SimTK Simmatrix™

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

Michael Sherman

ROUGH DRAFT Version 0.6, December 28, 2006

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. [Note: this document is only partially written at the moment.]

1 Purpose of this document

To describe the goals, design issues, theory, implementation and use of the SimTK Simmatrix C++ toolset.

2 Goals

Simmatrix is a part of the basic infrastructure for use of the SimTK Core tools. It is delivered as part of the SimTKcommon library, and developed within the SimTKcommon project on SimTK.org.

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 mathematicians' viewpoint 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 computational treatment as embodied in today's 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 ele-

ment 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) transpose "operator" performs 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.

All symbols introduced into a user program by Simmatrix are in the `SimTK` namespace, except for `#define` macros (used sparingly) which are prefixed by "`SimTK_`" instead. So if we mention a symbol named `MyType` below, its actual name is `SimTK::MyType`.

3 Design issues

Here we discuss our thoughts on some of the choices and dilemmas that are forced on anyone building a system like this, and the resulting design decisions we made.

3.1 Naming

Because there are two separate facilities, we are at times in need of several names for similar concepts, such as "matrix." A great deal of information must be embedded in the types of the small matrix objects, which in C++ risks either a proliferation of obscure type names or frequent use of templates. Further, there is a need to permit programs to be written as generically as is appropriate. For example, a programmer should not need to specify precision in a type name if the algorithm is precision-independent. Vector length is a critical attribute in programs which use the small-vector system; for example, 3d simulation code is written using 3-vectors and 3x3 matrices. Thus it is appropriate for small-vector type names to include lengths. On the other hand, for programs written to deal with large matrices, the specific size is an unimportant run-time specification and must not be embedded in the type name.

To address these issues, we have adopted what we hope is a suggestive convention of using short names and abbreviations for short, fixed-size vectors and longer ones for the large objects. Length always appears in the short type names and never in the long ones. So a 3-element vector is a `Vec<3>` (abbreviated `Vec3`) while a long but size-unspecified vector is called a `Vector`. Similarly, `Mat<3,4>` (abbreviated `Mat34`) is a 3x4 fixed-size matrix at default precision, while `Matrix` is an *m*x*n* adjustable-size matrix at default precision. The full set of names is provided later in this document.

