ISIS++ Reference Guide
(Iterative Scalable Implicit Solver in C++)
Version 1.1

Benjamin A. Allan, Robert L. Clay, Kyran D. Mish, Alan B. Williams

RECEIVED

JUL 0 1 1999

OSTI

Sandia National Laboratories

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

ISIS++ Reference Guide
(Iterative Scalable Implicit Solver in C++)
Version 1.1

Benjamin A. Allan, Robert L. Clay, Alan B. Williams
Distributed Applications Research Department
Sandia National Laboratories
and
Kyran D. Mish
Department of Mechanical Engineering
California State University, Chico

Abstract

ISIS++ (Iterative Scalable Implicit Solver in C++) Version 1.1 is a portable, object-oriented framework for solving sparse linear systems of equations. It includes a collection of Krylov solution methods and preconditioners, as well as both uni-processor (serial) and multi-processor (scalable) matrix and vector classes. Though it was developed to solve systems of equations originating from large-scale, 3-D, finite element analyses, it has application in many other fields.

This document supercedes the ISIS++ v1.0 Reference Guide [4], defines the v1.1 interface specification, and includes the necessary instructions for building and running ISIS++ v1.1 on Unix platforms. The interface is presented in annotated header format, along with background on design and implementation considerations. A finite difference modeling example problem is included to demonstrate the overall setup and use.
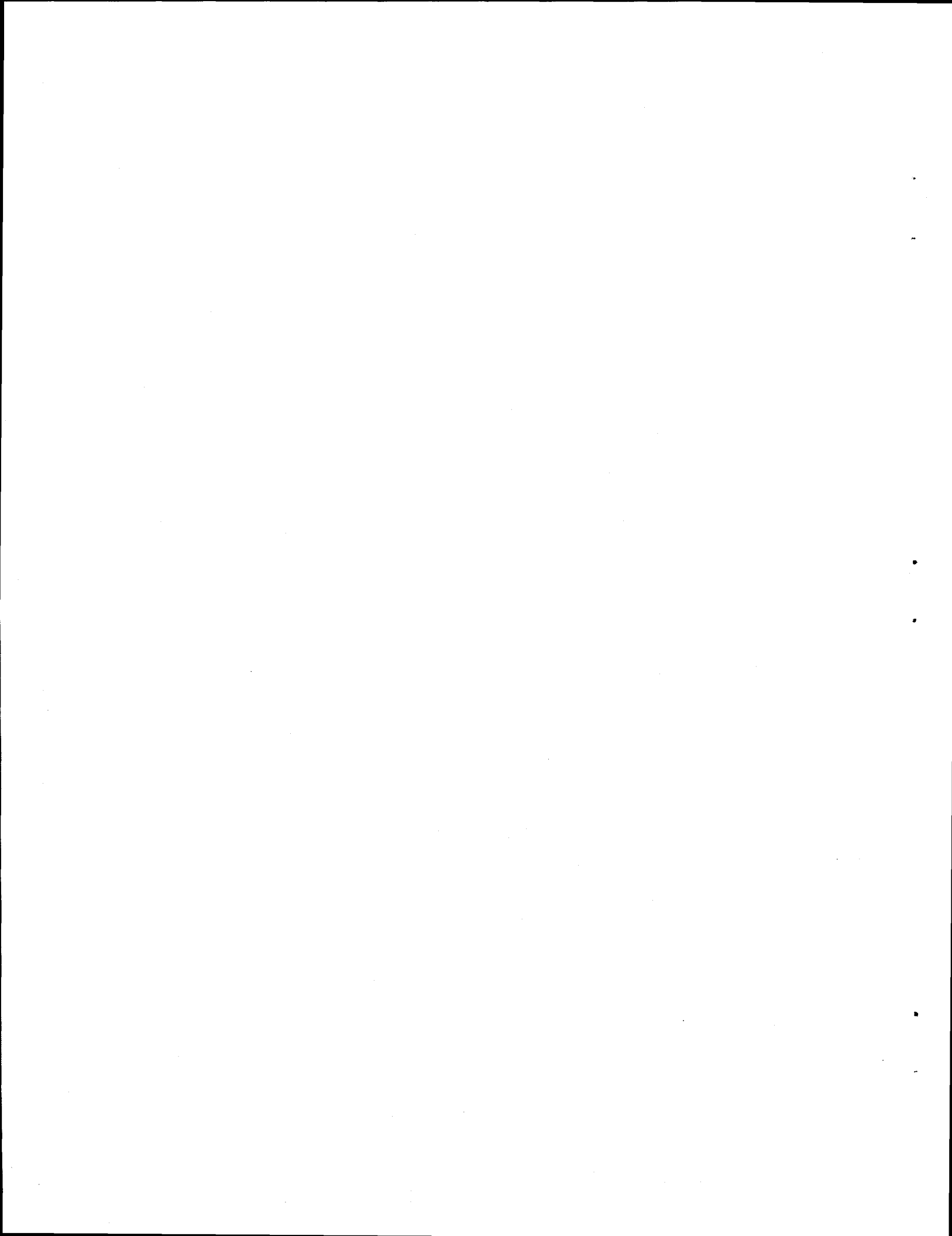
Table of Contents

1 Introduction

ISIS++ (Iterative Scalable Implicit Solver in C++) is a portable, object-oriented framework for solving sparse linear systems of equations. It includes a collection of Krylov subspace solution methods and preconditioners, as well as both uni-processor (serial) and multi-processor (scalable) matrix and vector classes (Figure 1). Though it was developed to solve systems of equations originating from large-scale, 3-D, finite element analyses, it has applications in many other fields.

ISIS++ is designed to provide simple interchangeability of components – both from within the ISIS++ system and from other packages. The ISIS++ framework facilitates integrating components from various libraries, and in particular the matrix-vector functional units and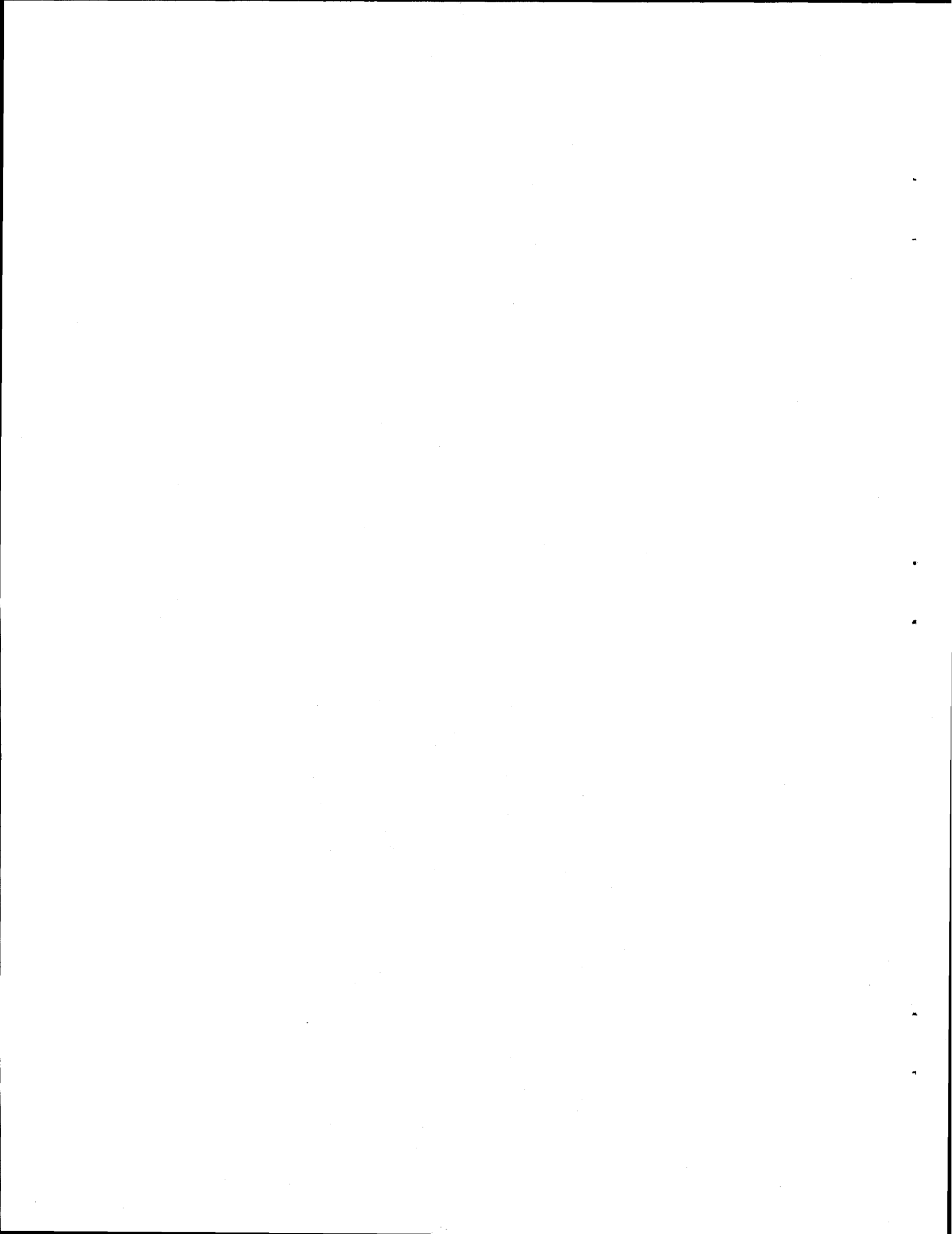 their corresponding data structures. The first practical test of this concept was the integration of the Aztec [14] DMSR matrix-vector classes.

A primary goal of the ISIS++ project is to decompose the problem space into a set of independent, object-oriented functional units, and in particular to decouple sparse matrix data structures and their implementations from their use in Krylov solvers and preconditioners. This can be viewed as developing archetypal interfaces between matrix, vector, solver and preconditioner objects. In this manner, matrix-vector objects can be implemented from various libraries while maintaining functional compatibility with the solvers and preconditioners.

The advantages of the framework design include improving the ability to leverage existing work. This facilitates usage of implementations and data structures tuned to a particular application and computing platform. The source code for the solver and preconditioner components is decoupled from the matrix-vector implementations[1]. Thus, ISIS++ can be built using the matrix-vector implementation best suited to the task and compute system at hand, with no changes to the solver or preconditioner source code.

For this concept to work in practice, the task of including library components must be relatively straightforward and efficient. This design objective was addressed by a policy of *minimal but sufficient* core components. That is, the abstract base classes define the core set of interactions between solvers, preconditioners, matrix and vector objects, regardless of their implementation. The purpose of keeping the core interface requirements minimal is to simplify (i.e., not unduly restrict or complicate) adding new implementations into the framework. To support parallel implementations, care has been taken to avoid inclusion of any function not deemed scalable.

---

[1] For preconditioners which access the matrix values directly, completely generalized access can potentially incur a large overhead cost (e.g., if a matrix is stored as CRS format and the preconditioner attempts to access the values column-wise this is an extremely inefficient process). We use a hierarchical mechanism to limit access to data structures so as to preclude these types of inefficiencies.
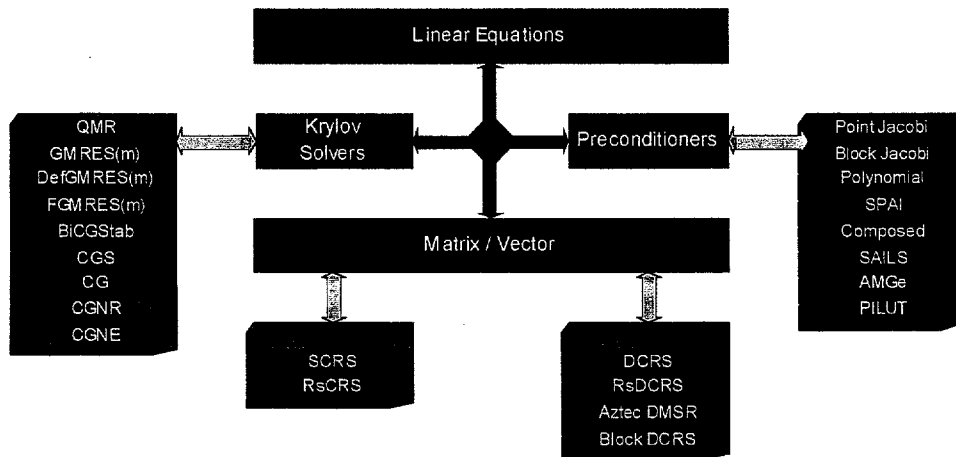
Figure 1. ISIS++ framework overview[2].

It is essential that the performance of the mathematical components incorporated from other libraries be comparable to that of their "native" state. That is, the overall performance of a combination of components within ISIS++ should be comparable to that observed when those components are run in their original stand-alone state. Our observation is that a penalty is incurred for providing fully generalized matrix access functions for use in the preconditioners. Specifically, in cases where the preconditioner needs an internal representation of the sparse matrix, there may be a memory and/or a "copy" time overhead associated with the translation. However, specializations of the matrix classes can be provided to give the preconditioner knowledge of the data structures, allowing direct (fast) access to the underlying data with minimal overhead.

The remainder of this document is organized as follows. The ISIS++ framework design and core base classes are discussed in section 2. The solver, preconditioner, and matrix/vector class implementations are presented in sections 3, 4, and 5, respectively. An example problem is described in section 6. The installation procedures are provided in section 7, and the references are in section 8.

Further information and the most current updates can be found on the ISIS++ web site at http://z.ca.sandia.gov/isis/.

---

[2] The color coding in the overview figure is as follows: blue represents the central abstractions in ISIS++, dark blue represents the implementations, dark magenta (AMGe & PILUT) represents the implementations still in progress at the time of this writing, and dark cyan (CGNE and CGNR solvers) are specializations which interact with only a subset of the preconditioners. The SAILS, AMGe, and PILUT preconditioners are LLNL/CASC (http://www.llnl.gov/CASC/) implementations.

2 Framework Overview

In this section we describe the ISIS++ framework, which is founded on the base classes *Solver, Preconditioner, Matrix, Vector,* and *LinearEquations.* These classes constitute the fundamental abstractions within ISIS++, and define the core interactions provided by the framework. As will be shown, specializations of these abstractions are provided in order to address the needs of certain methods (e.g., the SPAI preconditioner uses dynamic row resizing features not provided in the Matrix base class).

After an overview of the ISIS++ framework, we describe the public interfaces for the core and derived base classes. Additionally, we describe the public interfaces for the *Map* and *CommInfo* auxiliary classes.

2.1 Central abstractions

The ISIS++ framework includes an integrated collection of C++ classes which are designed for the scalable solution of large-scale, unstructured, sparse systems of linear of equations on distributed memory parallel computers[3].

At the core of the ISIS++ framework are the abstract base classes: Solver, Preconditioner, Matrix, Vector, and LinearEquations. These base classes are particularized to yield specialized base classes as follows:

Solver → IterativeSolver
Preconditioner → RowPreconditioner
Matrix → RowMatrix.

The hierarchical representation of this class structure is shown in Figure 2. The core base classes interact with each other through the functions defined in their public interfaces, and represent generalizations of the primary functional and data units within the framework., These classes and their immediate descendants define the basic framework, while the implementations of the classes provide the data structures and solution methods.

