SimTK Simmatrix™

A SimTK Toolset for efficient manipulation of vectors and matrices in C++

Michael Sherman

Version 2.1, January, 2010

Abstract

We describe the goals and design decision behind Simmatrix, the SimTK matrix and linear algebra library (toolset) for C++ programmers, and provide reference information for using it. The idea is to provide the power, naturalness, and flexibility of Matlab from within a C++ program, but with maximal performance and convenient interoperability with numerical libraries and custom code which may already exist in various languages, including C and FORTRAN.

1 Purpose of this document

To describe the goals, design issues, theory, implementation and use of the SimTK Simmatrix C++ toolset. Simmatrix is a part of the basic infrastructure for use of the SimTK Core tools. It is delivered as part of the SimTKcore project on simtk.org (https://simtk.org/home/simtkcore), and developed within the SimTKcommon project there (https://simtk.org/home/simtkcommon).

2 Goals

Here is what we hope to achieve with Simmatrix.

2.1 Speed

Perhaps it seems crass to start with this topic, but computer simulations are dominated in practice by performance considerations. As a result, few practitioners will make use of a facility, however lovely, that slows down their programs. A rule of thumb among computational scientists is that anything more than 20% overhead will begin to affect adoption by high-end users.

When designing a system for dealing with matrices, there are two completely different and mutually incompatible performance issues that must be addressed. Computations with small vectors and matrices, such as the 3-element ones needed for representing points and orientations in 3d, are dominated by overhead because each calculation is so small. Computations with large matrices, on the other hand, are dominated by large-scale repetitive computations and memory access patterns. Performance of large computations can be dramatically improved by a constant-time overhead spent determining the optimal execution strategy.

To deal with these conflicting concerns successfully, we have adopted the commonly-used approach of building what are essentially two completely independent facilities, one suited for small objects, and one for large. In the former, we do anything necessary to avoid overhead. In the latter, we do whatever it takes to optimize computation and memory access patterns. The facilities are then designed to work smoothly together, without having to compromise performance. A purely interpretive environment like Matlab can optimize only the performance of the large computations, while continuing to incur the same overhead on small ones.

2.2 Accuracy

Performing linear algebra correctly on a computer is a completely different field from linear algebra as taught in mathematics classes. The finite precision of computers is the dominant issue in performing correct computation, while that issue is completely irrelevant in mathematics. The best strategy here is to stand on the shoulders of the giants who have devoted their careers to computational methods for linear algebra. In current practice, that means performing linear algebra using software produced by the decades-long U.S. Department of Energy (DOE) effort to produce reliable linear algebra, as embodied in LAPACK and related public domain facilities. This is the technique adopted by the extremely successful Matlab package, and we use it here for our large-matrix facility. We cannot use it directly for the small matrix facility when overhead concerns dominate everything else, but it is always available even for small systems in those cases where highest accuracy is paramount.

2.3 Expressive power

The goal here is best stated in terms of Matlab, by far the most successful matrix handling environment in existence. In fact, before you proceed further you may wish to consider whether your needs might be better met by using Matlab itself. Our facility is not intended as a replacement for Matlab   we are interested in supporting programmers who need to work in C++ for one well-considered reason or another. For those programmers, we would like to provide a Matlab-like capability accessible in a natural way from within a C++ program. We want to minimize the mental gymnastics required to go from a mathematical statement of a computation to its functional implementation. Among other things, that means that real and complex numbers should be supported equally so that the occasional appearance of a complex result does not present an insurmountable disaster.

Like Matlab, we take the mathema w-point that a vector is a vertical object (a column) that is distinct from a horizontal object of the same dimension (a row or covector). We are able to enforce that distinction at compile time with no overhead and to do so provides significant benefits. A matrix is then seen conceptually as a collection of column vectors, making ours fully compatible with both the mathematical treatment of matrices and the compu
best linear algebra software. However, the facility is flexible enough so that anyone who prefers to treat matrices as collections of rows (as is common among computer scientists) can do so without penalty, although we would advise computer scientists not to do this reflexively why not follow mathematical conventions for mathematical objects?

