Chapter 7. Linear Algebra

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can [illegible] release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *learnmodcppfinance@gmail.com*.

Introduction

Linear algebra is an essential part of computational finance, and as such it is a necessary and fundamental component for financial C++ software development. Options existing at present in the Standard Library are mostly limited to the `valarray` container, to be discussed briefly below. Over the last 15 years or so, some very good open source matrix algebra libraries have emerged that have been adopted by the financial software industry, as well as other computationally intensive domains such as data science and medi-

cal research. Progress is also being made toward linear algebra capabilities eventually being adopted into the Standard Library in C++23 and C++26.

As C++ did not have all the convenient built-in multidimensional array capabilities that came with Fortran platforms, quantitative programmers making the transition to C++ back in the 1990's often found themselves in an inconvenient situation with limited options. These included building up this functionality mostly from scratch, wrestling with interfaces to numerical Fortran libraries such as BLAS and LAPACK, or somehow convincing management to invest in a third-party C++ commercial library.

Contrived DIY solutions that were sometimes employed, based on what was available in C++ at the time, included representing a matrix as a `vector` of `vector`(s), or holding data in a two-dimensional dynamic C-array. Neither of these was particular palatable, with the former being cumbersome and inefficient, and the latter exposing the software to the risks associated with raw pointers and dynamic memory management. One seemingly useful feature available in the Standard Library, but not without controversy, was `std::valarray`. It has survived to the current day, and it provides vectorized operations and functions highly suitable for matrix and vector math. Its pros and cons will be presented momentarily.

The situation has improved substantially over the years with the release of multiple open-source linear algebra libraries for C++. Among these, two that have gained considerable critical mass in computational finance are the Eigen library **{1}**, and Armadillo **{2}**. A third option that has risen to some prominence in high-performance computing (HPC) is the Blaze library **{3}**. An earlier library, uBLAS, **{4}** is also available as part of the Boost libraries; however, it does not include the matrix decompositions and other capabilities available in the aforementioned offerings.

As a side note, open source R interface packages are available for each of these libraries. **{5}** These packages enable the integration of C++ code dependent on one or more of these libraries into R packages, usually to enhance run-time performance.

More recently, NVIDIA has released GPU-accelerated C++ linear algebra libraries as part of its HPC SDK. **{6}**.

A comparative list of both open source and commercial C++ linear algebra libraries, including those mentioned here, can be found on Wikipedia.**{7}** A more in-depth view of the Eigen library follows later in this chapter.

New features planned for C++23 and C++26 look set to finally provide the Standard Library with long overdue robust and well-supported linear algebra capabilities. Central among these new features is the `std::mdspan` multi-dimensional array representation planned for C++23. C++26 should then be updated with both a standardized interface to external BLAS-compatible libraries, as well as its own set of linear algebra facilities.

Below, we will first take a trip back in time and examine convenient mathematical features of `valarray`, and then demonstrate how it can be used as a proxy for a matrix. Following that, we will dive into the Eigen library of the present and demonstrate the basic matrix operations, along with matrix decompositions frequently used in financial modeling. Finally, a glimpse of the proposals for near-future Standard Library releases will also be presented.

valarray and Matrix Operations

The workhorse STL container `std::vector`, as we have seen, is an option for representing a vector in the mathematical sense. Common vector arithmetic, such as inner products, can be performed using STL algorithms. However, having been "designed to be a general mechanism for holding values...and to fit into the architecture of containers, iterators, and algorithms" {2.5 Stroustrup, Tour 2E} **{8}**, the common arithmetic vector operators such as addition and multiplication were not included as members in its implementation.

A Standard Library container class separate from the STL, called `valarray`, does support arithmetic operators and provides "optimizations that are often considered essential for serious numerical work" {ibid}. With slice and stride functions also accompanying the `valarray` class, it can also facilitate representation arrays of higher dimension, in particular a matrix.

While `valarray` has these very useful properties that would seem to make it an obvious choice for matrix math, it has played to mixed reviews. This dates back to its original specification which was never fully complete due to debates over whether to require a new technique at the time, expression templates (to be introduced shortly), that can significantly optimize performance. In the end, this was not mandated. As a result, "initial implementations were slow, and thus users did not want to rely on it." **{9}**

As of the time of this writing, however, two of the mainstream Standard Library distributions have implemented expression template versions of `valarray`, namely those that accompany the gcc and Clang compilers. In addition, the Intel oneAPI DPC++/C++ Compiler **{10}** ships with its own high-performance implementation of `valarray`. And as an incidental remark, specializations of `begin` and `end` functions were included as enhancements in C++11.

The moral of the story seems to be: know the capabilities of the implementation you intend to use. If its performance is suitable for your needs, then it can potentially be a very convenient option for matrix/vector operations, and vectorized versions of common mathematical functions. In addition, examining the properties of `valarray` may provide some context for future linear algebra enhancements planned for the Standard Library, with similar functionality in some cases, even though the implementations behind the scenes will be considerably different.

Arithmetic Operators and Math functions

The `valarray` container supports the standard arithmetic operators on an element-by-element basis, as well as scalar multiplication.

For example, the vector sum expression $3\mathbf{v}_1 + \frac{1}{2}\mathbf{v}_2$ can be naturally transcribed from a mathematical statement into C++ using `valarray` objects:

```
import <valarray>;
. . .

      std::valarray<double> v1{ 1.0, 2.0, 3.0,
                                1.5, 2.5 };

      std::valarray<double> v2{ 10.0, -20.0, 30.0,
                                -15.0, 25.0 };

      double vec_sum = 3.0 * v1 + 0.5 * v2;    // vec_sum is also a valarray <double>
```

The result is

```
8 -4 24 -3 20
```

Element-by-element multiplication is also implemented with the `*` operator:

```
double prod = v1 * v2;
```

This gives us

```
10 -40 90 -22.5 62.5
```

The dot (or inner) product of v_1 and v_2 is easily obtained by summing the preceding result by invoking the `sum()` member function on the `valarray` class:

```
double dot_prod = prod.sum();  // Result = 100
```

In addition to `sum`, `valarray` also has `max` and `min` functions, along with an `apply(.)` member function that applies an auxiliary function similar to `std::transform`:

```
double v1_max = v1.max();  // 3.0
double v1_min = v1.min();    // 1.0

// u and w are valarray<double> types
auto u = v1.apply([](double x) -> double {return x * x; });
// Result: 1, 4, 9, 2.25, 6.25

auto w = v1.apply([](double x) -> double {return std::sin(x) + std::cos(x);});
// Result: 1.38177 0.493151 -0.848872 1.06823 -0.202671
```

A subset of the `cmath` functions is conveniently defined for vectorized operations on the entirety of a `valarray`. For example, the following operations will return a `valarray` containing the images of the respective functions applied to each element in `v1` and `neg_val` below. Note that we can also negate each element in the same way as a plain numerical type with the subtraction operator.

```
// The result in each is a valarray<double>
auto sine_v1 = std::sin(v1);
auto log_v1 = std::log(v1);
auto abs_v1 = std::abs(neg_val);
```

```
auto exp_v1 = std::exp(neg_val);
auto neg_v1 = - v1;
```

Finally, as of C++11, specializations of the `begin` and `end` functions analagous to those provided for STL containers have been implemented for `valarray`. A simple example is as follows:

```
template<typename T>
void print(T t) { cout << t << " "; }

std::for_each(std::begin(w), std::end(w), print<double>);
```

Given the `apply(.)` member function on `valarray` and the built-in vectorized mathematical functions that are already available, STL algorithms `for_each` and `transform` might not be needed as often in the case of `valarray` compared to STL containers, however.

valarray as a Matrix Proxy

`valarray` provides the facilities to represent multidimensional arrays. In our case, we are specifically concerned with representing a two-dimensional array as a proxy for a matrix. This can be achieved with the `slice(.)` member function that can extract a reference to an individual row or column.

To demonstrate this, let us first lighten the notation by defining the alias

```
using mtx_array = std::valarray<double>;
```

Then, create a `valarray` object `val`, with the code formatted in a way to make it look like a 4×3 matrix:

```
mtx_array val{ 1.0, 2.0, 3.0,
               1.5, 2.5, 3.5,
               7.0, 8.0, 9.0,
               7.5, 8.5, 9.5};
```

The first row can be retrieved using the `std::slice` function, defined for a `valarray`, using the square bracket operator.

```
auto slice_row01 = val[std::slice(0, 3, 1)];
```

What this says is:

- Go to the first element of the `valarray`: index 0, value = 1.0.
- Choose 3 elements, beginning with the first
- Using a *stride* of 1, which in this case means to choose three consecutive rowwise elements

Similarly, the second column can be retrieved, using in this case a stride of 3, the number of columns:

```
auto slice_col02 = val[std::slice(1, 4, 3)];       // The 2nd rowwise element has index 1
```

It is important to note the `slice(.)` function returns a lighter `slice_array` type—that acts as a reference to the selected elements—rather than a full `valarray`. It does not, however, provide the necessary member functions and operators to access individual elements or compute, say, new rows comprising a matrix product. If we want to apply these functions to row or column data, we will need to construct corresponding new `valarray` objects. This will be seen in the next example, computing the dot product of a row in one matrix by the column in another, a necessary option in carrying out matrix multiplication.