The framework insulates the implementation details of one base class from another. Implementations can be added or modified without requiring changes to associated (or indirectly related) classes. For example, adding a new matrix class is simply a matter of mapping the data and functionality of the matrix object into the Matrix (or derived matrix) base class. The solvers and preconditioners are immediately (and without modification to source code) able to use the new matrix class, since it behaves according to the definition of the base class. It is a simple matter to switch matrix and vector class implementations to run on uni-processor or multi-processor computers. Thus, parallel matrix implementations can be selected according to those best tuned for the platform.

---

[3] Uni-processor implementations are also supported.
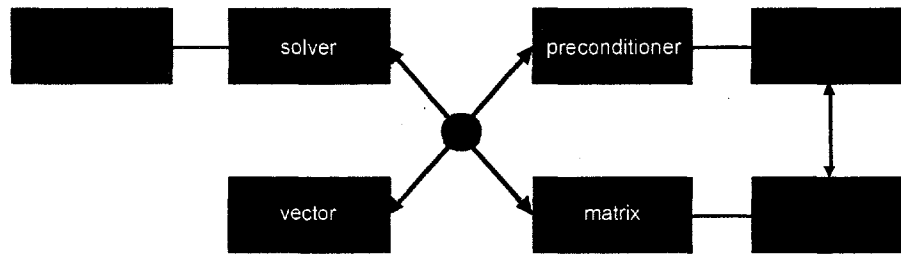
Figure 2. ISIS++ central abstractions.

2.2 Solver base classes

The *Solver* abstract base class[4] is essentially a placeholder for the function `solve`. The Solver class is the root for the *IterativeSolver* abstract base class. The Krylov methods implemented in ISIS++ are all derived from the IterativeSolver base class. The solver base classes share one important feature – they use generic representations of matrices and vectors (i.e., they have as arguments the classes Matrix and Vector). Consequently, solver implementations (such as the Krylov solvers derived from the IterativeSolver class) may interact with any matrix and vector objects so long as they do not introduce specialized requirements from the Matrix or Vector base classes.

In effect, the solver base classes are only relevant from the developer's point of view, since the user interacts with particularizations of the solver classes. Indeed, the solve function itself is accessed via the LinearEquations class (see below). The annotated public interfaces for the Solver and IterativeSolver base classes follow. The functions represented therein are inherited by (and hence are available from) all of the Krylov subspace iterative methods presented in section 3.

Solver class public interface

```
class Solver

// default constructor function
Solver() {};
// default destructor function
virtual ~Solver() {};


// solve function
virtual int solve(const Matrix& A, Vector& x, const Vector& b,
    Preconditioner& pc) = 0;
```

---

[4] A C++ class containing one or more pure virtual functions is by definition an abstract base class. Pure virtual functions *must* be implemented by derived classes, hence the notion of an *abstract* base class. It is not possible to instantiate an abstract base class object. Rather, derived objects must be instantiated. Ellis and Stroustrup [6].

10

```
// to pass in parameters
virtual void parameters(int numParams, char **paramStrings) = 0;


// to set amount of screen output
virtual void outputLevel(int level, int localRank, int masterRank);
```

Solver base class public interface reference notes:

- The return values for all `solve` functions are currently interpreted as follows:
    - 1   successful completion, normal convergence tolerance met.
    - 0   unsuccessful completion, failed to converge in `maxIterations()`.
    - -1   unsuccessful exit on stall condition.
    - -2   failed on memory allocation.

- The `parameters` function is to be implemented by each solver implementation. The format of the arguments is the same as that used to access command line arguments in general, and allows any number of any type of argument to be passed in. A simple example of this is given in the Example Problem section at the end of this document.

- The `outputLevel` function determines the amount of screen output that will be produced. The "level" parameter has the following effect:
    - 0   no screen output
    - 1   master node prints out parameter values and residual norms
    - 2   all nodes print out information. Intended for debugging purposes.

    The *localRank* and *masterRank* arguments are logical processor numbers for the parallel case; they can both take the value 0 for the serial case.

IterativeSolver class public interface

```
class IterativeSolver : public Solver

// default constructor function
IterativeSolver() :
     tolerance_(1.0e-13),  // default convergence tolerance
     maxIterations_(2000), // default max iterations
     maxStallCount_(0) {}; // disable stall check
// default destructor function
virtual ~IterativeSolver(){};


// solve function
virtual int solve(const Matrix& A, Vector& x, const Vector& b,
     Preconditioner& pc) = 0;


// get/set convergence tolerance
double tolerance();
void tolerance(double tol);
```

```
// get/set maximum number of iterations
int maxIterations();
void maxIterations(int maxIt);


// get/set scaled residual 2-norm
double normalizedResidual();
void normalizedResidual(double residual);


// get/set stall count
int maxStallCount();
void maxStallCount(int maxSC);
```

IterativeSolver base class public interface reference notes:

- The return values for the `solve` function are similar to the Solver base class.

- Current implementations of the `normalizedResidual` function use the 2-norm of the RHS vector b to scale the residual vector.

- The `maxStallCount` functions are used to control the stall checking algorithm within the Krylov solvers. The stall checking algorithm looks for progress toward convergence within *maxSC* iterations, and will terminate the iterations if a stall condition is observed. Setting `maxStallCount(0)` disables the stall checking algorithm. By default, the stall checking is disabled.

2.3 Preconditioner base classes

The *Preconditioner* abstract base class provides the fundamental interactions required by Krylov subspace iterative solvers, fashioned after the development presented in Barrett, et al. [2]. The basic interaction between solvers and preconditioners in ISIS++ is discussed below. It is worth noting the mathematical model assumed within ISIS++ before describing the Preconditioner and RowPreconditioner base classes.

The basic mathematical problem can be defined as follows. Given the linear system of equations:

$$Ax = b \ : \ A \in \mathfrak{R}^{n \times n}, x, \ b \in \mathfrak{R}^n, \tag{2.1}$$

where sparse matrix A and RHS vector b are known[5], we seek to determine the solution vector x within a predetermined accuracy. Consider a matrix M which approximates A in some sense. If M^{-1} is relatively cheap to compute and if $M^{-1}A \approx I$ (or is otherwise significantly better conditioned than A), then M can be considered an effective preconditioner. Assume a splitting of the approximation matrix such that:

$$M_1 M_2 = M \approx A \ : \ M_1, M_2, M \in \mathfrak{R}^{n \times n}. \tag{2.2}$$

Applying this to (2.1) gives

---

[5] Solution strategies can also use an initial guess of the solution vector x.

$$M_1^{-1} A M_2^{-1} M_2 x = M_1^{-1} b \; : \; A \in \mathfrak{R}^{n \times n}, \, x, \, b \in \mathfrak{R}^n, \tag{2.3}$$

which can be viewed as

$$By = c \; : \; B = M_1^{-1} A M_2^{-1}, \quad y = M_2 x, \quad c = M_1^{-1} b. \tag{2.4}$$

Typically, either $M_1 = I$ or $M_2 = I$, for *right* or *left* (one-sided) preconditioning, respectively. However, the basic model (and interface) supports two-sided preconditioning. The solvers form B, y, and c implicitly by applying the preconditioners through computational steps such as:

$$\text{solve for } z \text{ from } M_1 z = d,$$

which is available from the preconditioner interface as member function `solveM1(z, d)`. There are corresponding functions for M_1^T, M_2, and M_2^T, as shown below.

The *RowPreconditioner* base class provides specializations for implementations which require row-wise access to matrix values. As shown below, the Preconditioner and RowPreconditioner public interfaces are nearly identical, with only minor modifications in the constructor function to account for the use of RowMatrix objects. The primary difference in the interfaces is the requirement that a RowMatrix object be referenced. This restriction permits preconditioner implementations to access the row data within the reference matrix (passed in the constructor). By inheritance, all of the public functions from the Preconditioner class are available. The pure virtual functions shown for the RowPreconditioner class are essentially "passed down" from the Preconditioner class to derived implementations.

Preconditioner class public interface

```
class Preconditioner

// default constructor function
Preconditioner(const Matrix& A) : matrix_(A) {outputLevel_ = 0;};
// default destructor function
virtual ~Preconditioner() {};


// solver access functions
virtual void solveM1(Vector& y, const Vector& z) const = 0;
virtual void solveM1T(Vector& y, const Vector& z) const = 0;
virtual void solveM2(Vector& y, const Vector& z) const = 0;
virtual void solveM2T(Vector& y, const Vector& z) const = 0;


// calculate preconditioner
virtual void calculate() = 0;


// to pass in parameters
virtual void parameters(int numParams, char **paramStrings) = 0;


// clear memory
virtual void empty() = 0;
```

```
// left and right modifiers and query functions
virtual void setDefault() = 0;
void setLeft();
void setRight();
bool isLeft() const;
bool isRight() const;
```

Preconditioner base class public interface reference notes:

- Given a preconditioning matrix $M \approx A$, matrices $M_1 M_2 = M$, and vectors y and z, the solver access functions correspond to the following operations:

    `solveM1(y,z)` $\Rightarrow y = M_1^{-1}z$

    `solveM1T(y,z)` $\Rightarrow y = M_1^{-T}z$ (where M_1^{-T} is the inverse of M_1 transpose)

    `solveM2(y,z)` $\Rightarrow y = M_2^{-1}z$

    `solveM2T(y,z)` $\Rightarrow y = M_2^{-T}z$ (where M_2^{-T} is the inverse of M_2 transpose).

- The `calculate` function does any up-front computations for the preconditioner, and in general needs to be issued prior to invoking preconditioner services from within the solvers. For multi-step problems, the preconditioner results can be sub-cycled by calling the `calculate` function less frequently than the `LinearEquations::solve` function.

- The `parameters` function has the same functionality as the one in the Solver base class.

- The `empty` function is designed to release memory (after the preconditioner has been used) without deleting the preconditioner object. It is completely specific to the particular preconditioner implementation.

- The left and right modification and query functions allow the user to control the operation of the preconditioner in those cases where it may be applied from either the left or right. Generally, there is a preferred means of applying the preconditioner, and this is set by the `setDefault` function. By convention, the `setDefault` function is issued from within the constructor function[6].

RowPreconditioner class public interface

```
class RowPreconditioner : public Preconditioner

// default constructor function
RowPreconditioner(const RowMatrix& A) : Preconditioner(A), matrix_(A) {};
// default destructor function
virtual ~RowPreconditioner() {};
```

---

[6] Strictly speaking, the interface does not guarantee the user that the `setDefault` function will be issued upon construction.

```
// solver access functions
virtual void solveM1(Vector& y, const Vector& z) const = 0;
virtual void solveM1T(Vector& y, const Vector& z) const = 0;
virtual void solveM2(Vector& y, const Vector& z) const = 0;
virtual void solveM2T(Vector& y, const Vector& z) const = 0;


// calculate preconditioner
virtual void calculate() = 0;


// to pass in parameters
virtual void parameters(int numParams, char **paramStrings) = 0;


// clear some memory
virtual void empty() = 0;


// left and right modifiers
virtual void setDefault() = 0;
```

RowPreconditioner base class public interface reference notes:

- The RowPreconditioner class behaves precisely like the Preconditioner class, but adds the restriction that construction requires a RowMatrix object. This specialization permits the added (data access) functionality of RowMatrix objects to be employed by RowPreconditioner derived objects.

- The complete set of left and right modification and query functions are inheritted. The setDefault function is "passed through" as a pure virtual function to derived classes.

2.4 Auxiliary container classes

The Map and CommInfo classes are basic building blocks for sparse linear systems in ISIS++ on distributed-memory computing systems. These two classes contain information pertaining to the decomposition of the problem data. In a distributed-memory setting, these objects hold the partitioning and basic communications information. In the uni-processor setting (see section 5), the matrix/vector objects default to the trivial case and are constructed without need of partitioning information.

The *Map* base class is the primary container for partitioning information. Map class derived objects contain CommInfo objects. Consequently, CommInfo services can be reached via the Map class. The Map class contains a virtual representation of the nxm-matrix and vector partitioning information. Specifically, any matrix/vector can be thought of as corresponding to a linear partitioning of rows and columns across processors. Thus, there is the notion of a global matrix/vector addressing within the Map base class. Further, certain restrictions apply to this global addressing scheme, including:

- Rows and columns are globally numbered from 1 to n and 1 to m, respectively.

- The terms *startRow*, *endRow*, and *numLocalRows* retain the relationship

$$numLocalRows = endRow - startRow + 1$$

for all processors. The same relationship holds for the column equivalents.

Since this generalization is not suitable for all possible matrix/vector implementations (i.e., more complex data distributions may be desirable or necessary), the Map class is meant to be expanded as needed for each new matrix/vector class implementation. That is, a derivative Map class may be added for each new parallel matrix/vector implementation, depending on how the partitioning information is stored native to the new implementation. The public interface shown below contains the constructors for the ISIS++ "native" matrix/vector class implementations. The Map class specialization developed for the Aztec DMSR matrix/vector implementation is presented in section 5.10.

The *CommInfo* class contains information pertaining to processor IDs and the number of processors being used. This object exists trivially for the uni-processor case. For the parallel case, it is a repository of the communication subsystem information and is constructed by the end-user. CommInfo objects are referenced via Map objects which effectively own the information.

Map class public interface

```
class Map

// default distributed-memory constructor function
Map(int n, int startRow, int endRow, int startCol, int endCol, const CommInfo&
commInfo);

// non-square distributed-memory constructor function
Map(int n, int m, int startRow, int endRow, int startCol, int endCol, const
CommInfo& commInfo);

// default serial constructor function
Map(int n);

// copy constructor
Map(const Map& map);

// default destructor
virtual ~Map()   {};


// access functions
const CommInfo& getCommInfo() const;     // get CommInfo ref
int n() const;                           // get global row count
void initComplete();                     // mark object as fully constructed
bool isInitComplete()                    // TRUE if object is fully constructed
int startRow() const;                    // get local start row
void startRow(int startRow);             // set local start row
int endRow() const;                      // get local end row
void endRow(int endRow);                 // set local end row
int numLocalRows() const;                // get local number of rows
int startCol() const;                    // get local start column
void startCol(int startCol);             // set local start column
```

```
int endCol() const;                     // get local end column
void endCol(int endCol);                 // set local end column
int numLocalCols() const;                // get local number of cols
int numGlobalRows();                     // get global row count, n
void numGlobalRows(int numGR);           // set global row count, n
int numGlobalCols();                     // get global column count, m
void numGlobalCols(int numGC);           // set global column count, m
int* globalStartRow();                   // get processor rank indexed array of
int* globalEndRow();                     // starting or ending row indices
int* globalStartCol();                   // get processor rank indexed array of
int* globalEndCol();                     // starting or ending column indices
```

Map class public interface reference notes:

- The function *n*() is deprecated in favor of function *numGlobalRows*().

- The default distributed memory constructor requires the user to provide the total number of equations *n*, the processor-local values for the partitioning parameters, and a reference to a local CommInfo object. The distributed-memory constructor arguments are defined as follows:

n	global number of rows/cols (matrix must be square)
startRow	global index of lowest number row on local processor
endRow	global index of highest number row on local processor
startCol	global index of lowest number column on local processor
endRow	global index of highest number column on local processor.

- The non-square distributed memory constructor is as the distributed memory constructor except that the user must provide the total number of variables, *m*, also.

