Slate++

Version 1.2

Tutorial

Contents

1 Overview

1.1 What is `Slate++` ?

`Slate++` is an object-oriented collection of C++ templates for creating and manipulating vectors and matrices. It allows one to create vectors and matrices in C++ which "know" their dimensions, so they can be passed to functions without extra parameters to keep track of their size. They are dynamically created, which makes them useful for computational work where the dimensions aren't necessarily known ahead of time. `Slate++` vectors and matrices have methods which facilitate using their data directly in external C functions (for example, legacy code which hasn't been built with `Slate++` in mind). Most basic operations, like addition, subtraction and matrix multiplication, are defined with simple operators ('-', '+' and '*'), making code cleaner and easier to read. Some simple linear algebra algorithms have been implemented, such as determinants and inverses, so that simple computational problems can be solved without resorting to complicated libraries.

Consider the following piece of code which adds two `Slate++` vectors to one another, scales a `Slate++` matrix by 20.0, outputs the inverse of the scaled matrix, and then outputs the matrix product of the difference of the vectors with the scaled matrix:

```
v3 = v1 + v2;
m = 20.0 * m;
cout << inverse(m) << endl;
cout << (v1 - v2) * m << endl;
```

The point of this example is that matrix and vector manipulation using `Slate++` is incredibly simple. The equivalent C/C++ code using only C arrays would require external functions (not operator overloading) for every linear algebra operation. Moreover, these function calls would require extra parameters to describe the shape of the vectors and matrices. Finally, the insertion operator (`<<`) can't be used on ordinary C arrays. In a nutshell, the code would be much more complicated to write, and much more difficult to read.

If you're programming in C++, and you're looking for a way to handle vectors and matrices, `Slate++` may be a very good solution.

1.2 What is `Slate++` not?

`Slate++` is not a library of numerical methods. While some simple linear algebra routines have been included, they are far from complete, and in any case, most computational programmers will want to rely on their own algorithms and legacy code. The aim of `Slate++` is to provide a flexible and intuitive set of vector and matrix objects, which can easily interface with external functions.

`Slate++` is not designed for problems involving several dimensions. Its focus is on vectors and matrices, although it's certainly possible to construct arrays of `Slate++` vectors and matrices. Techniques for doing this are discussed later in the tutorial.

`Slate++` is not designed for problems requiring complicated or specialized data structures (e.g., trees, sparse matrices). While a programmer certainly may find a use for `Slate++` vectors and matrices within his or her problem, there are no explicit constructs for simplifying these kinds of structures.

1.3 Who is `Slate++` targeted toward?

`Slate++` is geared toward the C++ programmer who needs to use vectors and matrices for his work, but is unhappy with the standard C++ treatment of arrays. `Slate++` is designed to be extremely simple to use, so that it's appropriate for even programmers inexperiened in object-oriented methods or C++ templates. On the other hand, because `Slate++` vectors and matrices and be used with external legacy code without much difficulty, `Slate++` is equally appropriate for the advanced numerical programmer who wants to clean up his legacy code (say, by designing `Slate++` wrappers around the original code) and take advantage of the power of templates.

1.4 Who are the Authors?

Brian Thorndyke is a physics researcher at the University of Florida. His work centers around numerical studies of molecular systems, from a quantum mechanical point of view.

Robert Thorndyke is a faculty member in Computer Science at Camosun College in Victoria, British Columbia.

And yes, we're brothers. :)

2 New Features in Version 1.2

- The copy constructor and assignment ("=") now peform a deep copy by value. In version 1.0, they copied by reference. This new convention parallels object-oriented standards in C++, and should make Slate++ easier to use with other C++ software.

To copy by reference, one should use standard C++ conventions. A tutorial section describing passing Matrix and Vector objects in functions by value and by reference is included.

- Try/catch mechanisms have been put into the slate++ code, to throw exceptions for operations on non-conformant matrices or vectors (i.e., those that have unmatching sizes). Throwing exceptions is good practice for writing robust code, and gives flexibility to the designer in dealing with errors. A small tutorial section has been included on throwing exceptions.
- All element references in Vectors and Matrices use the notation (i,j) and (i). The C-style notations [][] and [] have been eliminated for now. They may be reinstated in later releases.
- Linear algebra routines now use pass-by-reference correctly, using the new convention for the copy constructor.

3 Review of C Arrays

`Slate++` could, in principle, eliminate the need to ever work with standard C arrays again. However, that would require all code to be built with `Slate++` from the beginning - hardly a practical solution! An alternative is to use `Slate++` alongside external functions which manipulate raw C arrays, and transfer between the two representations as needed. For this reason, it's important to have an understanding of the various forms of C arrays we might encounter.

There are various ways one can specify vectors and matrices using standard C constructs. In most numerical work, the data is stored as a set of contiguous elements, regardless of how it is accessed. When interfacing between `Slate++` and C arrays, we assume this is the storage method used by the external C routines. In this section, we'll describe four major methods of accessing a block of data.

The first representation is arrays of arrays. A matrix can be set up by statically allocating the array when it's initialized,

```
double a[5][6];
```

We'll call this form the *2DA* for "two-dimensional array." Rows (0-4) are indicated by the first index, columns (0-6) by the second. While this is a straightforward method of specifying matrices, there is no way of changing their size during program execution. Since often it's very difficult to know matrix dimensions ahead of time, and dynamic allocation of memory is much better for problems whose size may vary from run to run, the 2DA form is rarely used in computational science.

The second representation is a single array (we'll call it *1DA*). A vector can be set up by statically allocating an array on initialization,

```
double a[30];
```

In this example, `a` is a 30-element vector with indices running from 0 to 29. Alternatively, one could imagine that while it's a one-dimensional array, it actually represents a matrix. In C, multidimensional arrays are stored in row-major order, meaning that the block of data starts with the first row, then the second, and so forth. We could explicitly calculate element (i,j) of the 1DA if we knew the row size. If we assume `a` represents a 5×6 matrix, element (i,j) is `a[i*6 + j]`. The 1DA representation is natural for vectors, and useable for matrices, but suffers from the same ailment as the 2DA form: sizes must be statically allocated (i.e., you have to know the dimensions before compiling).

