1/12/96 JSD

SANDIA REPORT

SAND97-8200 • UC-405
Unlimited Release
Printed October 1996

The Scientific Computing (SC) Matrix Library

Curtis L. Janssen, Edward T. Seidl, Ida M. B. Nielsen, Michael E. Colvin

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94551
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Approved for public release; distribution is unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of the contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors or subcontractors.

This report has been reproduced from the best available copy.

Available to DOE and DOE contractors from:

Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge TN 37831

Prices available from (615) 576-8401, FTS 626-8401.

Available to the public from:

National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.
Springfield, VA 22161

DISCLAIMER

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

The SC Matrix Library

Curtis L. Janssen, Edward T. Seidl,
Ida M. B. Nielsen, and Michael E. Colvin

*MS 9214, Dept. 8117
Sandia National Laboratories
Livermore, CA 94551-0969*

September 11, 1996

Abstract

The scientific computing matrix library (SCMAT) is a C++ class library for representing and manipulating rectangular, symmetric, and diagonal matrices and vectors. The library is founded on an abstract base classes for each of these types of matrices. These are specialized to concrete matrix classes that are well suited for particular computational environments. The local matrix specializations do no communication. All elements reside on each node and all computations are duplicated on each node. The replicated matrix specializations hold all of the elements on each node, however do some communication in order to reduce computation time. The distributed matrix specializations spread the elements across all the nodes and thus require less storage than local matrices however these use more communication than replicated matrices. The blocked matrix specializations are used to implement point group symmetry. One of the other matrix specializations is used to hold the diagonal subblocks of a matrix. The offdiagonal subblocks are known to be zero and not stored. This results in considerable savings in storage and computation for those cases where it applies.

A variety of supporting classes is also provided. There is a dimension class that optionally contains application specific blocking information as well as the matrix dimension. Classes to assist in iterating through the elements of a matrix are available. Finally, a class is provided which acts as a class factory by returning mutually consistent matrix specializations.

1 Introduction

The scientific computing matrix library (SCMAT) is designed around a set of matrix abstractions that permit very general matrix implementations. This flexibility is needed to support diverse computing environments. For example, this library must support, at a minimum: simple matrices that provide efficient matrix computations in a uniprocessor environment, clusters of processors with enough memory to store all matrices connected by a relatively slow network (workstations on an LAN), clusters of processors with enough memory to store all matrices and a fast interconnect network (a massively parallel machine such as the Intel Paragon), and clusters of machines that don't have enough memory to hold entire matrices.

The design of SCMAT differs from other object-oriented matrix packages in two important ways. First, the matrix classes are abstract base classes. No storage layout is defined and virtual function calls must be used to access individual matrix elements. This would have a negative performance impact if users needed to frequently access matrix elements. The interface to the matrix classes is hopefully rich enough to avoid individual matrix element access for any computationally significant task. The second major difference is that symmetric matrices do not inherit from matrices, etc. The SCMAT user must know whether a matrix is symmetric at all places it is used if any performance gain, by virtue of symmetry, is expected.

Dimension information is contained objects of the SCDIMENSION type. In addition to the simple integer dimension, application specific blocking information can be provided. For example, in a quantum chemistry application, the dimension corresponding to the atomic orbital basis set will have block sizes that correspond to the shells. Dimensions are used to create new matrix or vector objects.

The primary abstract classes are SCMATRIX, SYMMSCMATRIX, DIAGSCMATRIX, and SCVECTOR. These represent matrices, symmetric matrices, diagonal matrices, and vectors, respectively. These abstract classes are specialized into groups of classes. For example, the locally stored matrix implementation specializes the abstract classes to LOCALSCMATRIX, LOCALSYMMSCMATRIX, LOCALDIAGSCMATRIX, LOCALSCVECTOR, LOCALSCDIMENSION, and LOCALSCMATRIXKIT. These specializations are all designed to work with each other. However, a given specialization is incompatible with other matrix specializations. An attempt to multiply a local matrix by a distributed matrix would generate an error at runtime.

Since the different groups of classes do not interoperate, some mechanism of creating consistent specializations is needed. This is done with SCMATRIXKIT objects. SCMATRIXKIT is an abstract base type which has specializations that correspond to each group of the matrix specializations. It is used to create matrices and vectors from that group. For example, the DISTSCMATRIXKIT is used to create objects of type DISTSCMATRIX, DISTSYMMSCMATRIX, DISTDIAGSCMATRIX, and DISTSCVECTOR.

The abstract matrix classes and their derivations are usually not directly

used by SCMAT users. The most convenient classes to use are the smart pointer classes REFSCMATRIX, REFSYMMSCMATRIX, REFDIAGSCMATRIX, REFSCDIMENSION, and REFSCMATRIXKIT. These automatically delete matrix objects when they are no longer needed. This is through a reference count mechanism that is supported by the VREFCOUNT base class from which the abstract matrix classes derive. The smart pointer classes also have matrix operations such as operator `*`(), operator `-`(), and operator `+`() defined as members for convenience. These forward the operations to the contained matrix object. The smart pointer classes also simplify creation of matrices by providing constructors that take as arguments one or more REFSCDIMENSION's and a REFSCMATRIXKIT. These initialize the smart pointer to contain a new matrix with a specialization corresponding to that of the REFSCMATRIXKIT. Matrix operations not provided by the smart pointer classes but present as member in the abstract classes can be accessed with operator `->`().

If a needed matrix operation is missing, mechanisms exist to add more general operations. Operations which only depend on individual elements of matrices can be provided by specializations of the SCELEMENTOP class. Sometimes we need operations on matrices with identical dimensions that examine each element in one matrix along with the corresponding element from the other matrix. This is accomplished with SCELEMENTOP2 for two matrices and with SCELEMENTOP3 for three.

Other features of SCMAT include run-time type facilities and persistence. Castdown operations (type conversions from less to more derived objects) and other run-time type information are provided by the DESCRIBEDCLASS base class. Persistence is not provided by inheriting from SAVABLESTATE base class as is the case with many other classes in the SC class hierarchies, because it is necessary to save objects in an implementation independent manner. If a calculation checkpoints a matrix on a single processor machine and later is restarted on a multiprocessor machine the matrix would need to be restored as a different matrix specialization. This is handled by saving and restoring matrices' and vectors' data without reference to the specialization.

The following include files are provided by the matrix library:

matrix.h Usually, this is the only include file needed by users of matrices. It declares reference counting pointers to abstract matrices.