As mentioned above, we have a goal of zero overhead for dealing with small objects, something which is not attainable in an interpreted system. C++ is one of the very few languages in which such a facility can be attempted. Its inheritance, templates, operator overloading, inline functions, and naughty loopholes permitting direct hardware access when necessary offer the opportunity to build extensions which provide elegant, type-safe abstractions with no runtime overhead whatsoever. We are able to provide zero-overhead operators for basic matrix operations like transpose, arithmetic, extraction (without copying) of elements and element subsets such as rows, columns, submatrices, diagonals, real or imaginary parts, etc. As a concrete example, matrix transpose can be seen as a

change in point of view rather than a physical operation and C++ is powerful enough to support that concept so that our (Hermitian) a-
s no computation or memory operations at all.

2.4 API stability

An interface (or API) must satisfy very restrictive criteria compared to general programming. Stability is probably the primary one an interface should be extremely stable once defined because many programs will depend on it for their correct functioning.

SimTK sets the bar higher by promising binary compatibility. That means that software that depends on the SimTK Core interface can take advantage of new releases without being recompiled or relinked in the case of dynamically linked library upgrades. This level of stability requires that objects which appear in the interface must have either very simple, obviously permanent implementations, or opaque implementations which protect client software from inevitable changes to the implementation.

```
Vec<3>            v; // a 3-vector of Reals
Vec3              w; // same thing, using abbr.
Vector            b,x; // vectors of reals
Matrix            M; // mxn matrix of reals
Matrix_<Complex> C; // mxn matrix of complex
Vector_<Vec3>     v3;// big vector of 3-vecs

// This type is a 2-element vector whose elements
// are 3-vectors. Memory layout and computational
// efficiency are identical to Vec<6>.

typedef Vec<2,Vec3> SpatialVec;
```

3.2 Indexing

One of the thorniest issues to decide is how to treat indices, specifically, what is the index of the first element? Scientific programmers used to FORTRAN, Matlab, and general mathematical conventions ex

Many packages address this by making the indexing offset a user choice. We have done this in past designs and found it unsatisfying and extremely prone to induce errors where some programmers (typically at the upper levels of the code) use 1-based indexing while others use 0-based. The resulting awkwardness and uncertainty produces either subtle off-by-one errors, or unnecessary copying as responsible programmers move data into compatibly-indexed vessels to avoid3>-30 1.f abbr.

i(exts)-2(uce)] TJETBT1 491.4 5060 8 671.52 heported precisions. The number types are the standard *real* and *complex* numbers, a-

conjugate, which is not typically used in user programs but is important for the efficient implementation of matrix operations.*

Each of the three precision types is also a real number type, with the SimTK type `Real` predefined equivalent to one of those as described above. The C++ standard template type `std::complex`, specialized as `complex<float>`, `complex<double>`, and `complex<long double>` serve as our complex numbers, with SimTK default-precision type `Complex` predefined as `complex<Real>`. The three conjugate types (from the `SimTK` namespace) are `conjugate<float>`, `conjugate<double>`, and `conjugate<long double>`, with default-precision type `Conjugate` predefined as `conjugate<Real>`.

4.3 Scalar types

Only `Real` and `Complex`, and vectors and matrices defined in terms of them, will appear in typical user programs. However, our complete set of scalar numeric types consists of the nine number types described above, plus a templatized adaptor class `negator<`*number*`>`, which may be applied to any number type to create a new type whose memory representation is unchanged but whose value is to be interpreted with the opposite sign. Like `conjugate`, `negator` is not expected to appear in user programs but permits efficient implementation of some matrix operations, in particular Hermitian transpose of a complex matrix. The imaginary part of a `conjugate` number has type `negator<`*real*`>` for the appropriate precision real.

* Conjugate types have the same representation as complex types, but the imaginary part is interpreted with opposite sign.