- The default serial constructor requires the user to provide the total number of equations *n*. The values of *startRow* and *startCol* are set equal to 1 and *endRow* and *endCol* to *n* upon construction. At present there is no non-square serial constructor.

- The copy constructor is used to create a duplicate Map object, which may then be independently modified.

- The array returned by `globalStartRow` contains the index of the starting row for each processor. For *k* processors, rank 1 to *k*, `globalStartRow()[i-1]` is the index of the first row stored on the *i*th processor. Similarly for `globalEndRow`, `globalStartCol`, and `globalEndCol`.

- Both the Matrix and Vector base classes contain the function *getMap* which can be used to retrieve a reference to a Map object.

- The `getCommInfo` function returns a reference to the CommInfo object associated with the Map object. As an example, consider the following code snippet:

```
Map map(n);                             // construct a simple serial map
SCRS_Matrix A(map);                     // construct a serial CRS matrix
int rowcount = A.getMap().n();          // matrix size, since square matrix
// determine local processor rank
```

17

```
int myRank = A.getMap().getCommInfo().localRank();
```

CommInfo class public interface

```
class CommInfo

// distributed-memory constructor function
CommInfo(int numProcessors, int masterRank, int localRank);

// serial constructor function
CommInfo();

// default destructor function
virtual ~CommInfo() {};


// access functions
int masterRank() const;
int localRank() const;
int numProcessors() const;
```

CommInfo class public interface reference notes:

- For the distributed-memory case, the processor ID *masterRank* is used primarily for output control. Native ISIS++ implementations use node *masterRank* as the primary synchronization point for some global operations.

- The serial constructor sets *masterRank* and *localRank* to zero, and *numProcessors* to one. While this is a trivial result, it permits the access functions to work interchangeably on serial and parallel platforms.

- Communications information may be retrieved via the function `Map::getCommInfo`, which returns a reference to a CommInfo object.

For example,

```
int n = 10;                          // 10 rows & cols
Map map(n);                          // construct serial map object
// query master processor rank
int masterRank = map.getCommInfo().masterRank();
```

2.5 Matrix base classes

The *Matrix* abstract base class represents the primary data structure within ISIS++, since matrix operations and storage typically dominate CPU and memory requirements, respectively. The Matrix class was designed to be as simple and general as possible, while providing the operations needed to support Krylov iterative solvers. Specializations of the matrix classes in large part revolve around the storage format of the data, and consequently the data access interface possibilities. Krylov solvers in general do not need to access the matrix data, but rather the mathematical operations of matrices on vectors. In this sense, the matrix data abstraction works ideally for solver/matrix interactions.

18

However, the same does not hold for preconditioner/matrix interactions. That is, some preconditioners need to access matrix values, and in some cases (e.g., SPAI) construct matrix objects internally. In these cases, a generalized data access interface (and underlying implementation) is desirable to keep the abstraction "intact". Unfortunately, this degree of generality appears to be impractical for high-performance implementations due to the overhead involved when the data does not naturally conform to the fully generalized access requirements. For example, consider the case whereby a sparse matrix object is stored in CRS (Compressed Row Storage) format (see Barrett et al.[2]), and a column of the matrix is needed. One does not even need to consider the complicating factor of partitioning the matrix across processors according to rows to realize that fetching a column of a CRS matrix is an extremely inefficient operation. Consequently, specializations of the Matrix class are needed to provide for efficient access to the internal data. As shown below, except for the ability to access the matrix diagonal, there are no provisions for data access in the Matrix base class. The data access specializations arise in the derived matrix classes.

The *RowMatrix* base class is derived from (and inherits the public interface of) the Matrix base class, and requires further specialization before objects can be constructed. The added access functions distinguish the RowMatrix class from the Matrix class. The common (pure virtual) functions are essentially passed through to classes which are derived from the RowMatrix class. A specialized set of direct pointer access functions is available for implementations which can support it. In particular, each row's data must be contiguous in memory for pointer access to be viable. This is currently provided for four ISIS++ matrix implementations, all derived from the RowMatrix class. A test function is provided for run-time determination of the viable existence of the pointer access functions.

We now present the public interfaces for the Matrix and RowMatrix base classes.

Matrix class public interface

```
class Matrix

// constructor function
Matrix(const Map& map);
// default destructor function
virtual ~Matrix() {};


// mathematical functions
virtual void vectorMultiply(Vector& y, const Vector& x) const = 0;
virtual void transposeVectorMultiply(Vector& y, const Vector& x) const = 0;


// data access functions
virtual void getDiagonal(Vector& diagVector) const = 0;
const Map& getMap() const;
virtual void put(double s);


// special functions
virtual void configure(const IntVector& rowCount) = 0;
```

```
virtual void fillComplete() = 0;
virtual bool readFromFile(char *filename) = 0;
virtual bool writeToFile(char *filename) const = 0;


// query functions
bool isFilled() const;
bool isConfigured() const;


// min/max functions
virtual bool rowMax() const {return false;};
virtual bool rowMin() const {return false;};
virtual double rowMax(int rowNumber) const (return -1.0;);
virtual double rowMin(int rowNumber) const (return -1.0;);
```

Matrix base class public interface reference notes:

- Upon construction the matrix object does not allocate the memory space for the data. Rather, the `configure` function passes the vector *rowCount* which contains the number of non-zeros per row. At that point the memory for the matrix values and indices can be allocated. This is not needed or even supported by all implementations, as will be seen later.

- The use of the matrix-vector multiply functions are illustrated in the following code snippet.

```
    DCRS_Matrix A(map);          // construct DCRS matrix A
    Dist_Vector y(map);          // construct distributed vector y
    Dist_Vector z(map);          // construct distributed vector z
  (initialize A and z)
    A.vectorMultipy(y,z);              //  y = Az
    A.transposeVectorMultiply(y,z);    //  y = A'z
```

- The `getDiagonal` function loads the reference vector with the matrix diagonal terms.

- The `getMap` function returns a reference to the associated Map object, and is the access point for Map and (indirectly) CommInfo information.

- The `configure` function allocates memory for the storage of all necessary terms to contain matrix data. Calling this function resets the internal state such that (for a given matrix) subsequent calls to `isConfigured` will return *true*. It is not necessary to use the `configure` function for all matrix implementations.

- The `fillComplete` function provides a placeholder for handling the data consistency checks as well as message-passing configuration information for the distributed-memory case. This function *must* be invoked once all the user data is loaded into the matrix object, and before computations are performed with it.

- The `put` function assigns the value given to all the non-zero elements of an already configured matrix.

- The `readFromFile` and `writeToFile` functions read/write to a user-named ASCII file in the Matrix Market exchange format. As yet, not all ISIS matrix implementations are fully MatrixMarket conformant, as documented in the file doc/file-format.txt in the ISIS++ source distribution.

- The `isFilled` function is used to verify that the matrix object has been loaded with data before attempting to mathematically operate on the matrix. When a matrix object is constructed, or subsequent to calling the `empty` function, the matrix state is internally set to *not-filled*. Only after calling `fillComplete` is the state reset to return *true*.

- The `isConfigured` function is used to query whether the matrix object has been configured (i.e., the memory has been allocated).

- The boolean `rowMax` and `rowMin` (query) functions indicate whether a valid implementation of the associated functions exist for a particular matrix implementation.

- New matrix implementations require development of the `configure` and `fillComplete` functions, and potentially a new variant of the Map class.

RowMatrix class public interface

```
class RowMatrix : public Matrix

// constructor function
RowMatrix(const Map& map) : Matrix(map) {};
// default destructor function
virtual ~RowMatrix() {};


// mathematical functions
virtual void vectorMultiply(Vector& y, const Vector& x) const = 0;
virtual void transposeVectorMultiply(Vector& y, const Vector& x) const = 0;


// special functions
virtual void configure(const IntVector& rowCount) = 0;
virtual void fillComplete() = 0;


// data access functions ...
virtual void getDiagonal(Vector& diagVector) const = 0;
virtual void getRowSum(Vector& rowSumVector) const = 0;
virtual int rowLength(int row) const = 0;


// ... to resize matrix rows (where applicable)
virtual bool setRowLength(int length, int rowNumber) {return false;};


// ... to read matrix rows.
virtual void getRow(int row, int& length, double* coefs, int* colInd) const = 0;
virtual void getRow(int row, int& length, double* coefs) const = 0;
virtual void getRow(int row, int& length, int* colInd) const = 0;
```

21

```
// ... to write matrix rows.
virtual int putRow(int row, int cardinality, double* coefs, int* colInd) = 0;
virtual int sumIntoRow(int row, int cardinality, double* coefs, int* colInd) =
0;


// specialized direct pointer access functions ...
// ... test for pointer access viability
virtual bool pointerAccess() {return false;};
// ... read-write pointer access to matrix data
virtual double* getPointerToCoef(int& length, int rowNumber) = 0;
virtual int* getPointerToColIndex(int& length, int rowNumber) = 0;
// ... read-only pointer access to matrix data
virtual const double* getPointerToCoef(int& length, int rowNumber) const=0;
virtual const int* getPointerToColIndex(int& length, int rowNumber) const=0;
```

RowMatrix base class public interface reference notes:

- Unless otherwise indicated, the RowMatrix functions are identical to the Matrix class equivalents.

- The `getRowSum` function returns (via the argument list) a vector whose elements are the sum of the absolute values of the entries of the corresponding rows.

- The `rowLength` function returns the number of (presumably non-zero) entries in the specified row.

- The `setRowLength` function is provided for all RowMatrix class implementations, but is only functional for those implementations which support dynamically resizing rows. For the statically sized implementations, the function will return *false*. For the dynamically sized implementations, the function will return *true*.

- Three variations of the `getRow` function are provided for reading matrix row data, each guaranteed not to modify the matrix data. Depending on the argument list, data is loaded into the buffers for the matrix coefficients and/or column indices, and the *length* of the row is returned as an argument.

- The `pointerAccess` function[7] provides a boolean test for the availability of the direct pointer access functions, which only apply to implementations with rows stored in contiguous memory.

  For implementations which do not support pointer access, the `pointerAccess` function returns *false* and the `getPointerToCoef` and `getPointerToColIndex` functions return pointer to NULL (i.e., zero value) and argument *length* = -1.

---

[7] This approach was chosen over more elegant solutions, as it proved completely portable on all C++ compilers. Dynamic casting would probably be preferable, but the compiler support was marginal at the time of this development.

For implementations which do support pointer access, the functions `getPointerToCoef` and `getPointerToColIndex` provide the means to directly access the matrix coefficients and column indices, respectively. The `const` versions of these functions provide read-only access to the data. The value of the return argument *length* is the number of entries in the specified row.

2.6 Vector base classes

There are two fundamental vectors abstractions represented in ISIS++ v1.1: real-valued (double-precision) and integer-value (int) vectors. A further delineation can be made regarding uni- or multi-processor implementations, but this is abstracted from the fundamental representation. The current implementations available in ISIS++ are discussed in section 5.

The *Vector* class represents the real-valued vector abstraction and, like the Matrix class, is designed to provide the operations necessary to support Krylov subspace iterative methods. Since data access is so much simpler for vectors than matrices, efficient, generalized data access can be provided.

The *IntVector* class represents the integer-valued vector abstraction, and is similar to the Vector class but with a more limited set of mathematical functions. Its primary use within the ISIS++ native implementations is as a container class for indices and cardinalities.

Vector base class public interface

```
class Vector

// default constructor function
Vector(const Map& map);
// default destructor function
virtual ~Vector() {};


// cloning constructor function
virtual Vector* newVector() const = 0;


// mathematical functions
virtual void addVec(double s, const Vector& y) = 0;
virtual double dotProd(const Vector& y) const = 0;
virtual void linComb(const Vector& y, double s, const Vector& z) = 0;
virtual double norm() const = 0;
virtual void put(double s) = 0;
virtual void scale(double s) = 0;
virtual void random(int seed=1) = 0;
virtual double norm1() const = 0;


// assignment operator
Vector& operator=(const Vector& rhs);
```

```
// access functions
virtual double& operator[](int index) = 0;
virtual const double& operator[](int index) const = 0;
const Map& getMap() const;
```

Vector base class reference notes:

- The `newVector` function is critical to the use of vector objects internally in other objects (e.g., solvers and preconditioners). When objects use vectors as internal auto-variables, the cloning facility permits the object to construct vectors from the passed-in prototype so that they are of similar type (and partitioning) as the prototype. This capability allows the abstracted vector types to be used essentially anywhere in a consistent manner.

- The mathematical functions correspond to the following operations, where x is the reference vector, y and z are vectors, and s is a scalar:

`x.addVec(s,y);`	$x_i \leftarrow x_i + s y_i \quad \forall i$
`x.dotProd(y);`	return $\sum_{\forall i} x_i y_i$
`x.linComb(y,s,z);`	$x_i = y_i + s z_i \quad \forall i$
`x.norm();`	return $\left(\sum_{\forall i} x_i^2\right)^{1/2}$
`x.norm1();`	return $\max(|x_i|)$
`x.put(s);`	$x_i = s \quad \forall i$
`x.scale(s);`	$x_i = s x_i \quad \forall i$
`x.random(seed);`	$x_i \in [0,1] \quad \forall i$

- The `operator=` function sets the LHS vector equal to the RHS vector, such as:

`x = y;`	$x_i = y_i \quad \forall i.$

- The `operator[]` functions provide read-only and read/write access to individual vector elements. In general, these functions are slower than direct (pointer-based) data access.

- The `getMap` function returns a `const` reference to the map object used to construct the vector object.

- The `random` function implemented in ISIS++ is relatively simple. It is drawn from Carl Pearson's 1986 book "Numerical Methods in Engineering and Science".

IntVector base class public interface

```
class IntVector

// default constructor function
IntVector(const Map& map);

// default destructor function
virtual ~IntVector() {};
```

24

```
// cloning constructor function
virtual IntVector* newIntVector() const = 0;


// mathematical functions
virtual void put(int scalar) = 0;


// operator= function
IntVector& operator=(const IntVector& rhs);


// access functions
virtual int& operator[](int index) = 0;
virtual const int& operator[](int index) const = 0;
const Map& getMap() const;
```

IntVector base class reference notes:

- The `newVector` function is similar in principle to that of the Vector class.

- The sole mathematical function is used for setting all vector elements to a scalar value. This corresponds to the following operation, where x is a reference integer-valued vector and s is an integer-valued scalar:

  `x.put(s);` $\qquad\qquad x_i = s \qquad \forall i.$

- The `operator=` function sets the LHS vector equal to the RHS vector, such as:

  `x = y;` $\qquad\qquad x_i = y_i \qquad \forall i.$

- The `operator[]` functions provide read-only and read/write access to individual vector elements. In general, these functions are slower than direct (pointer-based) data access.

- The `getMap` function returns a `const` reference to the map object used to construct the vector object.

2.7 LinearEquations class

The *LinearEquations* class binds the matrix, the solution vector, and RHS vector to form a system of linear equations (denoted $Ax = b$). The LinearEquations object provides a point of interaction to initiate and control the solution process, including setting the solver, preconditioner, and scaling functions. Another role of the LinearEquations class is to check for consistency with the associated matrix and vector objects. That is, the matrix and vector types can be compared, and the partitioning can be checked via the Map object (used to construct the matrix and vectors).

