Linear Algebra with The Eigen C++ Library

DANIEL HANSON

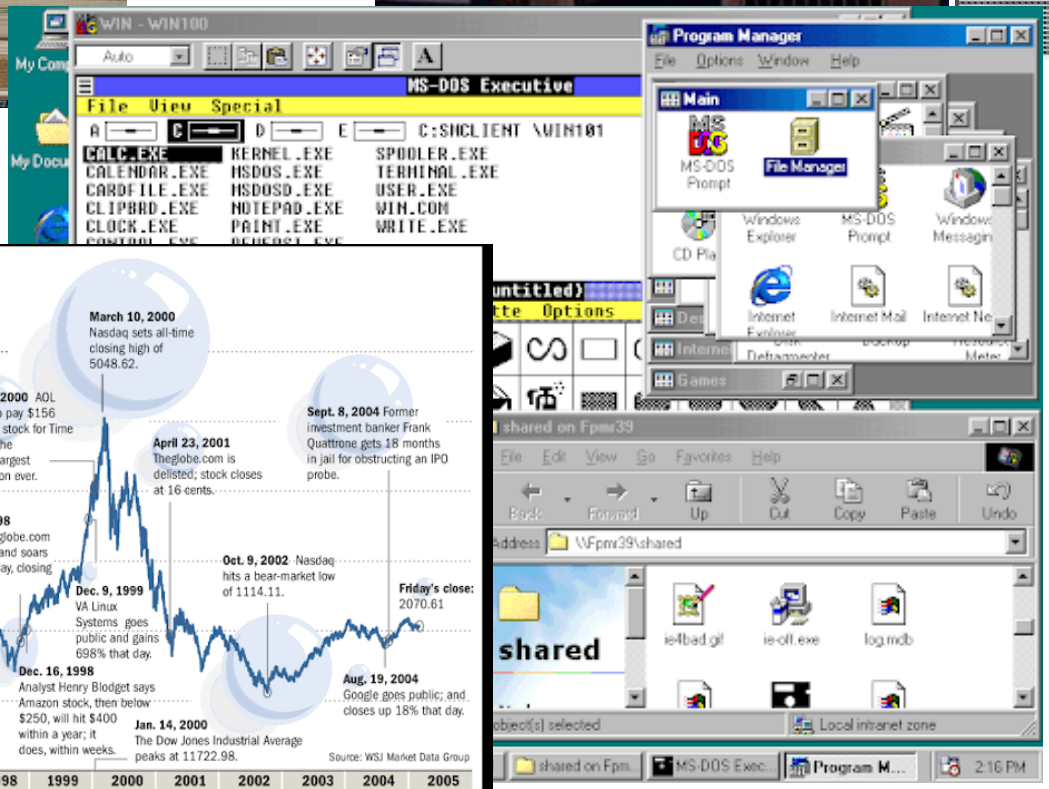
Outline

- A short history – linear algebra and C++ (1998 – Present)
- The Eigen C++ Template Library for Linear Algebra
- Linear Algebra Interface in C++26
  - Basics
  - Using with Eigen

Disclaimers/Caveats

- This presentation is on solving problems *using*
  - The Eigen linear algebra library
  - stdBLAS in C++26
- Not affiliated with Eigen but have used it in financial programming and teaching
- Goal is to share some topics you might find interesting (and hopefully useful!)

Take a Little Trip Back to 1998



A Little History

- Circa 1998:
  - C++ growing in popularity
  - Wide adoption in financial programming
  - But, no support for linear algebra (pining for Fortran…)
- Your options essentially were:
  - Write your own Matrix class and operations
  - Convince your boss to buy a commercial library

A Little History: Open-Source Libraries

- Boost uBLAS (BLAS: "Basic Linear Algebra Subroutines")
  - November 2002, release 1.29.0
  - BLAS functionality
    - Matrix and vector representations
    - Matrix addition and subtraction
    - Matrix multiplication
      - Matrix × Matrix
      - Matrix × Vector
      - Vector dot products
      - Scalar multiplication
    - Norms
    - Still on your own for linear solvers and matrix decompositions
  - Good (but not outstanding) performance
  - Last major improvement was in 2008 (See uBLAS FAQ's)