A third representation is a pointer to pointers, which we'll call *2DP* ("two-dimensional pointers"). This is the most common and versatile representation for matrices, and is the convention used by Numerical Recipes (Press et al.), for examples. First, a contiguous block of data is allocated (for example, using the `new` command). Then a pointer to a pointer is created,

```
double **a;
```

`a` points to an array of pointers. The first in this array of pointers points to the first element of the first row. The second element in the array of pointers points to the first element of the second row, etc. It's a bit confusing, and we recommend looking up some good web-based tutorials on pointers and arrays for a complete explanation. In the meantime, it's sufficient to know that element (i,j) can be accessed, in the 2DP representation, exactly the same way as the 2DA representation: `a[i][j]`. The advantage is that the matrix can be allocated dynamically (i.e., at run-time, after compilation).

The fourth representation is a pointer to the block of data, which we'll call *1DP*. This is the most common representation for vectors. First, a contiguous block of data is allocated to hold the vector. Then a pointer is created,

```
double *a;
```

`a` points to the first element of the vector. Element (i) of the vector has the same notation as a 1DA representation: `a[i]`. One could also imagine the 1DA to represent a matrix, provided one knows the column size. Then, for a 5×6 matrix, element (i,j) is accessed with `a[i*6 + j]`. While vectors are often represented with 1DPs, matrices are most commonly represented with 2DPs instead.

A big disadvantage to using 1DPs and 2DPs is that the programmer has to explicitly keep track of all memory allocated, and ensure that when the arrays go out of scope, the memory is deallocated. If this isn't done, it's called a *memory leak*, and the usable RAM memory gradually diminishes until the program can no longer run. A second problem to all these native C methods is that the dimensions have to be known outside the arrays themselves. None of the above representations (1DP, 2DP, 1DA, 2DA) saves information about the dimensions of the vector or matrix, and this is something that has to be provided externally.

Slate++ solves these problems, by automatically keeping track of allocated memory, and deallocating memory when the Slate++ Vector or Matrix goes out of scope. The Slate++ objects also hold the dimensions of the Vector or Matrix, and this information can be queried. This is convenient when passing Vectors and Matrices into functions, since their dimensions don't have to be passed as additional parameters. In this tutorial, switching between Slate++ and the native C representations will be described in detail.

4 Review of Exception Handling

Exceptions allow the programmer to handle potential runtime errors in a consistent and robust manner. This refers specifically to errors related to *user input*, not actual programming errors. For example, if the user tries to find the inverse of a singular matrix, one would expect an exception to be generated. Without exceptions, the programmer has to rely on either testing the input matrix first, or hope that an appropriate error code could be returned from the function. With exceptions, the programmer can simply call the function and assume everything works; if there's an error, they will have an opportunity to either deal with it or let it crash the application.

When a error condition has been detected by a function, it "throws" the exception to the caller. The caller can then "catch" the exception in a special program block, providing feedback to the user or deal with the error in some way. If the calling function wishes to do this, it places the call in a special `try` block; if the calling function does not wish to catch exceptions (and therefore allow them to crash the program), then it just calls the function normally.

Here is an example. The code below shows how the try/catch mechanism can be used to handle input errors.

```
// log_base_2() calculates the base-2 log of the input value.
double log_base_2(double value) {
    if (value <= 0) throw("Error: can't compute log of a non-positive value.");
    else return log(value) / log(2);
}

int main() {
    double input;
    cout << "Enter your value: ";
    cin >> input;
    try {
      double n = log_base_2(input);
      cout << "Log base-2 of " << input << " is " << n << endl;
    } catch (const char * error) {
      cout << error << endl;
    }
    return 0;
}
```

The function `log_base_2()` can fail if the input is non-positive. When this occurs, rather than allowing the math operation to fail (producing non-standard results), the condition is detected and an exception is thrown. In the `main()` function, the code that assumes the error did *not* occur is placed inside a `try` block; if the exception is thrown, the code that occurs after the error in the `try` block is skipped, and execution jumps directly to the `catch` block.

Exceptions can be of any type — that is, what is placed inside the `throws()` command can be any kind of expression. This allows for great flexibility in error-handling strategies. If a number of different types of errors can occur in a function, for example, then the exception could be an integer whose value has some coded meaning. Furthermore, a function can throw any number of exceptions, all of which can be of different types. The calling function can then selectively catch any or all of the types of exception, by using multiply `catch` blocks. For example, the following code shows how to catch three types of exceptions that can result from calling a function called `foo()`:

```
try {
    foo();
    ... // code that assumes foo() succeeds.
} catch (int error) {
    ... // handle int exception types.
} catch (double error) {
    ... // handle double exception types.
} catch (const char * error) {
    ... // handle char* (string) exception types.
}
```

(Note that automatic conversion does not occur for exceptions; if the `foo()` function throws a `double`, the value will not be converted to an `int` to allow the first `catch` block to handle it.)

Sometimes the programmer wants to handle *all* exception types with the same `catch` block. This can be done using `...` as the catch parameter:

```
try {
    foo();

    // code goes here that assumes foo() succeeds.

} catch (...) {

    // handle all exception types here
}
```

This can also be used in conjunction with other `catch` blocks; for example, the programmer can catch `int` exceptions with one block and all other types in another block by placing the `int` catch block before the `...` catch block. Be careful: placing the `...` block before other blocks will preempt the other blocks! The catch blocks are processes in order, and the first one matching the exception type will be the one to handle the exception.

In `Slate++` , exceptions are used where appropriate. The linear algebra functions that depend on non-singular matrices will throw a `const char *` exception with an appropriate error message. We are also working on putting proper exception handling in all functions where dimension mismatch could be an issue (*e.g.* multiplying matrices of incompatible sizes).