If kit for a matrix must be created, or a member specific to an implementation is needed, then that implementation's header file must be included.

elemop.h This is the next most useful include file. It defines useful SCELEMENTOP, SCELEMENTOP2, and SCELEMENTOP3 specializations.

abstract.h This include file contains the declarations for abstract classes that users do not usually need to see. These include SCDIMENSION, SCMATRIX, SYMMSCMATRIX, DIAGSCMATRIX, SCMATRIXKIT. This file is currently included by **matrix.h**.

block.h This file declares `SCMATRIXBLOCK` and specializations. It only need be include by users implementing new `SCELEMENTOP` specializations.

blkiter.h This include file declares the implementations of `SCMATRIXBLOCK-ITER`. It only need be include by users implementing new `SCELEMENTOP` specializations.

vector3.h This declares `SCVECTOR3`, a lightweight vector of length three.

matrix3.h This declares `SCMATRIX3`, a lightweight matrix of dimension three by three. It includes `vector3.h`.

local.h This include file is the matrix implementation for locally stored matrices. These are suitable for use in a uniprocessor environment. The `LOCALSCMATRIXKIT` is the default matrix implementation returned by the static member `SCMATRIXKIT::default_matrixkit`. This file usually doesn't need to be included.

dist.h This include file is the matrix implementation for distributed matrices. These are suitable for use in a distributed memory multiprocessor which does not have enough memory to hold all of the matrix elements on each processor. This file usually doesn't need to be included.

repl.h This include file is the matrix implementation for replicated matrices. These are suitable for use in a distributed memory multiprocessor which does have enough memory to hold all of the matrix elements on each processor. This file usually doesn't need to be included.

blocked.h This include file is the matrix implementation for blocked matrices. Blocked matrices store a matrix as subblocks that are matrices from another matrix specialization. These are used to save storage and computation time in quantum chemistry applications for molecules with other than C_1 point group symmetry.

2 Matrix Dimensions

In addition to the simple integer dimension, objects of the `SCDIMENSION` class contain application specific blocking information. This information is held in an object of class `SCBLOCKINFO`.

2.1 The SCDimension Class

The `SCDIMENSION` class is used to determine the size and blocking of matrices. The blocking information is stored by an object of class `SCBLOCKINFO`.

2.1.1 The Public SCDimension Interface

SCDimension(int n, const char* name = 0)
SCDimension(const RefSCBlockInfo&, const char *name = 0)
SCDimension(int n, int nblocks, const int *blocksizes = 0, const char* name = 0)
SCDimension(const RefKeyVal&)
SCDimension(StateIn&s)
Create a dimension with an optional name. The name is a copy of the '0' terminated string *name*.
equiv(const SCDimension*) const → int
Test to see if two dimensions are equivalent.
n() const → int
Return the dimension.
name() const → const char *
Return the name of the dimension. If no name was given to the constructor, then return 0.
blocks() → RefSCBlockInfo
Return the blocking information for this dimension.
print(ostream&o=cout)
Print information about this dimension to *o*.

2.2 The SCBlockInfo Class

SCBLOCKINFO contains blocking information for the SCDIMENSION class. There are really two ways that it can contain blocking information. In the first way, a vector of block offsets and block sizes is stored. The second method is only used by those specializations created by the BLOCKEDSCMATRIXKIT class. In this method the blocking information is stored as subdimensions of type SCDIMENSION. If both methods are used, they must be used consistently. That is, the number, sizes, and order of the blocks must match the number, sizes, and order of the SCDIMENSION objects.

2.2.1 The Public SCBlockInfo Interface

SCBlockInfo(int n, int nblocks = 0, const int *blocksizes = 0)
SCBlockInfo(StateIn&)
SCBlockInfo(const RefKeyVal& keyval)
Create a SCBLOCKINFO object.
equiv(SCBlockInfo *bi) → int
Return nonzero if *this* is equivalent to *bi*.
nelem() const → int
Return the total number of elements.
nblock() const → int
Return the number of blocks.
start(int i) const → int
Return the starting index for block *i*.

size(int i) const → int
 Return the size of block *i*.

fence(int i) const → int
 Return the last index +1 for block *i*.

subdim(int i) → RefSCDimension
 Retrieve subdimension information.

set_subdim(int i, const RefSCDimension &dim)
 Set subdimension information. The dimension *dim* and index *i* must be consistent with the *nblocks* and *blocksizes* information given to the constructor.

print(ostream&o=cout)
 Print the object to the stream *o*.

3 Matrix Reference Classes

The easiest way to use SCMAT is through the smart pointer classes REFSCMATRIX, REFSYMMSCMATRIX, REFDIAGSCMATRIX, REFSCVECTOR, REFSCDIMENSION, and REFSCMATRIXKIT. These are based on the REF reference counting package and automatically delete matrix objects when they are no longer needed. These reference classes also have common operations defined as members for convenience. This makes it unnecessary to also use the sometimes awkward syntax of `operator->()` to manipulate the contained objects.

3.1 The RefSCDimension Class

The REFSCDIMENSION class is a smart pointer to an SCDIMENSION specialization.

3.1.1 The Public RefSCDimension Interface

RefSCDimension()
 Initializes the dimension pointer to 0. The reference must be initialized before it is used.

RefSCDimension(const RefSCDimension& d)
 Make this and *d* refer to the same SCDIMENSION.

RefSCDimension(SCDimension *d)
 Make this refer to *d*.

operator =(SCDimension* d) → RefSCDimension &
 Make this refer to *d*.

operator =(const RefSCDimension & d) → RefSCDimension &
 Make this and *d* refer to the same SCDIMENSION.

operator int() const → int

n() const → int
 Return the dimension.

3.2 The RefSCVector Class

The REFSCVECTOR class is a smart pointer to an SCVECTOR specialization. Valid indices range from 0 to $n-1$.

3.2.1 The Public RefSCVector Interface

RefSCVector()

Initializes the vector pointer to 0. The reference must be initialized before it is used.

RefSCVector(const RefSCVector& v)

Make this and *v* refer to the same SCVECTOR.

RefSCVector(SCVector *v)

Make this refer to *v*.

operator =(SCVector* v) → RefSCVector &

Make this refer to *v*.

operator =(const RefSCVector& v) → RefSCVector &

Make this and *v* refer to the same SCVECTOR.