To demonstrate this, suppose we have a 5×3 and a 3×5 matrix, each represented as a `valarray`. Note that we are also storing the number of rows and columns of each in separate variables.

```
mtx_array va01{ 1.0, 2.0, 3.0,
                1.5, 2.5, 3.5,
                4.0, 5.0, 6.0,
                4.5, 5.5, 6.5,
                7.0, 8.0, 9.0 };

unsigned va01_rows{ 5 }, va01_cols{ 3 };

mtx_array va02{ 1.0, 2.0, 3.0, 4.0, 5.0,
                1.5, 2.5, 3.5, 4.5, 5.5,
                5.0, 6.0, 7.0, 8.0, 8.5 };

unsigned va02_rows{ 3 }, va02_cols{ 5 };
```

If we were to apply matrix multiplication, it would require taking the dot product of each row of the first "matrix" by each column of the second. As an example, in order to get the dot product of the third row by the second column, we would first need the slice for each:

```
auto slice_01_row_03 = va01[std::slice(9, va01_cols, 1)];
auto slice_02_col_02 = va02[std::slice(1, va02_rows, 5)];
```

However, neither element-by-element multiplication nor the `sum()` member function is defined on a `slice_array`, so we need to construct corresponding `valarray` objects:

```
mtx_array va01_row03{ slice_01_row_03 };
```

```
mtx_array va02_col02{ slice_02_col_02 };
```

The dot product is then computed in the usual way:

```
double dot_prod = (va01_row03 * va02_col02).sum();
```

CAUTION

As previously noted, a `slice_array` acts as a reference to a block within a `valarray`. Operations and member functions such as `\*` and `sum` are not defined on `slice_array`, but assignment operators such as `*=` and `+=` are. Therefore, modification of a `slice_array` such as in the following example will also be reflected in the `valarray` itself. If we take the first row of `va01` as a slice:

```
auto slice_01_row_01 = va01[std::slice(0, va01_cols, 1)];
```

and then apply assignment operators

```
slice_01_row_01[0] *= 10.0;
slice_01_row_01[1] += 1.0;
slice_01_row_01[2] -= 3.0;
```

then the `valarray` contents would be

```
10   3    0
1.5  2.5  3.5
7    8    9
7.5  8.5  9.5
```

In summary, `valarray` conveniently provides the ability to apply mathematical operators and functions on an entire array, similar to Fortran 90, as well as more math-focused languages such as R and Matlab . As a reminder, however, performance can be highly dependent on the implementation used in your Standard Library distribution.

More information about `valarray`, its history, and its pros and cons can be found in the online supplemental chapter accompanying Josuttis, The C++ Standard Library, second edition). **{11}**

Subsequent to `valarray` and C++98, there have been some very positive developments regarding linear algebra in C++, some of which will now be presented.

Eigen

The first release of the Eigen library became available in 2006. Since then, it has been expanded to version 3.4.0 as of August of 2021. Starting with version 3.3.1, it has been licensed under the reasonably liberal Mozilla Public License (MPL) 2.0.

Eigen is comprised of template code that makes inclusion into other C++ projects very easy, in that in its standard installation there is no linking necessary to external binary files. Its incorporation of expression templates, facilitating lazy evaluation, provides for enhanced computational performance. It also received a a further boost in popularity after being chosen for incorporation into the well-respected TensorFlow **{12}** machine learning library, as well as the Stan Math Library. **{13}** More background on its suitability and popularity in finance is presented in a recent Quantstart article **{14}**.

Finally, the Eigen library is very well documented, with a tutorial and examples to help the newcomer get up and running quickly.

Lazy Evaluation

Lazy evaluation defers and minimizes the number of operations required in matrix and vector operations. Expression templates in C++ are used to encapsulate arithmetic operations – that is, expressions – inside templates such that they are delayed until they are actually needed. This can reduce the total number of operations, assignments, and temporary objects that are created when using conventional approaches.

An example of lazy evaluation that follows is based on a more comprehensive and illustrative discussion in the book by Peter Gottschling on modern C++ for scientific programing. **{15}**

Suppose you have four vectors in the mathematical sense, each with the same fixed number of elements, say

$$\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4$$

and you wish to store their sum in a vector *y*. The traditional approach would be to define the addition operator, take successive sums and store them in temporary objects, ultimately computing the final sum and assigning it to *y*.

In code, the operator could be defined in the generic sense such that vector addition would be defined for any arithmetic type:

```
template <typename T>
std::vector<T> operator + (const std::vector<T>& a,
      const std::vector<T>& b)
{
      std::vector<T> res(a.size());
      for (size_t i = 0; i < a.size(); ++i)
            res[i] = a[i] + b[i];
```

```
    return res;
}
```

Computing the sum of four vectors as follows

```
vector<double> v1{ 1.0, 2.0, 3.0 };
vector<double> v2{ 1.5, 2.5, 3.5 };
vector<double> v3{ 4.0, 5.0, 6.0 };
vector<double> v4{ 4.5, 5.5, 6.5 };

auto y = v1 + v2 + v3 + v4;
```

results in the following:

- 4 - 1 = 3 `vector` instances created (two temporary plus one final `y` instance)
- (4 - 1) $\times$ 3 assignments of `double` variables

As the number of vectors (say *m*) and the number of elements in each vector (say *n*) get larger, this generalizes to

- m – 1 `vector` objects allocated on the heap: m - 2 temporary, plus one return object (`y`)
- $(m-1)n$ assignments

With lazy evaluation, we can reduce the total number of steps, and thus improve efficiency for "large" *m* and *n*. More specifically, this can be accomplished by delaying the addition until all the data is ready, and then only at that time perform the sums for each element in the result. In code, this could be accomplished by writing a function as follows.

```
template <typename T>
std::vector<T> sum_four_vectors(const std::vector<T>& a, const std::vector<T>& b,
      const std::vector<T>& c, const std::vector<T>& d)
{
      // Assume a, b, c, and d all contain the same
      // number of elements:
      std::vector<T> sum(a.size());

      for (size_t i = 0; i < a.size(); ++i)
      {
            sum[i] = a[i] + b[i] + c[i] + d[i];
      }

      return sum;
}
```

Now, in this case,

- There are *no* temporary `vector` objects created; only the `sum` result is necessary
- The number of assignments is reduced to $n = 4$

The Eigen documentation provides additional background in the section Lazy Evaluation and Aliasing.

The previous example demonstrates how lazy evaluation can work, but the obvious problem is it would be unrealistic to write individual sum functions for all possible fixed numbers of vectors. Generalizing with expression templates is a far more challenging problem that will not be included here, but more information can be found in the Gottschling book {ibid 12}, as well as in Chapter 27 of the comprehensive book on C++ templates by Vandevoorde, Josuttis, and Gregor **{16}**.

Like any other optimization tool, it should not be applied blindly in the belief it will automatically make your code more efficient, as again there are cases where performance could actually be degraded.

Finally, for a very interesting presentation of a real-world case study of expression templates in financial risk management, a talk on the subject presented by Bowie Owens at CppCon 2019 **{17}** is very well worth watching.

Eigen Matrices and Vectors

The heart of the Eigen library is, not surprisingly, the `Matrix` template class. It is scoped with the `Eigen` namespace and requires the `Dense` header file be included. At the time of this writing, corresponding module imports have not yet been standardized. This means the header file will need to be included in the global fragment of a module.

The `Matrix` class carries six template parameters, but a variety of aliases are provided as specific types. These include fixed square matrix dimensions up to a maximum of four, as well as dynamic types for arbitrary numbers of rows and columns. The numerical type that a `Matrix` holds is also a template parameter, but this setting is also incorporated into individual aliases. For example, the following code will construct and display a fixed 3×3 matrix of `double` values, and a 4×4 matrix of `int`s. Braced (uniform) initialization by row can be used to load the data at construction.

```
#include <Eigen/Dense>
. . .

Eigen::Matrix3d dbl_mtx             // Contains 'double' elements
{
    {10.6, 41.2, 2.16},
    {41.9, 5.31, 13.68},
    {22.47, 57.43, 8.82}
```

```
};

Eigen::Matrix4i int_mtx                 // Contains 'int' elements
{
      {24, 0, 23, 13},
      {8, 75, 0, 98},
      {11, 60, 1, 3 },
      {422, 55, 11, 55}
};

cout << dbl_mtx << endl << endl;
cout << int_mtx << endl << endl;
```

Note also that the `<<` stream operator is overloaded, so the result can be easily displayed on the screen (in row-major order).

```
 10.6  87.4 58.63
 41.9  53.1 13.68
22.47 57.43  88.2

 24   0  23  13
  8  75   0  98
 11  60   1   3
422  55  11  55
```

Individual rows and columns can also be accessed, using 0-based indexing. The first column of the first matrix, and the third column of the second, for example are obtained with respective accessor functions:

```
cout << dbl_mtx.col(0) << endl << endl;
```

```
cout << int_mtx.row(2) << endl << endl;
```

This results in the following screen output:

```
10.6
41.9
22.47

11 60  1  3
```

Technically speaking, the type returned by either the `row` or `col` accessor is an `Eigen::Block`. It is similar to a `slice_array` accessed from a `valarray`, in that it acts as a lighter weight reference to the data. Unlike `slice_array`, it does not carry any mathematical operators such as `+=`.

For most of the financial examples considered in this book, the dimensions of a matrix will not be known a priori, nor will they necessarily be of a square matrix. In addition, the contents will usually be real numbers. For these reasons, we will primarily be concerned with the Eigen dynamic form for `double` types, aliased as `Eigen::MatrixXd`.