5 Setting Up Slate Vectors and Matrices

`Slate++` Vectors and Matrices are set up in the same way other object-oriented objects are constructed. We won't go into great detail about object-oriented programming; rather, we'll show how to use `Slate++` through an exhaustive set of examples. In the rest of this tutorial, every example is complete, and located in the directory

```
/slate/examples/
```

Each example is self-contained, and should be easily compiled and run by a compiler like `g++`. Hopefully this approach (focusing on learning *how* to use `Slate++` , primarily through example) will prove more useful than describing the internal workings of `Slate++` ! On the other hand, we've kept the code documentation fairly clear, so it should be simple for an intermediate programmer to modify `Slate++` as he/she feels the need.

5.1 Construction

```
#include <slate.h>

int main() {

  //
  // Vector and Matrix dimensions can be specified on initialization.  The
  // following form makes indices begin at 1.
  //
  Matrix<double> a(2,3);
  Vector<int> b(6);

  //
  // Let's specify different ranges on initialization.
  //
```

```
Matrix<double> x(6,7,4,6);
Vector<int> y(2,7);

//
// We can use other types if we like.
//
Matrix<float> r(3,9);
Vector<long> p(23);

//
// Dimensions can also be specified later.
//
Matrix<double> aa;
Vector<int> bb;
aa.set_size(1,2,1,3);
bb.set_size(1,6);

//
// The size can be changed numerous times.  Let's increase the range
// of each dimension in aa and bb by 4.
//
// NOTE:  Changing the size of a Matrix or Vector results in a new
// Matrix or Vector being created.  If no other Vectors or Matrices
// refer to the old data, then it is lost.
//
aa.set_size(1,6,1,7);
bb.set_size(1,10);

//
// Since the lower indices were 1, we could have written:
//
aa.set_size(6,7);
bb.set_size(10);

//
// Let's set aa and bb to have the same size as x and y.  They will
// adopt x and y's index range as well.
//
aa.set_size(x);
bb.set_size(y);

//
// We can change the offsets anytime.  The following makes a and b
// have ranges a(10...11, 20...22) and b(-6...-1)
//
```

```
  a.new_offset(10,20);
  b.new_offset(-6);
}
```

5.2 Copy by Value

`Slate++` Vectors and Matrices copy by value. This means that the assignment operator creates a *deep* copy of all elements, which are then independent of the original.

```
#include <slate.h>

int main() {

  ////////////////////////////////////////////////////////////////////////////
  //
  // Matrices
  //
  //

  //
  // Matrices can be initialized to a scalar value on construction.  The
  // following creates a 2x3 Matrix with values 19.0.
  //
  Matrix<double> a(1,2,1,2,19.0);

  //
  // The Matrix elements can be given a different value later.
  //
  a = 6.0;   // all elements of a are now 6.0

  //
  // Elements can be specified individually.  Let's make a unit Matrix.
  //
  a = 0.0;
  a(1,1) = 1.0;
  a(2,2) = 1.0;

  //
  // Let's create some native C arrays, which we'll then assign to a.
  //
  double mat1[4] = { 3.0, 4.0, 5.0, 6.0 };         // 1DA matrix
  a = mat1;

  cout << a << endl;
```

```
double *mat2 = new double[4];                    // 1DP matrix
mat2[0] = 1.0; mat2[1] = 5.0; mat2[2] = 9.0; mat2[3] = 3.0;
a = mat2;

cout << a << endl;

double **mat3 = new (double *)[4];              // 2DP matrix
mat3[0] = new double[4];
mat3[1] = mat3[0]+2;
mat3[0][0] = 3.0; mat3[0][1] = 4.0; mat3[1][0] = 5.0; mat3[1][1] = 6.0;
a = mat3;

cout << a << endl;


////////////////////////////////////////////////////////////////////////////////
//
// Vectors
//
//

//
// Vectors can be initialized to a scalar value on construction.  The
// following creates a 3-element with values 19.0.
//
Vector<double> v(1,3,19.0);

//
// The Vector elements can be given a different value later.
//
v = 6.0;   // all elements of b are now 6.0

//
// Elements can be specified individually.
//
v(1) = 1.0;
v(2) = 1.0;
v(3) = 1.0;

//
// Let's create some native C arrays, which we'll then assign to v.
//
double vec1[3] = { 3.0, 4.0, 5.0 };         // 1DA vector
v = vec1;
```

```
  cout << v << endl;

  double *vec2 = new double[3];               // 1DP vector
  vec2[0] = 2.0; vec2[1] = 7.0; vec2[2] = 4.0;
  v = vec2;

  cout << v << endl;
}
```

5.3 Importing from Files

```
#include <slate.h>
#include <fstream>

int main() {

  // "data.txt" contents:
  //
  // 2 3 1 2\n
  // -2 4 -1 5\n
  // 3 7 1.5 1\n
  // 6 9 3 7

  // import into a slate matrix.
  ifstream fin1("data.txt");
  Matrix<double> m(4,4); // must specify correct dimensions.
  fin1 >> m;

  // import into several slate vectors.
  ifstream fin2("data.txt");
  Vector<double> v1(4), v2(4), v3(4), v4(4);  // must specify dimensions.

  // can do one at a time:
  fin2 >> v1;
  fin2 >> v2;

  // ... or can do several on one line, if enough data exists.
  fin2 >> v3 >> v4;

  cout << m << endl;
  cout << v1 << endl;
  cout << v2 << endl;
  cout << v3 << endl;
  cout << v4 << endl;
}
```