RefSCVector(const RefSCDimension& dim, const RefSCMatrixKit&)

Create a vector with dimension *dim*. The data values are undefined.

operator()(int) const → SCVectordouble

operator [](int) const → SCVectordouble

Return an l-value that can be used to assign or retrieve an element.

operator +(const RefSCVector&a) const → RefSCVector

Add two vectors.

operator -(const RefSCVector&a) const → RefSCVector

Subtract two vectors.

operator *(double) const → RefSCVector

Scale a vector.

outer_product(const RefSCVector& v) const → RefSCMatrix

Return the outer product between this and *v*.

symmetric_outer_product() const → RefSymmSCMatrix

The outer product of this with itself is a symmetric matrix.

set_element(int i, double val) const

accumulate_element(int i, double val) const

get_element(int) const → double

n() const → int

dim() const → RefSCDimension

kit() const → RefSCMatrixKit

clone() const → RefSCVector

copy() const → RefSCVector

maxabs() const → double

scalar_product(const RefSCVector&) const → double

dot(const RefSCVector&) const → double

normalize() const

randomize() const

```

assign(const RefSCVector& v) const
assign(double val) const
assign(const double* v) const
convert(double*) const
scale(double val) const
accumulate(const RefSCVector& v) const
accumulate_product(const RefSymmSCMatrix&, const Ref-
SCVector&)
accumulate_product(const RefSCMatrix&, const RefSCVector&)
element_op(const RefSCElementOp& op) const
element_op(const RefSCElementOp2&, const RefSCVector&)
const
element_op(const RefSCElementOp3&, const RefSCVector&,
const RefSCVector&) const
print(ostream&out) const
print(const char*title=0, ostream&out=cout, int precision=10)
const
save(StateOut&)
restore(StateIn&)

```

These call the SCMATRIX members of the same name after checking for references to 0.

3.3 The RefSCMatrix Class

The REFSCMATRIX class is a smart pointer to an SCMATRIX specialization.

3.3.1 The Public RefSCMatrix Interface

```

RefSCMatrix()
    Initializes the matrix pointer to 0. The reference must be initialized before
    it is used.
RefSCMatrix(const RefSCMatrix& m)
    Make this and m refer to the same SCMatrix.
RefSCMatrix(SCMatrix* m)
RefSCMatrix()
    Make this refer to m.
operator =(SCMatrix* m) → RefSCMatrix &
    Make this refer to m.
operator =(const RefSCMatrix& m) → RefSCMatrix &
    Make this and m refer to the same matrix.
RefSCMatrix(const RefSCDimension& d1, const RefSCDimen-
sion& d2, const RefSCMatrixKit&)
    Create a vector with dimension d1 by d2. The data values are undefined.
operator *(const RefSCVector&) const → RefSCVector
    Multiply this by a vector and return a vector.
operator *(const RefSCMatrix&) const → RefSCMatrix
operator *(const RefSymmSCMatrix&) const → RefSCMatrix

```

operator *(const RefDiagSCMatrix&) const → RefSCMatrix
 Multiply this by a matrix and return a matrix.
operator *(double) const → RefSCMatrix
 Multiply this by a scalar and return the result.
operator +(const RefSCMatrix&) const → RefSCMatrix
operator -(const RefSCMatrix&) const → RefSCMatrix
 Matrix addition and subtraction.
t() const → RefSCMatrix
 Return the transpose of this.
i() const → RefSCMatrix
 Return the inverse of this.
gi() const → RefSCMatrix
 Return the generalized inverse of this.
clone() const → RefSCMatrix
copy() const → RefSCMatrix
 These call the SCMATRIX members of the same name after checking for references to 0.
svd(const RefSCMatrix &U, const RefDiagSCMatrix &sigma, const RefSCMatrix &V)
 Compute the singular value decomposition. $\text{this} = U \sigma V.t()$. The dimension of *sigma* is the smallest dimension of this. *U*, *V*, and *sigma* must already have the correct dimensions and are overwritten.
solve_lin(const RefSCVector& v) const → double
 Solves this $x = v$. Overwrites *v* with *x*.
determ() const → double
 Returns the determinant of the referenced matrix.
operator()(int i, int j) const → SCMatrixdouble
 Assign and examine matrix elements.
operator *(double, const RefSCMatrix&) → RefSCMatrix
 Allow multiplication with a scalar on the left.

3.4 The RefSymmSCMatrix Class

The REFSYMMSCMATRIX class is a smart pointer to an SCSYMMSCMATRIX specialization.

3.4.1 The Public RefSymmSCMatrix Interface

RefSymmSCMatrix()
 Initializes the matrix pointer to 0. The reference must be initialized before it is used.
RefSymmSCMatrix(const RefSymmSCMatrix& m)
 Make this and *m* refer to the same SCMATRIX.
RefSymmSCMatrix(SymmSCMatrix *m)
RefSymmSCMatrix()
 Make this refer to *m*.

operator =(SymmSCMatrix* m) → RefSymmSCMatrix &
 Make this refer to *m*.
operator =(const RefSymmSCMatrix& m) → RefSymmSCMatrix &
 Make this and *m* refer to the same matrix.
RefSymmSCMatrix(const RefSCDimension& d, const RefSCMatrixKit&)
 Create a vector with dimension *d* by *d*. The data values are undefined.
operator *(const RefSCMatrix&) const → RefSCMatrix
 Multiply this by a matrix and return a matrix.
operator *(const RefSCVector&a) const → RefSCVector
operator *(double) const → RefSymmSCMatrix
 Multiply this by a vector and return a vector.
operator +(const RefSymmSCMatrix&) const → RefSymmSCMatrix
operator -(const RefSymmSCMatrix&) const → RefSymmSCMatrix
 Matrix addition and subtraction.
i() const → RefSymmSCMatrix
 Return the inverse of this.
gi() const → RefSymmSCMatrix
 Return the generalized inverse of this.
clone() const → RefSymmSCMatrix
copy() const → RefSymmSCMatrix
set_element(int, int, double) const
accumulate_element(int, int, double) const
get_element(int, int) const → double
 These call the SCMATRIX members of the same name after checking for references to 0.
accumulate.transform(const RefSCMatrix&a, const RefSymmSCMatrix&b) const
accumulate.transform(const RefSCMatrix&a, const RefDiagSCMatrix&b) const
accumulate.transform(const RefSymmSCMatrix&a, const RefSymmSCMatrix&b) const
 Add $a * b * a.t()$ to this.
solve_lin(const RefSCVector&) const → double
 Solves this $x = v$. Overwrites *v* with *x*.
determ() const → double
 Returns the determinant of the referenced matrix.
eigvals() const → RefDiagSCMatrix
 Returns the eigenvalues of the reference matrix.
eigvecs() const → RefSCMatrix
 Returns the eigenvectors of the reference matrix.
diagonalize(const RefDiagSCMatrix& eigvals, const RefSCMatrix& eigvecs) const
 Sets *eigvals* to the eigenvalues and *eigvecs* to the eigenvalues and eigenvectors of the referenced matrix.

operator `()(int i, int j) const` $\rightarrow$ `SymmSCMatrixdouble`
 Assign and examine matrix elements.
 operator `*(double, const RefSymmSCMatrix&)` $\rightarrow$ `RefSymmSCMatrix`
 Allow multiplication with a scalar on the left.

