resource_name,val)`, `BpGetDouble(resource_name,val)`, and `BpGetStr(resource_name,val)` return `int`, `double` or string values from the resource file, respectively. `resource_name` specifies the name of the resource, and `val` its default value, in case the resource is not specified. For `BpGetStr`, the caller is responsible for freeing the string that is returned.

`resource_name` has the format `class.attribute` where `class` is usually a BPKIT object name, and `attribute` is a particular attribute of the class. For instance, the printing of convergence history in the FGMRES object has the resource name `fgmres.history` and can take values `0` or `1` (off or on).

The default resource file name is `.BpResource`. Each line has the following format:

```
class.attribute: value  % comment
```

Note that a colon `:` immediately follows the resource name. Whitespace may follow the colon before the resource value is specified. A percent symbol begins a comment for the remainder of the line.

Adding Resources

Resources are stored as static members of a class. Thus they are initialized at source-file level at run time. To add a resource to a class, follow these steps:

1. Add the static member to the class.
2. Define the name of the resource in the header file.
3. Initialize the resource in the source file.
4. Use the static member somewhere.
5. Add the resource to the resource file if necessary.

Update: with some compilers, the static members are not initialized. In this case, the `bpinitialize` function must be used. See `FORTRAN interface`.

Implementation Notes

Resource databases are used extensively in the X-window environment and this is a very rudimentary implementation of it, to provide some basic functionality as conveniently as possible. The resource file is opened and closed whenever a resource is sought. The resource names are searched in the resource file linearly. Thus there can be significant overhead with this implementation, so they should not be used for classes for which hundreds of objects may be created (e.g., `DenseMat`). Resources are provided for options that are seldomly changed, so that naive users do not need to be aware of them.

Files

`.BpResource` — default resource file name