We now present the public interface for the LinearEquations class. It is worth noting that unlike the Solver, Preconditioner, Matrix, and Vector base classes LinearEquations objects may be directly constructed (i.e., without derivative implementations).

LinearEquations class

```
class LinearEquations

// constructor function
LinearEquations(Matrix& A, Vector& x, Vector& b);

// default destructor function
virtual ~LinearEquations() {};


// set preconditioner and solver functions
void setPreconditioner(Preconditioner& pc);
void setSolver(Solver& solver);


// invoke solution process
void solve();


// scaling functions
bool rowScale();
bool colScale();
```

LinearEquations class reference notes:

- The `setPreconditioner` and `setSolver` functions set internal pointers to Preconditioner and Solver objects, respectively. The pointers are subsequently used in the `solve` function.

- The `rowScale` and `colScale` functions invoke row and column scaling on the reference system $Ax = b$. These operate through related matrix and vector scaling services, and return *false* when scaling is not supported or otherwise *true*. All rows/columns are scaled according to the maximum absolute value over the corresponding row or column. Hence, the maximum value in a row/column is 1 immediately following row/column scaling.

3 Solver Implementations

The solvers currently implemented in ISIS++ are all Krylov subspace iterative methods. For mathematical background on Krylov methods for linear systems, we refer the reader to Barrett et al. [2], Freund, Golub and Nachtigal [10], Meier-Yang [16] and Tong [21], to name just a few of the many works that exist in this field. In this section we briefly describe the solvers included in the ISIS++ v1.1 framework.

All solver implementations are derived from the *IterativeSolver* base class, and inherit its public interface (see section 2.2). Consequently, the primary interaction with these solvers is defined by the IterativeSolver public interface. Here we present the functions particular to each of the methods. We have omitted the pure virtual function `solve`, which is common to all the solvers and is identical to that of the IterativeSolver class.

In some of the algorithm descriptions that follow, we refer to the initial residual, and the Krylov subspace corresponding to the initial residual. The initial residual is denoted by $r^{(0)} = b - Ax^{(0)}$, where $A \in \Re^{n \times n}$ is the coefficient matrix, $b \in \Re^n$ is the right-hand-side vector, and $x^{(0)} \in \Re^n$ is the initial guess of the solution vector supplied by the user. The Krylov subspace of dimension m corresponding to the initial residual is defined as $K_m(A, r^{(0)}) = span\{r^{(0)}, Ar^{(0)}, A^2 r^{(0)}, ..., A^{m-1} r^{(0)}\}$.

3.1 QMR

The QMR (Quasi-Minimal Residual) algorithm, introduced by Freund and Nachtigal [11], is based on the non-symmetric Lanczos process. It consists of constructing a bi-orthogonal pair of vector sequences, which are the Krylov subspace vectors for the matrix A and for the transpose of A. Several variants of QMR have been developed which add look-ahead techniques to avoid numerical break downs, and which use two- or three-term recursions to construct the vector iterates. Transpose-free variants have also been developed (see Freund [9]), in order to avoid the need to calculate a transpose matrix-vector product since some sparse matrix implementations don't provide that capability. The implementations currently in ISIS++, however, use the transpose product, and employ coupled two-term recurrences without look ahead.

The QMR algorithm may be applied to general linear systems; it requires neither symmetry nor positive-definiteness of the coefficient matrix. The major computational components of this algorithm are the two matrix-vector products (one of them a transpose), some vector updates and two vector dot products per iteration. In terms of memory requirements, it uses about 15 internal vectors in addition to the matrix and two vectors that are passed in from the calling program. No special control parameters are used for the QMR algorithm. Hence the constructor and destructor functions are all that are required from the user's point of view.

At present, there are two variants of QMR implemented in ISIS++. The QMR_Solver class is based on Freund and Nachtigal [11]. The QMR2_Solver is a variant by Buecker and Sauren [3] which reduces the number of global synchronization points to improve scalability. Our experience indicates that QMR is slightly more numerically stable than QMR2. We are investigating the addition of look-ahead mechanisms.

QMR_Solver class public interface

```
class QMR_Solver : public IterativeSolver

// constructor function
QMR_Solver();
// destructor function
virtual ~QMR_Solver() {};
```

QMR2_Solver class public interface

```
class QMR2_Solver : public IterativeSolver

// constructor function
QMR2_Solver();
// destructor function
virtual ~QMR2_Solver() {};
```

3.2 GMRES(m)

The Generalized Minimum Residual (GMRES) algorithm was introduced by Saad and Schultz [19]. It is based on the Arnoldi algorithm for reducing a general matrix to upper Hessenberg form. Basically, the Hessenberg matrix is a restriction of the coefficient matrix A onto the Krylov subspace corresponding to the initial residual. The approximate solution is obtained by minimizing the residual on the Krylov subspace and then projecting it back onto the space corresponding to A using the basis vectors that were produced by the Arnoldi process. This requires that all basis vectors be stored, so that the memory requirements increase linearly with the iteration count. The computational cost per iteration also increases linearly, since each new basis vector must be orthogonalized against all previous ones. In order to avoid prohibitive memory and computational costs, the algorithm is *restarted* periodically, at which point the dimension of the Krylov subspace is reset to 1 and the approximate solution is used for the initial guess in the next cycle. The optimal number of iterations to perform between restarts is problem-dependent, and it represents a compromise between memory and computational costs, and rate of convergence. In general, a smaller restart value causes poorer convergence behavior, and can in fact lead to a stall situation in some cases.

GMRES(m) can be applied to general linear systems, requiring neither symmetry nor positive definiteness. The restart value (m) is the only special control parameter used for the GMRES(m) algorithm. Functions are provided to query and set the restart value, which is by default set to 100 upon construction.

GMRES_Solver class public interface

```
class GMRES_Solver : public IterativeSolver

// constructor function
GMRES_Solver(int m);
// destructor function
virtual ~GMRES_Solver() {};


// restart interval
int m() const;                      // get restart interval
void m(int m);                      // set restart interval
```

3.3 FGMRES(m)

The FGMRES algorithm is a Krylov subspace method which is described in detail in Y. Saad [18]. It is a right-preconditioned version of GMRES, which allows the preconditioner to vary at each iteration. For example, other iterative solvers can be used

28

as preconditioners. At this time, preconditioners are being constructed for ISIS++ that will exploit this capability.

FGMRES_Solver class public interface

```
class FGMRES_Solver : public IterativeSolver

// constructor function
FGMRES_Solver(int m);
// destructor function
virtual ~FGMRES_Solver() {};


// restart interval
int m() const;                          // get restart interval
void m(int m);                          // set restart interval
```

3.4 DefGMRES(m)

DefGMRES(m) (Deflated GMRES(m)) is a modification of GMRES(m), based on an algorithm introduced by Erhel et al. [8]. When GMRES is restarted, the Krylov subspace that has been constructed is discarded. Contained in the Krylov subspace, is information about the extremal eigenvalues and eigenvectors of the coefficient matrix A, which are important to the convergence of the algorithm. Discarding this information is the reason why restarting harms the convergence of GMRES. The idea of Erhel et al. is to save some of this eigenvalue and eigenvector information (through deflation) and then apply it as a preconditioner after the restart, thus enhancing the convergence of the restarted algorithm. The implementation used in ISIS++ is a variant of the above idea, whereby the information saved by deflation is applied in addition to any arbitrarily chosen preconditioner passed in by the user. Our experience indicates that DefGMRES(m) can provide significant performance gains for many problems[8].

DefGMRES(m) has the same applicability as ordinary GMRES(m), requiring neither symmetry nor positive definiteness. However, the benefits provided by the deflation strategy will vary from case to case. The restart value (m) is the only special control parameter used for the deflated GMRES(m) algorithm. Functions are provided to query and set the restart value, which is by default set to 100 upon construction.

DefGMRES_Solver class public interface

```
class DefGMRES_Solver : public IterativeSolver

// constructor function
DefGMRES_Solver(int m);
// destructor function
virtual ~DefGMRES_Solver() {};
```

---

[8] Considering both qualitative convergence and wall-clock time.

29

```
// restart interval
int m() const;                          // get restart interval
void m(int m);                          // set restart interval
```

3.5 BiCGStab

The Bi-Conjugate Gradient Stabilized (BiCGStab) algorithm is another Lanczos-based algorithm, which is closely related to the Conjugate Gradient (CG) algorithm. It produces two mutually orthogonal vector sequences. For more detail regarding this algorithm see Barrett et al. [2].

BiCGStab may be applied to general linear systems, and has a computational cost of two matrix-vector products and four inner products per iteration. No special control parameters are used for the BiCGStab algorithm. Hence the constructor and destructor functions are all that are required from the user's point of view.

BiCGStab_Solver class public interface

```
class BiCGStab_Solver : public IterativeSolver

// constructor function
BiCGStab_Solver();
// destructor function
virtual ~BiCGStab_Solver() {};
```

3.6 CGS

The Conjugate Gradient Squared (CGS) algorithm was described by Sonneveld [20]. It is applicable to non-symmetric linear systems, but has highly irregular convergence behavior. Barrett et al. [2] state that it tends to diverge when the initial guess is close to the solution.

No special control parameters are used for the CGS algorithm. Hence the constructor and destructor functions are all that are required from the user's point of view.

CGS_Solver class public interface

```
class CGS_Solver : public IterativeSolver

// constructor function
CGS_Solver(int m);
// destructor function
virtual ~CGS_Solver() {};
```

3.7 CG

The Conjugate Gradient (CG) algorithm, due to Hestenes and Stiefel [13], is the oldest and most well known of the Krylov subspace methods for linear systems. Like all of the other Krylov methods, it constructs the approximate solution vector as a linear

combination of the orthogonal basis vectors for the Krylov subspace generated from the initial residual. It is closely related to the Lanczos method for symmetric matrices.

The CG algorithm is only guaranteed to converge for linear systems with symmetric, positive definite (SPD) matrices (though this sufficiency condition for convergence is useful only in the theoretical setting where round-off error is absent: hence this guarantee is of limited utility in practice, as ill-conditioned matrices may result in extremely slow convergence rates in the absence of an effective preconditioner). Computationally, it requires one matrix-vector product and two vector inner products per iteration. Due to its reduced operation count, it is an excellent choice when the linear system is SPD. No special control parameters are used for the CG algorithm. Hence the constructor and destructor functions are all that are required from the user's point of view.

CG_Solver class public interface

```
class CG_Solver : public IterativeSolver

// constructor function
CG_Solver();

// destructor function
virtual ~CG_Solver() {};
```

3.8 CGNE

Conjugate Gradients on the Normal equations to minimize the Error (CGNE) is a simple variant of CG which allows the solution of systems with non-symmetric matrices (see Barrett et al. [2], Kelly [15] and Nachtigal, Reddy and Trefethen [17], among others). The idea is to apply the method of conjugate gradients to the linear system $AA^T y = b$, and then set $x = A^T y$ to obtain the solution to the original system $Ax = b$. The disadvantage of this approach is that the condition number of AA^T is the square of that of A.

CGNE may be applied to non-symmetric linear systems, and requires a transpose matrix-vector product in addition to the work performed by ordinary CG at each iteration. No special control parameters are used for the CGNE algorithm. Hence the constructor and destructor functions are all that are required from the user's point of view.

The CGNE solver requires specialized preconditioner implementations. The valid preconditioners have prefix "CGNE_". Since the resulting system is SPD, a few specialized preconditioners are generally sufficient. We have implemented the polynomial and Block Jacobi preconditioning methods for use with CGNE.

CGNE_Solver class public interface

```
class CGNE_Solver : public IterativeSolver

// constructor function
CGNE_Solver();
```

```
// destructor function
virtual ~CGNE_Solver() {};
```

3.9 CGNR

Conjugate Gradients on the Normal equations to minimize the Residual (CGNR) is another variant of CG which allows the solution of systems with non-symmetric matrices (see Barrett et al. [2], Kelley [15] and Nachtigal, Reddy and Trefethen [17], among others). It is similar in principle to CGNE, with the idea being to apply the method of conjugate gradients to the linear system $A^T A x = A^T b$. The resulting solution vector is in principle identical to the solution of $Ax = b$. This approach is also affected by the fact that the condition number of AA^T is the square of that of A. See Kelley [15] for an explanation of the theoretical difference between CGNE and CGNR.

CGNR may be applied to non-symmetric linear systems, and requires a transpose matrix-vector product in addition to the work performed by ordinary CG at each iteration. No special control parameters are used for the CGNR algorithm. Hence the constructor and destructor functions are all that are required from the user's point of view.

Like CGNE, the CGNR solver requires specialized preconditioner implementations. The valid preconditioners have prefix "CGNR_". Since the resulting system is SPD, a few specialized preconditioners are generally sufficient. We have implemented the polynomial preconditioning method for use with CGNR.

CGNR_Solver class public interface

```
class CGNR_Solver : public IterativeSolver

// constructor function
CGNR_Solver();

// destructor function
virtual ~CGNR_Solver() {};
```

4 Preconditioner Implementations

The current implementation of ISIS++ includes a collection of preconditioner implementations, all derived from either the *Preconditioner* or *RowPrecontioner* base class. The basic preconditioning model used in ISIS++ is described in section 2.3. For brevity, we have omitted the interface components common to the base class in the implementation specifications that follow. Detailed descriptions of the preconditioner base classes can be found in section 2.3.

A number of unsupported preconditioners based on other packages (SuperLU, SPARSKIT2, BPKIT, HYPRE, and AMG) are optional in ISIS++. If the user has access to a package, the corresponding preconditioners can be enabled from the configure script used when installing ISIS++. We now present the public interfaces for the preconditioners currently supported in ISIS++.

32

4.1 Identity

The identity preconditioner is provided in ISIS++ to establish a base line for unconditioned systems. The identity preconditioner is essentially a non-operation, whereby the solution vector from the preconditioner "solve" functions simply returns the passed vector. There are no special parameters associated with the identity preconditioner, so the public interface simply provides constructor and destructor functions, as follows.

Identity_PC class public interface

```
class Identity_PC : public Preconditioner

// constructor function
Identity_PC(const Matrix& A);

// destructor function
virtual ~Identity_PC() {};
```

4.2 Diagonal scaling

Diagonal scaling can be viewed as the simplest of the incomplete factorization schemes applied to the matrix A to form an approximation to A^{-1}. The diagonal scaling (Point Jacobi) preconditioner takes the form given by:

$$m_{ij} = \begin{pmatrix} a_{ii} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{pmatrix}$$

where $a_{ij} = (i, j)$ element of the matrix A, and $m_{ij} = (i, j)$ element of the (diagonal) preconditioning matrix M. Since multiplication on the left by a diagonal matrix merely scales the corresponding rows of the coefficient matrix, the preconditioned coefficient matrix A takes the simple form:

$$M^{-1}A = \begin{bmatrix} 1 & a_{11}^{-1}a_{12} & a_{11}^{-1}a_{13} & \cdots & a_{11}^{-1}a_{1n} \\ a_{22}^{-1}a_{21} & 1 & a_{22}^{-1}a_{23} & \cdots & a_{22}^{-1}a_{2n} \\ a_{33}^{-1}a_{31} & a_{33}^{-1}a_{32} & 1 & \cdots & a_{33}^{-1}a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{nn}^{-1}a_{n1} & a_{nn}^{-1}a_{n2} & a_{nn}^{-1}a_{n3} & \cdots & 1 \end{bmatrix}$$

There are many practical advantages to this *diagonal scaling* scheme, and this simple method works remarkably well on many problems. Among its chief useful characteristics are that it is very simple and hence readily implemented, and that it converts all of the diagonal elements to the same sign. This latter feature is helpful when dealing with matrices that have elements of both signs on the diagonal, such as those arising from mixed finite-element analyses involving required implementations of constraint relations (at least, those mixed analyses not characterized by zero diagonal matrix elements, in accordance with the caveats given below). In addition, if the matrix

is sparse and stored in a row-oriented format, then this approach is easy to implement and fast to compute, as the diagonal scaling is applied row-wise.