3.5 The RefDiagSCMatrix Class

The `REFDIAGSCMATRIX` class is a smart pointer to an `DIAGSCMATRIX` specialization.

3.5.1 The Public RefDiagSCMatrix Interface

`RefDiagSCMatrix()`
 Initializes the matrix pointer to 0. The reference must be initialized before it is used.

`RefDiagSCMatrix(const RefDiagSCMatrix& m)`
 Make this and *m* refer to the same SCMATRIX.

`RefDiagSCMatrix(DiagSCMatrix *m)`
`RefDiagSCMatrix()`
 Make this refer to *m*.

operator `=(DiagSCMatrix* m)` $\rightarrow$ `RefDiagSCMatrix &`
 Make this refer to *m*.

operator `=(const RefDiagSCMatrix & m)` $\rightarrow$ `RefDiagSCMatrix &`
 Make this and *m* refer to the same matrix.

`RefDiagSCMatrix(const RefSCDimension&, const RefSCMatrixKit&)`
 Create a vector with dimension *d* by *d*. The data values are undefined.

operator `*(const RefSCMatrix&) const` $\rightarrow$ `RefSCMatrix`

operator `*(double) const` $\rightarrow$ `RefDiagSCMatrix`
 Multiply this by a matrix and return a matrix.

operator `+(const RefDiagSCMatrix&) const` $\rightarrow$ `RefDiagSCMatrix`
 Matrix addition and subtraction.

`i() const` $\rightarrow$ `RefDiagSCMatrix`
 Return the inverse of this.

`gi() const` $\rightarrow$ `RefDiagSCMatrix`
 Return the generalized inverse of this.

`clone() const` $\rightarrow$ `RefDiagSCMatrix`

`copy() const` $\rightarrow$ `RefDiagSCMatrix`

`set_element(int, double) const`

`accumulate_element(int, double) const`

`get_element(int) const` $\rightarrow$ `double`

`randomize() const`

`assign(const RefDiagSCMatrix&) const`

`scale(double) const`

```

assign(double) const
assign(const double*) const
convert(double*) const
accumulate(const RefDiagSCMatrix&) const
element_op(const RefSCElementOp&) const
element_op(const RefSCElementOp2&, const RefDiagSCMa-
trix&) const
element_op(const RefSCElementOp3&, const RefDiagSCMa-
trix&, const RefDiagSCMatrix&) const
n() const → int
dim() const → RefSCDimension
kit() const → RefSCMatrixKit
trace() const → double
print(ostream&) const
print(const char*title=0, ostream&out=cout, int =10) const
save(StateOut&)
restore(StateIn&)

```

These call the SCMATRIX members of the same name after checking for references to 0.

```
determ() const → double
```

Returns the determinant of the referenced matrix.

```
operator()(int i) const → DiagSCMatrixdouble
```

Assign and examine matrix elements.

```
operator*(double, const RefDiagSCMatrix&) → RefDiagSCMa-
trix
```

Allow multiplication with a scalar on the left.

4 Abstract Matrix Classes

This section documents the primary abstract classes: SCMATRIX, SYMMSCMATRIX, DIAGSCMATRIX, and SCVECTOR, as well as the SCMATRIXKIT class which allows the programmer to generate consistent specializations of matrices. These represent matrices, symmetric matrices, diagonal matrices, and vectors, respectively.

This section is primarily for implementers of new specializations of matrices. Users of existing matrices will be most interested in the matrix reference classes.

4.1 The SCMatrixKit Class

4.1.1 The Public SCMatrixKit Interface

```
default_matrixkit() static → SCMatrixKit *
```

```
set_default_matrixkit(const RefSCMatrixKit &) static
```

This returns a LOCALSCMATRIXKIT, unless the default has been changed with set_default_matrixkit.

```
matrix(const RefSCDimension&, const RefSCDimension&) pure
→ SCMatrix *
```

```

symmmatrix(const RefSCDimension&) pure → SymmSCMatrix
*
diagmatrix(const RefSCDimension&) pure → DiagSCMatrix *
vector(const RefSCDimension&) pure → SCVector *

```

Given the dimensions, create matrices or vectors.

```

restore_matrix(StateIn&, const RefSCDimension&, const Ref-
SCDimension&) → SCMatrix *
restore_symmmatrix(StateIn&, const RefSCDimension&) →
SymmSCMatrix *
restore_diagmatrix(StateIn&, const RefSCDimension&) → Di-
agSCMatrix *
restore_vector(StateIn&, const RefSCDimension&) → SCVector
*

```

Given the dimensions and a STATEIN object, restore matrices or vectors.

4.2 The SCVector Class

The SCVECTOR class is the abstract base class for double valued vectors.

4.2.1 The Public SCVector Interface

```

save(StateOut&) virtual
restore(StateIn&) virtual
    Save and restore this in an implementation independent way.
kit() const → RefSCMatrixKit
    Return the SCMATRIXKIT used to create this object.
copy() virtual → SCVector *
    Return a vector with the same dimension and same elements.
clone() virtual → SCVector *
    Return a vector with the same dimension but uninitialized memory.
n() → int
    Return the length of the vector.
maxabs() virtual → double
    Return the maximum absolute value element of this vector.
normalize() virtual
    Normalize this.
randomize() virtual
    Assign each element to a random number between -1 and 1
assign(double val) virtual
    Assign all elements of this to val.
assign(const double* v) virtual
    Assign element i to v[i] for all i.
convert(double* v) virtual
    Assign v[i] to element i for all i.
convert(SCVector*) virtual
convert_accumulate(SCVector*) virtual
    Convert an SCVECTOR of a different specialization to this specialization

```


and possibly accumulate the data.

assign(SCVector* v) virtual
 Make this have the same elements as *v*. The dimensions must match.