A Little History: Open-Source Libraries

- "Next Generation" linear algebra libraries (not exhaustive):
  - Eigen (2006)
  - Armadillo (2009)
  - Blaze (2012)
  - Expression template-based libraries
  - Include the usual BLAS functionality plus decompositions and solvers
- For more information on Blaze
  - CppCon 2016 talk by Klaus Iglberger (library author)
  - Performance tests included comparisons with Eigen

A Little History: C++23 and C++26

- **`mdspan`** (P0009), C++23
  - *A mutating multidimensional array reference (view)…*
  - *…of a container whose elements reside in contiguous memory*
    - **`std::vector`**
    - **`Eigen::VectorXd`**
    - **`std::mdarray`** (p1684) (C++26)

- stdBLAS (P1673), C++26
  - A free function linear algebra *interface* based on the BLAS
  - Matrix represented by a 2-D **`mdspan`**
  - Default BLAS with major compilers/library vendors

The Eigen Library

$$\sigma_p = \sqrt{\omega^\mathsf{T} \Sigma \omega}$$

$$\hat{\beta} = \begin{bmatrix} \hat{\beta}_1 \\ \hat{\beta}_2 \\ \vdots \\ \hat{\beta}_n \end{bmatrix} \qquad y = X\hat{\beta}$$

$$\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}$$

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

$$w_t = Lz_t^\mathsf{T}$$

```
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}
};
```

Eigen

- *A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms*
  - v 01 released December 2006
  - Latest release: v 3.4.0 (August 2021)
  - Mozilla Public License (MPL) 2.0
  - Header only
  - Uses expression templates and lazy evaluation

- Popular within various domains:
  - Finance (Armadillo is also popular)
  - Medical/Pharmaceutical research
    - Stan Math Library (Differentiable C++ for linear algebra and probability)
    - Cancer modeling research (talk by Ruibo Zhang, CppCon 2024)
  - Data Science/Machine Learning (TensorFlow Library)
  - Experimental Physics
    - ATLAS Experiment tracking software
    - CERN Large Hadron Collider

Eigen

- Dense and Sparse matrix representations
- Core class for dense operations: **`Eigen::Matrix`**
- Part of a class template hierarchy:

```
EigenBase<Matrix>
  <-- DenseCoeffsBase<Matrix>
    <-- DenseBase<Matrix>
      <-- MatrixBase<Matrix>
        <-- PlainObjectBase<Matrix>
          <-- Matrix
```

- Most member functions are defined on **`Eigen::MatrixBase`**

- **`Eigen::Vector`** class: $m \times 1$ column vector

- The **`Eigen::Matrix`** class supports the (usual) numerical types:
  - **`int`**
  - **`float`**
  - **`double`**
  - **`std::complex<double>`**

- Various aliases of the **`Matrix`** class
  - Fixed square dimensions (up to 4)
    - **`Eigen::Matrix4i`**: fixed 4 x 4 matrix of **`int`** types
    - **`Eigen::Matrix3f`**: fixed 3 x 3 matrix of **`float`** types
  - Dynamic dimensions (m x n)
    - **`Eigen::MatrixXd`**: dynamic $m \times n$ matrix of **`double`** types
    - **`Eigen::VectorXd`**: dynamic $m \times 1$ vector of **`double`** types
    - **`Eigen::RowVectorXd`**: dynamic $1 \times m$ vector of **`double`** types
    - **`Eigen::MatrixXcd`**: dynamic $m \times n$ matrix of **`complex<double>`** types