Unfortunately, there is a practical problem that commonly occurs with Point Jacobi schemes, in that a zero on the diagonal causes numerical difficulties that add complexity to this otherwise simple preconditioning scheme. This pathological case is representative of a general problem where relying *only* on the diagonal entry for scaling information provides insufficient information to construct a good preconditioner. We have currently implemented a heuristic in ISIS++ that sets a zero diagonal to one for the purposes of the diagonal Jacobi preconditioner.

The row scaling operation is closely related to the diagonal preconditioner, whereby the maximum absolute value on each row is used instead of the element on the diagonal. . There is a CGNE variant of this preconditioner in ISIS++.

Diagonal_PC class public interface

```
class Diagonal_PC : public Preconditioner

// constructor function
Diagonal_PC(const Matrix& A);
// destructor function
virtual ~Diagonal_PC() {};
```

4.3 Block Jacobi

Block Jacobi preconditioning is a generalization of the Point Jacobi or diagonal scaling scheme. Block Jacobi creates a block diagonal preconditioning matrix, whose blocks correspond to the coefficient matrix A. Inverting these blocks gives M_1^{-1}, where the right-preconditioning matrix $M_2 = I$. The implementation in ISIS++ currently supports two blocking strategies, namely, allowing the blocks to overlap or not. Interface functions are supplied to allow the user to set and query the block size as well as the blocking strategy.

Non-overlapping or overlapping blocking is controlled by setting *strategy* equal to 1 or 2, respectively, with the `blockStrategy` set function. There is a CGNE variant of this preconditioner in ISIS++.

BlockJacobi_PC class public interface

```
class BlockJacobi_PC : public RowPreconditioner

// constructor function
BlockJacobi_PC(const RowMatrix& A);
// destructor function
virtual ~BlockJacobiPC() {};


// control/access functions
int blockSize();                    // get block size
void blockSize(int size);           // set block size
```

34

```
int blockStrategy();                    // get blocking strategy
void blockStrategy(int strategy);// set blocking strategy
```

4.4 Block LU

The Block LU preconditioner uses the SuperLU factorization package of Demmel et al. [5],[6] to invert exactly the diagonal blocks of a block-wise distributed matrix. In the case where a single processor is being used, this preconditioner is in fact a direct solver. Naturally its performance (in terms of convergence) on a given problem deteriorates rapidly as the number of processors is increased, since there is a corresponding decrease in the size of the diagonal blocks that are being inverted and used to approximate the matrix inverse. Obviously the diagonal blocks must be non-singular (this is not always the case in practice).

Memory overheads are severe for this preconditioner. Since it performs a full exact factorization, there is a lot of fill-in for the factors L and U that are produced and stored internally. A set of ILU preconditioners have been added to ISIS++, but are not officially supported at time of this writing.

Currently, the Block LU preconditioner only works with the BDCRS (see description in the implementation section) matrix class.

BLU_PC class public interface

```
class BLU_PC : public Preconditioner

// constructor function
BLU_PC(const BDCRS_Matrix& A);

// destructor function
virtual ~BLU_PC() {};
```

4.5 Polynomial

In general, a polynomial preconditioner approximates the inverse of the matrix A by constructing a matrix $M_1^{-1} = P_n(A)$, a polynomial in A. The polynomial preconditioner currently implemented in ISIS++ provides two choices for the type of the polynomial: Neumann and Least Squares. The polynomial type and order can be set and queried by the user through public interface functions.

ISIS++ currently supports polynomial orders up to 10. Polynomial type is controlled by setting *type* to 1 or 2 for Neumann or least squares, respectively, with the polyType set function. Variants of this preconditioner are included for use with CGNR and CGNE solvers.

Poly_PC class public interface

```
class Poly_PC : public Preconditioner
```

```
// constructor function
Poly_PC(const Matrix& A, Vector& sample);

// composed preconditioning constructor function
Poly_PC(const Matrix& A, Vector& sample, Preconditioner& PC1);

// destructor function
virtual ~Poly_PC() {};


// control/access functions
int polyOrder();                    // get the polynomial order
void polyOrder(int order);          // set the polynomial order
int polyType();                     // get the polynomial type
void polyType(int type);            // set the polynomial type
virtual bool isMatrixFree();        // TRUE if preconditioner is matrix free
```

4.6 SPAI

The SPAI (SParse Approximate Inverse) preconditioner is an incomplete factorization method which explicitly calculates and stores the approximate inverse matrix. It uses an algorithm that is due to Grote and Huckle [12] and was implemented by Barnard (see Barnard and Clay [1]). It has a lengthy calculation phase, but it is fully parallel and produces dramatic improvements in convergence for most problems. Unlike the Block LU scheme, it doesn't restrict its attention to the diagonal blocks of the matrix. Thus, the convergence performance does not degrade when using many processors.

SPAI has several control parameters which are set and queried by interface functions. Its public interface is given below.

SPAI_PC class public interface

```
class SPAI_PC : public RowPreconditioner

// constructor function
SPAI_PC(const RowMatrix& A);

// destructor function
virtual ~SPAI_PC() {};


//functions for setting/querying parameters
double spai_epsilon() const;        // get epsilon
void spai_epsilon(double epsilon);  // set epsilon
int spai_nbsteps() const;           // get number of steps
void spai_nbsteps(int nbsteps);     // set number of steps
int spai_maxapi() const;            // get maxapi
void spai_maxapi(int maxapi);       // set maxapi
int spai_maxnew() const;            // get maxnew
void spai_maxnew(int maxnew);       // set maxnew
int spai_max() const;               // get max
void spai_max(int max);             // set max
int spai_cache_size() const;        // get cache size
void spai_cache_size(int cache_size);   // set cache size
```

```
int spai_info() const;            // get info
void spai_info(int info);         // set info
```

Notes on SPAI_PC parameter control functions:

- The term *epsilon* is a convergence tolerance relating to the Froebenius norm, influencing the accuracy of the approximate inverse in a trade-off with memory requirements and calculation speed. A smaller value of *epsilon* implies a more accurate approximate inverse.

- The term *nbsteps* is the maximum number of "improvement" steps per column.

- The term *maxapi* is the maximum number of non-zeros in a row or column of the preconditioning matrix M.

- The term *maxnew* is the maximum number of new entries per "improvement" step.

- The term *max* is the maximum dimension of the QR subproblems.

- The term *cache_size* is used to control local row caching. The values have the following mapping:
  0 - 101
  1 – 503
  2 - 2503
  3 - 12503
  4 - 62501.
  Values of 3 or 4 are recommended as a starting point, although the optimal values are problem dependent.

- The term *info* is used to control output specific to the SPAI algorithm. Setting *info* to 0 will disable SPAI output, while setting it to 1 will enable output.

5 Matrix/Vector Implementations

ISIS++ was designed for scalable, distributed-memory computations. However, some people are also interested in running in serial mode, particularly on platforms which do not support the MPI message-passing library. In response to this need, we have implemented mirror image serial versions of the statically and dynamically sized native ISIS++ matrix classes.

The static-size implementations are generally faster, since they have more latitude as regards the underlying data structures, and hence can be more fully optimized for performance. The resizable matrix classes are based on storing and processing one row at a time, where each row's data is stored in contiguous memory. Rows are however, not necessarily contiguous with each other, and hence the resizable matrix is effectively a collection of rows, each of which may be resized.

What follows is a description of the matrix/vector implementations currently in ISIS++. Most matrix implementations are variations on the standard CRS (Compressed

Row Storage) sparse matrix format, delineated by serial/distributed-memory data and static/dynamic sizing properties, blocked data distribution, etc. An exception to this is the Aztec [14] DMSR matrix/vector class. For each matrix class we give an annotated header showing the functions provided by that class. In general though, we omit the interface components common to the base matrix class from which the class is derived. The "native" vector implementations (Seq_Vector and Dist_Vector) can be used with multiple matrix implementations, with only the AztecDMSR matrix class requiring its own vector class.

5.1 Sequential vector classes

The native sequential vector class implementation Seq_Vector is a direct implementation of the Vector base class (see section 2.6), with an added copy constructor. The default is to use the Fortran BLAS routines for the internal computations. An alternate version which doesn't rely on this library is also available.

Similarly, the Seq_IntVector class is a direct implementation of the IntVector class, specialized for a serial computing model and with a copy constructor. Both of the serial vector classes are designed to inter-operate efficiently with the native serial matrix implementations (i.e., SCRS_Matrix and RsSCRS_Matrix classes). In both vector classes, the copy constructor creates an exact clone of the original vector.

Seq_Vector class public interface

```
class Seq_Vector: public Vector

// default constructor function
Seq_Vector(const Map& map);
// copy constructor function
Seq_Vector(const Seq_Vector& source);
// default destructor function
virtual ~Seq_Vector() {};
```

Seq_IntVector class public interface

```
class Seq_IntVector: public IntVector

// default constructor function
Seq_IntVector(const Map& map);
// copy constructor function
Seq_IntVector(const Seq_IntVector& source);
// default destructor function
virtual ~Seq_IntVector () {};
```

5.2 Sequential static-size matrix class

The SCRS_Matrix matrix class is a CRS (Barrett et al. [2]) row matrix abstraction specialized for serial computing. That is, the underlying model for the implementation is uni-processor, global memory. The primary advantages inherent with this specialization

include performance and simplicity related to replacing MPI-based message passing with a global-memory model. Further, the data is contiguous in memory to fully support the direct pointer access functions.

Unless otherwise shown, the RowMatrix public interface is replicated within the SCRS_Matrix class. The reader is referred to section 2.5 for the complete specification of the RowMatrix base class public interface.

As with all the native matrix implementations, the Map object is based on a virtual global coefficient mapping numbered from 1 to n, the characteristic size of the system. The simple Map constructor `map(n)` is designed for the serial case.

The `configure` function must be called to allocate memory before any data can be loaded into the matrix.

The function `setRowLength` returns *false*, since dynamic row resizing is not supported for this implementation.

SCRS_Matrix class public interface

```
class SCRS_Matrix : public RowMatrix

// constructor function
SCRS_Matrix(const Map& map);
// destructor function
virtual ~SCRS_Matrix() {};


// information functions
int nonZeros();                        // return number of nonzeros in matrix
// direct pointer data access functions
bool pointerAccess() {return true;};
```

5.3 Sequential re-sizable matrix class

The RsSCRS_Matrix re-sizable matrix class is a CRS (Barrett et al.[2]) row matrix abstraction specialized for serial computing. It essentially duplicates the functionality of the static equivalent (see preceding), but fully enables the function `setRowLength`. The direct pointer access functions are fully supported.

While the two serial matrix classes share near-identical functionality, we note that each are completely different implementations. In particular, the SCRS_Matrix implementation uses one contiguous block of memory for the matrix values and similarly for the column indices. This data/memory configuration is extremely inefficient for row resizing, to the extent that we make no attempt to support it. In order to support dynamic row resizing, the RsSCRS_Matrix effectively treats a matrix as a collection of rows, each of which is contiguous in memory but may be disjoint from other rows. As a consequence of this data/memory layout, each row may be resized without affecting other rows, and hence is relatively efficient for row resizing operations. Furthermore, there is no need to pre-configure the matrix before loading data into it. Coefficients may be

inserted using the `putRow` function, and the appropriate rows will be adjusted as necessary.

RsSCRS_Matrix class public interface

```
class RsSCRS_Matrix : public RowMatrix

// constructor function
RsSCRS_Matrix(const Map& map);
// destructor function
virtual ~RsSCRS_Matrix() {};


// set row length -- fully implemented, return value = true
virtual bool setRowLength(int length, int rowNumber);
// information functions
int nonZeros();                         // return number of nonzeros in matrix
// direct pointer data access functions
bool pointerAccess() {return true;};
```

5.4 Distributed-memory vector classes

The "native" distributed-memory vector class implementation Dist_Vector is a direct implementation of the Vector base class (see section 2.6), with an added copy constructor. As with all the distributed-memory components in ISIS++, the MPI message-passing library is used for communications. The default is to use the Fortran BLAS routines for the internal computations. An alternate version which doesn't rely on that library is also available.

Similarly, the Dist_IntVector class is a direct implementation of the IntVector class, specialized for a distributed-memory computing model and with a copy constructor. Both of the distributed-memory vector classes are designed to inter-operate efficiently with the native distributed-memory matrix implementations (i.e., DCRS_Matrix and RsDCRS_Matrix classes). In both vector classes, the copy constructor creates an exact clone of the original vector.

Dist_Vector class public interface

```
class Dist_Vector: public Vector

// default constructor function
Dist_Vector(const Map& map);
// copy constructor function
Dist_Vector(const Dist_Vector& source);
// default destructor function
virtual ~Dist_Vector() {};
```

Dist_IntVector class public interface

```
class Dist_IntVector: public IntVector
```

```
// default constructor function
Dist_IntVector(const Map& map);
// copy constructor function
Dist_IntVector(const Dist_IntVector& source);
// default destructor function
virtual ~Dist_IntVector () {};
```

5.5 Distributed-memory static-size matrix class

The DCRS_Matrix matrix class is a CRS (Barrett et al. [2]) row matrix abstraction specialized for distributed-memory computing, utilizing the MPI message-passing library for communications. The matrix data is in contiguous memory to fully support the direct pointer access functions.

As with all the "native" matrix implementations, the map object is based on a virtual global equation mapping numbered from 1 to n, the characteristic size of the system. The parallel Map constructor (see section 2.4) is designed for this case. As with the sequential static-size matrix class, memory must be pre-allocated by calling the configure function before any data may be loaded.

The function setRowLength returns *false*, since dynamic row resizing is not supported for this implementation.

DCRS_Matrix class public interface

```
class DCRS_Matrix : public RowMatrix

// constructor function
DCRS_Matrix(const Map& map);
// destructor function
virtual ~DCRS_Matrix() {};


// direct pointer data access functions
bool pointerAccess() {return true;};
```

5.6 Distributed-memory re-sizable matrix class

The RsDCRS_Matrix re-sizable matrix class is a mirror image of the RsSCRS_Matrix class, but specialized for distributed-memory computing, utilizing the MPI message-passing library for communications. The RsDCRS_Matrix class essentially duplicates the functionality of the static equivalent (see preceding), but fully enables the function setRowLength. The direct pointer access functions are fully supported. Also, it is not necessary to pre-allocate memory for this matrix by calling the configure function.