The one difference between our complex type and the C++ standard is that we *do not* initialize unused values in a Release build. That is, we treat complex identically to real in this regard, while the C++ standard specifies that complex variables are initialized to (0,0). We feel that is an inappropriate default for large, complex matrices where avoiding unnecessary memory accesses is of primary concern.

4.4 Scalar summary

The above defines a set of exactly 18 scalar types: three kinds of numbers in each of three precisions, in normal or negated form. Despite the use of templates, this is not a user-extendable set.

Stated as a grammar, scalars are defined like this:

scalar	::= *number* | <*number*>
number	::= *standard* | *conjugate*
standard	::= *real* | *complex* rge. t448.88] [2> BDC te 1 0 0 1 134.42 412.
conjugate	::= |

18F3 10.05 Tf 1405.1 1 B 119M 592.9 Tm 0.015 Tc[( )]5 0.03 Tc[(n-)] reET B 0 1 324.76 616.36 Tm 0 Tc[(sc)7(ala)0 [(c5TJ ET BT /F1 symmetric matrix has a different type from a general one, so that the appropriate access pattern can be determined at compile time. CNTs make extensive use of C++ templates and inline functions, and the fact that type casting is free, to permit compile-time optimization into the optimal set of machine instructions.

The table below presents the available Composite Numerical Types, or more precisely the templates available for constructing CNTs. Note that predefined abbreviations are available for certain common combinations of template arguments (up to 9 rows and columns). We provide those because we know from experience that programmers will de- <> eated by the C++ template syntax, and we would like to encourage a consistent set of common abbreviations. These abbreviations are *not* distinct types (they are `typedef`s), so may be freely intermingled with the spelled-out types. For example, a `Vec3` can be passed to a routine written to take a `Vec<3>` argument.

Type	Description
`Real` `Complex`	A floating point number at default precision, either real or complex.

`Vec<`*m*`>`
`Vec<`*m,C*`>`
`Vec<`*m,C,stride*`>`
 	
`Vec2 Vec3 Vec4`
`Vec5 … Vec9`
	A short, fixed-length column vector of *m* elements of composite numerical type C (default `Real`) with optional element-to-element stride (default 1). Typedef abbreviations are provided as shown, with `Vec3  Vec<3>`, etc.

`Row<`*n*`>`
`Row<`*n,C*`>`
`Row<`*n,C,stride*`>`
 	
`Row2 Row3 Row4`
`Row5 … Row9`
	A short, fixed-length row vector of *n* elements of composite numerical type C (default `Real`) with optional element-to-element stride (default 1). `Row`s are not typically used in user programs, but occur as intermediate results in expressions.

`Mat<`*m,n*`>`
`Mat<`*m,n,C*`>`
`Mat<`*m,n,C,cs,rs*`>`
 	
`Mat22 Mat33 …`
`Mat66 … Mat76`
`… Mat89 Mat99`
	A small, fixed-size matrix of *m* rows and *n* columns of composite numerical type C (default `Real`) with optional column-to-column spacing *cs* (default *m*), and row-to-row spacing *rs* (default 1).
	These typedefs are provided, with `Mat33  Mat<3,3>`, etc.

`SymMat<`*m*`>`
`SymMat<`*m,C*`>`
`SymMat<`*m,C,rs*`>`
 	
`SymMat22`
`SymMat33`
`SymMat44`
`… SymMat99`
	A small, fixed-size *m*x*m* symmetric (Hermitian if complex) matrix of composite numerical type C (default `Real`) with optional element-to-element spacing (default 1). Only the elements of the diagonal and lower triangle are stored.
	These typedefs are provided, with `SymMat33  SymMat<3>`, etc.

Note that the short `Vec` and `Mat` types are themselves Composite Numerical Types and can thus be composed recursively. That is, it is reasonable to have a 2x2 matrix of 3x3 matrices and in fact this can be quite useful. This can be declared[*]

```
    Mat<2,2,Mat<3,3> >
```

or more pleasantly using a predefined typedef

```
    Mat<2,2,Mat33>
```

Note that these types are exactly equivalent and thus interchangeable, and that the resulting type is itself a CNT.