NOTE

1. As just mentioned, we will primarily use the dynamic Eigen `MatrixXd` form of a matrix (with `d` indicating `double` numerical elements); however, member and non-member functions will usually apply to any class derived from the `Matrix` template class. Where these functions are discussed, their relations with `Matrix` rather than `MatrixXd` may also be mentioned. Similarly, vector representation in Eigen will use `VectorXd`.
2. Linear algebra will inevitably involve subscripts and superscripts, eg x_{ij}, where in mathematical notation *i* might run from 1 to *m*, and *j* from 1 to *n*. However, C++ is 0-indexed, so a mathematical statement where *i* = 1 will be represented by `i = 0` in C++, *j* = *n* by `j = n - 1`, and so forth.

Construction of a `MatrixXd` can take on many forms. Data can be entered as before in row-major order, with the number of rows and columns implied by uniform initialization of individual rows. Alternatively, the dimensions can be used as constructor arguments, with the data input by streaming in row-major order. And, one more approach is to set each element one-by-one. An example of each is shown here:

```
using Eigen::MatrixXd;
. . .

MatrixXd mtx0
{
      {1.0, 2.0, 3.0},
      {4.0, 5.0, 6.0},
      {7.0, 8.0, 9.0},
      {10.0, 11.0, 12.0}
};

MatrixXd mtx1{4, 3};          // 4 rows, 3 columns
mtx1 << 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0;

MatrixXd mtx3{2, 2};
```

```
mtx3(0, 0) = 3.0;
mtx3(1, 0) = 2.5;
mtx3(0, 1) = -1.0;
mtx3(1, 1) = mtx3(1, 0) + mtx3(0, 1);
```

Note that the round bracket operator serves as both a mutator and an accessor, as demonstrated in the third example.

Two special cases, where either the number of columns or rows is one, are aliased as `VectorXd` and `RowVectorXd`. Construction options are similar to the `MatrixXd` examples above, again as shown in the Eigen documentation:

```
using Eigen::VectorXd;
using Eigen::RowVectorXd;
. . .

VectorXd a    { {1.5, 2.5, 3.5} };          // A column-vector with 3 coefficients
RowVectorXd b { {1.0, 2.0, 3.0, 4.0} };     // A row-vector with 4 coefficients

Eigen::VectorXd v(2);
v(0) = 4.0;
v(1) = v(0) - 1.0;
```

Matrix and Vector Math Operations

Matrix addition and subtraction are are conveniently implemented as overloads of the `+` and `-` operators, not unlike `valarray`. Similarly, these apply to vectors. Unlike `valarray`, however, the multiplication operator `*` refers to matrix multiplication rather than an element-by-element product. A separate set of functions is available for a wide range of element-by-element operations.

To multiply two matrices `A` and `B`, the code follows in natural mathematical order:

```
MatrixXd A
{
      {1.0, 2.0, 3.0},
      {1.5, 2.5, 3.5},
      {4.0, 5.0, 6.0},
      {4.5, 5.5, 6.5},
      {7.0, 8.0, 9.0}
};

MatrixXd B
{
      {1.0, 2.0, 3.0, 4.0, 5.0},
      {1.5, 2.5, 3.5, 4.5, 5.5},
      {5.0, 6.0, 7.0, 8.0, 8.5}
};

MatrixXd prod_ab = A * B;
```

This gives us as output:

```
  19    25    31    37  41.5
22.75 30.25 37.75 45.25    51
 41.5  56.5  71.5  86.5  98.5
45.25 61.75 78.25 94.75   108
  64    88   112   136 155.5
```

CAUTION

In the Eigen documentation, it is strongly recommended to "not use the auto keywords with Eigen's expressions, unless you are 100% sure about what you are doing. In particular, do not use the auto keyword as a replacement for a `Matrix<>` type."

The reasons behind this require an advanced discussion about templates that tie in with lazy evaluation. Lazy Evaluation provides advantages in efficiency, but it also can involve return types using `auto` that might be in the form of a reference rather than a full `Matrix` type. This can result in unexpected or undefined behavior. It becomes less of an issue as you become more familiar with various Eigen types, but for this introductory presentation, we will mostly heed this admonishment.

More information can be found in the documentation. **{18}**

The `*` operator is also overloaded for matrix-vector and row vector-matrix multiplication. As an example, suppose we have a portfolio of three funds, with correlation matrix of the returns and the vector of individual fund volatilities given (annualized). As is a typical problem, we might need to construct the covariance matrix given the data in this form in order to calculate the portfolio volatility. First, to form the covariance matrix, we would pre- and post-multiply the correlation matrix by diagonal matrices containing the fund volatilities:

```
MatrixXd corr_mtx
{
      {1.0, 0.5, 0.25},
      {0.5, 1.0, -0.7},
      {0.25, -0.7, 1.0}
};

VectorXd vols{ {0.2, 0.1, 0.4 } };
```

```
MatrixXd cov_mtx = vols.asDiagonal() * corr_mtx * vols.asDiagonal();
```

Note how the `VectorXd` member function `asDiagonal()` conveniently forms a diagonal matrix with the vector elements along the diagonal.

Then, given a vector of fund weights ω adding to 1, the portfolio volatility is then the square root of the quadratic form

$$\omega^{\mathsf{T}} \Sigma \omega$$

where Σ is the covariance matrix:

```
VectorXd fund_weights{ {0.6, -0.3, 0.7 } };
double port_vol = std::sqrt(fund_weights.transpose() * cov_mtx * fund_weights);
```

For element-by-element matrix multiplication, use the `cwiseProduct` member function. As an example, to multiply the individual elements in matrices of like dimension, say $\mathbf{A}$ and $\mathbf{B}^{\mathsf{T}}$, we would write:

```
MatrixXd cwise_prod = A.cwiseProduct(B.transpose());
```

There is in fact a set of `cwise...` (meaning *coefficient-wise*) member functions on an Eigen `Matrix` that perform element-by-element operations on two compatible matrices, such as `cwiseQuotient` and `cwiseNotEqual`. There are also unary `cwise` member functions that return the absolute value and square root of each element. These can be found in the Eigen documentation here. **{19}**

The result of the `*` operator when applied to two vectors depends upon which vector is transposed. For two vectors u and v, the dot (inner) product is computed as

$$\mathbf{u}^{\mathsf{T}}\mathbf{v}$$

while the outer product results when the transpose is applied to v:

$$uv^T$$

So, one needs to be careful when using the `*` operator with vectors. Suppose we have:

```
VectorXd u{ {1.0, 2.0, 3.0} };
VectorXd v{ {0.5, -0.5, 1.0} };
```

The respective results of the following vector multiplications will be different:

```
double dp = u.transpose() * v;          // Returns 'double'
MatrixXd op = u * v.transpose();        // Returns a Matrix
```

The first would result in a real value of 2.5, while the second would give us a singular 3×3 matrix:

```
 0.5 -0.5   1
   1   -1   2
 1.5 -1.5   3
```

To make life easier, Eigen provides a member function, `dot`, on the `VectorXd` class. By instead writing

```
dp = u.dot(v);
```

it should perhaps make it clearer which product we want. The result would be the same as before, plus the operation is commutative.

STL Compatibility

A very nice feature of both the Eigen `Vector` and `Matrix` classes is their compatibility with the Standard Template Library. This means you can iterate through an Eigen container, apply STL algorithms, and exchange data with STL containers.

STL and `VectorXd`

As a first example, suppose you wish to generate 12 random variates from a t-distribution and place the results in a `VectorXd` container. The process is essentially the same as what we saw using a `std::vector` and applying the `std::generate` algorithm with a lambda auxiliary function:

```
VectorXd u(12);                                  // 12 elements
std::mt19937_64 mt(100);                    // Mersenne Twister engine, seed = 100
std::student_t_distribution<> tdist(5);     // 5 degrees of freedom
std::generate(u.begin(), u.end(), [&mt, &tdist]() {return tdist(mt); });
```

NOTE

Prior to the recent Eigen 3.4 release, the `begin` and `end` member functions were not defined. In this case, you would need to instead use the `data` and `size` functions, as follows:

```
std::generate(u.data()), u.data() + u.size(),
[&mt, &tdist]() {return tdist(mt); });
```

Non-modifying algorithms such as `std::max_element` are also valid:

```
auto max_u = std::max_element(u.begin(), u.end());          // Returns iterator
```

Numeric algorithms, for example `std::inner_product`, can also be applied:

```
double dot_prod = std::inner_product(u.begin(), u.end(), v.begin(), 0.0);
```

The cleaner C++20 range versions are also supported on `VectorXd`:

```
VectorXd w(v.size());
std::ranges::transform(u, v, w.begin(), std::plus{});
```

Constructing a Matrix from STL Container Data

Matrix data can also be obtained from STL containers. This is convenient, as data can often arrive via interfaces from other sources where Eigen might not be included. The key is to use an `Eigen::Map`, which sets up a reference to (view of) the `vector` data rather than taking a copy of it.

