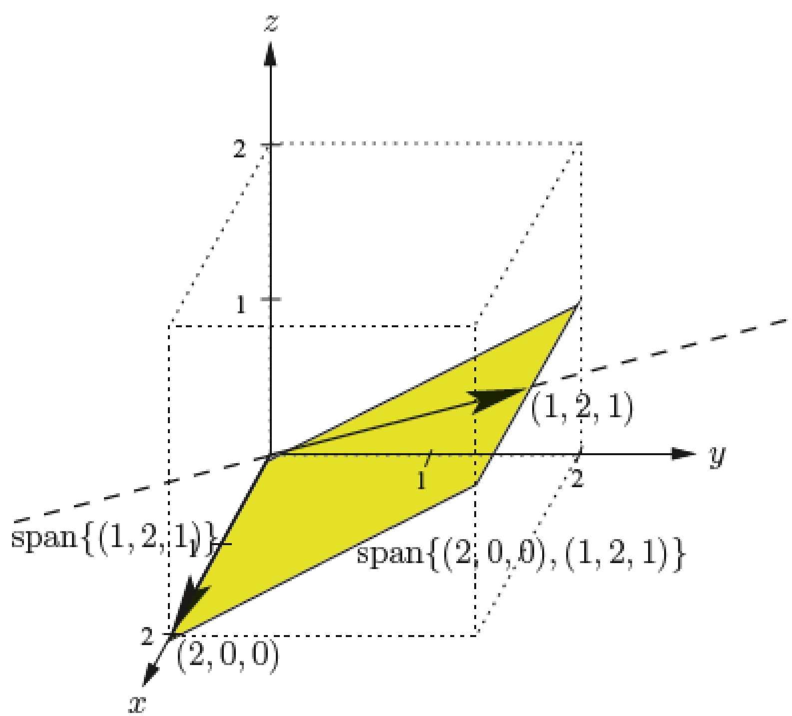
Linear Algebra in C++

The Eigen Library

C++26 Linear Algebra Interface

$$
\begin{aligned}
\left\|\mathbf{x}^{(k)}\right\| &= \left\|c_1\lambda_1^k\mathbf{v}_1 + c_2\lambda_2^k\mathbf{v}_2 + \cdots + c_n\lambda_n^k\mathbf{v}_n\right\| \\
&\leq |\lambda_1|^k \|c_1\mathbf{v}_1\| + |\lambda_2|^k \|c_2\mathbf{v}_2\| + \cdots + |\lambda_n|^k \|c_n\mathbf{v}_n\| \\
&\leq \|c_1\mathbf{v}_1\| + \|c_2\mathbf{v}_2\| + \cdots + \|c_n\mathbf{v}_n\| \,.
\end{aligned}
$$



$$
\begin{aligned}
\theta) &= \left(\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}\right)^T \begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix} \\
&= \begin{bmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{bmatrix} \begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix} \\
&= \begin{bmatrix}\cos^2\theta + \sin^2\theta & \cos\theta\sin\theta - \sin\theta\cos\theta \\ -\cos\theta\sin\theta + \sin\theta\cos\theta & \cos^2\theta + \sin^2\theta\end{bmatrix} = \begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix}
\end{aligned}
$$

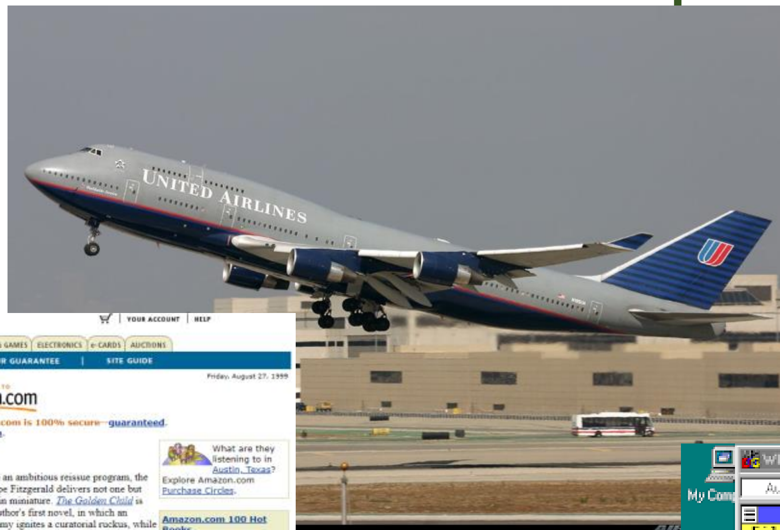
Thomas S Shores, *Applied Linear Algebra and Matrix Analysis*, Springer (2000)