5.1 Memory layout of CNTs; packed CNTs

The default layout for any CNT is to pack the elements into the least amount of consecutive storage as logically required to hold the value. We refer to a CNT stored this way as a *packed CNT*. In addition, each of the CNT templates provides arguments which can be used to specify regular gaps in the storage layout, where the gap size is always an integer multiple of the storage requirement of ement type. For example, the `Vec` and `Row` templates allow specification of a *stride*, which gives the spacing between consecutive elements in terms of those elements. So a packed `Vec` or `Row` has stride 1, which is the default.

Non-packed CNTs exist to facilitate reinterpretation of existing data in terms of CNTs. For example, if one has a `Mat<4,3>` stored as three consecutive packed `Vec<4>` columns, the rows can be viewed as a 3-element `Row` CNT with a stride of 4 (that is, four `Real` elements), which could be specified as `Row<3,Real,4>`. However, such declarations do not normally appear in user programs;

[*] C++ unfortunately requires the extra space for nested tem >>

instead, they are the hidden return types of methods and operators which select out portions of an existing object. In this case, the row index operator `m[i]` acting on a `Mat<4,3>` matrix `m` returns the *i*th row of `m` as a 3-element `Row` with stride 4, meaning it references the elements of `m` without copying, and can serve as an lvalue (target of an assignment) which alters the appropriate elements of `m`. Because this type has the same semantics as any other `Row<3>`, most users will never need to think about how it is implemented.

When we discuss the large-matrix facility below, the distinction between packed and non-packed CNTs is somewhat more significant, since only packed CNTs can serve as element types for the large `Vector` and `Matrix` classes.

5.1.1 CNT packing vs. compiler packing

Some C++ compilers attempt to improve execution

size which is particularly efficient for the targeted hardware. For example, a class containing three 4-byte `float` values (e.g., `Vec<3,float>`) might be allocated 16 bytes rather than 12, by some compilers on some machines, with some compile-time options. That means that C++ arrays of such objects would contain 4-byte gaps between the elements.

Because a great deal of our performance comes from the ability to change our point of view (i.e., recast) rather than copy or compute, we depend on predictable storage layouts for our classes. To ensure that, we define packed CNTs in terms of the storage requirements of their underlying scalar types, which are packed the same way by all compilers on all machines. We guarantee, for example, that a CNT `Vec<2,Vec3>` can be recast to a `Vec<6>` with the obvious interpretation and memory layout identical to a C++ array `Real[6]`. This would not necessarily be possible if the CNT stored the two `Vec3` `Vec3[2]` since the compiler might choose to allocate space for two `Vec4` nstead!

The key point to remember is that CNTs are packed internally as arrays of scalars, so that a composite CNT whose elements are also composite CNTs may occupy less storage than a C++ array of those same elements. So while you can always cast a CNT into a C++ scalar array with predictable results (for any of the supported scalar types including complex), you cannot safely cast to a C++ array of composite CNT types; cast to a CNT `Vec` of those types instead.

5.2 *Construction and assignment of CNTs*

All CNTs define a default constructor. In Debug mode (that is, when the C++ standard `NDEBUG` preprocessor symbol is *not* defined), the default constructor initializes all elements to `NaN`. In Release mode (i.e., `NDEBUG` *is* defined), all elements are left uninitialized, so that declaring CNTs, or arrays of CNTs, has no cost if the variables are not used.

Constructors are also available for initializing data elements from individual element values, or by copying compatible CNTs. Initialization values can be provided in the constructor or via a pointer (or C array) to values of the appropriate type.

Assignment operators are available for copying one CNT to another, and for setting single elements, subvectors and submatrices.