As a first example, data residing in a `std::vector` container can be transferred to an `Eigen::Map`, which in turn can be used as the constructor argument for a `MatrixXd`. Note that the `Map` takes in a pointer to the first element in the `vector` (using the `data` member function), and the number of rows and columns, in its constructor.

```
std::vector<double> v{ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
      7.0, 8.0, 9.0, 10.0, 11.0, 12.0 };
```

```
Eigen::Map<MatrixXd> mtx_map(v.data(), 4, 3);
```

By default, `mtx_map` will provide row/column access to the data. Note that unlike creating a `MatrixXd` as before with an initializer list of data, the order will be column-major rather than row-major. Using `cout` yields the following output:

```
1 5 9
2 6 10
3 7 11
4 8 12
```

As a `Map` is a lighter weight view of the data in `v`, it does not carry with it all of the functionality as found on a `Matrix` object, in a sense similar to a slice taken from a`valarray`. If you need this functionality, then you can construct a `MatrixXd` instance by putting the `Map` object in its constructor:

```
MatrixXd mtx_from_std_vector{ mtx_map };
```

The default arrangement of the data in a `Map` will be in *column-major order*, which is different from the earlier `MatrixXd` examples constructed with numerical data. If row-major is required, you can specify row-major at the outset, but this will require an `Eigen::Matrix` template parameter with storage explicitly set to `RowMajor`, as `MatrixXd` does not have its own template parameter for storage method:

```
Eigen::Map<Eigen::Matrix<double, 4, 3, Eigen::RowMajor>>
   mtx_row_major_map{ v.data(), 4, 3 };
```

If the matrix is square, you can just transpose the `Map` in place to put it in row-major order:

```
// Square matrix, place in row-major order:
std::vector<double> sq_data{ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
      7.0, 8.0, 9.0 };

Eigen::Map<MatrixXd> sq_mtx_map{ sq_data.data(), 3, 3 };
sq_mtx_map.transposeInPlace();
```

The result is then:

```
1 2 3
4 5 6
7 8 9
```

WARNING

Attempting to transpose a non-square matrix using `Map` can result in a program crash. In this case, you will need to create a full `MatrixXd` object before applying `transposeInPlace`.

Applying STL Algorithms to a Matrix

STL algorithms can also be applied to matrices row by row, or column by column. Suppose we have a 4 × 3 matrix as follows:

```
      MatrixXd vals
      {
            { 9.0, 8.0, 7.0 },
```

```
        { 3.0, 2.0, 1.0 },
        { 9.5, 8.5, 7.5 },
        { 3.5, 2.5, 1.5 }
    };
```

The `rowwise()` member function on an Eigen `Matrix` sets up an iteration by row. Each `row` is a reference to (view of) the respective data, so we can square each element of the matrix in place as follows:

```
for (auto row : vals.rowwise())
{
    std::ranges::transform(row, row.begin(), [](double x) {return x * x; });
}
```

The `colwise()` member function is similar, in this case sorting each column of the matrix:

```
for (auto col : vals.colwise())
{
    std::ranges::sort(col);
}
```

The end result after applying both algorithms gives us

```
   9    4    1
12.25  6.25  2.25
  81   64   49
90.25 72.25 56.25
```

Each element has been squared, and each column has been rearranged in ascending order.

Financial programmers often need to write code that will compute the log returns on a set of equity or fund prices. For example, suppose we have a set of 11 monthly prices for three ETF's, in the following three columns:

```
25.5  8.0 70.5
31.0  7.5 71.0
29.5  8.5 77.5
33.5  5.5 71.5
26.5  9.5 72.5
34.5  8.5 75.5
28.5  9.0 72.0
23.5  7.5 73.5
28.0  8.0 72.5
31.5  9.0 73.0
32.5  9.5 74.5
```

The first step in computing log returns is to compute the natural log of each price. This can be done by applying the `transform` algorithm row by row:

```
for (auto row : prices_to_returns.rowwise())
{
      std::ranges::transform(row, row.begin(), [](double x) {return std::log(x); });
}
```

Then, to get the log returns, we need to subtract from each log price its predecessor. For this, we can apply the `adjacent_difference` numeric algorithm to each column:

```
for (auto col : prices_to_returns.colwise())
{
      std::adjacent_difference(col.begin(), col.end(), col.begin());
}
```

This result is still an 11×3 matrix, with the first row still containing the logs of the prices in the first row.

```
  3.23868    2.07944    4.25561
 0.195309 -0.0645385 0.00706717
-0.0495969   0.125163  0.0875981
 0.127155  -0.435318 -0.0805805
-0.234401   0.546544  0.0138891
 0.263815  -0.111226  0.0405461
-0.191055  0.0571584 -0.0474665
-0.192904  -0.182322  0.0206193
 0.175204  0.0645385 -0.0136988
 0.117783   0.117783 0.00687288
0.0312525  0.0540672  0.0203397
```

What we want are the monthly returns alone, so we need to remove the first row. This can be achieved by applying the `seq` function, introduced in Eigen 3.4, which provides an intuitive way of extracting a sub-matrix view (an `Eigen::Block`) from a `Matrix` object. The example here shows how to extract all rows below the first:

```
MatrixXd returns_mtx{ prices_to_returns(Eigen::seq(1, Eigen::last),
      Eigen::seq(0, Eigen::last)) };
```

What this says is:

1. Start with the second row (index 1) and include all rows down to the last row: `Eigen::seq(1, Eigen::last)`
2. Take all columns from the first (index 0) to the last: `Eigen::seq(0, Eigen::last)`
3. Use this submatrix data alone in the constructor for the resulting `returns_mtx`

The results held in `returns_mtx` are then the log returns alone:

```
  0.195309 -0.0645385 0.00706717
-0.0495969   0.125163  0.0875981
  0.127155  -0.435318 -0.0805805
 -0.234401   0.546544  0.0138891
  0.263815  -0.111226  0.0405461
 -0.191055  0.0571584 -0.0474665
 -0.192904  -0.182322  0.0206193
  0.175204  0.0645385 -0.0136988
  0.117783   0.117783 0.00687288
 0.0312525  0.0540672  0.0203397
```

Now, suppose the portfolio allocation is fixed at 35%, 40%, and 25% for each respective fund (column-wise). We can get the monthly portfolio returns by multiplying the vector of allocations by `returns_mtx`:

```
VectorXd monthly_returns = returns_mtx * allocations;
```

The result is

```
 0.0443094
 0.0546058
-0.149768
  0.14005
```

```
 0.0579814
-0.0558726
 -0.13529
 0.0837121
 0.0900555
 0.0376502
```

Matrix Decompositions and Applications

Matrix decompositions are of course essential for a variety of financial engineering problems. Below, we will discuss a few examples.

Systems of Linear Equations and the LU Decomposition

Systems of linear equations are ubiquitous in finance and economics, particularly in optimization, hedging, and forecasting problems. To show how solutions can be found in Eigen, let us just look at a generic problem and implement it in code. The LU (Lower/Upper-triangular matrix) decomposition is a common approach in numerical methods. As opposed to coding it ourselves, Eigen can get the job done in two lines of code.

Suppose we want to solve the following system of linear equations for x_1, x_2, and x_3:

$$3x_1 - 5x_2 + x_3 = 0$$

$$-x_1 - x_2 + x_3 = -4$$

$$2x_1 - 4x_2 + x_3 = -1$$

This sets up the usual matrix equation

$$\mathbf{Ax} = \mathbf{b}$$

where

$$\mathbf{x} = x_1 \quad x_2 \quad x_3]^\mathsf{T}$$

The matrix $\mathbf{A}$ contains the coefficients, and the column vector $\mathbf{b}$ contains the constants on the right-hand side of the equations. The LU algorithm decomposes the matrix $\mathbf{A}$ into the product of lower- and upper-triangular matrices $\mathbf{L}$ and $\mathbf{U}$ in order to solve for $\mathbf{x}$.

In Eigen, form the matrix $\mathbf{A}$ and the vector $\mathbf{b}$:

-

```
MatrixXd A          // Row-major
{
      {3.0, -5.0, 1.0},
      {-1.0, -1.0, 1.0},
      {2.0, -4.0, 1.0}
};

VectorXd b
{
      {0.0, -4.0, 1.0}
};
```

The next step is to create an instance of the `Eigen::FullPivLU` class with template parameter `MatrixXd`. This sets up the LU decomposition. To find the vector containing the solution, all that is left to do is call the `solve(.)` member function on this object. This means two lines of code, as promised:

```
Eigen::FullPivLU<MatrixXd> lu(A);
VectorXd x = lu.solve(b);
```

The solution for `x` is then (from top to bottom x_1, x_2, x_3)

```
2.5
1.5
  0
```

Several other decomposition methods are available in Eigen for solving linear systems, where as is often the case a choice between speed and accuracy needs to be considered. The LU decomposition in the example above is among the best for accuracy, although there are others that may be faster but do not offer the same level of stability. The complete list can be found here:

Basic linear solving **{20}**

A comparison of the matrix decomposition methods in Eigen is also available:

Catalogue of decompositions offered by Eigen **{21}**

Fund Tracking with Multiple Regression and the Singular Value Decomposition

Another common programming problem in finance is fund tracking with multiple regression. Examples include

- Tracking whether a fund of hedge funds is following its stated allocation targets by regressing its returns on a set of hedge fund style index returns.
- Tracking the sensitivity of a portfolio to changes in different market sectors.
- Tracking the goodness of fit of mutual funds offered in guaranteed investment products such as variable annuities, with respect to their respective fund group benchmarks.