5.4 Importing from Stdin

```
#include <slate.h>

int main() {

  // Importing a matrix from stdin.
  //
  // Note: row*col entries will be taken from stdin; if too few
  // are supplied, then the process will block until more entries
  // are given; if too many are given, the extra will be ignored and
  // will perhaps end up in the *next* import from stdin!
  //
  // Entries will be populated in row-major form.
  //
  // Importing works fine with file redirection, pipes, and direct
  // import from the keyboard.
  //
  Matrix<double> m(4, 4); // must specify proper dimensions.
  cin >> m;

  cout << "Matrix entered: " << endl;
  cout << m;


  // Importing a vector from stdin.
  //
  // Note: n entries will be taken from stdin; if too few
  // are supplied, then the process will block until more entries
  // are given; if too many are given, the extra will be ignored and
  // will perhaps end up in the *next* import from stdin!
  //
  // Importing works fine with file redirection, pipes, and direct
  // import from the keyboard.
  //
  Vector<double> v(4); // must specify proper dimension.
  cin >> v;

  cout << "Vector entered: " << endl;
  cout << v;
}
```

5.5 Extracting Subsections

```
#include <slate.h>

int main() {

  //
  // Matrix and Vector elements can be used as lhs and rhs values.
  //
  Matrix<float> a(1,4,1,4,2.0);
  cout << a << endl;

  a(1,3) = 9.0;
  cout << a << endl;
  cout << a(2,2) << endl;

  Vector<int> b(1,6,0);
  cout << b << endl;

  b(3) = 6;
  cout << b << endl;
  cout << b(1) << endl;

  //
  // We can extract subsets of any Matrix or Vector.  Submatrix is
  // specified as submatrix(low row, high row, low col, high col).
  //
  Matrix<float> sub1(2,3);
  sub1 = a.submatrix(1,2,2,4);   // sub1 will start at (1,1) like a
  cout << sub1 << endl;

  Vector<int> sub2(2);
  sub2 = b.subvector(2,4);
  cout << sub2 << endl;

  //
  // We can also take entire rows or columns of Matrices.  These are
  // created as Vectors.
  //
  cout << a.row(2) << endl;      // a.row(2) is a Vector
  cout << a.col(3) << endl;      // a.col(3) is a Vector
}
```

5.6 Converting Matrices to Vectors

```
#include <slate.h>

int main() {

  //
  // Suppose we have a 5x5 Matrix, with index offset (5,2).
  //
  Matrix<float> a(5,9,2,6,2.0);

  //
  // We can create an equivalent Vector, where all elements are in a
  // row-major order.  The first index of the Vector will be 1.
  //
  Vector<float> b(25);
  b = a.pack();

  cout << a << endl;
  cout << b << endl;

}
```

5.7 Viewing Vectors and Matrices

```
#include <slate.h>

int main() {

  //
  // All Vectors and Matrices (and their elements) can be displayed with
  // the insertion operator ('<<') to cout:
  //
  Matrix<double> a(1,2,1,2,19.0);
  Vector<float> b(7,10,4.3);
  a(1,2) = 45.6;
  b(9) = 12.3;

  cout << a << endl;
  cout << b << endl;

  cout << a(1,1) << "  " << a(1,2) << endl;
  cout << b(9) << "  " << b(10) << endl;
  //
  // We can also display the number of row and column dimensions of a
  // Matrix.
  //
  cout << a.row_low() << "  " << a.row_high() << endl;
  cout << a.col_low() << "  " << a.col_high() << endl;
  cout << a.row_size() << "  " << a.col_size() << endl;

  //
  // Total number of elements in a Matrix.
  //
  cout << a.size() << endl;

  //
  // Dimensions and size of a Vector.
  //
  cout << b.low() << "  " << b.high() << endl;
  cout << b.size() << endl;
}
```

5.8 Using As Function Parameters

When a `Slate++` Matrix or Vector is passed into a function, the *copy constructor* is called, which creates a deep copy of the original Matrix or Vector. Then, whatever is done to the object within the function does *not* affect the original object. Sometimes you'll want to change the original function rather than make a copy. This is most easily done by passing by reference, as one could do for any C++ object. When one passes by reference, the copy constructor is not called, and the function permanently modifies the original Vector or Matrix. The example below demonstrates both passing by value and passing by reference.

```
#include <slate.h>

//
// Sample function which modifies a Matrix by adding 1.0 to each element.
// The original Matrix is not changed.
//
void matmod(Matrix<double> a) {
  a = a + 1.0;
  cout << a << endl;
}

//
// Sample function which modifies a Vector by adding 1.0 to each element.
// The original Vector is not changed.
```

```
//
void vecmod(Vector<double> v) {
  v = v + 1.0;
  cout << v << endl;
}

//
// Sample function which modifies a Matrix by adding 1.0 to each element.
// The original Matrix *is* changed, since it's passed by reference.
//
void matmod2(Matrix<double> &a) {
  a = a + 1.0;
  cout << a << endl;
}

//
// Sample function which modifies a Vector by adding 1.0 to each element.
// The original Vector *is* changed, since it's passed by reference.
//
void vecmod2(Vector<double> &v) {
  v = v + 1.0;
  cout << v << endl;
}


int main() {

  //
  // Pass a Matrix as a parameter - passing by value (so orig not changed)
  //
  Matrix<double> a(1,5,1,5,3.0);
  matmod(a);
  cout << a << endl;

  //
  // Pass a Vector as a parameter - passing by value (so original not changed)
  //
  Vector<double> v(1,6,4.0);
  vecmod(v);
  cout << v << endl;

  //
  // Pass a Matrix as a parameter - passing by reference (so orig changed)
  //
  Matrix<double> a2(1,5,1,5,3.0);
  matmod2(a2);
  cout << a2 << endl;

  //
  // Pass a Vector as a parameter - passing by reference (so orig changed)
  //
  Vector<double> v2(1,6,4.0);
  vecmod2(v2);
  cout << v2 << endl;

}
```

6 Basic Operations

6.1 Addition

```
#include <slate.h>

int main() {

  //
  // Vectors and Matrices with the same dimensions and type can be added
  // together.
  //
  Matrix<double> a(1,2,1,2,19.0);
  Matrix<double> b(3,4,1,2,10.0);
  Matrix<double> c(2,2);

  c = a + b;
  cout << c << endl;

  Vector<int> d(3,9,4);
  Vector<int> e(1,7,2);
  Vector<int> f;

  f = d + e;
  cout << f << endl;

  //
  // Scalar elements can be added to Vectors and Matrices.
  //
  c = c + 14.0;
  cout << c << endl;
  c = a + 12.0 + b + 1.0 + c;
  cout << c << endl;

  f = d + 7;
  cout << f << endl;

  f = 9 + d + 24;
  cout << f << endl;
}
```