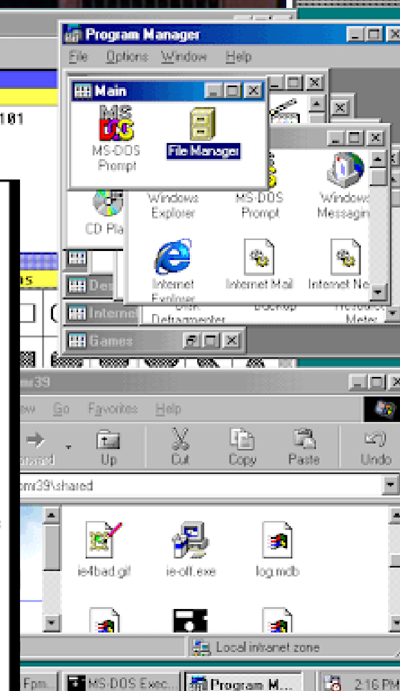
Daniel Hanson, NWCPP 21 March 2024

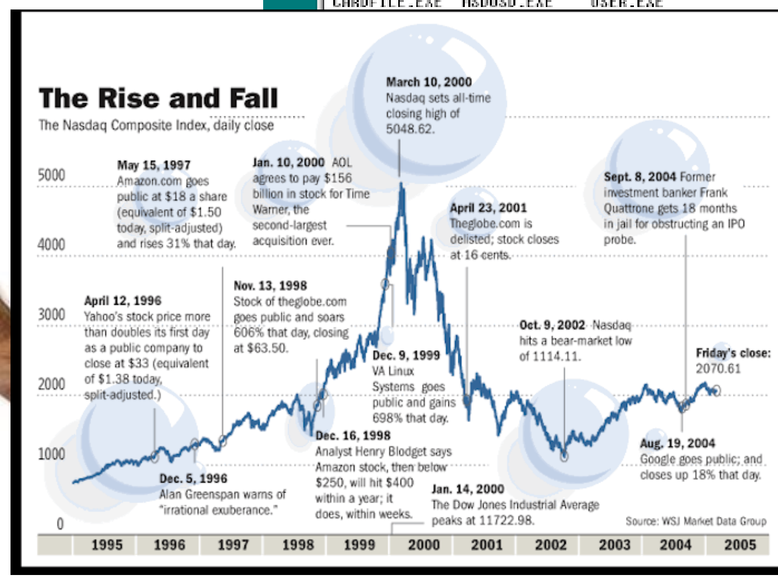
Outline

- Some history
  - Starting with the C++98 State of the World
  - Open Source Linear Algebra Libraries (2000's)
  - **`mdspan`**, C++26 stdBLAS (Basic Linear Algebra Subroutines)
- The Eigen Library
- Using stdBLAS and Eigen matrix decompositions

Take a Little Trip Back to 1998







C++98 Linear Algebra

- Fortran platform support for linear algebra:
  - Multidimensional Arrays
  - BLAS (Basic Linear Algebra Subroutines)
  - LAPACK (Linear Algebra Package)
  - IMSL (International Mathematics and Statistics Library)
  - Vanished with transition to C++
  - Now what?!
- Contrived C++ solutions
  - **`std::vector`** of **`std::vector`**(s)
  - 2-dimensional dynamic C array

C++98 Linear Algebra

- **`std::valarray`**
  - 2-dimension **`valarray`**: proxy for a matrix
  - Element-by-element +, -, *, /
  - Element-by-element **`<cmath>`**-like functions (**`std::sin`**, **`std::log`**, **`std::abs`** etc)
  - **`apply(.)`** member function similar to **`std::transform`**
    - ➢ Row by column dot products
    - ➢ Easy to implement matrix multiplication
  - Controversy and downsides
    - ➢ Inconsistent implementations (expression templates, or not)
    - ➢ **`slice_array`** for row or column – does not support (most) **`valarray`** functionality
- Another option was: Try to convince your boss to invest in a decent commercial linear algebra library

C++ Open Source Linear Algebra Libraries

- Matrix Template Library (MTL):
  - First released in 1998
  - Sparse and dense matrix formats
  - Arithmetic linear algebra operations on vectors and matrices
  - Inner products and norms, etc

- Boost uBLAS
  - November 2002, release 1.29.0
  - Similar functionality as MTL ("BLAS")
  - Good (but not outstanding) performance
  - Last major improvement was in 2008 (See uBLAS FAQ's)

- "Next Generation" linear algebra libraries (not exhaustive):
  - Eigen (2006)
  - Armadillo (2009)
  - Blaze (2012)
  - Include the usual BLAS functionality plus decompositions and solvers

Standard Library Support for Linear Algebra

- **`mdspan`** (p0009), C++23
  - *A polymorphic and mutating multidimensional array reference (view)…*
  - *…of a container whose elements reside in contiguous memory*, ex's:
    - **`std::vector`**
    - **`Eigen::VectorXd`**
    - **`std::mdarray`** (p1684)
- stdBLAS (p1673, **`std::linalg`**), C++26
  - A free function linear algebra *interface* based on the BLAS
  - Matrix represented by a 2-D **`mdspan`**
  - Default BLAS with major compilers