While the two "native" distributed-memory matrix classes share near-identical functionality, each are completely different implementations. In particular, the DCRS_Matrix implementation uses one contiguous block of memory for the "local" matrix values and similarly for the column indices. In order to support row resizing, the

41

RsDCRS_Matrix effectively treats a matrix (or sub-matrix when partitioned row-wise) as a collection of rows, each of which is contiguous in memory but may be disjoint from other rows. As a consequence of this data/memory layout, each row may be resized relatively efficiently without affecting other rows.

RsDCRS_Matrix class public interface

```
class RsDCRS_Matrix : public RowMatrix

// constructor function
RsDCRS_Matrix(const Map& map);
// destructor function
virtual ~RsDCRS_Matrix() {};


// set row length -- fully implemented, return value = true
virtual bool setRowLength(int length, int rowNumber);


// direct pointer data access functions
bool pointerAccess() {return true;};
```

5.7 Block DCRS matrix class

The BDCRS_Matrix class is intended primarily for the multiple processor case, and distributes the matrix data block-wise in 2 dimensions so that the global matrix consists of $p \times p$ (where p is the number of processors) sub-matrices. Each processor owns a row of sub-matrices. This implementation is not derived from the RowMatrix base class, but rather from the Matrix base class. While there is no provision for getting a pointer to a row, it is possible to get a pointer to a sub-block which is itself a CRS_Matrix object. The CRS_Matrix class, which will be described in the next section, provides many of the capabilities of the sequential CRS matrix classes described earlier.

The BDCRS_Matrix class can not be pre-configured. It is loaded with data using the putRow function, which temporarily stores the data in re-sizable sub-blocks. When all of the data has been loaded (and the fillComplete function is called), it is transferred into static size CRS_Matrix blocks. A form of direct pointer access to the data is available by first getting a pointer to a sub-block of the matrix, and then getting a pointer to that block's coefficients.

BDCRS_Matrix class public interface

```
class BDCRS_Matrix : public Matrix

// constructor function
BDCRS_Matrix(const Map& map);
// destructor function
virtual ~BDCRS_Matrix();


// initialization function
void put(double s);
```

```
//access functions
virtual void getDiagonal(Vector& diagVector);
void putRow(int row, int length, double *coef, int *colInd);


// direct sub-block access function
CRS_Matrix* getBlockPtr(int blockNumber);
```

5.8 CRS and RsCRS matrix classes

The CRS_Matrix and RsCRS_Matrix classes are essentially stripped-down versions of the previously described SCRS and RsSCRS matrix classes. They are not part of the Matrix/RowMatrix hierarchy, require no map at instantiation, and are purely sequential. They were specially created for use as sub-blocks of the BDCRS format (which was described in the previous section). Their functionality mirrors that of the SCRS and RsSCRS matrices, with the main difference lying in the constructor functions. The other difference, in the case of the CRS_Matrix class, is that the getPointerToCoef and getPointerToColIndex functions are overloaded to simply return pointers to the beginning of the data block as well as to a particular row.

The CRS_Matrix class also has a specialized function copyToCCS which copies its contents into a CCS_Matrix object.

We now give annotated partial headers for these two classes, showing only those functions which differ from the SCRS and RsSCRS matrix classes, respectively. All other functions are identical.

CRS_Matrix class public interface

```
class CRS_Matrix
// constructor function
CRS_Matrix(int rows, int cols, int nnz);


// access functions
double* getPointerToCoef(int& nnz);
int* getPointerToColIndex(int& nnz);


// conversion to CCS storage
void copyToCCS(CCS_Matrix **B);


//inquiry functions
int rows();
int columns();
int nonZeros();
```

RsCRS_Matrix class public interface

```
class RsCRS_Matrix
// constructor function
RsCRS_Matrix(int rows, int cols);


// access functions
double* getPointerToCoef(int& nnz);
int* getPointerToColIndex(int& nnz);


//inquiry functions
int rows();
int columns();
int nonZeros();
```

5.9 CCS matrix class

The CCS (Compressed Column Storage) matrix is a column-oriented equivalent to the CRS matrix described above. It is also intended for use as a local sub-block of a block-wise distributed matrix. It was created for use inside the Block LU preconditioner, since the internal algorithm in Block LU requires a column-oriented matrix. As with the CRS matrix, no map is required at instantiation. Its functionality mirrors that of the CRS matrix, but with operations being column-oriented.

Those functions which are different are shown in the public interface below.

CCS_Matrix class public interface

```
class CCS_Matrix
// constructor function
CCS_Matrix(int rows, int cols, int nnz);


// access functions
void getColSum(Vector& colSumVector);
int colLength(int col);
int getCol(int col, int length, double *coef, int &rowInd);
int getCol(int col, int length, double *coef);
int getCol(int col, int length, int &rowInd);
int putCol(int col, int cardinality, double *coef, int *rowInd);
int* getPointerToRowIndex(int& nnz);
int* getPointerToRowIndex(int& length, int colNumber);
int* getPointerToColPtr(int& nnz);


//inquiry functions
int rows();
int columns();
int nonZeros();


// min/max functions
```

```
bool colMax() const;
double colMax(int col) const;
bool colMin() const;
double colMin() const;
```

5.10 Aztec DMSR matrix/vector and map classes

The AztecDMSR_Matrix class is simply a wrapper which encapsulates the DMSR matrix storage format from the Aztec package and allows it to function as an ISIS++ matrix class, working with other ISIS++ components such as solvers, preconditioners, etc. This allows the core computational kernels (matrix-vector product) to be used by an ISIS++ Krylov solver, for instance. Details of the DMSR storage format may be found in the Aztec documentation [14]. No Aztec source code is included in the ISIS++ code distribution. The user is responsible for ensuring that a copy of the Aztec library is available to be linked against. Additionally, the Aztec header file must be available, and needs to be slightly modified. In the file "az_aztec.h", all function prototypes which are declared as "extern ..." need to be declared as "extern "C" ...".

From the user's point of view, the AztecDMSR matrix class behaves similarly to the DCRS class, inheriting most of the RowMatrix base class functionality. The most significant difference is that the Aztec data decomposition doesn't require that each processor own contiguous blocks of rows. Instead, the mapping is defined by an arbitrary lists of rows. Obviously, contiguous blocks of rows are still a possibility, with a linear decomposition being supplied by default. A specialized Map class, the Aztec_Map (described below) must be used at construction. The direct pointer access functions (getPointerToCoef and getPointerToColIndex) are not supported. After the matrix has been configured, data may be loaded using the putCoef function. Additionally, these matrices must be used in conjunction with a specialized Vector class, the Aztec_Vector (also described below).

Below is a partial annotated header for the AztecDMSR class, again showing only those functions differing from other RowMatrix subclasses.

AztecDMSR_Matrix class public interface

```
class AztecDMSR_Matrix : public RowMatrix

// constructor function
AztecDMSR_Matrix(const Aztec_Map& map);


//modified configure function
void configure(int **rowCount);


//data load function
void putCoef(int row, int col, double value);
```

The `configure` function takes an "int **" as an argument instead of the IntVector accepted by previously described implementations. The reason for this is that there is no IntVector implementation that is compatible with the AztecDMSR data structures. In this case *rowCount* is still a simple (single-dimensional) array of row lengths as before, but it is declared as a double pointer since it can be allocated and initialized inside other functions.

Although the AztecDMSR storage format and data decomposition is different to those used by other ISIS++ matrix classes, the user doesn't see any difference, for the most part. The primary point at which the differences affect the user are in the map object, Aztec_Map. Below is an annotated header for the Aztec_Map class.

Aztec_Map class public interface

```
class Aztec_Map : public Map

// constructor functions
Aztec_Map(int n, const CommInfo& commInfo);
Aztec_Map(int n, int **update, int N_update, const CommInfo& commInfo);


//query and data mapping access functions
int inUpdate(int globalIndex, int& localIndex) const;
int **getUpdate() const;
int **getUpdateOrdering() const;
int **getOrderingUpdate() const;
int **getExternal() const;
int **getExternIndex() const;
int *getProcConfig() const;
const int* getN_update() const;
int **getDataOrg() const;
```

Aztec_Map reference notes:

- There are two ways to construct the Aztec_Map. If only the overall dimension *n* and a CommInfo object are supplied, then the AZ_linear option is used internally to form a linear (contiguous blocks of rows) decomposition. Alternatively, the user can supply the decomposition in the form of a list of local rows or an "update set" in the array *update*, and the number of local rows *N_update*.

- The `inUpdate` function determines whether the row with global number *globalIndex* is in this processor's local update set. If it is, `inUpdate` has a return value of 1 and that row's local index is returned in *localIndex*. If row *globalIndex* is not in the local update set, `inUpdate` has a return value of 0.

- The `getUpdate` function returns a pointer to the list of global row numbers which make up the local update set.

46

- The `getUpdateOrdering` function returns a pointer to the list of local indices which reflects how the local rows were re-ordered by the internal function `AZ_transform` when the `fillComplete` function was called.

- The `getOrderingUpdate` function returns a pointer to the list of local indices which maps back from the reordered local row numbers to the original ordering: i.e., the inverse of the list returned by the `getUpdateOrdering` function.

- The `getExternal` function returns a pointer to this processor's list of external rows, or rows from which information is needed for local calculations.

- The `getExternIndex` function returns a pointer to the list which gives the local numbering (after being reordered) of this processor's external rows.

- The `getN_update` function returns a pointer to the integer which is the number of rows in the local update set. This is the Aztec equivalent of `numLocalRows`.

- The `getProcConfig` and `getDataOrg` functions are primarily used internally by `AztecDMSR_Marix` and `Aztec_Vector` functions. They return arrays which store information about the processor configuration and the data organization, respectively. For detailed information, see the `proc_config` and `data_org` descriptions in the Aztec documentation [14].

As an example of how some of the Aztec_Map variables relate to each other, consider the following declarations.

```
const int **update = map.getUpdate();
const int **updateOrdering = map.getUpdateOrdering();
const int **orderingUpdate = map.getOrderingUpdate();
const int **external = map.getExternal();
const int **externIndex = map.getExternIndex();
```

Then `*update` contains a sorted list of (global) row numbers to be updated on this processor. Before `fillComplete()` has been called (i.e., before the internal matrix data has been re-ordered), the following relation holds: if a row's local index *i* is known, `(*update)[i]` gives that row's global row number. The mapping arrays `updateOrdering` and `orderingUpdate` are available only after `fillComplete()` has been called. `(*updateOrdering)[i]` gives the local index of global row `(*update)[i]`. If only the local (reordered) index *i* is known, then we can use the relation *j* = `(*orderingUpdate)[i]` and then the global row number is `(*update)[j]`. For the external rows, `*external` contains a sorted list of this processor's external rows (global row numbers), and `(*externIndex)[i]` gives the local (reordered) index of `(*external)[i]`.

As mentioned, the AztecDMSR_Matrix class must be used with the Aztec_Vector class. This is simply because the other ISIS++ vector classes can't be instantiated with the Aztec_Map and must have data corresponding to a contiguous block decomposition. In the public interface given below for the Aztec_Vector class, only the constructor is

shown, because this is the only way in which it differs from the other vector classes from the user's point of view. All other differences are internal.

Aztec_Vector class public interface

```
class Aztec_Vector : public Vector

// constructor function
Aztec_Vector(const Aztec_Map& map);
```
---

6 Example Problem

A simple example problem is presented in this section as a concrete example of the process of calling the ISIS++ package from an analysis program. This problem is solved using the algebraic interface. A finite element interface has been developed, and is addressed in a separate document. Information on the finite element interface can be found at URL http://z.sandia.gov/fei. This example problem utilizes a two-dimensional heat conduction problem to generate a set of finite-difference equations that are solved by ISIS++. The entire process of mathematical modeling, linear equation construction, and solution is presented in sufficient detail so that this example can serve as an intermediary to more complicated problems to be solved using ISIS++.

This section demonstrates the entire process of "solving a problem", starting with the underlying mathematical statement, proceeding through the discretization process that results in a system of linear algebraic equations, and terminating in the solution of those simultaneous relations. Also, the example problem is sufficiently "generic" so that it can be readily modified by the user to handle large or small equations, or a whole range of solution response, ranging from smooth to singular.

6.1 Problem statement

The example problem represents heat conduction on a rectangular domain, which is a class of problems that includes many other important engineering and scientific analyses, such as steady-state diffusion, dispersion, electrical conduction, and membrane displacement. This problem admits relatively simple discretizations (such as the finite-difference scheme presented below), but can be easily generalized to more complex discretization schemes.

The geometry of the problem is a rectangular plate lying in the x-y plane, as shown in the figure below. The rectangle has dimensions of a and b, and its natural directions are aligned with the coordinate directions as shown. On the perimeter of the rectangle, the temperature field vanishes, and within the interior, steady-state heat conduction occurs with a balance of conductance of heat energy within the plate, lateral convection from the plate's area (in the direction perpendicular to the page), and arising from sources within the plate.
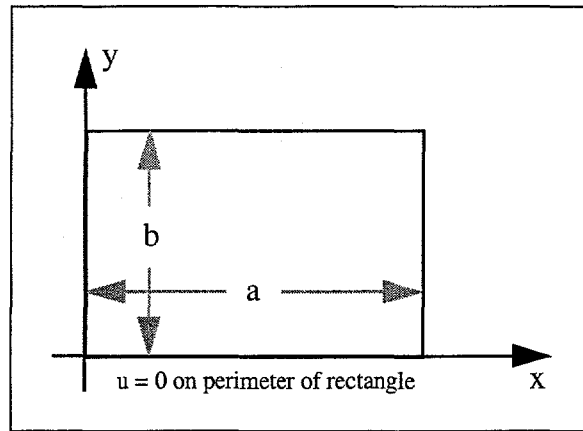
Figure 3. Geometry of Sample Problem

The balance of heat energy can be made precise by introducing appropriate definitions. First, define $u(x,y)$ to be the temperature distribution within the rectangular plate. Let $c(x,y)$ represent the thermal conductivity of the plate, which is taken to be a scalar because the plate is composed of an isotropic material (in general, the thermal conductivity is a second-rank tensor, but that complicates the sample problem in a manner not germane to the demonstration purposes desired here). Take $p(x,y)$ to represent the surface convection coefficient, which captures the (linearized) temperature-dependent transfer of heat from the top and bottom of the plate. Finally, let $s(x,y)$ represent the sources and sinks distributed throughout the plate, with a positive sense representing a source of heat energy.

With these definitions, the governing boundary-value problem (BVP) for heat conduction in the plate is given by the partial-differential equation (PDE) and boundary-conditions (BC's) defined by:

$$-\left[\frac{\partial}{\partial x}\left(c(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(c(x,y)\frac{\partial u}{\partial y}\right)\right] + p(x,y)u(x,y) = s(x,y)$$

$u(0,y) = u(a,y) = 0$ for $0 < y < b$ and $u(x,0) = u(x,b) = 0$ for $0 < x < a$

In order to simplify the example problem development, take each of the material properties to be a constant, so that a uniform isotropic problem is modeled, in that:

$c(x,y) = c_o$ = constant, $p(x,y) = p_o$ = constant, $s(x,y) = s_o$ = constant

With this simplification, the BVP takes the following form, which is amenable to an elementary finite-difference discretization:

$$-c_o\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + p_o u(x,y) = s_o$$

$u(0,y) = u(a,y) = 0$ for $0 < y < b$ and $u(x,0) = u(x,b) = 0$ for $0 < x < a$

6.2 Derivation of equation set

The governing BVP can be discretized by introducing standard difference relations to replace the partial differential operators, and the resulting discrete problem is readily cast into the form of a system of linear equations. The process of constructing this set of equations begins by constructing a grid of difference nodes, and then introducing a standard finite-difference approximation for the Laplacian operator on each of these nodes.