6.2 Subtraction

```
#include <slate.h>

int main() {

  //
  // Vectors and Matrices with the same dimensions and type can be
  // substracted from one another.
  //
  Matrix<double> a(1,2,1,2,19.0);
  Matrix<double> b(3,4,1,2,10.0);
  Matrix<double> c(2,2);

  c = a - b;
  cout << c << endl;

  Vector<int> d(3,9,4);
  Vector<int> e(1,7,2);
  Vector<int> f;

  f = d - e;
  cout << f << endl;

  //
  // Scalar elements can be subtracted from Vectors and Matrices.
  //
  c = c - 14.0;
  cout << c << endl;

  c = 2.0 - a - 12.0 - b - 1.0 - c;
  cout << c << endl;
  f = d - 7;
  cout << f << endl;

  f = 9 - d - 24;
  cout << f << endl;

  //
  // The unary operator ('-') can also be used to negate a Vector or Matrix.
  //
  cout << -f << endl;
  cout << -c << endl;
}
```

6.3 Multiplication

```
#include <slate.h>

int main() {

  //
  // Vectors and Matrices with the same dimensions and type can be
  // multiplied element-by-element.
  //
  Matrix<double> a(1,2,1,2,19.0);
  Matrix<double> b(3,4,1,2,10.0);
  Matrix<double> c(2,2);

  c = elem_mul(a,b);
  cout << c << endl;

  Vector<int> d(3,9,4);
  Vector<int> e(1,7,2);
  Vector<int> f;

  f = elem_mul(d,e);
  cout << f << endl;

  //
  // Scalar elements can be multiplied Vectors and Matrices.
  //
  c = c * 0.1;
  cout << c << endl;

  c = 2.0 * a * 2.0;
  cout << c << endl;

  f = d * 7;
  cout << f << endl;

  f = 9 * d * 3;
  cout << f << endl;

  //
  // Matrix multiplication between conformant Matrices.  The result
  // is another Matrix.
  //
  Matrix<int> p(1,3,1,3,2);
  Matrix<int> q(1,3,1,3,3);
  p(1,1) = 4;
  q(3,3) = 5;
  cout << p * q << endl;
  cout << q * p << endl;
  cout << q * p * p * p << endl;

  //
  // Matrix multiplication between conformant Matrices and Vectors.  The
  // result is another Vector.
  //
  Vector<int> r(1,3,10);
  r(3) = 5;
  p(1,3) = 3;
  p(3,1) = 1;
  cout << p << endl;
  cout << r << endl;
  cout << p * r << endl;
  cout << r * p << endl;

  //
  // Dot product between two Vectors.
  //
  cout << dot(r,r) << endl;
}
```

6.4 Division

```
#include <slate.h>

int main() {
  //
  // Vectors and Matrices with the same dimensions and type can be
  // divided by one another element-by-element.
  //
  Matrix<double> a(1,2,1,2,19.0);
  Matrix<double> b(3,4,1,2,10.0);
  Matrix<double> c(2,2);

  c = a / b;
  cout << c << endl;

  Vector<double> d(3,9,4.0);
  Vector<double> e(1,7,2.0);
  Vector<double> f;

  f = d / e;
  cout << f << endl;

  //
  // Scalar elements can be divided by Vectors and Matrices, and
  // vice versa.  This is an element-by-element division.
  //
  c = c / 14.0;
  cout << c << endl;

  c = 2.0 / a / 12.0 / b / 1.0 / c;
  cout << c << endl;

  f = d / 7.0;
  cout << f << endl;

  f = 9.0 / d / 24.0;
  cout << f << endl;
}
```

6.5 Comparisons

```
#include <slate.h>

int main() {

  //
  // Vectors and Matrices with the same dimensions and type can compared
  // element-by-element.
  //
  Matrix<double> a(1,2,1,2,19.0);
  Matrix<double> b(3,4,1,2,10.0);
  a(1,1) = 10.0;

  cout << (a < b) << endl;    // TRUE only if *all* elements of a < b
  cout << (a <= b) << endl;
  cout << (a == b) << endl;
  cout << (a >= b) << endl;
  cout << (a > b) << endl;

  //
  // Comparisons can be made with scalars.
  //
  cout << (a < 10.0) << endl;
  cout << (a <= 10.0) << endl;
  cout << (a == 10.0) << endl;
  cout << (a >= 10.0) << endl;
  cout << (a < 10.0) << endl;

  cout << (10.0 < a) << endl;
  cout << (10.0 <= a) << endl;
  cout << (10.0 == a) << endl;
  cout << (10.0 >= a) << endl;
  cout << (10.0 > a) << endl;
}
```

6.6 Complex Values

```
#include <slate.h>

int main() {

  //
  // Complex values can be used, drawing from C's <complex> classes.
  //
  Matrix< complex<double> > a(1,2,1,2,complex<double>(19.0,0));
  Matrix< complex<double> > b(3,4,1,2,complex<double>(10.0,0));
  a(1,1) = complex<double> (0.0,3.0);

  cout << a << endl;

  //
  // All arithmetic operations defined over complex numbers can be
  // performed the same as any other type.
  //
  cout << a + a << endl;
  cout << complex<double>(3.0,4.0) * a << endl;
}
```

6.7 Changing Type

```
#include <slate.h>

int main() {

  //
  // New Vectors and Matrices with different types can be formed.
  // Suppose we have a double Matrix which we want to float.  We create
  // a new float Matrix as follows:
  //
  Matrix<double> a(1,2,1,2,19.0);
  Matrix<float> b;
  b = a.convert(b);
  cout << b << endl;

  Vector<double> c(1,5,12.5);
  Vector<int> d;
  d = c.convert(d);
  cout << d << endl;
}
```