One important convention we follow, which is different than that of most similar systems, is the treatment of scalar assignment. We follow this convention: (1) when a scalar *s* is assigned to a vector, every element of the vector is set to *s* (this is the typical convention), and (2) when a scalar *s* is assigned to a matrix, the *diagonal* elements of the matrix are set to *s* while the off-diagonals are set to zero. Examples:

```
Vec3   v;
Mat22  m;
Vector b(10);     // initial size 10 reals
Matrix M(20,10); // initial size 20x10
v=0; // v=(0,0,0)
v=3; // v=(3,3,3)
m=0; // m=( 0,0 )
     //   ( 0,0 )
m=1; // m=( 1,0 )
     //   ( 0,1 )
b=0; // b=10 zeroes
M=1; // M=0, except M(i,i)=1, 0<=i<10
```

This convention is especially apt for matrices, because the matrix resulting from such a scalar as- alar. That is, if you multiply by this matrix the result is identical to a scalar multiply by the original scalar. Two impor-

tant special cases are: (1) setting a matrix to the
a-
trix of that shape, and (2) setting a matrix to the

that shape. In general, in any operation involving a scalar *s* and a `Matrix` or `Mat`, the scalar is treated as if it were a conforming matrix whose main diagonal consists of all *s* with all other elements zero. So `Matrix` *m* += *s* will result in *s* being added to *m* *s* were replaced by `diag`(*s*) of the same dimension as *m*. *m*−=`1` thus subtracts an identity matrix from *m*, *without* touching any of the off-diagonal elements. Note that for multiply and divide this convention yields the ordinary scalar multiply and divide operations: e.g., *m***s* (=*m**`diag`(*s*)) multiplies every element of *m* by *s*.

5.3 Operators on CNTs

There are numerous operators available which act on CNTs, for element access, computation, reinterpretation of the data, and obtaining information about the type and its contents.

5.3.1 Element access

These operators provide access to individual elements, or subsets of elements, of composite numerical types, where we use letters to indicate types: s=scalar, e=element (of whatever CNT), v=Vec, r=Row, m=Mat, sy=SymMat. *i*,*j* are integer indices, with *i* a row index and *j* a column index.
left hand side of an assignment statement, with the result affecting the original element values.

Operator	Applied to	Meaning	Lvalue?	Cost	Notes
v[*i*] v(*i*) r[*j*] r(*j*)	Vec Row	select *i*th (*j*th) element	yes	native array index	all indexing is 0-based
m[*i*][*j*] m(*i*,*j*) sy[*i*][*j*] sy(*i*,*j*)	Mat SymMat	obtain *i*,*j* element of m. Only diag & lower triangle of SymMat; i.e., *i* *j*.	yes	same as native matrix index for Mat; extra integer operations for SymMat.	
m[*i*] m.row(*i*) m(*j*) m.col(*j*)	Mat	obtain *i*th row or *j*th column of m as Row or Vec, resp.	yes	native array index	Size and spacing are taken from Mat; typically columns are packed while Rows have stride>1.

m.diag()
sy.diag()
	Mat
	SymMat
		obtain diagonal as a Vec	yes	zero	For rectangular Mat<*m*,*n*>, result has dimension min(*m*,*n*).

sy[

2- and 3- vectors, and the usual C arithmetic assignment operat s with scalar elements are as expected; behavior for CNTs with composite elements are defined analogously and generally work well, but most users will not have a well-developed intuition for those objects at first.

Operator	**Applied**

5.4 Summary of CNTs

Stated loosely as a grammar, we build CNTs recursively like this:

```
CNT         ::=   scalar
                | composite  size [  CNT [  packing] ]c
composite   ::=       |      |      |
size        ::= nrow [  ncol]
packing     ::= stride | colSpacing  rowSpacing
```

The unbolded terminals `nrow`, `ncol`, `stride`, `colSpacing`, and `rowSpacing` are integers, or compile-time expressions that evaluate to integers.

This grammar permits unlimited nesting of these constructs, and the implementation does work that way, but we would counsel restraint here and note that there is unlikely to be much utility (or clarity) beyond two or three levels deep.

6 Types for linear algebra

[This part of the document is very sparse at the moment (no pun intended).]