Eigen MatrixXd Examples

```
#include <Eigen/Dense>
using Eigen::MatrixXd;

MatrixXd mtx
{
    {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 mtx{4, 3};        // 4 rows, 3 columns
mtx << 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 mtx{2, 2};
// 0-index as is the case in C++ generally:
mtx(0, 0) = 3.0;
mtx(1, 0) = 2.5;
mtx(0, 1) = -1.0;
mtx(1, 1) = mtx3(1, 0) + mtx3(0, 1);

// Similar for Vector types
```

Note: Although data is entered in row-major order, it is stored in column-major order

More info (documentation):
https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html

Eigen: Matrix Multiplication

- The ∗ operator conveniently implements the *matrix product*
- *Not* element-by-element multiplication

```
using Eigen::MatrixXd;
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;
cout << "A (5 x 3) * B (3 x 5) = \n\n" << prod_ab << "\n";
```

```
A (5 x 3) * B (3 x 5) =

   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
```

Eigen: Scalar Multiplication

- The $*$ operator is also overloaded for scalar multiplication:

```
MatrixXd scale_a = 0.5 * A;
```

```
scale_a =
 0.5    1  1.5
0.75 1.25 1.75
   2  2.5    3
2.25 2.75 3.25
 3.5    4  4.5
```

- Also works with multiplication assignment:

```
B *= 2.0;
```

```
B =
 2  4  6  8 10
 3  5  7  9 11
10 12 14 16 17
```

- Applies to **`Eigen::Vector`** types (e.g. **`VectorXd`**) as well

- Matrix and vector addition/subtraction are similarly defined by $+$ and $-$ operators

Eigen: Matrix and Vector addition and subtraction

```
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 C
{
    {10.0, 20.0, 30.0},
    {10.5, 20.5, 30.5},
    {40.0, 50.0, 60.0},
    {40.5, 50.5, 60.5},
    {70.0, 80.0, 90.0}
};
```

```
MatrixXd mtx_sum = A + C;
cout << mtx_sum << "\n";
```

```
11 22 33
12 23 34
44 55 66
45 56 67
77 88 99
```

```
VectorXd vec_diff = u - v;        // u = [1.0  2.0  3.0]^T
                                  // v = [0.5 -0.5  1.0]^T

cout << vec_diff<< "\n";          //   = [0.5  2.5  2.0]^T
```

Evaluating Matrix Expressions

- Suppose we have matrices $\mathbf{A}_{n\times n}, \mathbf{B}_{n\times n}, \mathbf{C}_{n\times n}$
- And n-dimensional vectors $\mathbf{u}$ and $\mathbf{v}$
- It is thus easy to take linear algebra expressions such as

$$\mathbf{w} = \mathbf{A}\mathbf{u} + \mathbf{B}\mathbf{v}$$

$$\mathbf{M} = \mathbf{A}\mathbf{B} + (\mathbf{u}\mathbf{v})\mathbf{C}$$

and map them into C++ using Eigen in the same mathematical syntax:

```
MatrixXd A, B, C...
VectorXd u, v...

VectorXd w = A * u + B * v;
MatrixXd M = A * B + u.dot(v) * C;    // u.dot(v): dot product uv
```

Eigen and STL Compatibility

- One can also iterate through an Eigen **`Vector`** container and apply STL algorithms
- As an example:
  - Populate a **`VectorXd`** with random numbers drawn from a t-distribution using **`<random>`** and the **`std::generate`** algorithm
  - And then apply **`std::max_element`** to find the maximum random value in the result

```
#include <random>
#include <algorithm>
// . . .

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

Eigen and STL Compatibility

- **`Eigen::Vector`**(s) can also be used together with STL containers in standard algos
- **Example**: Dot (inner) product of **`Eigen::VectorXd`** and **`std::vector`**

```
#include <numeric>
// . . .