7 Mathematical Functions

7.1 Minimum and Maximum

```
#include <slate.h>

int main() {

  //
  // Vectors and Matrices with the same dimensions and type can be
  // compared element-by-element, producing a new Vector or Matrix with
  // the maximum of each element pair.
  //
  Matrix<double> a(1,2,1,2,19.0);
  Matrix<double> b(1,2,1,2,10.0);
  a(1,1) = 4.0;
  cout << max(a, b) << endl;

  Vector<int> d(1,6,4);
  Vector<int> e(1,6,2);
  d(1) = 0;
  cout << max(d, e) << endl;

  //
  // Scalar values can be compared with Vector or Matrix elements to
  // produce a new Vector or Matrix.
  //
  cout << max(5.0, a) << endl;
  cout << max(a, 5.0) << endl;
  cout << max(3, d) << endl;
  cout << max(d, 3) << endl;

  //
  // Forming Vectors and Matrices with *minimum* values.
  //
  cout << min(a,b) << endl;
  cout << min(d,e) << endl;

  cout << min(5.0, a) << endl;
  cout << min(a, 5.0) << endl;
  cout << min(3, d) << endl;
  cout << min(d, 3) << endl;

  //
  // We can also return the minimum or maximum element of a Vector or Matrix.
  //
  cout << min(a) << "  " << max(a) << endl;
  cout << min(d) << "  " << max(d) << endl;
}
```

7.2 Sum of Elements

```
#include <slate.h>

int main() {

  //
  // We can sum all elements of a Vector or Matrix.
  //
  Matrix<double> a(1,2,1,2,19.0);
  Vector<double> b(1,9,10.0);

  cout << sum(a) << endl;
  cout << sum(b) << endl;
}
```

7.3 Power

```
#include <slate.h>

int main() {

  //
  // Take each element to some power.
  //
  Matrix<double> a(1,2,1,2,2.0);
  Vector<double> b(1,9,3.0);
  cout << pow(a,4) << endl;
  cout << pow(b,3) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << pow(c,4) << endl;
  cout << pow(d,3) << endl;
}
```

7.4 Absolute Value

```
#include <slate.h>

int main() {

  //
  // Take the absolute values of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,-3.0);
  cout << abs(a) << endl;
  cout << abs(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << abs(c) << endl;
  cout << abs(d) << endl;
}
```

7.5 Square Root

```
#include <slate.h>

int main() {

  //
  // Take the square roots of each element.
  //
  Matrix<double> a(1,2,1,2,2.0);
  Vector<double> b(1,9,3.0);
  cout << sqrt(a) << endl;
  cout << sqrt(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << sqrt(c) << endl;
  cout << sqrt(d) << endl;
}
```

7.6 Square

```
#include <slate.h>

int main() {

  //
  // Take the square of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << sqr(a) << endl;
  cout << sqr(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << sqr(c) << endl;
  cout << sqr(d) << endl;
}
```

7.7 Cube

```
#include <slate.h>

int main() {

  //
  // Take the cube of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << cube(a) << endl;
  cout << cube(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << cube(c) << endl;
  cout << cube(d) << endl;
}
```

7.8 Sine

```
#include <slate.h>

int main() {
  //
  // Take the sine of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << sin(a) << endl;
  cout << sin(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << sin(c) << endl;
  cout << sin(d) << endl;
}
```

7.9 Cosine

```
#include <slate.h>

int main() {

  //
  // Take the cosine of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << cos(a) << endl;
  cout << cos(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << cos(c) << endl;
  cout << cos(d) << endl;
}
```

7.10 Tangent

```
#include <slate.h>

int main() {

  //
  // Take the tangent of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << tan(a) << endl;
  cout << tan(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << tan(c) << endl;
  cout << tan(d) << endl;
}
```

7.11 Hyperbolic Sine

```
#include <slate.h>

int main() {

  //
  // Take the hyperbolic sine of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << sinh(a) << endl;
  cout << sinh(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << sinh(c) << endl;
  cout << sinh(d) << endl;
}
```

7.12 Hyperbolic Cosine

```
#include <slate.h>

int main() {

  //
  // Take the hyperbolic cosine of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << cosh(a) << endl;
  cout << cosh(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << cosh(c) << endl;
  cout << cosh(d) << endl;
}
```

7.13 Hyperbolic Tangent

```
#include <slate.h>

int main() {

  //
  // Take the hyperbolic tangent of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << tanh(a) << endl;
  cout << tanh(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << tanh(c) << endl;
  cout << tanh(d) << endl;
}
```

7.14 Exponential

```
#include <slate.h>

int main() {

  //
  // Take the exponential of each element.
  //
  Matrix<double> a(1,2,1,2,-2.0);
  Vector<double> b(1,9,3.0);
  cout << exp(a) << endl;
  cout << exp(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << exp(c) << endl;
  cout << exp(d) << endl;
}
```

7.15 Logarithms

```
#include <slate.h>

int main() {

  //
  // Take the natural (base-e) logarithm of each element.
  //
  Matrix<double> a(1,2,1,2,2.0);
  Vector<double> b(1,9,3.0);
  cout << ln(a) << endl;
  cout << ln(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << ln(c) << endl;
  cout << ln(d) << endl;

  //
  // Take the base-10 logarithm of each element.
  //
  cout << log10(a) << endl;
  cout << log10(b) << endl;

  //
  // The Matrix or Vector can have complex elements.
  //
  cout << log10(c) << endl;
  cout << log10(d) << endl;
}
```

7.16 Complex Conjugate

```
#include <slate.h>

int main() {

  //
  // Take the complex conjugate of each element.  This is only sensible
  // for complex Matrices and Vectors.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << conj(c) << endl;
  cout << conj(d) << endl;
}
```

7.17 Real Component

```
#include <slate.h>

int main() {

  //
  // Take the real component of each element.  This is only sensible
  // for complex Matrices and Vectors.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << real(c) << endl;
  cout << real(d) << endl;
}
```