6.1 Large Vector and Matrix types

T   Composite Numerical Types limits their flexibility. For larger vectors and matrices, some constant-time overhead is acceptable since we expect time to be dominated by floating point calculations and memory accesses done on the (large) operands. In fact, this overhead is desirable since it is used to set up optimal large-scale operations which can then be performed at machine speeds. The basic types, and general behavior, are modeled after the very successful Matlab system with the expectation (but not requirement) of LAPACK and BLAS style implementation. We assume that these objects will be very large and the classes are carefully designed to avoid unnecessary data copying and memory references.

The underlying element type stored in our large matrix objects can be any scalar type or other *packed* composite numerical type (see section 5.1). These elements will be packed adjacent in memory in our large matrix objects regardless of whether the C++ compiler would pack them that tightly when creating its own arrays. Other data layouts are available if explicitly requested, but packing of elements is always done by packing the underlying scalars as discussed for CNTs in section 5.1.1.

Type	**Description**
`Vector` `Vector_`<*C*>	An arbitrary-length column of `Real` values, or of values of packed composite numerical type *C* (e.g., `Vector_<Complex>` or `Vector_<Mat<2,2,Mat33>` >).
`RowVector` `RowVector_`<*C*>	Same as `Vector` but horizontal. Usually not used explicitly in code, but is the type of a `Matrix` row or `Vector` transpose.
`Matrix` `Matrix_`<*C*>	An arbitrary-size, two dimensional matrix of `Real` values, or of values of packed composite numerical type *C*.

Standard linear algebra operations, matrix decompositions, and interconversions with composite numerical types are provided. Note that SimTK `Vector` and `Matrix` are not themselves composite numerical types and may *not* be composed recursively. A `Vector` may be considered an *m*x1 `Matrix`, and a `RowVector` a 1x*n* `Matrix` when convenient. Thus the discussion below which refers to matrices applies to `Vector` and `RowVector` as well.

Unlike the Composite Numerical Types, very little is encoded in the type here   only the basic shape outline (1 or 2d object) and the element type. Dimensions, spacing, and internal data layout are determined at runtime. This provides a great deal of useful flexibility but imposes a constant-time cost for every operation.

SimTK provides 0-based indexing using the `[]` operator. If the `Matrix` is modifiable (non-const) then the indexed element can be modified and that change affects the contents of the object. The `[]` operator applied to a `Matrix` returns a row, which may in turn be indexed to obtain an element in C style. SimTK also permits indexing using round brackets `()` yielding identical results to `[]` for `Vector` but selecting a column rather than a row when applied to a `Matrix`. A two-argument round bracket operator accesses a `Matrix` element, and unlike for CNTs, it is more efficient to use the two-argument form here since the overhead cost is paid only once.

```
Matrix m; Vector v; …
v[i]          // ref to i^th element of v, 0-based
```

```
v(i)        //   same
m[i][j]     // ref to i,jth element of m, 0-based
m(i,j)      //   same, but faster
m[i]        // ref to ith row of m, 0-based
m(j)        // ref to jth column of m, 0-based
```

There are also operators for selecting subvectors and submatrices. Like the indexing operators, these return references into the *original* object, not copies. Submatrices are t l- ogy) meaning that they can appear on the left hand side of an assignment.

```
Matrix m; Vector v; …
v(i,m)          // ref to m-element subvector whose 0th
                //   element is v's ith element
m(i,j,m,n)      // ref to mxn submatrix whose (0,0)
                //   element is m's (i,j) element
```

References of this type are called *views* since they provide alternate views of the same data. They retain all properties of the original object except that they cannot be resized. They are in fact represented identically to the original objects in the sense that they can be used wherever a `Vector` or `Matrix` reference is expected, *without* memory allocation or data copying.

The implementations of these types are opaque to a C++ program using them. That is, the header files define these classes which contain only a pointer to an undefined type (essentially a `void*`). The object refere[(68.t)-154(is)1426(an)-144(ols)]TJETBT1 0 0 1 63.36 331.44 Tm[(vi)-2ddnan pleme[(th)-2(ai(or

Thfs

these-4( )716(cle)-2(spes)786(in)-2( )716Sinm infapes whpleprese-4rygin
otms operatoral(e)-3ss withCNT( )] TJETBT1 0 0 1 63.36 807.04 Tm[seleme[(th)-2(s)-43willy oi cemto matrices
erfpoingli - ae( )-118(op)-4(eraicon)-3ie at

m(

a y aso of these

(Th)8is)-06sual(y)-7)--4ccurh)-2s)-06unt

by hiding it under a `Matrix`
to worry about whether the integer indices start at zero.)