// Recall: VectorXd u{12}: Contains random t-dist variates from previous slide
std::vector<double> v(u.size());    // u is a VectorXd, v is an STL vector
std::generate(std::begin(v), std::end(v), [&mt, &tdist]() {return tdist(mt);});

// Inner product of Eigen::VectorXd and std::vector
double dot_prod = std::inner_product(u.begin(), u.end(), v.begin(), 0.0);
```

Eigen Unary Expressions

- Similar to **`std::transform(.)`**

- Can be applied to an entire **`Matrix`**

```
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}
};

vals = vals.unaryExpr([](double x) {return x * x;});
```

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

Eigen Decompositions

- A list and description of matrix decompositions (and solvers) in Eigen is available in the documentation

- We will look at two examples:

  - QR (**HouseholderQR**):
    - $\mathbf{A}_{n\times p} = \mathbf{Q}_{n\times p}\mathbf{R}_{p\times p}, \quad n \geq p,$ Rank($\mathbf{A}$) = p
    - $\mathbf{Q}\mathbf{Q}^T = \mathbf{I}, \quad \mathbf{R}$ upper triangular

  - Cholesky (**LLT**) :
    - $\mathbf{A}_{n\times n} = \mathbf{L}\mathbf{L}^T$
    - $\mathbf{A}$ positive definite, $\mathbf{L}$ is lower triangular

- Also in Eigen (among others):
  - LU: Lower-Upper decomposition (**PartialPivotLU**)
  - Singular Value Decomposition (**JacobiSVD**, **BDCSVD**)

Eigen Decompositions and Linear Regression

- Example: Fund portfolio with return y
- Three index returns to predict fund performance x_1, x_2, x_3
- n months of data (observations)
- Seek least squares estimates $\widehat{\beta_0}$, $\widehat{\beta_1}$, $\widehat{\beta_2}$, $\widehat{\beta_3}$:

$$\hat{y} = \widehat{\beta_0} + \widehat{\beta_1} x_1 + \widehat{\beta_2} x_2 + \widehat{\beta_3} x_3$$

- Number of observations $n \gg p =$ number of index predictors $= 3$
- Index returns (predictor variables) not highly correlated
- Can use the Householder QR Decomposition in Eigen

Eigen Decompositions and Linear Regression

- As an example, suppose we have data over 30 months, with the vector $\mathbf{Y}$ containing the monthly target fund returns

$$\mathbf{Y} = \begin{bmatrix} -0.039891 \\ 0.001787 \\ \vdots \\ 0.011249 \end{bmatrix}$$

- The matrix $\mathbf{X}$ will contain monthly returns for each index in separate columns (plus the intercept term column at left):

$$\mathbf{X} = \begin{bmatrix} 1 & 0.04470 & -0.01900 & -0.03063 \\ 1 & -0.00789 & 0.02604 & 0.02419 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0.00144 & 0.52195 & -0.00430 \end{bmatrix}$$

- In matrix form the regression equation is $\widehat{\mathbf{Y}} = \mathbf{X}\widehat{\boldsymbol{\beta}}$
- Apply QR Decomposition on $\mathbf{X}$
  - $\mathbf{X} = \mathbf{QR}$ => $\mathbf{Q}^T\mathbf{Y} = \mathbf{R}\widehat{\boldsymbol{\beta}}$
  - Triangular system ($\mathbf{R}$ is upper-triangular), easier to solve numerically

Eigen Decompositions and Linear Regression

- Seek best fit estimates $\widehat{\beta_0}, \widehat{\beta_1}, \widehat{\beta_2}, \widehat{\beta_3}$: $\hat{y} = \widehat{\beta_0} + \widehat{\beta_1}x_1 + \widehat{\beta_2}x_2 + \widehat{\beta_3}x_3$

```
// 3 predictor funds, 30 months of data, plus intercept column:
MatrixXd X{30, 4};
X.col(0) = VectorXd::Ones(30);
X.col(1) = VectorXd {{-0.044700, -0.007888, . . .,  0.001440}};
X.col(2) = VectorXd {{-0.019003,  0.026037, . . .,  0.052195}};
X.col(3) = VectorXd {{-0.030629,  0.024919, . . ., -0.004396}};