7.18 Imaginary Component

```
#include <slate.h>

int main() {

  //
  // Take the imaginary component of each element.  This is only sensible
  // for complex Matrices and Vectors.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  cout << imag(c) << endl;
  cout << imag(d) << endl;
}
```

7.19 Complex Argument

```
#include <slate.h>

int main() {

  //
  // Take the complex argument of each element.  This is only sensible
  // for complex Matrices and Vectors.
  //
  Matrix< complex<double> > c(1,2,1,2,complex<double>(2.0,1.0));
  Vector< complex<double> > d(1,9,complex<double>(3.0,2.0));
  Vector< complex<double> > e(1,9,complex<double>(-3.0,-2.0));
  Vector< complex<double> > f(1,9,complex<double>(3.0,-2.0));
  cout << arg(c) << endl;
  cout << arg(d) << endl;
  cout << arg(e) << endl;
  cout << arg(f) << endl;
}
```

8 Linear Algebra

8.1 Transpose

```
#include <slate.h>

int main() {

  //
  // Take the tranpose of a Matrix.
  //
  Matrix<double> a(1,2,1,3,3.4);
  a(1,2) = 5.9;
  cout << a << endl;
  cout << transpose(a) << endl;

  //
  // Also works for complex Matrices.
  //
  Matrix< complex<double> > c(1,2,1,3,complex<double>(2.0,1.0));
  c(2,1) = complex<double>(4.0,-1.0);

  cout << c << endl;
  cout << transpose(c) << endl;
}
```

8.2 Hermitian Conjugate

```
#include <slate.h>

int main() {

  //
  // Take the Hermitian conjugate of the Matrix.  This is only sensible
  // for complex Matrices.
  //
  Matrix< complex<double> > c(1,2,1,3,complex<double>(2.0,1.0));
  c(2,1) = complex<double>(4.0,-1.0);

  cout << c << endl;
  cout << dag(c) << endl;

}
```

8.3 Norms

```
#include <slate.h>

int main() {

  ////////////////////////////////////////////////////////////////////////////
  //
  // Matrix
  //

  //
  // Frobenius norm.
  //
  Matrix<double> a(1,3,1,3,3.4);
  a(1,1) = 5.9;
  a(3,3) = 18.0;
  cout << a << endl;
  cout << normFro(a) << endl;

  //
  // Matrix 1-norm.
  //
  cout << norm1(a) << endl;

  //
  // Matrix infinity-norm.
  //
  cout << normInf(a) << endl;

  //
  // Also works for complex Matrices.
  //
  Matrix< complex<double> > b(1,3,1,3,complex<double>(1.0,2.0));
  b(1,1) = complex<double>(0.0,-1.0);
  b(3,3) = complex<double>(5.0,5.0);
  cout << b << endl;

  cout << normFro(b) << endl;
  cout << norm1(b) << endl;
  cout << normInf(b) << endl;

  // ////////////////////////////////////////////////////////////////////////
  //
  // Vector
  //

  //
  // Vector 1-norm
  //
  Vector<double> c(1,4,-3.0);
  c(1) = 7.0;
  cout << c << endl;
  cout << norm1(c) << endl;

  //
  // Vector 2-norm
  //
  cout << norm2(c) << endl;

  //
  // Vector infinity-norm
  //
  cout << normInf(c) << endl;

  //
  // Works for complex Vectors as well.
  //
  Vector< complex<double> > d(1,4,complex<double>(2.0,-6.0));
  d(1) = complex<double>(9.0,10.0);

  cout << d << endl;
  cout << norm1(d) << endl;
  cout << norm2(d) << endl;
  cout << normInf(d) << endl;
}
```

8.4 Trace

```
#include <slate.h>

int main() {

  //
  // Take the trace of a Matrix.  Only sensible for square matrices.
  //
  Matrix<double> a(1,3,1,3,3.4);
  a(1,1) = 5.9;
  cout << a << endl;
  cout << trace(a) << endl;

  //
  // Also works for complex Matrices.
  //
  Matrix< complex<double> > c(1,3,1,3,complex<double>(2.0,1.0));
  c(2,2) = complex<double>(4.0,-1.0);

  cout << c << endl;
  cout << trace(c) << endl;
}
```

8.5 Determinant

```
#include <slate.h>

int main() {

  //

  // Want to find the determinant of:
  //
  //    [  2   3   1    2 ]
  //    [ -2   4  -1    5 ]
  //    [  3   7  1/2   1 ]
  //    [  6   9   3    7 ]
  //

  // create the matrix.
  Matrix<double> m(4, 4);
  m(1, 1) = 2.0; m(1, 2) = 3.0; m(1, 3) = 1.0; m(1, 4) = 2.0;
  m(2, 1) = -2.0; m(2, 2) = 4.0; m(2, 3) = -1.0; m(2, 4) = 5.0;
  m(3, 1) = 3.0; m(3, 2) = 7.0; m(3, 3) = 0.5; m(3, 4) = 1.0;
  m(4, 1) = 6.0; m(4, 2) = 9.0; m(4, 3) = 3.0; m(4, 4) = 7.0;

  // find the determinant.
  double d = det(m);
  cout << d << endl;
}
```

8.6 Inverse

```
#include <slate.h>

int main() {

  // Want to find the inverse of the following:
  //
  //     [  2   0   1   2 ]
  //     [  1   1   0   2 ]
  //     [  2  -1   3   1 ]
  //     [  3  -1   4   3 ]
  //

  // create the matrix.
  Matrix<double> m(4, 4);
  m(1, 1) = 2; m(1, 2) = 0; m(1, 3) = 1; m(1, 4) = 2;
  m(2, 1) = 1; m(2, 2) = 1; m(2, 3) = 0; m(2, 4) = 2;
  m(3, 1) = 2; m(3, 2) = -1; m(3, 3) = 3; m(3, 4) = 1;
  m(4, 1) = 3; m(4, 2) = -1; m(4, 3) = 4; m(4, 4) = 3;

  try {
    Matrix<double> inv = inverse(m);
    cout << m * inv << endl;
  } catch (int error) {
    cout << "matrix is singular" << endl;
  }
}
```