The discretization geometry is shown in the figure below.
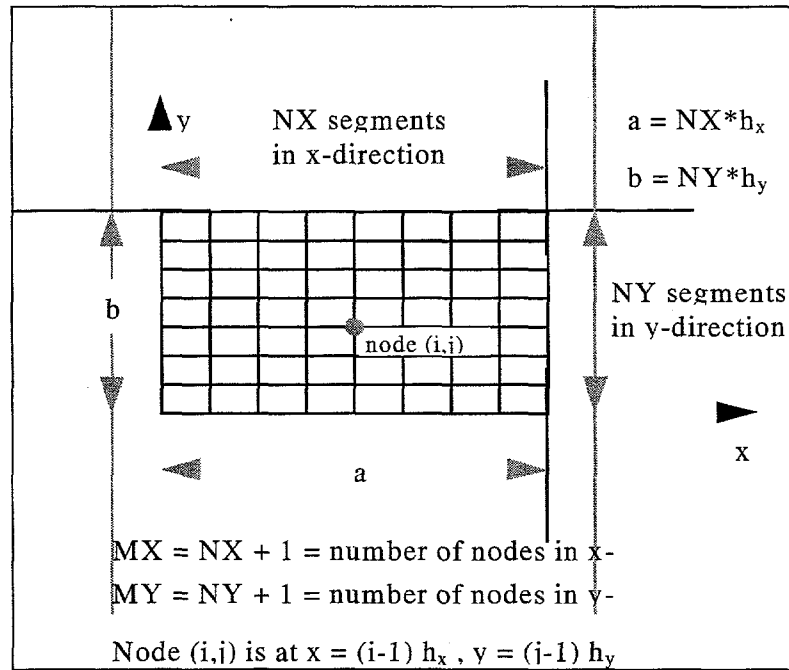
Figure 4. Geometry of Discretization

There are two classes of nodes present in the discretization: interior nodes, where a difference relation for the Laplacian operator can be written, and exterior nodes, where the problem's boundary conditions must be satisfied. In either case, an independent mathematical relation can be written for each node, which results in a N × N system of linear relations that can be solved using solution services provided by ISIS++.

In the case of interior nodes, the Laplacian difference relation is given by:

$$\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)_{node(i,j)} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\left(h_x \right)^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\left(h_y \right)^2}$$

On the exterior, the boundary condition specification is given by:

$$u_{i,1} = u_{i,MY} = 0 \qquad u_{1,j} = u_{MX,j} = 0$$

50

A vector storage format can be used to represent all of the nodal temperature values by introducing a single subscript k defined by:

$$k = i(MY) + j + 1 \quad \text{for } i = 0, 1, 2, \dots NX; \ j = 0, 1, 2, \dots NY$$

This culminates in the following set of linear algebraic relations:

Left Edge:

$$u_k = 0 \quad 1 \le k \le MY$$

Right Edge:

$$u_k = 0 \quad MX * NY + 1 \le k \le MX * MY$$

Top Edge:

$$u_k = 0 \quad k \bmod MY = 0$$

Bottom Edge:

$$u_k = 0 \quad k \bmod MY = 1$$

Interior Nodes:

$$-u_{k_1} h_y^{\,2} - u_{k_2} h_x^{\,2} + u_{k_3}\left[2\!\left(h_x^{\,2} + h_y^{\,2}\right) + \frac{h_x^{\,2} h_y^{\,2} p_o}{c_o} \right] - u_{k_4} h_x^{\,2} - u_{k_5} h_y^{\,2} = \frac{h_x^{\,2} h_y^{\,2} s_o}{c_o}$$

where

$$k_1 = (i-1)*MY + j + 1$$
$$k_2 = i*MY + j$$
$$k_3 = k = i*MY + j + 1$$
$$k_4 = i*MY + j + 2$$
$$k_5 = (i+1)*MY + j + 1$$

In practice, the matrix would be populated by looping over all the *(i,j)* nodes using a program control structure such as the following:

```
for (i = 0 to NX)
    for (j = 0 to NY)
        k = i*MY + j + 1
        case:
            left edge node
                generate simple equation for left edge
            right edge node
                generate simple equation for right edge
            top edge node
                generate simple equation for top edge
            bottom edge node
                generate simple equation for top edge
            interior node
                generate complicated difference equation
```

The structure of the resulting system of linear equations is pictured below, for the case of NX = NY = 10. Note that because of the simple manner in which the boundary

51

conditions are implemented, the matrix is not symmetric (e.g., examine the initial rows and columns). It is relatively straightforward to symmetrize matrices arising from self-adjoint differential and symmetric difference relations, but since ISIS++ is capable of solving non-symmetric systems of equations, no attempt at symmetrizing these difference relations will be made.
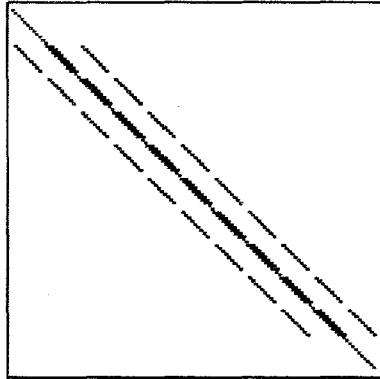

Figure 5. Sample Problem Matrix Structure

6.3 Overview of code to generate problem

The C++ code required to generate the equations derived above is outlined below. (The complete working program is supplied with the ISIS++ code distribution in file "isis/drivers/FDexample.cc.") The first block shown is the declaration of the various geometric and material parameters, named to match (or augment, in the case of single-letter terms, like "a") the actual parameter names from the heat conduction problem:

```
// parameters defining the physical and discretized problem
int nx, ny, mx, my;
double conduct, convect, source;
double a_length, b_length;
```

Next, various program variables are declared, including those required to construct the solution, such as the column indices (k_column[]) and matrix equation terms (matrix_terms[]) required to store the various nonzero elements of a given row of the matrix.

```
// variables to construct matrices for the discretization
int i, j, k, m, n, num_cols;
int k_columns[5];
double x_size, y_size, sq_x_size, sq_y_size;
double sol_factor, rhs_factor;
double rhs_term, matrix_terms[5];
```

The next block of code expresses the initialization of the program data, including all geometric, material, and discretization data required to express the matrix equations in a simple form.

```
// initialize the physical problem parameters here
conduct = 100.0;    // isotropic thermal conductivity
```

52

```
convect = 1.0;       // lateral convection coefficient
source = 1.0;     // distributed heat source term
a_length = 20.0;     // length of rectangle in x-direction
b_length = 10.0;     // length of rectangle in y-direction

nx = 24;             // nx = number of elements in x-direction
ny = 16;             // ny = number of elements in y-direction
mx = nx + 1;         // mx = number of nodes in x-direction
my = ny + 1;         // my = number of nodes in y-direction
n = mx*my;           // size (number of equations) of problem

x_size = a_length/nx;
y_size = b_length/ny;
sq_x_size = x_size*x_size;
sq_y_size = y_size*y_size;
sol_factor = 2.0*(sq_x_size + sq_y_size) +
             sq_x_size*sq_y_size*convect/conduct;
rhs_factor = sq_x_size*sq_y_size*source/conduct;


// now that we know the size of the problem, we can
// initialize the sparse matrix structures appropriately
Map map(n);

// construct vector for configuring the matrix structure.
Seq_IntVector myRowCount(map);

// get the row lengths to establish matrix structure.
// determine the number of nonzero terms in each row (k)
// of the matrix by stepping over the entire finite-
// difference grid (i,j)
for (i = 0; i <= nx; i++) {
    for (j = 0; j <= ny; j++) {
        k = i*my + j + 1;

        // check for interior (differenced) vs. exterior (boundary) nodes
        // (these cases are treated separately in case we wish to
        // generalize the boundary conditions)
        if ((1 <= k) && (k <= my))               // left edge
            num_cols = 1;
        else if ((my*nx+1<=k) && (k<=mx*my))     // right edge
            num_cols = 1;
        else if ((k % my) == 0)                  // top edge
            num_cols = 1;
        else if ((k % my) == 1)                  // bottom edge
            num_cols = 1;
        else                         // interior node (differenced)
            num_cols = 5;
            myRowCount[k] = num_cols;
    }
}

// construct solution and RHS vectors.
Seq_Vector x(map), b(map);

// construct an "empty" matrix, then set its structure.
RsSCRS_Matrix A(map);
A.configure(myRowCount);
```

The following code generates the matrix on a row-by-row basis, using the same mathematical relations presented earlier.

```
for (i = 0; i <= nx; i++) {
    for (j = 0; j <= ny; j++) {
        k = i*my + j + 1;

        // check for interior (differenced) vs. exterior (boundary) nodes
        // (these cases are treated separately to simplify generalization
        // to more complicated boundary conditions
```

```
        if ((1 <= k) && (k <= my)) {              // left edge
            num_cols = 1;
            k_columns[0] = k;
            matrix_terms[0] = 1.0;
            rhs_term = 0.0;
        }
        else if ((my*nx+1<=k) && (k<=mx*my)) {  // right edge
            num_cols = 1;
            k_columns[0] = k;
            matrix_terms[0] = 1.0;
            rhs_term = 0.0;
        }
        else if ((k % my) == 0) {                 // top edge
            num_cols = 1;
            k_columns[0] = k;
            matrix_terms[0] = 1.0;
            rhs_term = 0.0;
        }
        else if ((k % my) == 1) {                 // bottom edge
            num_cols = 1;
            k_columns[0] = k;
            matrix_terms[0] = 1.0;
            rhs_term = 0.0;
        }
        else {                    // interior node (differenced)
            num_cols = 5;
            k_columns[0] = (i-1)*my + j + 1;
            k_columns[1] = i*my + j;
            k_columns[2] = i*my + j + 1;
            k_columns[3] = i*my + j + 2;
            k_columns[4] = (i+1)*my + j + 1;

            matrix_terms[0] = -sq_y_size;
            matrix_terms[1] = -sq_x_size;
            matrix_terms[2] = sol_factor;
            matrix_terms[3] = -sq_x_size;
            matrix_terms[4] = -sq_y_size;
            rhs_term = rhs_factor;
        }

        double *coeffs = A.getPointerToCoef(num_cols, k);
        int *column_indices = A.getPointerToColIndex(num_cols, k);
        for (m = 0; m < num_cols; m++) {
            column_indices[m] = k_columns[m];
            coeffs[m] = matrix_terms[m];
        }
        b[k] = rhs_term;
    }
}

// indicate matrix data is loaded and internal structures
// can be checked and finalized.
A.fillComplete();
```

Now the data structures are loaded and the linear system can be solved. So the following code instantiates and uses a preconditioner and solver.

```
// construct the preconditioner
Identity_PC preconditioner(A);

// construct the solver.
QMR_Solver solver;

// declare strings for passing parameters to the solver.
char **paramStrings;
paramStrings = new char*[2];        // we'll pass in 2 parameters
paramStrings[0] = new char[32];   //set max string length=32
paramStrings[1] = new char[32];
```

54

```
// now set the parameters
sprintf(paramStrings[0],"%s","maxIterations 10000");
sprintf(paramStrings[1],"%s","tolerance 1.e-10");

// now pass the parameters to the solver
int numParams = 2;
solver.parameters(numParams,paramStrings);

// construct the system of equations object.
LinearEquations lse(A, x, b);

// set solver & preconditioner for the lse object.
lse.setSolver(solver);
lse.setPreconditioner(preconditioner);

// compute the preconditioner.
preconditioner.calculate();

// solve linear system Ax = b.
int solveStatus = lse.solve();
```

At this point, if the solve was successful, the solution vector is available in x and can be used by the application code.

6.4 Results

The following figures apply to the problem data in the program listing above. This set of parameters results in a small algebraic system (425 equations) which was solved by ISIS++ using the Quasi-Minimum Residual (QMR) and Conjugate-Gradient-Squared (CGS) algorithm utilizing an identity preconditioner (i.e., no preconditioning). The convergence histories of these two solution methods are shown in the figure below.

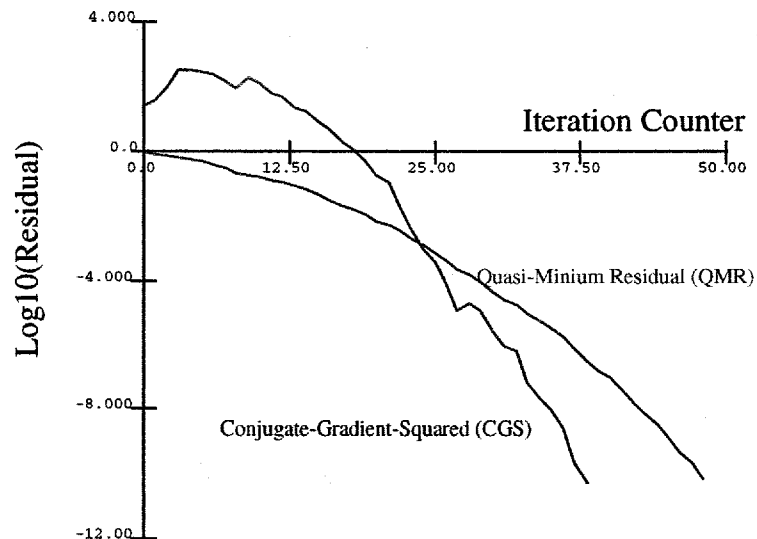Figure 6. Thermal Example Problem Convergence Histories

The results of this simulation are graphed in the contour plot shown below. Note that the problem is doubly symmetric, although no attempt has been made in this example to take advantage of this symmetry to reduce the size of the equation set. This neglect of symmetry arises from two causes. The first reason is that the implementation of

55

symmetry lines in a simple finite-difference model requires specification of a Neumann (normal derivative) boundary condition along the line of symmetry, and such a modification of a centered-difference relation is non-trivial, and beyond the desired scope of this simple example problem. The other reason is that since ISIS++ is designed to solve systems with millions of equations, there is no real need to utilize symmetry in this simple example merely to reduce the already very small equation set size.
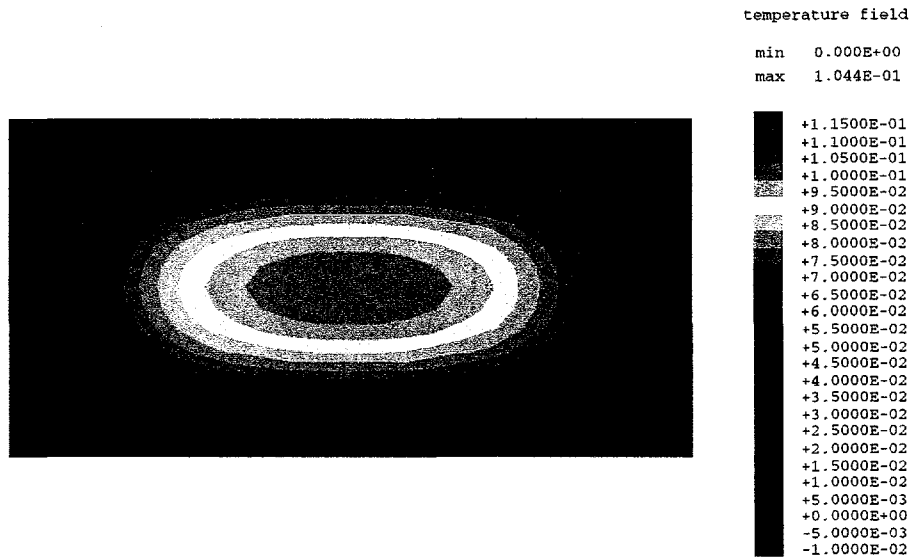

Figure 7. Contour Plot of Temperature Field

7 Installation Procedures

7.1 System requirements

The primary requirement for building ISIS++ is a sound C++ compiler. In fact, most of ISIS++ can be built and used (serially) with that alone. However, the distributed-memory components rely on the MPI message-passing library. Also, several of the algorithms use dense linear algebra methods provided by the LAPACK and BLAS libraries.