TODO: ideas for more: scalar matrices for zero and identity and scalar*identity; sparse matrices in some DOE-compatible format(?)

Where possible, similar storage options are available for any element type, however factoring, pivoting and so on are only defined for scalar elements.

6.3 *Matrix characteristics*

`Matrix m;` `m` as an *uncommitted* matrix handle. That is, although `m` has the semantics of a 0x0 matrix of reals, it has not yet been committed to using a particular data layout. It can be used to hold the results of any matrix operation, and will take on the characteristics of that result. A subsequent use of the handle might leave it with completely different characteristics; only the element type can never change. If an uncommitted handle is asked to allocate space for some data (for `m.resize(10,20);` it will use a dense, column oriented allocation identical to LAPACK .

However, handles can optionally be restricted to narrower ranges of behavior, via *commitments*

the kinds of characteristics that a matrix can possess.

A Simmatrix `Matrix`, `Vector` or `RowVector` object is characterized by the following seven attributes:

1. Element type (a CNT)
2. Outline
3. Size
4. Structure
5. Conditioning
6. Sparsity
7. Storage format

Collectively, we refer to a set of particular values of these attributes as a *matrix character*. There are two matrix characters associated with every matrix commitment, and its current character. The current character always

6.3.1 Matrix character commitments

In general, a `Matrix` handle will be committed with respect to some of the above attributes, and uncommitted to the rest. A character commitment

actual matrix referenced by the handle. For example, a matrix handle committed to symmetric structure cannot be assigned to a nonsymmetric result, but would accept a diagonal result, since every diagonal matrix is also symmetric.

Every `Matrix` (and `Vector` and `RowVector`) is committed to a particular element type, since an element type is required as a template argument in the `Matrix` type itself (recall that `Matrix` itself is an abbreviation for `Matrix_<Real>`). In addition, `Vector` and `RowVector` handles are committed to a particular outline: column and row, respectively.

Any other desired commitments must be added explicitly before the handle is used to hold any data. Many of the characteristics represent categories of acceptable attributes, rather than specific ones. For example, a commitment to a square outline still permits flexibility with regard to the size, as long as both dimensions are the same. The matrix characteristics are not completely independent some imply others. For example, a symmetric structure implies a square outline.

atrix characteristics one by one.

6.3.2 Element type

As mentioned above, every matrix handle is committed to a particular element type by its own templatized type. Only packed CNTs are permitted as element types.

Most operations require exactly matching element types among the operands, with the exception that operands which differ only in the negation or conjugation status of their underlying scalars can be intermingled.

6.3.3 Outline

There are five possible outline attributes:

1. Rectangular (mxn)
2. Square (nxn)
3. Column (mx1)
4. Row (1xn)

    a. Specify leading dimension (default is number of rows)
    b. Symmetric, triangular, diagonal can exist within full storage
    c. Specify upper/lower for symmetric & triangular
    d. Specify whether unit diagonal is assumed
2. Packed (default for banded matrices, space saving for symmetric & triangular but usually considerably slower than using full storage)
    a. Specify whether unit diagonal is assumed
3. Householder product (LAPACK representation for orthogonal matrices)
4. Pivot array (set of integers used to represent Permutation matrices)
5. TODO: look into use of rectangular full-packed storage (Gustavson & Wasniewski) instead of LAPACK packed claim is up to 20X faster.
6. TODO: should we include a stride as a data storage option or is that just a view?