scale(double val) virtual
 Multiply each element by *val*.

dim() const → RefSCDimension
 Return the REFSCDIMENSION corresponding to this vector.

set_element(int i, double val) pure
 Set element *i* to *val*.

accumulate_element(int, double) pure
 Add *val* to element *i*.

get_element(int i) pure → double
 Return the value of element *i*.

accumulate_product(SymmSCMatrix* m, SCVector* v) virtual
accumulate_product(SCMatrix* m, SCVector* v) pure
 Sum the result of *m* times *v* into this.

accumulate(SCVector*v) pure
 Sum *v* into this.

accumulate(SCMatrix*m) pure
 Sum *m* into this. One of *m*'s dimensions must be 1.

scalar_product(SCVector*) pure → double
 Return the dot product.

element_op(const RefSCElementOp&) pure
element_op(const RefSCElementOp2&, SCVector*) pure
element_op(const RefSCElementOp3&, SCVector*, SCVector*)
pure
 Perform the element operation *op* on each element of this.

print(ostream&) virtual
print(const char* title=0, ostream& out=cout, int =10) pure
 Print out the vector.

messagegrp() → RefMessageGrp
 Returns the message group used by the matrix kit

local_blocks(SCMatrixSubblockIter::Access) pure → RefSCMatrixSubblockIter
 Returns iterators for the local (rapidly accessible) blocks used in this vector. Only one iterator is allowed for a matrix is it has Accum or Write access is allowed. Multiple Read iterators are permitted.

all_blocks(SCMatrixSubblockIter::Access) pure → RefSCMatrixSubblockIter
 Returns iterators for the all blocks used in this vector.

4.3 The SCMatrix Class

The SCMATRIX class is the abstract base class for general double valued *n* by *m* matrices. For symmetric matrices use SYMMSCMATRIX and for diagonal matrices use DIAGSCMATRIX.

4.3.1 The Public SCMatrix Interface

save(StateOut&) virtual

restore(StateIn&) virtual

Save and restore this in an implementation independent way.

kit() const → RefSCMatrixKit

Return the SCMATRIXKIT used to create this object.

nrow() const → int

ncol() const → int

Return the number of rows or columns.

maxabs() virtual → double

Return the maximum absolute value element.

randomize() virtual

Assign each element to a random number between -1 and 1

assign(double val) virtual

Set all elements to *val*.

assign(const double* m) virtual

Assign element *i, j* to *m[i*nrow()+j]*.

assign(const double m) virtual**

Assign element *i, j* to *m[i][j]*.

convert(double*) virtual

convert(double) virtual**

Like the assign members, but these write values to the arguments.

convert(SCMatrix*) virtual

convert_accumulate(SCMatrix*) virtual

Convert an SCMATRIX of a different specialization to this specialization and possibly accumulate the data.

assign(SCMatrix* m) virtual

Make this have the same elements as *m*. The dimensions must match.

scale(double val) virtual

Multiply all elements by *val*.

scale_diagonal(double val) virtual

Scale the diagonal elements by *val*.

shift_diagonal(double val) virtual

Shift the diagonal elements by *val*.

unit() virtual

Make this equal to the unit matrix.

copy() virtual → SCMatrix *

Return a matrix with the same dimension and same elements.

clone() virtual → SCMatrix *

Return a matrix with the same dimension but uninitialized memory.

rowdim() const → RefSCDimension

coldim() const → RefSCDimension

Return the row or column dimension.

get_element(int, int) pure → double

set_element(int, int, double) pure

`accumulate_element(int, int, double) pure`
 Return or modify an element.

`get_subblock(int br, int er, int bc, int ec) pure → SCMatrix *`
 Return a subblock of this. The subblock is defined as the rows starting at br and ending at er, and the columns beginning at bc and ending at ec.

`assign_subblock(SCMatrix *m, int, int, int, int, int=0, int=0) pure`
 Assign m to a subblock of this.

`accumulate_subblock(SCMatrix *m, int, int, int, int, int=0, int=0) pure`
 Sum m into a subblock of this.

`get_row(int i) pure → SCVector *`
`get_column(int i) pure → SCVector *`
 Return a row or column of this.

`assign_row(SCVector *v, int i) pure`
`assign_column(SCVector *v, int i) pure`
 Assign v to a row or column of this.

`accumulate_row(SCVector *v, int i) pure`
`accumulate_column(SCVector *v, int i) pure`
 Sum v to a row or column of this.

`accumulate(SCMatrix* m) pure`
`accumulate(SymmSCMatrix* m) pure`
`accumulate(DiagSCMatrix* m) pure`
`accumulate(SCVector*) pure`
 Sum m into this.

`accumulate_outer_product(SCVector*, SCVector*) pure`
`accumulate_product(SCMatrix*, SCMatrix*) pure`
`accumulate_product(SCMatrix*, SymmSCMatrix*) virtual`
`accumulate_product(SCMatrix*, DiagSCMatrix*) virtual`
`accumulate_product(SymmSCMatrix*, SCMatrix*) virtual`
`accumulate_product(DiagSCMatrix*, SCMatrix*) virtual`
 Sum into this the products of various vectors or matrices.

`transpose_this() pure`
 Transpose this.

`trace() pure → double`
 Return the trace.

`invert_this() pure → double`
 Invert this.

`determ_this() pure → double`
 Return the determinant of this. this is overwritten.

`svd_this(SCMatrix *U, DiagSCMatrix *sigma, SCMatrix *V) virtual`
`solve_this(SCVector*) pure → double`
`gen_invert_this() pure`
 Compute the singular value decomposition for this, possibly destroying this.

schmidt_orthog(SymmSCMatrix*, int n) pure
 Schmidt orthogonalize this. S is the overlap matrix. n is the number of columns to orthogonalize.

element_op(const RefSCElementOp&) pure
element_op(const RefSCElementOp2&, SCMatrix*) pure
element_op(const RefSCElementOp3&, SCMatrix*, SCMatrix*) pure
 Perform the element operation *op* on each element of this.

print(ostream&) virtual
print(const char* title=0, ostream& out=cout, int =10) pure
 Print out the matrix.

messagegrp() → RefMessageGrp
 Returns the message group used by the matrix kit

local_blocks(SCMatrixSubblockIter::Access) pure → RefSCMatrixSubblockIter
 Returns iterators for the local (rapidly accessible) blocks used in this matrix.

all_blocks(SCMatrixSubblockIter::Access) pure → RefSCMatrixSubblockIter
 Returns iterators for the all blocks used in this matrix.