In multiple regression, one is tasked with finding a vector

$$\hat{\beta} = [\beta_1 \quad \beta_2 \quad \dots \quad \beta_n]^T$$

that satisfies the matrix form of the normal equations

$$\hat{\beta} = [X^T X]^{-1} X^T Y$$

$$\hat{\beta} = [X^T X]^{-1} X^T y$$

where $\mathbf{X}$ is the design matrix containing *p* columns of independent variable data, and *n* observations (rows), with the number of observations *n* "comfortably" greater than the number of data columns *p* to ensure stability. This is typically the case in these types of fund tracking applications.

NOTE

For fund tracking applications, the intercept term β_0 can usually be dropped, so it is omitted here. For regression cases where an intercept is required, a final column of ones needs to be appended to the design matrix when using Eigen. _

A commonly employed solution is the compact Singular Value Decomposition (SVD), which replaces the matrix $\mathbf{X}$ with the decomposition $\mathbf{U}\Sigma\mathbf{V}^T$, where $\mathbf{U}$ is an $n \times p$ matrix, $\mathbf{V}$ is $p \times p$, and Σ is a $p \times p$ diagonal matrix of strictly positive values. Making this substitution in the original formula for $\widehat{\beta}$, we get

$$\widehat{\beta} = \mathbf{V}\Sigma\mathbf{U}^{\mathsf{T}}\mathbf{y}$$

Eigen provides two SVD solvers that can be used for obtaining the least squares estimates of the regression coefficients. The first of these, as described in the Eigen documentation, is the Jacobi SVD decomposition of a rectangular matrix_ **{22}**. The documentation also states the Jacobi version is recom-

mended for design matrices of 16 columns or less, which is sometimes sufficient for fund tracking problems.

The way this works is given the design matrix predictor data contained in a `MatrixXd X`, Eigen sets up the SVD logic inside the class template `Eigen::JacobiSVD`. Then, given the response data contained in a `VectorXd Y`, solving for $\widehat{\beta}$ is again just two lines of code:

```
Eigen::JacobiSVD<MatrixXd> svd{ X, Eigen::ComputeThinU | Eigen::ComputeThinV };
VectorXd beta = svd.solve(Y);
```

NOTE

The `Eigen::ComputeThinU | Eigen::ComputeThinV` bitwise-or parameter instructs the program to use the compact version of SVD. If the full SVD is desired, then the above `JacobiSVD` instantiation would be:

```
Eigen::JacobiSVD<MatrixXd> svd{ X, Eigen::ComputeFullU | Eigen::ComputeFullV };
```

In this case, the $\mathbf{U}$ matrix will be $n \times n$, and Σ will be an $n \times p$ pseudoinverse.

There are no default settings.

As an example, suppose we have three sector ETF's and wish to examine their relationship to the broader market (eg the S&P 500), and suppose we have 30 daily observations.

The design matrix will contain the three ETF returns and is stored in a `MatrixXd` called `X`, as follows:

```
MatrixXd X{ 3, 30 };    // 3 sector funds, 30 observations (will transpose)
X    <<
     // Sector fund 1
     -0.044700388, -0.007888394, 0.042980064, 0.016416586, -0.01779658, -0.016714149,
     0.019472031, 0.029853293, 0.023126097, -0.033879088, -0.00338369, -0.018474493,
     -0.012509815, -0.01834808, 0.010626754, 0.036669407, 0.010811115, -0.035571742,
     0.027474007, 0.005406069, -0.010159427, -0.006145632, -0.0103273, -0.010435171,
     0.011127197, -0.023793709, -0.028009362, 0.00218235, 0.008683152, 0.001440032,

     // Sector fund 2
     -0.019002703, 0.026036835, 0.03782709, 0.010629292, -0.008382267, 0.001121697,
     -0.004494407, 0.017304537, -0.006106293, 0.012174645, -0.003305029, 0.027219671,
     -0.036089287, -0.00222959, -0.015748493, -0.02061919, -0.011641386, 0.023148757,
     -0.002290732, 0.006288094, -0.012038397, -0.029258743, 0.011219297, -0.008846992,
     -0.033738048, 0.02061908, -0.012077677, 0.015672887, 0.041012907, 0.052195282,

     // Sector fund 3
     -0.030629136, 0.024918984, -0.001715798, 0.008561614, 0.003406931, -0.010823864,
     -0.010361097, -0.009302434, 0.008142014, -0.004064208, 0.000584335, 0.004640294,
     0.031893332, -0.013544321, -0.023573641, -0.004665085, -0.006446259, -0.005311412,
     0.045096308, -0.007374697, -0.00514201, -0.001715798, -0.005176363, -0.002884991,
     0.002309361, -0.014521608, -0.017711709, 0.001192088, -0.00238233, -0.004395918;

X.transposeInPlace();
```

Similarly, the market returns are stored in a `VectorXd` array `Y`:

```
VectorXd Y{ 30 }; // 30 observations of market returns
Y <<
     -0.039891316, 0.00178709, -0.0162018, 0.056452057, 0.00342504, -0.012038314,
     -0.009997657, 0.013452043, 0.013485674, -0.007898137, 0.008111428, -0.015424523,
```

```
-0.002161451, -0.028752191, 0.011292655, -0.007958389, -0.004002386, -0.031690771,
0.026776892, 0.009803957, 0.000886608, 0.01495181, -0.004155781, -0.001535225,
0.013517306, -0.021228542, 0.001988701, -0.02051788, 0.005841347, 0.011248933;
```

Obtaining the regression coefficients is just a matter of compiling and running the SVD using the two lines of code shown at the outset, which gives us for `beta`:

```
 0.352339
-0.0899004
 0.391252
```

The $\mathbf{U}$ and $\mathbf{V}$ matrices can also be obtained if desired, along with the Σ matrix, using the following accessor functions:

```
cout << svd.matrixU() << endl;              // U: n x p = 30 x 3
cout << svd.matrixV() << endl;              // V: p x p = 3 x 3
cout << svd.singularValues().asDiagonal() << endl;          // Sigma: p x p = 3 x 3
```

An alternative SVD solver, the *Bidiagonal Divide and Conquer SVD*, **{23}** is also available in Eigen. Per the documentation, it is recommended for design matrices with greater than 16 columns for better performance.

The setup is the same as the Jacobi case, but using the `BDCSVD` class in place of `JacobiSVD`:

```
Eigen::BDCSVD<MatrixXd> svd(X, Eigen::ComputeThinU | Eigen::ComputeThinV);
VectorXd beta = svd.solve(Y);
```

It should be noted Eigen also provides QR decompositions as well as the capability of just coding in the normal equations with Eigen matrices and operations. As noted in the documentation, *Solving linear least squares systems*, these can be faster than SVD methods, but they can be less accurate. **{24}** These are alternatives you might want to consider if speed is an issue, but under the right conditions.

Correlated Random Equity Paths and the Cholesky Decomposition

The Cholesky decomposition is a popular tool in finance for generating correlated Monte Carlo equity path simulations. For example, when pricing basket options, covariances between movements in the basket securities need to be accounted for, specifically in generating correlated random normal draws. This is in contrast to generating a random price path for a single underlying security, as we saw in Chapter 8:

$$S_t = S_{t-1} e^{\left(\frac{r-\sigma^2}{2}\right)\Delta t + \sigma\varepsilon_t\sqrt{\Delta t}}$$

where again $\varepsilon_t \sim N(0,1)$, σ is the equity volatility, and r represents the risk-free interest rate,

In the case of a basket option, we now need to generate a path for each of say *m* assets at each time t, where the $\sigma\varepsilon_t$ term is replaced by a random term $w_t^{(i)}$ that is again based on a standard normal draw, but whose fluctuations also contain correlations with the other assets in the basket. Therefore, we need to generate a set of prices $\left\{S_t^{(i)}\right\}$ where

$$S_t^{(1)} = S_{t-1}^{(1)} e^{\left(\frac{r-\sigma^2}{2}\right)\Delta t + w_t^{(1)}\sqrt{\Delta t}} \; (*)$$

$$\vdots$$

$$S_t^{(m)} = S_{t-1}^{(m)} e^{\left(\frac{r-\sigma^2}{2}\right)\Delta t + w_t^{(m)}\sqrt{\Delta t}}$$

for each asset $i = 1, \cdots, m$ at each time step $t_j, j = 1, \cdots, n$. Our task is to calculate the random but correlated vector

$$w_t^{(1)} \quad \cdots \quad w_t^{(m)}\Big]^{\mathsf{T}}$$

for each time t. This is where the Cholesky decomposition – available in Eigen – comes into play. For an $n \times n$ covariance matrix Σ, assuming it is positive definite, will have a Cholesky decomposition

$$\Sigma = LL^{\mathsf{T}}$$

where *L* is a lower triangular matrix. Then, for a vector of standard normal variates $\mathbf{z}$,

$$z_1 \quad z_2 \quad \cdots \quad z_m]^{\mathsf{T}}$$

the $n \times 1$ vector generated by $\mathbf{Lz}^{\mathsf{T}}$ will provide a set of correlated volatilities that can be used to generate in a single time step a random scenario of prices for each underlying security. For each time step t, we replace $\mathbf{z}$ with $\mathbf{z_t}$ to then arrive at our desired result:

$$\mathbf{w_t} = \mathbf{L}\mathbf{z}_t^{\mathsf{T}}$$

This can then be extended an arbitrary number of n time steps by placing each vector $\mathbf{z_t}$ into a column of a matrix, say $\mathbf{Z}$. Then, we can generate the entire set of vectors of correlated random variables in one step, and place the results in a matrix

$$\mathbf{W} = \mathbf{LZ}$$

Eigen provides a Cholesky decomposition of a `MatrixXd` object, using the `Eigen::LLT` class template with parameter `MatrixXd`. It again consists of creating an object of this class, and then calling a member function, `matrixL`, which returns the matrix $\mathbf{L}$ above.