VectorXd Y{30};    // 30 observations
Y << -0.039891, 0.001788, . . ., 0.011249;
```

1	-0.0447004	-0.0190027	-0.0306291
1	-0.00788839	0.0260368	0.024919
1	0.0429801	0.0378271	-0.0017158
1	0.0164166	0.0106293	0.00856161
1	-0.0177966	-0.00838227	0.00340693
1	-0.0167141	0.0011217	-0.0108239
1	0.019472	-0.00449441	-0.0103611
1	0.0298533	0.0173045	-0.00930243
1	0.0231261	-0.00610629	0.00814201
1	-0.0338791	0.0121746	-0.00406421
1	-0.00338369	-0.00330503	0.000584335
1	-0.0184745	0.0272197	0.00464029
1	-0.0125098	-0.0360893	0.0318933
1	-0.0183481	-0.00222959	-0.0135443
1	0.0106268	-0.0157485	-0.0235736
1	0.0366694	-0.0206192	-0.00466509
1	0.0108111	-0.0116414	-0.00644626
1	-0.0355717	0.0231488	-0.00531141
1	0.027474	-0.00229073	0.0450963
1	0.00540607	0.00628809	-0.0073747
1	-0.0101594	-0.0120384	-0.00514201
1	-0.00614563	-0.0292587	-0.0017158
1	-0.0103273	0.0112193	-0.00517636
1	-0.0104352	-0.00884699	-0.00288499
1	0.0111272	-0.033738	0.00230936
1	-0.0237937	0.0206191	-0.0145216
1	-0.0280094	-0.0120777	-0.0177117
1	0.00218235	0.0156729	0.00119209
1	0.00868315	0.0410129	-0.00238233
1	0.00144003	0.0521953	-0.00439592

- Then, apply the Householder QR decomposition:

```
VectorXd beta = X.householderQr().solve(Y);
```

```
(Beta estimates:)
 0.000504336
 0.353685
-0.0926719
 0.394509
```

$$\hat{y} = 0.0005 + 0.3537x_1 - 0.0927x_2 + 0.3945x_3$$

2024: stdBLAS and the Eigen Library

ChatGPT 3.5

COMPILER EXPLORER Add... More Templates

C++ source #1

Save/Load + Add new... Vim CppInsights Quick-bench

```
184
185 using std::vector, std::size_t, std::cout, std::format;
186 namespace stdex = std::experimental;
187
188 cout << "***\ncol_major_mtx_vec_mult() col major overload\n\n ***";
189 std::vector col_maj_data{ 10.1, 10.3, 10.5, 10.2, 10.4, 10.6 };
190 std::vector col_vec_data{ 1.0, 1.0 };
191 std::vector prod_result_init{ 0.0, 0.0, 0.0 };  // For the resulting product
192
193 // Create mdspan objects
194 stdex::layout_left::mapping col_major{ stdex::extents{ 3, 2 } };
195 stdex::mdspan col_maj{ col_maj_data.data(), col_major };
196 cout << "Col-major matrix:\n";
197 print_dynamic_mdspan(col_maj);
198
199 stdex::mdspan col_vec{ col_vec_data.data(), stdex::extents{ 2 } };
200 cout << "Column vector to be multiplied:\n";
201 print_dynamic_mdspan(col_vec);
202
203 stdex::mdspan prod_result{prod_result_init.data(), stdex::extents{ 3 }};
204 stdex::linalg::matrix_vector_product(col_maj, col_vec, prod_result);
205 cout << "Column vector result:\n";
206 print_dynamic_mdspan(prod_result);
207
208 std::transform(col_maj_data.begin(), col_maj_data.end(), col_maj_data.begin(), [](double x)
209 stdex::linalg::matrix_vector_product(col_maj, col_vec, prod_result);
210 cout << "Column vector result after modifying original data vector:\n";
211 print_dynamic_mdspan(prod_result);
212
213
214
```