4.4 The SymmSCMatrix Class

The SYMMSCMATRIX class is the abstract base class for symmetric double valued matrices.

4.4.1 The Public SymmSCMatrix Interface

kit() const → RefSCMatrixKit
 Return the SCMATRIXKIT object that created this object.

save(StateOut&) virtual
restore(StateIn&) virtual
 Save and restore this in an implementation independent way.

maxabs() virtual → double
 Return the maximum absolute value element of this vector.

randomize() virtual
 Assign each element to a random number between -1 and 1

assign(double val) virtual
 Set all elements to *val*.

assign(const double* m) virtual
 Assign element *i, j* to $m[i*(i+1)/2+j]$.

assign(const double m) virtual**
 Assign element *i, j* to $m[i][j]$.

convert(double*) virtual
convert(double) virtual**
 Like the assign members, but these write values to the arguments.

convert(SymmSCMatrix*) virtual

convert_accumulate(SymmSCMatrix*) virtual
 Convert an SCSYMMSCMATRIX of a different specialization to this specialization and possibly accumulate the data.

assign(SymmSCMatrix* m) virtual
 Make this have the same elements as *m*. The dimensions must match.

scale(double) virtual
 Multiply all elements by *val*.

scale_diagonal(double) virtual
 Scale the diagonal elements by *val*.

shift_diagonal(double) virtual
 Shift the diagonal elements by *val*.

unit() virtual
 Make this equal to the unit matrix.

n() → int
 Return the dimension.

copy() virtual → SymmSCMatrix *
 Return a matrix with the same dimension and same elements.

clone() virtual → SymmSCMatrix *
 Return a matrix with the same dimension but uninitialized memory.

dim() const → RefSCDimension
 Return the dimension.

get_element(int, int) pure → double

set_element(int, int, double) pure

accumulate_element(int, int, double) pure
 Return or modify an element.

get_subblock(int br, int er, int bc, int ec) pure → SCMatrix *

get_subblock(int br, int er) pure → SymmSCMatrix *
 Return a subblock of this. The subblock is defined as the rows starting at *br* and ending at *er*, and the columns beginning at *bc* and ending at *ec*.

assign_subblock(SCMatrix *m, int, int, int, int) pure

assign_subblock(SymmSCMatrix *m, int, int) pure
 Assign *m* to a subblock of this.

accumulate_subblock(SCMatrix *m, int, int, int, int) pure

accumulate_subblock(SymmSCMatrix *m, int, int) pure
 Sum *m* into a subblock of this.

get_row(int i) pure → SCVector *
 Return a row of this.

assign_row(SCVector *v, int i) pure
 Assign *v* to a row of this.

accumulate_row(SCVector *v, int i) pure
 Sum *v* to a row of this.

diagonalize(DiagSCMatrix*d, SCMatrix*m) pure
 Diagonalize this, placing the eigenvalues in *d* and the eigenvectors in *m*.

accumulate(SymmSCMatrix* m) pure
 Sum *m* into this.

accumulate_symmetric_sum(SCMatrix*) pure
accumulate_symmetric_product(SCMatrix*) virtual
accumulate_transform(SCMatrix*, SymmSCMatrix*) virtual
accumulate_transform(SCMatrix*, DiagSCMatrix*) virtual
accumulate_transform(SymmSCMatrix*, SymmSCMatrix*) virtual
accumulate_symmetric_outer_product(SCVector*) virtual
 Sum into this the products of various vectors or matrices.
scalar_product(SCVector* v) virtual $\rightarrow$ double
 Return the scalar obtained by multiplying this on the left and right by *v*.
trace() pure $\rightarrow$ double
 Return the trace.
invert_this() pure $\rightarrow$ double
 Invert this.
determ_this() pure $\rightarrow$ double
 Return the determinant of this. this is overwritten.
element_op(const RefSCElementOp&) pure
element_op(const RefSCElementOp2&, SymmSCMatrix*) pure
element_op(const RefSCElementOp3&, SymmSCMatrix*, SymmSCMatrix*) pure
 Perform the element operation *op* on each element of this.
print(ostream&) virtual
print(const char* title=0, ostream& out=cout, int =10) virtual
 Print out the matrix.
messagegrp() $\rightarrow$ RefMessageGrp
 Returns the message group used by the matrix kit
local_blocks(SCMatrixSubblockIter::Access) pure $\rightarrow$ RefSCMatrixSubblockIter
 Returns iterators for the local (rapidly accessible) blocks used in this matrix.
all_blocks(SCMatrixSubblockIter::Access) pure $\rightarrow$ RefSCMatrixSubblockIter
 Returns iterators for the all blocks used in this matrix.

4.5 The DiagSCMatrix Class

The SYMMSCMATRIX class is the abstract base class for diagonal double valued matrices.

4.5.1 The Public DiagSCMatrix Interface

kit() const $\rightarrow$ RefSCMatrixKit
 Return the SCMATRIXKIT used to create this object.
save(StateOut&) virtual
restore(StateIn&) virtual
 Save and restore this in an implementation independent way.

maxabs() virtual → double
 Return the maximum absolute value element of this vector.

randomize() virtual
 Assign each element to a random number between -1 and 1

assign(double val) virtual
 Set all elements to *val*.

assign(const double*) virtual
 Assign element *i*, *i* to *m[i]*.

convert(double*) virtual
 Like the assign member, but this writes values to the argument.

convert(DiagSCMatrix*) virtual

convert.accumulate(DiagSCMatrix*) virtual
 Convert an SCDIAGSCMATRIX of a different specialization to this specialization and possibly accumulate the data.

assign(DiagSCMatrix*) virtual
 Make this have the same elements as *m*. The dimensions must match.

scale(double) virtual
 Multiply all elements by *val*.

n() const → int
 Return the dimension.

copy() virtual → DiagSCMatrix *
 Return a matrix with the same dimension and same elements.

clone() virtual → DiagSCMatrix *
 Return a matrix with the same dimension but uninitialized memory.

dim() const → RefSCDimension
 Return the dimension.

get_element(int) pure → double

set_element(int, double) pure

accumulate_element(int, double) pure
 Return or modify an element.

accumulate(DiagSCMatrix* m) pure
 Sum *m* into this.

trace() pure → double
 Return the trace.

determ_this() pure → double
 Return the determinant of this. this is overwritten.

invert_this() pure → double
 Invert this.

gen_invert_this() pure
 Do a generalized inversion of this.

element_op(const RefSCElementOp&) pure

element_op(const RefSCElementOp2&, DiagSCMatrix*) pure

element_op(const RefSCElementOp3&, DiagSCMatrix*, DiagSCMatrix*) pure
 Perform the element operation *op* on each element of this.

print(ostream&) virtual

```

print(const char* title=0, ostream& out=cout, int =10) virtual
    Print out the matrix.
messagegrp() → RefMessageGrp
    Returns the message group used by the matrix kit
local_blocks(SCMatrixSubblockIter::Access) pure → RefSCMa-
trixSubblockIter
    Returns iterators for the local (rapidly accessible) blocks used in this ma-
    trix.
all_blocks(SCMatrixSubblockIter::Access) pure → RefSCMatrix-
SubblockIter
    Returns iterators for the all blocks used in this matrix.