As an example, suppose we have four securities in the basket, with the following covariance matrix.

```
MatrixXd cov_basket
{
   { 0.01263, 0.00025, -0.00017, 0.00503},
   { 0.00025, 0.00138,  0.00280, 0.00027},
   {-0.00017, 0.00280,  0.03775, 0.00480},
   { 0.00503, 0.00027,  0.00480, 0.02900}
};
```

The Cholesky decomposition is set up when the matrix data is used to construct the `Eigen::LLT` object. Calling the member function `matrixL()` computes the decomposition and returns the resulting lower triangle matrix:

```
Eigen::LLT<Eigen::MatrixXd> chol{ cov_basket };
MatrixXd chol_mtx = chol.matrixL();
```

This gives us

```
   0.1124        0        0        0
 0.002226 0.0370332        0        0
-0.0015544 0.0756179 0.178975        0
 0.0447889 0.00464393 0.0252348 0.162289
```

Suppose now there will be six time steps in the Monte Carlo model over a period of one year. This means we will need six vectors containing four standard normal variates each. For this, we can use the Standard Library `<random>` functions and generate a 4×6 matrix, and place the random vectors in consecutive columns.

First, create a `MatrixXd` object with four rows and six columns:

```
MatrixXd corr_norms{ 4, 6 };
```

The first step will be to populate this matrix with uncorrelated standard normal draws. As we did previously (in Chapter 8), we can set up a random engine and distribution, and capture these in a lambda to generate the standard normal variates:

```
std::mt19937_64 mt_norm{ 100 };          // Seed is arbitrary, just set to 100 again
std::normal_distribution<> std_nd;

auto std_norm = [&mt_norm, &std_nd](double x)
{
   return std_nd(mt_norm);
};
```

Because each column in a `MatrixXd` can be accessed as a `VectorXd`, we can again iterate columnwise through the matrix and apply the `std::ranges::transform` algorithm to each in a range-based `for` loop.

```
for (auto col : corr_norms.colwise())
{
   std::ranges::transform(col,
      col.begin(), std_norm);
}
```

This interim result would resemble the following, with actual results depending on the compiler (in this case using the Microsoft Visual Studio 2022 compiler):

```
  0.201395   0.197482    1.22857    1.40751    1.82789  -0.150014
-0.0769593  0.0830647    1.86252   0.122389  -0.949222   0.667817
  0.936051    1.16233  -0.642932   0.538005   -1.82688  -0.451039
-0.00916217  -2.79186  -0.434655 -0.0553752    1.46312   0.345527
```

Then, to get the *correlated* normal values, multiply the above result by the Cholesky matrix, and reassign `corr_norms` to the result.

```
MatrixXd corr_norms = chol_mtx * corr_norms;
```

This result, which corresponds to the matrix $\mathbf{W}$ in the mathematical derivation, comes out to be:

```
  0.0226368   0.0221969    0.138091    0.158204    0.205455  -0.0168616
-0.00240174  0.00351574   0.0717097  0.00766557  -0.0310838   0.0243974
  0.161397    0.214002   0.0238613    0.103356   -0.401585  -0.0299926
 0.0307971   -0.414526  -0.0230881   0.0681989    0.268808   0.0410757
```

Each successive column in `corr_norms` will provide a set of four correlated random variates that can be substituted in for $w_t^{(1)} \cdots w_t^{(4)}$ at each time $t = 1, \cdots, 6$ in (*) above. First, we will need the spot prices of the four underlying equities, which can be stored in an Eigen `VectorXd` (pretend they are retrieved from a live market feed). For example:

```
VectorXd spots(4);        // Init spot prices from market
spots << 100.0, 150.0, 25.0, 50.0;
```

Next, we will need a matrix in which to store a random price path for each equity, starting with each spot price, to be stored in an additional first column, making it a 4×7 matrix.

```
MatrixXd integ_scens{ corr_norms.rows(), corr_norms.cols() + 1 }       // 4 x 7 matrix
integ_scens.col(0) = spots;
```

Suppose then, the time to maturity is one year, divided into six equal time steps. From this, we can get the Δt value, say `dt`.

```
double time_to_maturity = 1.0;
unsigned num_time_steps = 6;
double dt = time_to_maturity / num_time_steps;
```

Each successive price for a given underlying equity, as shown in (*), can be computed in a lambda, where `price` is the previous price in the scenario, and `vol` is the volatility of the particular equity.

```
auto gen_price = [dt, rf_rate](double price, double vol, double corr_norm) -> double
{
   double expArg1 = (rf_rate - ((vol * vol) / 2.0)) * dt;
   double expArg2 = corr_norm * std::sqrt(dt);
   double next_price = price * std::exp(expArg1 + expArg2);
   return next_price;
};
```

Finally, for each underlying equity at each time step, we can set up an iteration and call the lambda at each step:

```
for (unsigned j = 1; j < integ_scens.cols(); ++j)
{
   for (unsigned i = 0; i < integ_scens.rows(); ++i)
   {
      integ_scens(i, j) = gen_price(integ_scens(i, j - 1), vols(i), corr_norms(i, j - 1));
   }
}
```

For this example, the results are

```
100     100.99      101.972     107.952     115.226     125.384     124.6
150     150.086     150.535 155.248 155.976 154.248 156.034
25      26.6633     29.0545     29.2955     30.5129 25.8607     25.5082
50      50.5946     42.6858     42.2536     43.414      48.4132 49.1949
```

Again, results may vary due to differences in the implementation of `<random>` between different Standard Library releases among vendors.

Yield Curve Dynamics and Principal Components Analysis

Principal Components Analysis (PCA) is a go-to tool for determining the sources and magnitudes of variation that drive changes in the shape of a yield curve. Given a covariance matrix of daily changes in yields spanning a range of bond maturities, PCA is employed by first calculating the eigenvalues of this matrix, and ordering them from highest to lowest. Then, the weightings are calculated by dividing each eigenvalue by the some of all the eigenvalues.

Empirical research has shown the contribution of first three eigenvalues will comprise nearly the entirety of the weightings, where the first weighting corresponds to parallel shifts in the yield curve, the second

corresponds to variations in its "tilt" or "slope", and the third to the curvature. The reasons and details behind this can be found in Chapter 18 of the very fine computational finance book by Ruppert and Matteson **{25}**, and in Chapter 3 of the classic text on interest rate derivatives by Rebonato **{26}**.

Rigorous statistical tests exist for measuring significance, but the weights alone can provide a relative estimated measure of each source of variation.

Section 18.2 of the text by Ruppert and Matteson {op cit 25} provides an excellent example on how to apply principal components analysis to publicly available US Treasury yield data {put URL here}. The resulting covariance matrix—input as constructor data for the following `MatrixXd` object, is based on fluctuations in differenced US Treasury yields **{28}** for eleven different maturities, ranging from one month to 30 years. The underlying data is taken from the period from January 1990 to October 2008.

To calculate the eigenvalues, first load the covariance matrix data into the upper triangular region of a `MatrixXd` instance.

```
MatrixXd term_struct_cov_mtx
{
    // 1 month
    { 0.018920,  0.009889, 0.005820,     0.005103, 0.003813,  0.003626,
        0.003136, 0.002646, 0.002015, 0.001438, 0.001303 },

    // 3 months
    { 0.0, 0.010107, 0.006123, 0.004796, 0.003532, 0.003414,
        0.002893, 0.002404, 0.001815, 0.001217, 0.001109},

    // 6 months
    { 0.0, 0.0, 0.005665, 0.004677, 0.003808, 0.003790,
        0.003255, 0.002771, 0.002179, 0.001567, 0.001400 },

    // 1 year
```

```
            { 0.0, 0.0, 0.0, 0.004830, 0.004695, 0.004672,
                0.004126, 0.003606, 0.002952, 0.002238, 0.002007},

        // 2 years
        { 0.0, 0.0, 0.0, 0.0, 0.006431, 0.006338,
            0.005789, 0.005162, 0.004337, 0.003343, 0.003004},

        // 3 years
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.006524,
            0.005947, 0.005356, 0.004540, 0.003568, 0.003231 },

        // 5 years
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.005800, 0.005291, 0.004552, 0.003669, 0.003352 },

        // 7 years
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.004985, 0.004346, 0.003572, 0.003288 },

        // 10 years
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.003958, 0.003319, 0.003085 },

        // 20 years
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.003062, 0.002858 },

        // 30 years
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.002814 }
    };
```

As a real symmetric matrix is trivially self-adjoint (no complex components), Eigen can apply a function `selfadjointView<.>()` to an upper (or lower) triangular matrix to define a view of a symmetric covariance matrix. The eigenvalues (all real) can then be obtained by applying the `eigenvalues()` function on the symmetric result:

```
VectorXd eigenvals = term_struct_cov_mtx.selfadjointView<Eigen::Upper>().eigenvalues();
```

In order to determine the weight of each principal component, we need to divide each eigenvalue by the sum of all the eigenvalues:

```
double total_ev = std::accumulate(eigenvals.cbegin(), eigenvals.cend(), 0.0);
std::ranges::transform(eigenvals, eigenvals.begin(),
    [total_ev](double x) {return x / total_ev; });
```

And finally, to view the results in order of the principal components, the weighted values need to be arranged from largest to smallest:

```
std::ranges::sort(eigenvals, std::greater{});
```

Examining the contents of `eigenvals`, we would find the following results:

```
0.67245 0.213209 0.073749
0.023811 0.00962511 0.00275773
0.0016744 0.00114298 0.000740139
0.000528862 0.000311391
```

From this, we can see the effects of parallel shifts and the "tilt" of the yield curve are the dominant effects, with relative weights of 67.2% and and 21.3%, while the curvature effect is smaller at 7.4%.