One minor annoyance in C++ is that the standard does not guarantee that a templatized type can appear without a template argument list, even if there are defaults for all the arguments. Thus if we want to allow `Vector<Complex>` some compilers will insist on the default invocation being written `Vector<>`, which we deem too ugly for human consumption. We want the simple name `Vector` to be used for the by-far-most-common case, `Vector<Real>`. To address this, the templatized version of the large vector and matrix types are written with a trailing underscore, like `Vector_<Complex>`. Then `Vector` is a `typedef` to `Vector_<Real>`, the default. This is not necessary for small vector and matrix types since they always have at least a length argument supplied. Typical declarations look like this:

```
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 expect the first element to have index "1." C programmers expect indexing to begin with "0." 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 avoid errors.

So we believe that a single indexing scheme must be chosen and used exclusively. Following the reasoning of the VXL design team (see http://paine.wiau.man.ac.uk/pub/doc_vxl/books/core/book_6.html) we have opted for 0-based indexing as "least weird" in a C++ numerical package. This does not preclude a FORTRAN-compatible 1-based interface to the facility, but it prevents any leakage of incompatible indices into the guts of the package. This will not please everyone, but in past efforts to do so many packages have ended up pleasing no one.

3.3 More design issues TBD

Topics: choice of indexing operators, subvector and submatrix operations, layout of symmetric matrices, mixed-precision operations, choice of transpose operator, use of non-conforming operands, why can't large matrix and vector types be used as element types, packing of data in memory, treatment of scalar assignment.

4 Scalars

We start by describing the Simmatrix scalar types, that is, types which represent a single floating point number (real or complex). We take a somewhat novel view of these types which allows us to achieve some dramatic performance improvements when we begin to aggregate them into matrices. The Simmatrix scalar types comprise three levels we call *precisions*, *numbers*, and *scalars*.

4.1 Precision types

The precision types are the built-in C++ floating point types `float`, `double`, and `long double` (called `single`, `double`, and `quad` or `real*4`, `real*8`, and `real*16` in FORTRAN). These convey the level of accuracy required in computations and stored values. On all currently supported platforms, `float` is 4 bytes, and `double` is 8 bytes. Depending on the compiler being used, `long double` may be the same as `double`, or it may be 10 or 16 bytes. *All* SimTK-supported computers must adhere to the IEEE 754 standard for floating point arithmetic, and SimTK-compliant code may depend on that fact, for example by depending on the existence of NaN (not-a-number) and Infinity.

Under compile-time control, one of the precisions is chosen as the default precision and given the type name `Real` (reminder: this is really `SimTK::Real`). We expect most SimTK code to be written in terms of `Real` rather than the explicit precisions, so that it may be compiled at different precisions without changing the code. Our default for this default is that `Real` is equivalent to `double`, that is, an 8-byte floating point value with approximately 16 digits of precision and an exponent range of about ±300.

Contributors to SimTK code should avoid writing precision-specific code whenever possible. In addition to our predefined default-precision types, C++ provides the standard template type `std::numeric_limits`, and we provide all the additional tools needed occasionally for writing precision-independent code, such as generic-precision constants and convergence limits.

4.2 Numbers

From the above precision types, we construct our numbers, each of which can exist in all three sup-

ported precisions. The number types are the standard *real* and *complex* numbers, and a novel "flavor" of complex number called 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. 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* | **`negator`**`<`*number*`>`
number	::=	*standard* | *conjugate*
standard	::=	*real* | *complex*
conjugate	::=	**`Conjugate`** | **`conjugate`**`<`*precision*`>`
complex	::=	**`Complex`** | **`complex`**`<`*precision*`>`
real	::=	**`Real`** | *precision*
precision	::=	**`float`** | **`double`** | **`long double`**

That completes our discussion of the scalar types. Next we'll look at how these can be used to construct the much more interesting *composite numerical types*.

5 Composite numerical types (fixed-size vectors & matrices)

SimTK defines the following composite numerical types (CNTs), built up recursively from scalars and other composite numerical types. As a consequence of our definition of "scalar" above, all composite numerical types support both real and complex arithmetic equally.

CNTs can be scalars as described above, or small fixed-size vectors and matrices whose elements are scalars or other CNTs. For example, one may easily define a 2-element vector whose elements are ordinary 3-element vectors.‡ CNTs are characterized by *zero* overhead and *minimal* storage requirements. For example, an *m*x*n* matrix of scalars is stored as exactly *mn* consecutive scalars (except when otherwise requested), with no flags, counters, heap pointers, or other sources of inefficiency. The exact storage layout in memory is part of the definition of these types (and is machine-, operating system-, and compiler-independent), so they may be recast to native machine types or other CNTs with predictable results. This implies, for example, that a

‡ That is very useful as the representation of a Spatial Vector which collapses both rotational and translational effects into a single quantity in the Spatial Operator Algebra used by SimTK's Simbody™ package.

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 some predefined abbreviations are available for certain common combinations of template arguments. We provide those because we know from experience that programmers will define them to reduce the ugly "<>" clutter created 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 Vec6`