```

5 Matrix Storage

All elements of matrices and vectors are kept in blocks. The choice of blocks and where they are kept is left up to each matrix specialization.

5.1 The SCMatrixBlock Class

SCMATRIXBLOCK is the base class for all types of blocks that comprise matrices and vectors.

5.1.1 The Public SCMatrixBlock Interface

```

deepcopy() const virtual → SCMatrixBlock *
    Return of copy of this. A runtime error will be generated for blocks that
    cannot do a deepcopy. These routines are only used internally in the
    matrix library.
dat() virtual → double *
ndat() const virtual → int
    Return a pointer to the block's data and the number of elements in the
    block. Some blocks cannot provide this information and a runtime error
    will be generated if these members are called. These routines are only
    used internally in the matrix library.
process(SCElementOp*) pure
process(SCElementOp2*, SCMatrixBlock*) pure
process(SCElementOp3*, SCMatrixBlock*, SCMatrixBlock*)
pure
    These routines are obsolete.

```

5.2 The SCMatrixRectBlock Class

The SCMATRIXRECTBLOCK describes a rectangular piece of a matrix. The following bit of code illustrates the data layout:

```

fill(double **matrix, SCMatrixRectBlock &b)
{

```



```

    int offset=0;
    for (int i=b.istart; i<b.iend; i++) {
        for (int j=b.jstart; j<b.jend; j++,offset++) {
            matrix[i][j] = b.data[offset];
        }
    }
}

```

5.3 The SCMatrixRectSubBlock Class

The SCMATRIXRECTSUBBLOCK describes a rectangular piece of a matrix. The following bit of code illustrates the data layout:

```

fill(double **matrix, SCMatrixRectSubBlock &b)
{
    int offset=b.istart * b.istride + b.jstart;
    for (int i=b.istart; i<b.iend; i++) {
        for (int j=b.jstart; j<b.jend; j++,offset++) {
            matrix[i][j] = b.data[offset];
        }
        offset += b.istride - (b.jend - b.jstart);
    }
}

```

5.4 The SCMatrixLTriBlock Class

The SCMATRIXLTRIBLOCK describes a triangular piece of a matrix. The following bit of code illustrates the data layout:

```

fill(double **matrix, SCMatrixLTriBlock &b)
{
    int offset=0;
    for (int i=b.start; i<b.end; i++) {
        for (int j=b.start; j<=i; j++,offset++) {
            matrix[i][j] = b.data[offset];
        }
    }
}

```

5.5 The SCMatrixLTriSubBlock Class

The SCMATRIXLTRISUBBLOCK describes a triangular subblock of a matrix. The following bit of code illustrates the data layout:

```

fill(double **matrix, SCMatrixLTriSubBlock &b)
{
    int offset=(b.istart*(b.istart+1)>>1) + b.jstart;

```

```

    for (int i=b.start; i<b.end; i++) {
        for (int j=b.start; j<=i && j<b.jend; j++,offset++) {
            matrix[i][j] = b.data[offset];
        }
        if (j>i) offset += b.istart;
        else offset += i + b.jstart - b.jend;
    }
}

```

5.6 The SCMatrixDiagBlock Class

The SCMATRIXDIAGBLOCK describes a diagonal piece of a matrix. The following bit of code illustrates the data layout:

```

fill(double **matrix, SCMatrixDiagBlock &b)
{
    int i,j,offset=0;
    for (i=b.istart,j=b.jstart; i<b.iend; i++,j++,offset++) {
        matrix[i][j] = b.data[offset];
    }
}

```

5.7 The SCMatrixDiagSubBlock Class

The SCMATRIXDIAGSUBBLOCK describes a diagonal subblock of a matrix. The following bit of code illustrates the data layout:

```

fill(double **matrix, SCMatrixDiagSubBlock &b)
{
    int i,j,offset=b.offset;
    for (i=b.istart,j=b.jstart; i<b.iend; i++,j++,offset++) {
        matrix[i][j] = b.data[offset];
    }
}

```

5.8 The SCVectorSimpleBlock Class

The SCVECTORSIMPLEBLOCK describes a piece of a vector. The following bit of code illustrates the data layout:

```

fill(double *vector, SCVectorSimpleBlock &b)
{
    int i,offset=0;
    for (i=b.istart; i<b.iend; i++,offset++) {
        vector[i] = b.data[offset];
    }
}

```

```

    }
  }
}

```

5.9 The SCVectorSimpleSubBlock Class

The SCVECTORSIMPLESUBBLOCK describes a subblock of a vector. The following bit of code illustrates the data layout:

```

fill(double *vector, SCVectorSimpleSubBlock &b)
{
    int i,offset=b.offset;
    for (i=b.istart; i<b.iend; i++,offset++) {
        vector[i] = b.data[offset];
    }
}

```

6 Manipulating Matrix Elements with Element Operations

6.1 The SCElementOp Class

Objects of class SCELEMENTOP are used to perform operations on the elements of matrices. When the SCELEMENTOP object is given to the element_op member of a matrix, each block the matrix is passed to one of the process, process_base, or process_base members.