Future Directions: Linear Algebra in the Standard Library

Three proposals related to linear algebra have been submitted to the ISO C++ committee.

- `std::mdspan` : A polymorphic multidimensional array reference (P0009) **{28}**
- A free function linear algebra interface based on the BLAS (P1673) **{29}**
- Add linear algebra support to the C++ standard library (P1385) **{30}**

`mdspan` (P0009) can impose a multidimensional array structure on a reference to a container, such as an STL `vector`. Using the example of a `vector` containing the data, and a referring `mdspan` representing a matrix, the number of rows and columns are set at construction of the `mdspan`. An `mdspan` can also take the form of higher dimensional arrays, but for our purposes we will concern ourselves with the two-dimensional case for matrix representations. `mdspan` is officially slated for release in C++23.

The second proposal (P1673) is for a standard interface to external linear algebra "based on the dense Basic Linear Algebra Subroutines (BLAS)", corresponding "to a subset of the BLAS Standard." {op cit 29}. In other words, code could be written independently of whichever external linear algebra library is used, making code maintenance much easier and less error-prone. As noted in the proposal, the "interface is designed in the spirit of the C++ Standard Library's algorithms", and "uses mdspan..., to represent matrices and vectors" {ibid}. Furthermore, the "interface is designed in the spirit of the C++ Standard Library's algorithms"{ibid}. It is currently planned for C++26.

The third proposal (P1385) is to provide actual BLAS-type functionality within the Standard Library. Its primary goal is to "provide a (sic) matrix vocabulary types for representing the mathematical objects and fundamental operations relevant to linear algebra" (op cit {30}). This proposal is also currently planned for release with C++26.

mdspan (P0009)

As mentioned above, for representing a matrix, `mdspan` establishes a reference to a contiguous container and then imposes the number of rows and columns. If these parameters are known at compile time, creating an `mdspan` is easy:

```
vector<int> v{ 101, 102, 103, 104, 105, 106 };
auto mds1 = std::mdspan{ v.data(), 3, 2 };
```

Note that `mdspan` uses the `data()` member function on `vector` to access its contents.

In `mdspan` parlance, rows and columns are referred to as *extents*, with the number of rows and columns accessed by the index of each, 0 for rows and 1 for columns. The total number of extents is referred to as the *rank*, so in this case, the rank is 2. For higher-order multidimensional arrays, the rank would be greater than two.

```
size_t n_rows{ mds1.extent(0) };          // 3
size_t n_cols{ mds1.extent(1) };      // 2
size_t n_extents{ mds1.rank() };      // 2
```

NOTE

The term *rank* as applied to `mdspan` is not the same as the mathematical definition of the rank of a matrix. This naming might unfortunately seem confusing, but it's something one should be aware of.

Elements of the `mdspan` object will be accessible with another new feature in C++23, namely the square bracket operator with multiple indices:

```
for (size_t i = 0; i < mds1.extent(0); ++i)
{
      for (size_t j = 0; j < mds1.extent(1); ++j)
            cout << mds1[i, j] << "\t";

      cout << "\n";
}
```

This is a welcome improvement, as it will no longer be necessary to put each index in a separate bracket pair, as was the case with C-style arrays:

```
double ** a[3][2];       // Ugh

// Could put a[2, 1] if it were an mdspan instead:
a[2][1] = 5.4;

...

delete [] a;
```

The run-time result of the previous nested loop displays a 3 × 2 matrix:

```
101   102
103   104
105   106
```

It is also possible to define a matrix with different dimensions to the same data:

```
auto mds2 = std::mdspan(v.data(), 2, 3);        // 2 x 3
```

Applying the same loop above, but replacing `mds1` with `mds2` would then display as expected:

```
101   102   103
104   105   106
```

One thing to be careful of is modifying data in the `mdspan` object or in the original `vector` will change it in both locations due to the referential relationship between the two. For example, modification of the last element of the vector

```
v[5] = 874;
```

will show up in both of the `mdspan` objects. `mds1` becomes

```
101   102
103   104
105   874
```

and `mds2` is now

```
101   102   103
104   105   874
```

Likewise, changing the value of an element in `mds2` will be reflected in both `mds1` and the vector `v`.

In quant finance applications, it will often be the case where the fixed number of rows and columns are not known at compile time. Suppose `m` and `n` are the numbers of rows and columns to be determined at runtime. These dimensions can be set dynamically by replacing the fixed settings of 2 and 3 in the previous example

```
auto mds2 = std::mdspan(v.data(), 2, 3);
```

with the `std::extents{m, n}` object, as shown in the `mdspan` definition here:

```
void dynamic_mdspan(size_t m, size_t n, vector<double> vec)
{

    std::mdspan md{ vec.data(), std::extents{m, n} };

    . . .

}
```

The `std::extents{m, n}` parameter represents the number of elements in each extent—ie in each row (`m`) and column (`n`)—which are determined dynamically at runtime.

Pretend we have the following data set at runtime:

```
vector<double> w{ 10.1, 10.2, 10.3, 10.4, 10.5, 10.6 };
size_t m{3};
size_t n{2};
```

Using these as inputs to the `dynamic_mdspan(.)` function above will then generate a 3×2 `mdspan` matrix:

```
10.1   10.2
10.3   10.4
10.5   10.6
```

NOTE

The above examples make use of *Class Template Auto Deduction*, commonly referred to as *CTAD*. Both `mdspan` and `extents` are class templates, but because `w` in the example just above is a `vector` of `double` types, and the size of the `vector` being of type `size_t`, when substituted in for `vec` in the `dynamic_mdspan` function, the compiler will deduce that the `mdspan` and `extents` objects are to use `double` and `size_t` as template parameters.

Without CTAD, the function would be written something like:

```
template<typename T, typename S>
void dynamic_mdspan(S m, S n, T vec)
{
      using ext_t = std::extents<S, std::dynamic_extent, std::dynamic_extent>;
      std::mdspan<T, ext_t> mds_dyn{ vec.data(), m, n };

      . . .

}
```

The discussion in this chapter will rely on CTAD, but in cases where more generality is required, it will be necessary to to write out the template arguments in full.

One more thing to note is in each of the examples so far, `mdspan` will arrange the data in row-major order by default. Arranging it in column-major order requires defining a `layout_left` policy *mapping*, in this case called `col_major`, as shown here:

```
std::layout_left::mapping col_major{ std::extents{ m, n } };
```

The corresponding column-major matrix can then be defined by substituting this mapping in the extents argument in the `mdspan` constructor.

```
std::mdspan md{ v.data(), col_major };
```

The result is the column-major version of the matrix:

```
10.1   10.4
10.2   10.5
10.3   10.6
```

NOTE

Row-major order can also be set explicitly with the `std::layout_right` mapping.

The `mdspan` proposal also includes a "slicing" function called `submdspan(.)` to return a reference to an individual row or column from a matrix represented by an mdspan. More generally, this would extend to subsets of higher-dimensional arrays.

Returning to the row-major integer 3×2 `mds1` example, if we wanted to extract a reference to the first row (index 0), it could be obtained as follows, with the index in the first extent (row) argument of `submdspan`:

```
auto row_1 = std::submdspan(mds1, 0, std::full_extent)
```

This would give us:

```
row_1[0] = 101  row_1[1] = 102
```

Rows 2 and 3 could also be referenced by replacing the `0` with `1` and `2`, respectively.

By explicitly setting the second (column) extent argument to the column size less 1, we could also access the last column:

```
auto col_last = std::submdspan(mds1, std::full_extent, mds1.extent(1)-1);
```

This would then be comprised of:

```
col_2[0] = 102  col_2[1] = 104  col_2[2] = 106
```

As `submdspan` is a reference (view) to an `mdspan` containing a row or column, it will obviate generating an additional `mdspan`, in contrast to the issue with a `slice_array` taken from a `valarray`. Any public member function on `mdspan` can be applied to a `submdspan`. On the downside, this would not include the vectorized mathematical operators or functions that are provided with `valarray`, although a different approach to the operators will be provided in `P1673`, to be discussed in the next section.

On the flip side, also because it is a reference, modifying an element of a `submdspan` will also modify the underlying `mdspan` object. Suppose the last element in `col_last` is reset:

```
col_2[2] = 3333;
```

The original `mds1` would then become:

```
101   102
103   104
105   3333
```

One final remark is a proposal for a multi-dimensional array (P1684), called `mdarray`, is also in review. As noted in the proposal **{31}**, "`mdarray` is as similar as possible to mdspan, except with container semantics instead of reference semantics". In other words, an `mdarray` object "owns" its data—similar to a `vector`—as opposed to existing as a reference to data "owned" by another container as in the case of `mdspan`. The earliest it would be released is also C++26.