	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 Row6`
	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`
`Mat44 Mat55`
`Mat66`
	A small, fixed-size matrix of *m* rows and *n* columns of composite numerical type C (default `Real`) with optional column-to-column spacing (default *m*), and row-to-row spacing (default 1).
	These typedefs are provided, with `Mat33` ≡ `Mat<3,3>`, etc.

`SymMat<`*m*`>`
`SymMat<`*m,C*`>`
`SymMat<`*m,C,rs*`>`
 	
`SymMat22`
`SymMat33`
`SymMat44`
`SymMat55`
`SymMat66`
	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 CNT's 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 the CNT's element 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 templates, to avoid confusion with the operator ">>".

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 speed by "rounding up" memory requirements to a 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`'s in a C++ array `Vec3[2]` since the compiler might choose to allocate space for two `Vec4`'s instead!

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 assignment "acts like" that scalar. That is, if you multiply by this matrix the result is identical to a scalar multiply by the origin scalar. Two important

special cases are: (1) setting a matrix to the scalar "1" results in the multiplicative identity matrix of that shape, and (2) setting a matrix to the scalar "0" results in the additive identity matrix of that shape.

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. An "Lvalue" can appear on the 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., $i \geq j$.	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, rep.	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[*i*] sy.row(*i*) sy(*j*) sy.col(*j*)	SymMat	obtain i^{th} row or j^{th} column of SymMat as a Row or Vec	no	must copy elements to temporary Row or Vec; avoid if possible	Return type is packed regardless of original SymMat spacing.
v.getSubVec<*m*>(*i*) v.updSubVec<*m*>(*i*)	Vec	return a reference to a Vec<*m*> whose 0^{th} element is v's i^{th} element	get – no upd – yes	native array index	Element type and stride are the same as the original vector.
r.getSubRow<*n*>(*j*) r.updSubRow<*n*>(*j*)	Row	return a reference to a Row<*n*> whose 0^{th} element is r's j^{th} element	get – no upd – yes	native array index	Element type and stride are the same as the original row.
m.getSubMat<*m*,*n*>(*i*,*j*) m.updSubMat<*m*,*n*>(*i*,*j*)	Mat	return a reference to a Mat<*m*,*n*> whose 0,0 element is m's *i*,*j* element	get – no upd – yes	native array index	Element type and spacing are the same as the original matrix.
m.getSubVec<*m*>(*i*,*j*) sy.getSubVec<*m*>(*i*,*j*) m.getSubRow<*n*>(*i*,*j*) sy.getSubRow<*n*>(*i*,*j*) (upd also available)	Mat SymMat	return reference to a Vec<*m*> or Row<*n*> whose 0^{th} element is m's *i*,*j* element. Only strictly lower triangle can be referenced for SymMat; that is, $i > j$.	get – no upd – yes	native array index	Element type and stride are the same as the original vector.

5.3.2 Arithmetic

The expected arithmetic operators are overloaded for use with CNTs, plus probably some unexpected ones. This includes add, subtract, matrix multiply, and divide for conforming objects and scalars, cross product for

3 vectors, and the usual C arithmetic assignment operators like "+=". Behavior for CNTs 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 to	Meaning	Lvalue?	Cost	Notes
+ - * /	Any CNT	matrix arithmetic	no	same as explicit code	conformant matrices always work; some non-conformant operations are useful also
TBD					

5.4 Summary of CNTs

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

CNT ::= *scalar*
| *composite*`<`*size* `[,` *CNT* `[,` *packing*`] ]>`
composite ::= **`Vec`** | **`Row`** | **`Mat`** | **`SymMat`**
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

The "zero overhead" requirement on the 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 (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,j^th element of m, 0-based
m(i,j)    //   same, but faster
m[i]      // ref to i^th row of m, 0-based
m(j)      // ref to j^th 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 thus "lvalues" (in C terminology) 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 i^th 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 as "handle" classes which contain only a pointer to an undefined type (essentially a `void*`). The object referenced is an instance of a hidden implementation class.* This permits use of these classes in SimTK interfaces while preserving binary compatibility. It also allows use of these objects from other languages, since the `void*` can serve as a "lowest common denominator" representation.

As in Matlab, there is a substantial performance penalty to work with `Vector` and `Matrix` objects element-by-element; "bulk" operators should always be used in performance critical code. For some operations, matrices with structured CNT elements will perform poorly compared to matrices of scalar elements. When used properly, Simmatrix objects containing scalar elements are capable of performing large-scale operations at full machine speed; that is, as fast as LAPACK. Note that complex and conjugate types are still scalars, not structured, and that a negated scalar is still a scalar. See section 4 for more information about scalars.

Many operations with structured CNT elements can be performed at full speed also, provided that there is an equivalent scalar operation. For example, scalar multiplication on structured elements is equivalent to the same operation done on the elements' underlying scalars.

The underlying data representations for these objects are documented and stable, and in many cases map directly onto standard dense storage. In those cases a pointer to the raw data can be obtained if necessary for performance or compatibility purposes.

6.2 Available storage types TBD

Most Simmatrix physical data layouts are designed to be directly compatible with one of the LAPACK-defined layouts. The default layout assumes that an *m*X*n* matrix is stored by columns using *m*X*n* elements. An option is to provide a "leading dimension" ($\geq m$) so that there are regular memory gaps between the columns.