6.1.1 The Public SCElementOp Interface

has_collect() virtual → int

defer_collect(int) virtual

collect(const RefMessageGrp&) virtual

If duplicates of the SCELEMENTOP exist (that is, there is more than one node), then if has_collect returns nonzero then collect is called with a MESSAGEGRP reference after all of the blocks have been processed. The default return value of has_collect is 0 and collect's default action is do nothing. If defer_collect member is called with nonzero, collect will do nothing (this is only used by the blocked matrices).

has_side_effects() virtual → int

By default this returns nonzero. If the ELEMENTOP specialization will change any elements of the matrix, then this must be overridden to return nonzero.

process(SCMatrixBlockIter&) pure

This is the fallback routine to process blocks and is called by process_spec members that are not overridden.

process_base(SCMatrixBlock*block)

Lazy matrix implementors can call this member when the type of block specialization is unknown. However, this will attempt to castdown *block* to a block specialization and will thus be less efficient.

process_spec(SCMatrixRectBlock*) virtual

process_spec(SCMatrixLTriBlock*) virtual

process_spec(SCMatrixDiagBlock*) virtual

process_spec(SCVectorSimpleBlock*) virtual

process_spec(SCMatrixRectSubBlock*) virtual

process_spec(SCMatrixLTriSubBlock*) virtual

process_spec(SCMatrixDiagSubBlock*) virtual

process_spec(SCVectorSimpleSubBlock*) virtual

Matrices should call these members when the type of block is known. ELEMENTOP specializations should override these when efficiency is important, since these give the most efficient access to the elements of the block.

6.2 The SCElementOp2 Class

The SCELEMENTOP2 class is very similar to the SCELEMENTOP class except that pairs of blocks are treated simultaneously. The two matrices involved must have identical storage layout, which will be the case if both matrices are of the same type and dimensions.

6.3 The SCElementOp3 Class

The SCELEMENTOP3 class is very similar to the SCELEMENTOP class except that a triplet of blocks is treated simultaneously. The three matrices involved must have identical storage layout, which will be the case if all matrices are of the same type and dimensions.

6.4 The SCMatrixBlockIter Class

The SCMATRIXBLOCKITER class is used to described iterates that loop through the elements in a block.

6.4.1 The Public SCMatrixBlockIter Interface

i() pure → int

Returns the row index.

j() pure → int

Returns the column index.

set(double val) pure

Set the current element to *val*.

accum(double val) virtual

Add *val* to the current element.

get() pure $\rightarrow$ double
 Return the value of the current element.
operator int() pure $\rightarrow$ int
 Return nonzero if there are more elements.
operator ++() pure
operator ++(int)
 Move to the next element.
reset() pure
 Start the iteration over.

7 SCElementOp Specializations

Several commonly needed element operations are already coded up and available by including `math/scmat/elemop.h`. Below are descriptions of these classes:

SCElementScalarProduct This SCELEMENTOP2 computes the scalar product of two matrices or vectors. The result is available after the operation from the return value of the `result()` member.

SCDestructiveElementProduct This SCELEMENTOP2 replaces the elements of the matrix or vector whose `element_op` member is called. The resulting values are the element by element products of the two matrices or vectors.

SCElementScale This scales each element by an amount given in the constructor.

SCElementRandomize This generates random elements.

SCElementAssign Assign to each element the value passed to the constructor.

SCElementSquareRoot Replace each element with its square root.

SCElementInvert Replace each element by its reciprocal.

SCElementScaleDiagonal Scales the diagonal elements of a matrix by the argument passed to the constructor. Use of this on a vector is undefined.

SCElementShiftDiagonal Add the value passed to the constructor to the diagonal elements of the matrix. Use of this on a vector is undefined.

SCElementMaxAbs Find the maximum absolute value element in a matrix or vector. The result is available as the return value of the `result()` member.

SCElementDot The constructor for this class takes three arguments: `SCElementDot(double**a, double**b, int length)`. The length of each vector given by *a* and *b* is given by *length*. The number of vectors in *a* is the number of rows in the matrix and the number in *b* is the number of columns. To each element in the matrix m_{ij} the dot product of the a_i and b_j is added.

SCElementAccumulateSCMatrix This is obsolete—do not use it.

SCElementAccumulateSymmSCMatrix This is obsolete—do not use it.

SCElementAccumulateDiagSCMatrix This is obsolete—do not use it.

SCElementAccumulateSCVector This is obsolete—do not use it.

8 Manipulating Matrix Elements with Block Iterators

8.1 The SCMatrixSubblockIter Class

Objects of class **SCMATRIXSUBBLOCKITER** are used to iterate through the blocks of a matrix. The object must be deleted before using the matrix that owns the blocks that **SCMATRIXSUBBLOCKITER** is iterating through.

8.1.1 The Public SCMatrixSubblockIter Interface

SCMatrixSubblockIter(Access access)

The *access* variable should be one of **Read**, **Write**, **Accum**, and **None**, with the **SCMatrixSubblockIter::scope** operator applied.

begin() pure

Start at the beginning.

ready() pure → int

Returns nonzero if there is another block.

next() pure

Proceed to the next block.

block() pure → **SCMatrixBlock** *

Return the current block.

access() const → Access

Return the type of Access allowed for these blocks.

8.2 Local Matrices

Local matrices do no communication. All elements reside on each node and all computations are duplicated on each node.

8.3 Replicated Matrices

Replicated matrices hold all of the elements on each node, however do some communications in order to reduce computation time.

8.4 Distributed Matrices

Distributed matrices spread the elements across all the nodes and thus require less storage than local matrices however these use more communications than replicated matrices.

8.5 Blocked Matrices

Blocked matrices are used to implement point group symmetry. Another matrix specialization is used to hold the diagonal subblocks of a matrix. The offdiagonal subblocks are known to be zero and not stored. This results in considerable savings in storage and computation for those cases where it applies.

**UNLIMITED RELEASE
INITIAL DISTRIBUTION:**

MS 9001 Tom Hunter, 8100
 Attn: 2200 J. B. Wright
 5200 E. E. Ives
 8100 M. E. John
 8200 L. A. West
 8300 W. J. McLean
 8400 R. C. Wayne
 8500 P. N. Smith
 8700 T. M. Dyer
 8800 P. E. Brewer
 8900 D. L. Crawford

MS 9214 L. M. Napolitano, Jr., 8117
MS 9214 M. E. Colvin, 8117 (3)
MS 9214 C. L. Janssen, 8117
MS 9214 R. S. Judson, 8117
MS 9214 W. E. Mason, Jr., 8117
MS 9214 J. C. Meza, 8117
MS 9201 M. M. Johnson, 8114
MS 9201 Ann Yoshimura, 8114

MS 9021 Technical Communications Dept., 8815 for OSTI (10)
MS 9021 Technical Communications Dept., 8815/Technical Library, MS 0899, 4414
MS 0899 Technical Library, 4414 (4)
MS 9018 Central Technical Files, 8940-2 (3)