6.4 Matrix views

In addition to being an owner of element data, a matrix handle can also serve as a *view* into some- provides a logical matrix whose elements consist of a subset and/or rearrangement of the elements described by the data descriptor, sometimes augmented with a few generic read-only data elements like 0, 1, and NaN. Views are commonly used to select blocks or diagonals from within a large matrix or to provide a matrix which appears to be transposed relative to the original data. A Matrix view is an lvalue and assignment to the view results in changes to the actual elements of the original data.

The most commonly encountered views are those created by operators such as transpose or row, column and submatrix selection.

A Matrix view is still a Matrix, and can be passed to any Matrix argument. Views may be made of views, with the logically correct result, however the views are combined into a single view rather than nested. So a Matrix always contains at most one view.

Many views can be manipulated as efficiently as the original data, particularly when the original is a full-storage matrix.

6.4.1 Element filters

Whenever possible, we construct a matrix view simply by finding another high-performance description of the desired subset of the data. For example, a view which is a row of an ordinary dense, column-oriented matrix can be represented as a -dimensional object, which is one of the formats which can be manipulated efficiently by the BLAS routines.

It is possible that a desired view cannot be expressed in one of the available high-performance data descriptors. In that case the matrix supplements the data descriptor with an *element filter*. An element filter presents a logical matrix whose elements consist of a subset and/or reordering of the in-memory elements, done in a way that does not map to a supported high speed format. For example, one could construct a view which picked out particular elements, with arbitrary spacing and ordering compared to the originals. These could appear as a contiguous `Vector`, for example, although the individual elements might be widely scattered. Operations on such an object are unlikely to be very efficient, but in many cases the clarity of code will matter more.

6.5 Factorizations

For equation solving, one may always calculate a matrix inverse and then multiply by it; however,

computa
acceptable results in finite precision arithmetic. Simmatrix does allow that approach but it is not recommended. As a better option for one-time use, the divide operator is overloaded to allow casual solution to $\mathbf{Mx}=\mathbf{b}$ by writing $\mathbf{x}=\mathbf{b}/\mathbf{M}$, which means $\mathbf{x}=\mathbf{M}^{-1}\mathbf{b}$ (or $\mathbf{x}=\mathbf{M}^{+}\mathbf{b}$ if a pseudoinverse is necessary). **M** can contain information about its conditioning and structure which permits the operator to make a reasonable choice of solution method; otherwise, Simmatrix will make a conservative choice yielding good numerical results but perhaps suboptimal performance. In any case the divide operator

does not actually form the inverse, but works directly with the factorization which is numerically preferable.

For more control, or for repeated use of the same matrix while factoring only once, one must construct explicit factorizations and then solve equations using the factorization directly rather than using it to invert the original matrix.

Matrix factorizations are objects which can be used similarly to matrix inverses, but with optimal numerical accuracy. See the Simmath documentation example:

```
Matrix M; Vector b1,b2,x1,x2; …
FactorLU f(M); // LU factorization of M
x1 = f*b1; // instead of x1=b1/M
x2 = f*b2;
```

These can be made to yield the best possible results with the highest efficiency, and the `Factor` classes can provide many useful methods such as rank determination. Typically the `Factor` constructor will obtain the layout and known properties from M and then call the appropriate LAPACK routines to perform the factorization. Options exist to allow the `Factor` class to steal the original memory from the matrix being factored.

6.6 Available factorizations TBD

Square, well conditioned matrix: LU with pivoting

Symmetric, general matrix: LL^T

Symmetric, positive definite: Cholesky (LDL^T?)

Rectangular: QR with pivoting, LQ

Rectangular, ill conditioned: QTZ, SVD

Symmetric and nonsymmetric eigenvalues routines and Schur factorization

Access to the underlying factors (without copying!).

Condition number, rank determination/setting, equation solve, inverse and pseudoinverse.

How error conditions are handled.

6.7 Operator reference TBD

Basic Matlab and BLAS equivalents.

Acknowledgments

This work was funded by the National Institutes of Health through the NIH Roadmap for Medical Research,[1] Grant U54 GM072970.

References

[1] Information on the National Centers for Biomedical Computing can be obtained from http://nihroadmap.nih.gov/bioinformatics.