```
    for (int i = 0; i < eigenValues.size(); ++i) {
        eigenPairs.push_back({eigenvalues(i), eigenvectors.col(i)});
    }
    std::sort(eigenPairs.begin(), eigenPairs.end(), [](auto &left, auto &right) {
        return left.first > right.first;
    });

    // Display the results
    cout << "Eigenvalues:\n" << eigenvalues << "\n\n";
    cout << "Eigenvectors:\n" << eigenvectors << "\n\n";
    cout << "Principal Components:\n";
    for (int i = 0; i < eigenPairs.size(); ++i) {
        cout << "PC" << i + 1 << ":\n" << eigenPairs[i].second << "\n\n";
    }

    return 0;
}
```

This program generates a random dataset, computes the mean, centers the data, calculates the covariance matrix, performs Eigen decomposition, and then displays the eigenvalues, eigenvectors, and principal components. Make sure you have the Eigen library installed and properly configured in your C++ project. You can find Eigen at: https://eigen.tuxfamily.org/

StdBLAS

- P1673: *A C++ Standard Library dense linear algebra interface based on the dense Basic Linear Algebra Subroutines (BLAS)*
- **`std::linalg`** namespace in code

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

Eigen and StdBLAS

- Portfolio management:
  - A covariance matrix $\boldsymbol{\Sigma}$ of asset returns in a portfolio
  - A vector of portfolio weights $\boldsymbol{\omega}$
- Common problems in quant finance:
  - Calculate the portfolio variance $\boldsymbol{\omega}^T \boldsymbol{\Sigma} \boldsymbol{\omega}$
  - Calculate the Cholesky decomposition of $\boldsymbol{\Sigma}$
- Portfolio variance: can use functions in the BLAS
- Use Eigen to compute the decomposition (not in BLAS)

Eigen and StdBLAS: Portfolio Variance

Compute right hand side first: $\boldsymbol{\omega}^T(\boldsymbol{\Sigma}\boldsymbol{\omega})$

```
using std::vector;
namespace stdex = std::experimental;
// Data originates somewhere in contiguous storage (ex: std::vector)
vector cov_mtx_data         // Cov Matrix (Sigma) – Use CTAD
{
    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
};

vector omega{0.25, -0.25, 0.50, 0.50};      // Vector of asset weights (omega)

long n = omega.size();              // Use long types for Eigen (later)

vector<double> inner_mtx_vec(n);    // Intermediate storage of Sigma * omega (RHS)
```

Eigen and StdBLAS: Portfolio Variance

- Recall: calculate as $\boldsymbol{\omega}^T(\boldsymbol{\Sigma}\boldsymbol{\omega})$:

```
// mdspan views of data (can use CTAD):
stdex::mdspan md_cov_mtx{cov_mtx_data.data(), stdex::extents{n, n}};
stdex::mdspan md_wgts{omega.data(), stdex::extents{n}};
stdex::mdspan md_inner_mtx_vec{inner_mtx_vec.data(), stdex::extents{n}};

// Compute the quadratic form using stdBLAS:
// 1) RHS product 1st: Sigma * omega:
stdex::linalg::matrix_vector_product(md_cov_mtx, md_wgts, md_inner_mtx_vec);
// 2) Now apply LHS product: omega^T * (Sigma * omega)
double port_var = stdex::linalg::dot(md_wgts, md_inner_mtx_vec);  // 0.02038
```

Next task: Cholesky Decomposition

- Covariance matrix $\boldsymbol{\Sigma}$, symmetric, positive definite
- Then, there is a lower triangular matrix $\mathbf{L}$ such that $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$ (Cholesky)
- This gives us the "square root" of the covariance matrix $\mathbf{L} = \text{“ } \boldsymbol{\Sigma}^{\frac{1}{2}} \text{ ”}$
- Decompositions not in stdBLAS => use Eigen
- An **`Eigen::Map`** is a view
  - Of the matrix data referred to by **`md_cov_mtx`** (**`mdspan`**): view of a view
  - And can be used in place of a full **`Matrix`** object in Eigen

Eigen and StdBLAS: Cholesky Decomposition