Our default for symmetric and triangular matrices is what LAPACK calls "conventional" storage; that is, space is allocated for the *whole* matrix, but only half the space is used. The space-saving "packed" format is also available but can be expected to run significantly slower for many operations. Note that many LAPACK operations pack *two* symmetric or triangular results into a conventional matrix; that provides both optimal speed and space.

Banded matrices are also available, always in packed storage. These can be full, symmetric, or triangular.

We provide direct support for permutation matrices, which are permuted identity matrices resulting from pivot operations required for numerical stability during factoring. The underlying storage for these is typically just a sequence of integers defining how the columns or rows were pivoted. (TODO: I think LAPACK has two different layouts of pivot matrices; we'll have to support them both.

* This is a standard C++ design pattern usually called "PIMPL" for "private implementation."

It should be possible to extract the underlying integer array and pass it straight through to LAPACK; by hiding it under a `Matrix` handle we don't have 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

A declaration like "`Matrix m;`" declares `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 example, via "`m.resize(10,20);`") it will use a dense, column oriented allocation identical to LAPACK's conventional matrix storage format.

However, handles can optionally be restricted to narrower ranges of behavior, via *commitments* which will be discussed below. First we'll discuss 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. Shape
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 handle: the handle's character commitment, and its current character. The current character always satisfies the handle's commitment.

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 specifies "minimum acceptable" properties for the 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 shape: column or 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 shape 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 shape.

Next we'll look at the individual matrix 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 Shape

There are five possible shape attributes:

1. Rectangular (*m*x*n*)

2. Square (*n*x*n*)
3. Column (*m*x1)
4. Row (1x*n*)
5. Scalar (1x1)

A matrix handle with no shape commitment can hold a general (rectangular) matrix, which of course includes all the other shapes as well.

`Vector` handles are always committed to column shape, `RowVector` handles to row shape.

6.3.4 Size

Size can be variable or fixed, in one or both dimensions. When a dimension is fixed, the matrix handle must commit to a particular size for that dimension.

`Vector` handles are always committed to exactly one column, `RowVector` to exactly one row. Even a zero-length `Vector` has one column, that is, it is 0x1, a zero-length `RowVector` is 1x0.

6.3.5 Structure

Structure refers to an inherent mathematical (or at least algorithmic) property of the matrix rather than a storage strategy. Symmetry is the clearest example of this; it is far more significant than just a way to save storage and reduce operation count.

1. Full (the default)
2. Symmetric (includes Hermitian); implies Square shape; Hermitian implies real diagonal.
3. Triangular (includes trapezoidal)
    a. Upper/lower
    b. Hessenberg (triangular except for one sub- or super-diagonal)
    c. Quasi-triangular (triangular except for 2x2 blocks on the diagonal)
4. Diagonal (also symmetric & triangular); rectangular shape is OK
5. Scalar (diagonal, and all diagonals have the same value); doesn't imply Scalar shape
6. Permutation (of rows or columns); implies Square shape

6.3.6 Conditioning

Matrix condition is a statement about the numerical properties of a Matrix. It can be set as a result of an operation, or by a knowledgeable user. Simmatrix is entitled to rely on the correctness of these assertions, although it will try to check them when it is possible to do so without sacrificing performance.

1. Unknown (the default)
2. Singular (expect terrible conditioning)
3. Full rank (for rectangular, this means rank is the same as the shortest dimension)
4. Well conditioned (implies full rank)
5. Positive definite (symmetric only)
6. Orthogonal

Most operations will result in Unknown conditioning, which will not satisfy a more restrictive commitment. In the case that a matrix handle commits to particular conditioning, only operations which preserve that conditioning will be successful.

6.3.7 Sparsity

This is a statement about the number and/or placement of non-zero entries in a matrix. When we know that few entries will be non-zero, exploiting this fact can yield significant speed gains. It is particularly easy and efficacious to work with a matrix whose non-zero elements cluster narrowly about the main diagonal.

1. Full (default)
2. Banded: diagonal plus specified upper and lower bandwidths
    a. If symmetric or triangular structure, only one bandwidth
    b. Bandwidth may be left uncommitted or commit to particular bandwidth
3. TODO: Sparse.

6.3.8 Storage formats

These refer to the physical layout of data in the computer's memory. Whenever possible we attempt to store data in a format that enables use of special high performance methods, such as those available in the SimTK LAPACK/BLAS implementation.

1. Full (default for full, symmetric & triangular matrices)
    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 someone else's element data. A view 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. This applies only to bulk operations like multiplication and factoring; element-by-element access may incur additional overhead when going through a view.

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 matrix can be represented as a "strided" one-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, this is the mathematician's approach rather than the computational scientist's and at times will not yield 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. The classes haven't been defined yet but will probably look something like

```
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.