Additionally, the automated building of the library requires a "make" facility. There is also a "configure" script which uses the standard "sh" shell. As discussed below, the distribution includes automated facilities for configuring and building ISIS++ using UNIX makefiles.

7.2 Building the library

At the time of this writing, ISIS++ has been built and run on the following types of computers:

- Cray T3D MPP (native and KAI C++ compilers)

56

- Cray T3E MPP (native C++ compiler)
- ASCI Pacific Blue (native compilers)
- ASCI Red TFLOP (Portland Group compilers)
- SGI O2K (native compilers)
- IBM SP2 (native compilers)
- Meiko CS-2 MPP (KAI C++ compiler)
- HP workstations (native and gnu compilers)
- SGI workstations (native and gnu compilers)
- DEC alpha workstations running Digital Unix (native, gnu, and KAI C++ compilers)
- DEC alpha workstations running Linux (gnu compilers)
- Sun Solaris workstations (native and gnu compilers)
- Intel x86 processors running Linux (gnu compiler)
- Intel x86 processors running Windows NT 4.0 (MS compilers).

The basic development environment and target platforms are UNIX systems, although building and running ISIS++ on other platforms is a relatively straightforward matter based on our experience so long as adequate compilers are available. Naturally, the distributed-memory components are not available without MPI.

The installation scripts that are distributed with ISIS++ are targeted at UNIX systems. Essentially, the installation involves running a configure script followed by "making" the code. The entire process is set up to be run from the root ISIS++ directory, without need to modify any of the lower-level make files. The installation process is also documented on the web site and in the INSTALL file included in the distribution. We now present the basic installation procedure.

To perform the standard UNIX installation process, carry out the following steps:

1. From the root ISIS++ directory, type the command "configure".

   This will ask you a couple of questions such as whether to build for serial or parallel execution, and (if it can't find them) the paths to your system's MPI directory and to your data files (for when you run the test programs in the drivers directory). Note that the script first searches the typical paths for auxiliary libraries (MPI, LAPACK, BLAS) and will only prompt for information if it cannot find the libraries. The default answers obtained by hitting "Enter" or "Return" at all the prompts are usually adequate.

2. Type the command: "make".

   If all went well, you now have the ISIS++ library, located in:

   $ISIS_ROOT/lib/$ISIS_ARCH/libisis_mpi.a   (parallel case)
   or $ISIS_ROOT/lib/$ISIS_ARCH/libisis_ser.a   (serial case)

   where $ISIS_ROOT is the path to the top-level ISIS++ directory and $ISIS_ARCH

represents your computer's architecture. If it takes more than one attempt to make the library (e.g., because one or more of your path variables was wrong), type the command "make clean" before trying "make" again, to make sure that all of the objects get made correctly.

3. Type the command: "make test".

   This will solve the example of section 6. If libisis_mpi.a was compiled this will also solve a test matrix (extrude_1590) using QMR on first one and then two processors. You can compare the results to those found for extrude_1590 in the file $ISIS_ROOT/drivers/verify. The problem should converge in 600-700 iterations.

4. If there is still difficulty with the build after several attempts at using the configure script, the file $ISIS_ROOT/make.options can be edited directly. make.options is generated by the configure script. Here the compiler, linker, and library options can be manually customized. The authors would like to know of any changes necessary to make.options when ISIS++ is ported to a new platform.

5. Type the command: "make install".

   This will install the libraries and headers in /usr/local/isis++. You may need root privileges to complete this step. If you specified a different installation path to configure, the libraries and headers will go there instead.

6. To compile against ISIS++ headers, the flag required is –I/usr/local/isis++, and the include statement is

   ```
   #include <isis-mpi.h> // if you have MPI
   ```
   or
   ```
   #include <isis-ser.h> // if you don't have MPI.
   ```

   To link against the library, give the full path to the ISIS++ library, e.g.,

   /usr/local/isis++/lib/LINUX/libisis-mpi.a

   where you substitute the name of your architecture for LINUX.

8 Overall Schema

The class hierarchy used to implement ISIS++ follows a patter of single inheritance, as shown below.

```
class Map
     class Aztec_Map :      public Map


class IntVector
```

```
         class Seq_IntVector :          public IntVector
         class Dist_IntVector :         public IntVector


     class Vector
         class Aztec_Vector :  public Vector
         class Seq_Vector :    public Vector
         class Dist_Vector :   public Vector


     class Matrix
          class BDCRS_Matrix : public Matrix
          class RowMatrix :      public Matrix
                 class AztecDMSR_Matrix :    public RowMatrix
                 class AztecDVBR_Matrix :    public RowMatrix
                 class CRS2_Matrix :         public RowMatrix
                 class DCRS_Matrix :         public RowMatrix
                 class RsDCRS_Matrix :       public RowMatrix
                 class RsRDCRS_Matrix :      public RowMatrix
                 class RsSCRS_Matrix :       public RowMatrix
                 class SCRS_Matrix :         public RowMatrix


     class LinearEquations


     class Preconditioner
         class Composed_PC :         public Preconditioner
         class Diagonal_PC :         public Preconditioner
         class Identity_PC :         public Preconditioner
         class Poly_PC :             public Preconditioner
         class RowPreconditioner :  public Preconditioner
                 class BlockJacobi_PC :              public RowPreconditioner
                 class CGNE_BlockJacobi_PC :         public RowPreconditioner
                 class CGNE_Diagonal_PC :            public RowPreconditioner
                 class CGNE_Poly_PC :                public RowPreconditioner
                 class CGNR_Poly_PC :                public RowPreconditioner
                 class SAILS_PC :                    public RowPreconditioner
                 class SPAI_PC :                     public RowPreconditioner


     class Solver
         class IterativeSolver :     public Solver
                 class BiCGStab_Solver :     public IterativeSolver
                 class CGNE_Solver :         public IterativeSolver
                 class CGNR_Solver :         public IterativeSolver
                 class CGS_Solver :          public IterativeSolver
                 class CG_Solver :           public IterativeSolver
                 class DefGMRES_Solver :     public IterativeSolver
                 class FGMRES_Solver :       public IterativeSolver
                 class GMRES_Solver :        public IterativeSolver
                 class QMR2_Solver :         public IterativeSolver
                 class QMR_Solver :          public IterativeSolver


     class CCS_Matrix
     class CRS_Matrix
```

```
class DenseMatrix
class RsCRS_Matrix

class IntArray
class DenseVector
class GlobalIDArray

class CommInfo
class MVComm
class MVCommunicator
class LocalCommData
class hash_tables
template<class T> class SmartArray       // to disappear soon
```

9 Changes from ISIS++ v1.0

The changes from version 1.0 are small, involving two new preconditioners and new member functions in several classes. The preconditioners are Composed_PC and SAILS_PC. The changes to class member functions are as follows:

Map
> constructor for rectangular matrices.
> initComplete/isInitComplete, to support matrix assembly protocols.
> numGlobalRows, numGlobalCols replace function n(), which is deprecated.
> globalStartRow, globalEndRow, globalStartCol, globalEndCol.

Matrix
> put(s) to fill the nonzeros of a matrix with value s.

Vector
> random, to generate uniform reals between 0 and 1.
> norm1, to compute the one-norm of a vector.

Poly_PC
> isMatrixFree, to return boolean value of matrix usage.

SCRS_Matrix, RsSCRS_Matrix
> nonZeros, to return the entry count of the sparse matrix.

10 Acknowledgements

for improving ISIS++; Lee Taylor and Jim Schutt, our collaborators on interface design; and finally, Rich Palmer for having vision when it was really needed.

11 References

[1] **Barnard, S.T. and Clay, R. L.,** *A Portable MPI Implementation of the SPAI preconditioner in ISIS++*, Proceedings of Eighth SIAM Conference on Parallel Processing for Scientific Computing, 1997

[2] **Barrett, R. and Berry, M. and Chan, T. and Demmel, J. and Donato, J. and Dongarra, J. and Eijkout, V. and Pozo, R. and Romine, C. and van der Vorst, H.,** *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994

[3] **Buecker, H. and Sauren, M.,** *A Parallel Version of the Unsymmetric Lanczos Algorithm and its Application to QMR*, Technical Report KFA-Juelich, 1996

[4] **Clay, R.L., Mish, K.D., and Williams, A.B.,** *ISIS++ Reference Guide Version 1.0*, Sandia National Laboratories Technical Report SAND97-8535, September 1997

[5] **Demmel, J. and Eisenstat, S. and Gilbert, J. and Li, X. and Liu, J.** *A SuperNodal Approach to Sparse Partial Pivoting*, Technical Report UCB//CSD-95-883, Computer Science Division, U. C. Berkeley, July 1995

[6] **Demmel, J. and Gilbert, J. and Li, X.** *SuperLU User's Guide*, Technical Report UCB//CSD-97, Computer Science Division, U. C. Berkeley, Novemeber 1997

[7] **Ellis, M. and Stroustrup, B.,** *The Annotated C++ Reference Manual, Addison Wesley*, 1990

[8] **Erhel, J. and Burrage, K. and Pohl, B.,** *Restarted GMRES preconditioned by Deflation*, J. Comp. Appl. Math., Dec, 1995

[9] **Freund, R.,** *A Transpose Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems*, SIAM J. Sci. Comp., 14:470-482, 1993

[10] **Freund, R. and Golub, G. and Nachtigal, N.,** *Iterative Solution of Linear Systems*, Acta Numerica, 57-100, 1992

[11] **Freund, R. and Nachtigal, N.,** *QMR: A Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems*, Numer. Math., 60:315-339, 1991

[12] **Grote, M. and Huckle, T.,** *Parallel Preconditioning with Sparse Approximate Inverses*, SIAM J. Sci. Comp., 18:838-853, 1997

[13] **Hestenes, M. and Stiefel, E.,** *Methods of Conjugate Gradients for solving Linear Systems*, J. Res. Nat. Bur. Stand., 49:409-436, 1952

[14] **Hutchinson, S. and Shadid, J. and Tuminaro, R.,** *Aztec Users Guide version 2,0*, Technical Report SAND95-1559, Sandia National Laboratories, Albuquerque NM, Revised July 1998

[15] **Kelley, C.,** *Iterative Methods for Linear and Nonlinear Equations*, SIAM, 1995

[16] **Meier-Yang, U.,** *Preconditioned Conjugate Gradient-like Methods for Nonsymmetric Linear Systems,* Technical Report, CSRD, University of Illinois, Urbana, IL, April 1992

[17] **Nachtigal, N. and Reddy, S. and Trefethen, L.,** *How Fast are Nonsymmetric Matrix Iterations?,* SIAM J. Matrix Anal. and App., 13:778-795, 1992

[18] **Saad, Y.,** *A Flexible Inner-outer Preconditioned GMRES Algorithm,* SIAM J. Sci. Comput., 14:461-469, 1993

[19] **Saad, Y. and Schultz, M.,** *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems,* Math. Comp., 44:417-424, 1986

[20] **Sonneveld, P.,** *CGS: A Fast Lanczos-type Solver for Nonsymmetric Linear Systems,* SIAM J. Sci. Statist. Comput., 10:36-52, 1989

[21] **Tong, C.H.,** *A Comparative Study of Preconditioned Lanczos Methods for Nonsymmetric Linear Systems,* Tech. Report SAND91-8240, Sandia Nat. Lab., Livermore, CA, 1992

DISTRIBUTION:

Dr.-Ing. Achim Basermann
NEC Europe Ltd.
C&C Research Laboratories
Rathausallee 10
D-53757 Sankt Augustin
Germany

Joe Fridy
Aluminum Company of America
Alcoa Technical Center
100 Technical Drive
Alcoa Center, Pennsylvania 15069–0001

Kyran D. Mish (5)
Department of Mechanical Engineering
California State University, Chico
95929-0789

J. Robert Neely
Mailstop L035
Lawrence Livermore National Laboratory
Livermore, CA 94550

Ivan J. Otero
Mailstop L035
Lawrence Livermore National Laboratory
Livermore, CA 94550

John V. Reynders
ACL, MS B287
Los Alamos National Laboratory
Los Alamos, New Mexico 87544

Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S Cass Ave
Argonne, IL 60439

1	MS 9003	D. R. Henson, 8900/8980
1	MS 9003	D. L. Lindner, 8902
1	MS 9011	P. W. Dean, 8903
1	MS 9011	B. V. Hess, 8910
1	MS 1202	M. R. Fox, 8920

1	MS 9012	R. A. Whiteside, 8920
1	MS 9012	S. C. Gray, 8930
1	MS 9037	J. C. Berry, 8930-1
1	MS 9019	B. A. Maxwell, 8940
1	MS 9011	J. C. Meza, 8950
1	MS 9214	C. H. Tong, 8950
1	MS 9011	M. H. Rogers, 8960
1	MS 9011	J. A. Larson, 8970
1	MS 9214	B. A. Allen, 8980
1	MS 9214	R. C. Armstrong, 8980
5	MS 9214	R. L. Clay, 8980
1	MS 9214	W. E. Mason, 8980
1	MS 9214	K. J. Perano, 8980
5	MS 9214	A. B. Williams, 8980
1	MS 9011	P. Wyckoff, 8980
1	MS 9012	K. R. Hughes, 8990
1	MS 9001	T. O. Hunter, 8000

Attn: R. C. Wayne, 2200/8400
            M. E. John, 8100
            L. A. West, 8200 (A)
            W. J. McLean, 8300
            P. N. Smith, 8500
            P. E. Brewer, 8600
            T. M. Dyer, 8700
            L. A. Hiles, 8800

1	MS 9051	J. Grcar, 8345
1	MS 9405	P. E. Nielan, 8743
3	MS 0836	J. H. Biffle, 9121
1	MS 0836	H. C. Edwards, 9121
3	MS 0836	L. M. Taylor, 9121
1	MS 1111	S. A. Hutchinson, 9221
1	MS 1111	J. N. Shadid, 9221
1	MS 1110	R. S. Tuminaro, 9222
1	MS 0819	J. S. Peery, 9231
3	MS 9018	Central Technical Files, 8940-2
1	MS 0899	Technical Library, 4916
1	MS 9021	Technical Communications Dept. 8815/Technical Library, 4916
1	MS 9021	Technical Communications Dept. 8815 for DOE/OSTI

This page intentionally left blank