```
using Eigen::MatrixXd;

// An Eigen::Map is also a view but can be used like a MatrixXd:
Eigen::Map<MatrixXd> cov_mtx_map{&md_cov_mtx(0, 0), n, n};

// Create an Eigen::LLT (Cholesky Decomposition) object:
Eigen::LLT<Eigen::MatrixXd> cholesky{cov_mtx_map};

// Member function matrixL() returns the Cholesky L matrix:
MatrixXd cholesky_mtx = cholesky.matrixL();

// Try storing results as a view in a new mdspan (proceed naively. . .):
stdex::mdspan md_cholesky_mtx{cholesky_mtx.data(), stdex::extents{n, n}};
```

```
0.11238     0.00222    -0.00151     0.04476
0.00000     0.03708     0.07560     0.00460
0.00000     0.00000     0.17898     0.02526
0.00000     0.00000     0.00000     0.16229
```

- Result is upper triangular, not lower triangular

Cholesky Decomposition:

```
// Use a column-major (layout_left) mapping for the mdspan:

stdex::layout_left::mapping col_major{stdex::extents{n, n}};
stdex::mdspan md_chol_mtx{chol_mtx.data(), col_major};
```

```
 0.11238     0.00000     0.00000     0.00000
 0.00222     0.03708     0.00000     0.00000
-0.00151     0.07560     0.17898     0.00000
 0.04476     0.00460     0.02526     0.16229
```

Summary

- Linear algebra options originally very limited for C++98

- Robust open-source libraries mid/late 2000's~
  - Eigen
  - Armadillo
  - Blaze

- Eigen
  - Header only/expression templates
  - Basic Linear algebra, linear solvers, matrix decomposition
  - STL compliant – can use STL algorithms
  - Unary expression function **`unaryExpr(.)`**, and other features

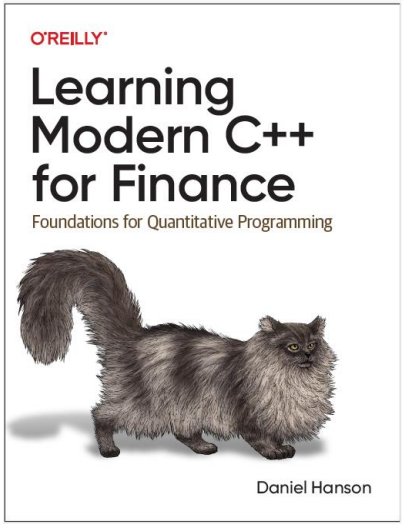
- BLAS interface proposal 1673 accepted for C++26 Standard Library
  - Matrix and vector addition, subtraction, multiplication, norms, etc.
  - Can use **`Eigen::Map`** as view of BLAS matrix data

References and Sample Code

- Eigen documentation: https://eigen.tuxfamily.org/
- stdBLAS WG21 Proposal P1673: https://wg21.link/p1673
- Reference implementation of P1673: https://github.com/kokkos/stdBLAS
- Mark Hoemann: "**`std::linalg`**: Linear Algebra Coming to Standard C++", CppCon 2023
  - Video: https://www.youtube.com/watch?v=-UXHMlAMXNk
  - Slides: https://github.com/CppCon/CppCon2023/blob/main/Presentations/stdlinalg_linear_algebra_coming_to_standard_cpp.pdf
- Golub and Van Loan, *Matrix Computations*, § 5.3.3: LS Solution Via QR Factorization

- Sample Code: https://github.com/QuantDevHacks/CppCon2024LinAlgEigen

- Shameless plug: Hanson, Ch 8,
  Learning Modern C++ for Finance (O'Reilly)
  (publication later this year)



Contact/ Questions

- Contact:
  - `daniel (at) cppcon.org` (Student Program Chair, CppCon)
  - https://www.linkedin.com/in/danielhanson/
- Thank You!

- Questions?