NOTE

The code examples for `mdspan` above can be compiled in C++20 using the working code currently available on the P1673 GitHub site **{31}**. Installation and building instructions are included with the repository, but two particular items to know are first, the code is currently under the namespace `std::experimental`, and second, as the square bracket operator for multiple indices is set for C++23, you can replace it with the round bracket operator for C++20 and earlier; viz,

```
namespace stdex = std::experimental;
auto mds1 = stdex::mdspan{ v.data(), 3, 2 };

// Replace the square brackets here:
for (size_t i = 0; i < n_rows; ++i)
{
      for (size_t j = 0; j < n_cols; ++j)
            cout << mds1[i, j]              << "\t";
}

// with round brackets:
for (size_t i = 0; i < n_rows; ++i)
{
      for (size_t j = 0; j < n_cols; ++j)
            cout << mds1(i, j) << "\t";
}
```

BLAS Interface (P1673)

This proposal is for "a C++ Standard Library dense linear algebra interface based on the dense Basic Linear Algebra Subroutines (BLAS)" {op cit 29}, also simply referred to as "stdBLAS". BLAS libraries date back a number of decades and were originally written in Fortran, but it evolved into a standard in the

early 2000's, with implementations in other languages such as C (OpenBLAS) and CUDA C++ (NVIDIA) now available, as well as C bindings to Fortran.

Fortran BLAS distributions support four numerical types : `FLOAT` , `DOUBLE` , `COMPLEX` , and `DOUBLE COMPLEX` . The C++ equivalents are `float` , `double` , `std::complex<float>` , and `std::complex<double>` . BLAS libraries contain several matrix formats (standard, symmetric, upper/lower triangular), matrix and vector operations such as element-by-element addition and matrix/vector multiplication.

With implementation of this proposal, it would be possible to apply the same C++ code base to any compatible library containing BLAS functionality, provided an interface has been made available, presumably by the library vendor. This will allow for portable code, independent of the underlying library being used. It remains to be seen at this stage which vendors will eventually come on board, but one major development is NVIDIA's implementation both of `mdspan` and stdBLAS, now available in their HPC SDK **{33}**.

It should be noted stdBLAS itself would only provide access to a particular subset of matrix operations—to be discussed next—even if the underlying library provides additional features such as matrix decompositions, linear and least squares solvers, etc.

BLAS functions are preceded by the type contained in the matrix and/or vector to which they are applied. For example, the function for multiplication of a matrix by a vector is of the form

```
xGEMV(.)
```

where the `x` can be `S` , `D` , `C` , or `Z` , meaning single precision (`REAL` in Fortran), double precision (`DOUBLE`), complex (`COMPLEX`), and double precision complex (`DOUBLE COMPLEX`) respectively.

The C++ equivalent in the proposal, `matrix_vector_product` would instead take in `mdspan` objects representing a matrix and a vector. For example, we can look at a case involving `double` values, using `m` and

`n` for the number of rows and columns as before.

```
std::vector<double> A_vec(m * n);
std::vector<double> x_vec(n);

// A_vec and x_vec are then populated with data...

std::vector<double> y_vec(n);          // empty vector

std::mdspan A{ A_vec.data(), std::extents{m, n} };
std::mdspan x{ x_vec.data(), std::extents{n} };
std::mdspan y{ y_vec.data(), std::extents{m} };
```

Then, performing the multiplication, the vector product is stored in `y`:

```
std::linalg::matrix_vector_product(A, x, y);      // y = A * x
```

The following table provides a subset of BLAS functions proposed in P1673 that should be useful in financial programming. The BLAS functions are assumed to be double precision, and any given matrix/vector expressions can be assumed to be of appropriate dimensions.

Table 7-1. Selected BLAS Functions in Proposal P1673

BLAS Function	P1673 Function	Description
DSCAL	`scale`	Scalar multiplication of a vector
DCOPY	`copy`	Copy a vector into another
DAXPY	`add`	Calculates $\alpha\mathbf{x} + \mathbf{y}$, vectors $\mathbf{x}$ & $\mathbf{y}$, scalar α
DDOT	`dot`	Dot product of two vectors
DNRM2	`vector_norm2`	Euclidean norm of a vector
DGEMV	`matrix_vector_product`	Calculates $\alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$, matrix $\mathbf{A}$, vector $\mathbf{y}$, scalars α & β
DSYMV	`symmetric_matrix_vector_product`	Same as DGEMV (`matrix_vector_product`) but where $\mathbf{A}$ is symmetric
DGEMM	`matrix_product`	Calculates $\alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$, for matrices $\mathbf{A}$, $\mathbf{B}$, & $\mathbf{C}$, and scalars α & β

Linear Algebra (P1385)

The authors of this proposal specifically recognized the importance of linear algebra in financial modeling, along with other applications such as medical imaging, machine learning, and high performance computing. {op cit 28}.

Initial technical requirements for a linear algebra library in C++ were outlined in a preceding proposal (P1166) **{34}**, directly quoted here (in italics):

The set of types and functions should be the minimal set required to perform functions in finite dimensional spaces. This includes:

- *A matrix template*
- *Binary operations for addition, subtraction and multiplication of matrices*
- *Binary operations for scalar multiplication and division of matrices*

Building on this, the P1385 proposal states two primary goals, namely that the library should be easy to use, with "run-time computational performance that is close to what {users} could obtain with an equivalent sequence of function calls to a more "traditional" linear algebra library, such as LAPACK, Blaze, Eigen, etc." {op cit 30}

At a high level, a matrix is generically represented by a *MathObj* type, and an *engine* "is an implementation type that manages the resources associated with a *MathObj* instance." Discussions are ongoing over the details, but "a *MathObj* might own the memory in which it stores its elements, or it might employ some non-owning view type, like `mdspan`, to manipulate elements owned by some other object". {ibid}. An *engine* object is proposed to be a private member on a *MathObj*, and a *MathObj* may have either fixed or dynamic dimensions.

More details should emerge over the next few years on the form the P1385 linear albegra library will assume, but for now, this hopefully provides an initial high-level glimpse of something that should be a very welcome addition to C++ for financial software developers.

Summary (Linear Algebra Proposals)

Recalling the results we saw with `valarray`, with the above proposals in place, some of the same convenient functionality should be in place by C++26, this time with rigorous, efficient, and consistent specifications that should avoid the problems that plagued `valarray`. While `mdspan` differs from `valarray` as a

non-owning reference, it will still allow array storage, such as a `vector`, to be adapted to a matrix proxy by specifiying the number of rows and columns. `submdspan` will assume a similar role as a `valarray` slice, but without the performance penalty due to object copy. P1673 will provide a common interface to libraries containing BLAS functions, and function naming, as as with `matrix_vector_product`, will be more expressive than their cryptic Fortran equivalents such as `DGEMV(.)`. And, the `+`, `-`, and `*` operators in P1385 will provide the ability to to implement linear algebra expressions in a natural mathematical format similar to the results we saw with `valarray`.

This is an exciting development that will finally provide efficient and reliable methods for implementing basic matrix calculations that are long overdue for C++. Hopefully we will also eventually see implementations of P1673 BLAS interfaces for popular open source libraries such as Eigen and others mentioned above, but as of the time of this writing, this remains to be seen.

Chapter Summary

This chapter has examined the past, present, and expected future of two-dimensional array management and linear algebra in C++. `valarray`, dating back to C++98, offered matrix-like capabilities for which quantitative developers certainly would have found ample use cases. It is unforunate it never received the attention and support of the Committee and community, even as C++ was being touted in the late 1990's as the language of the future for computational finance. For specific compilers, it might remain a viable option, but given the inconsistency of its implementations across Standard Library vendors, it can limit code reuse across different platforms.

In the late 2000's, high quality open-source linear algebra libraries such as Eigen and Armadillo came on the scene and were well-received by the financial quant C++ programming community. In the present day, these libraries contain not only much of the same functionality found in the BLAS standard, but also a plethora of matrix decompositions that are frequently used in financial applications.

Finally, the future also looks brighter for ISO C++ with the three proposals— `mdspan` (P0009), BLAS interface (P1673), and linear algebra library (P1385)—slated for inclusion in the Standard Library within the next three years. These are arguably features that are long overdue, but they will provide a big step in satisfying demand not just from the financial industry, but also other computationally intensive domains.

References

US Daily Treasury Par Yield Curve Rates *https://home.treasury.gov/resource-center/data-chart-center/interest-rates/TextView?type=daily_treasury_yield_curve&field_tdr_date_value_month=202211* _

Supplemental Chapter, _The C++ Standard Library, 2E

{2.5} Stroustrup, *A Tour of C++* (2E, but now in 3E)

Quantstart article on Eigen *https://www.quantstart.com/articles/Eigen-Library-for-Matrix-Algebra-in-C/*

Gottschling _Discovering Modern C++

Bowie Owens, CppCon 2019, *https://www.youtube.com/watch?v=4IUCBx5fIv0*

More information on the Jacobi and Bidiagonal Divide and Conquer SVD classes:

Two-sided Jacobi SVD decomposition of a rectangular matrix

Bidiagonal Divide and Conquer SVD

asciidoctor-latex -b html Ch10_v10_Split_LinearAlgebraOnly.adoc

{5} Rcpp Packages:

RcppEigen

RcppArmadillo

RcppBlaze3

Boost Headers, including uBLAS