8.7 System of Linear Equations

```
#include <slate.h>

int main() {

  // Want to solve the following:
  //
  //     [  1  -1   2  -1 ]     [  6 ]
  //     [  1   0  -1   1 ]     [  4 ]
  //     [  2   1   3  -4 ] x = [ -2 ]
  //     [  0  -1   1  -1 ]     [  5 ]
  //

  // create the matrix.
  Matrix<double> m(4, 4);
  m(1, 1) = 1; m(1, 2) = -1; m(1, 3) = 2; m(1, 4) = -1;
  m(2, 1) = 1; m(2, 2) = 0; m(2, 3) = -1; m(2, 4) = 1;
  m(3, 1) = 2; m(3, 2) = 1; m(3, 3) = 3; m(3, 4) = -4;
  m(4, 1) = 0; m(4, 2) = -1; m(4, 3) = 1; m(4, 4) = -1;

  // create the resultant vector.
  Vector<double> b(4);
  b(1) = 6;
  b(2) = 4;
  b(3) = -2;
  b(4) = 5;

  // solve for x.
  try {
    Vector<double> x = solve(m, b);
    cout << x << endl;
    cout << m * x << endl;
  } catch (int error) {
    cout << "matrix is singular" << endl;
  }
}
```

9 Interfacing Slate with External Functions

```
#include <slate.h>

//
// External function which takes an array as a pointer to a pointer, and
// adds 0.1 to each element.  The function assumes all element indices
// begin at 0.
//
void tweak(double **mat, int nrows, int ncols) {
  for (int i=0; i<nrows; i++)
    for (int j=0; j<ncols; j++)
      mat[i][j] += 0.1;
}

//
// Similar external function, but it takes a pointer to a contiguous block
// of values.
//
void tweak2(double *mat, int nrows, int ncols) {
  for (int i=0; i<nrows*ncols; i++)
    mat[i] += 0.1;
}

//
// Similar to tweak2(), but assumes a vector of data is input.
//
void tweak3(double *vec, int nelems) {
  for (int i=0; i<nelems; i++)
    vec[i] += 0.1;
}

//
// Generates a unit NxN matrix, in a pointer to data form.
//
void unit(double *u, int n) {
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++) {
      if (i==j) u[i*n+j] = 1.0;
      if (i!=j) u[i*n+j] = 0.0;
    }
}

int main() {

  //
  // Start with a slate++ matrix. We're interfacing with external functions
  // that expect indices starting at 0, so we need to specify slate Matrices
  // with corresponding index bounds (i.e., all starting at 0).
  //
  Matrix<double> a(0,2,0,2,3.4);
  cout << a << endl;

  //
  // Send it to the first function.  The raw() method points directly to the
  // pointer-to-pointer inside the Matrix, where data is stored.  Anything
  // done by tweak() to the data will be permanent.
  //
  tweak(a.raw(), a.row_size(), a.col_size());
  cout << a << endl;

  //
  // Send it to the second function.  There is nothing particularly different
  // here, except that we must dereference the pointer-to-pointer once
  // because tweak2() accepts a pointer to the data.
  //
  tweak2(*a.raw(), a.row_size(), a.col_size());
  cout << a << endl;

  //
  // For Vectors, routines will likely accept a pointer to the data,
  // which again is exposed through raw().
  //
  Vector<double> b(0,4,0.4);
  cout << b << endl;
  tweak3(b.raw(), b.size());
  cout << b << endl;

  //
  // Recall that native C data can be copied into slate Vectors and Matrices
  // either by reference ('=') or value ('copy()').  For example, we
  // can use the external function unit() to generate a unit slate Matrix.
  //
  Matrix<double> m(1,4,1,4);
  double *u = new double[16];
  unit(u,4);
  m = u;
  cout << m;
}
```

10 Multiple Slate Matrices

10.1 Arrays of Slate Matrices

```
#include <slate.h>

int main() {

  //
  // Make an array of slate Matrices.  Each Matrix is uninitialized.
  //
  Matrix<double> a[10];

  //
  // Initialize each Matrix.
  //
  for (int i=0; i<10; i++) {
    a[i].set_size(1,4,1,4);
    a[i] = (double)i;
  }

  cout << a[0] << endl;
  cout << a[5] << endl;
  cout << a[9] << endl;

  //
  // Make an array of array of slate Matrices.
  //
  Matrix<int> b[4][5];

  //
  // Initialize each Matrix.
  //
  for (int i=0; i<4; i++)
    for (int j=0; j<5; j++) {
      b[i][j].set_size(i,2*i+1,j,2*j+1);
      b[i][j] = i+j;
    }

  cout << b[0][0] << endl;
  cout << b[2][3] << endl;
  cout << b[3][4] << endl;
}
```

10.2 Pointers to Slate Matrices

```
#include <slate.h>

int main() {

  //
  // Make a pointer to slate Matrices.  Each Matrix is uninitialized.
  //
  Matrix<double> *a;
  a = new Matrix<double>[6];

  //
  // Initialize each Matrix.
  //
  for (int i=0; i<6; i++) {
    a[i].set_size(1,4,1,4);
    a[i] = (double)i;
  }

  cout << a[0] << endl;
  cout << a[2] << endl;
  cout << a[5] << endl;
}
```

11 Final Thoughts

All the examples in this tutorial are taken directly from the `examples` subdirectory. They should compile using `g++` without any special options and run without any modification.

We're actively developing `Slate++` , and will respond to questions as best we can. If you find any bugs, *please* consider submitting them to the `Slate++` repository on sourceforge:

`http://www.sourceforge.net/projects/slate`

And of course, if you feel you'd like to contribute to the `Slate++` project in any way, don't hesitate to send us a line.

Brian (thorndyb@phys.ufl.edu)
Robert (rthorndy@camosun.bc.ca)