The Eigen Library

$$\sigma_p = \sqrt{\boldsymbol{\omega}^\top \boldsymbol{\Sigma} \boldsymbol{\omega}}$$

$$\hat{\boldsymbol{\beta}} = \begin{bmatrix} \hat{\beta}_1 \\ \hat{\beta}_2 \\ \vdots \\ \hat{\beta}_n \end{bmatrix}$$

$$\boldsymbol{y} = \boldsymbol{X}\hat{\boldsymbol{\beta}}$$

$$\boldsymbol{Q}^\top \boldsymbol{y} = \boldsymbol{R}\boldsymbol{X}$$

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

$$\boldsymbol{\Sigma} = LL^\top$$

$$\boldsymbol{w}_t = \boldsymbol{L}\boldsymbol{z}_t^\top$$

```
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
- Now at v 3.4.0 (August 2021)
- Mozilla Public License (MPL) 2.0
- Decent documentation
- Popular within various domains:
  - Finance
  - Medical/Pharmaceutical research
  - Data Science
  - Experimental Physics
- Included in
  - TensorFlow Library
  - Stan Math Library
  - 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 uses expression templates and lazy evaluation
- **`Eigen::Vector`** class: special case of **`Matrix`**, m x 1 column vector

Eigen

- 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 x n matrix of **`double`** types
    - ➢ **`Eigen::VectorXd`**: dynamic m x 1 matrix of **`double`** types
    - ➢ **`Eigen::MatrixXcd`**: dynamic m x n matrix of **`complex<double>`** types

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}
};
```

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

```
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);
```

VectorXd Examples:

```
using Eigen::VectorXd;

VectorXd vec{ {1.0, 2.0, 3.0, 4.0, 5.0, 6.0} };

VectorXd vec{ 6 };          // 6 elements
vec << 1.0, 2.0, 3.0, 4.0, 5.0, 6.0;

VectorXd vec{ 3 };           // 3 elements
vec(0) = 3.19;
vec(1) = 2.58;
vec(2) = vec(0) + vec(1);

// cout is also for Eigen MatrixXd and VectorXd:
cout << "Contents of the VectorXd vec are:\n" << vec << "\n\n";
```

Eigen: Matrix multiplication

- As an example, a common problem in finance, given a correlation matrix of three assets in a portfolio:

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

  - And a vector of volatilities of individual asset returns (standard deviations):

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

  - …is to compute the covariance matrix **Σ**

$$\boldsymbol{\Sigma} = \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.4 \end{bmatrix} \begin{bmatrix} 1 & 0.5 & 0.25 \\ 0.5 & 1.0 & -0.7 \\ 0.25 & -0.7 & 1.0 \end{bmatrix} \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.4 \end{bmatrix}$$

  - First step is to form the diagonal matrices from the vector of volatilities

Eigen: Matrix multiplication

- Member function **`asDiagonal()`** returns a lighter weight view of a diagonal matrix containing the vector elements

- Compute the covariance matrix

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

$$\boldsymbol{\Sigma} = \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.4 \end{bmatrix} \begin{bmatrix} 1 & 0.5 & 0.25 \\ 0.5 & 1.0 & -0.7 \\ 0.25 & -0.7 & 1.0 \end{bmatrix} \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.4 \end{bmatrix}$$

- Operator * is defined: (3 x 3) * (3 x 3) * (3 x 3) = 3 x 3 covariance matrix

$$\begin{bmatrix} 0.4 & 0.01 & 0.02 \\ 0.01 & 0.01 & -0.028 \\ 0.02 & -0.028 & 0.16 \end{bmatrix}$$

Eigen Vector by vector multiplication

```
// Define two Vectors u and v
VectorXd u{ {1.0, 2.0, 3.0} };
VectorXd v{ {0.5, -0.5, 1.0} };
```

- Need to be careful:

```
double dp = u.transpose() * v;          // Returns 'double' = 2.5
MatrixXd op = u * v.transpose();        // Returns a matrix
```

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

- If you know you will need a dot product, use the **`dot(.)`** member function:

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

- Safer, commutative

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;
```

```
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;                 // = [-0.5 -2.5 -2.0]^T
```

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 apply the **`std::generate`** algorithm
  - 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 with STL containers in standard algos

  - Example 1: Dot (inner) product of **`Eigen::VectorXd`** and **`std::vector`**

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

// 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(v.begin(), v.end(), [&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);
```

  - Example 2: Element-by element addition of **`Eigen::VectorXd`** and **`std::vector`**, using ranges form of **`std::transform`**

```
// Results placed in new VectorXd:
VectorXd w(v.size());
std::ranges::transform(u, v, w.begin(), std::plus{});
```

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; });
```

- Can also apply a function object or external lambda

- QR and Singular Value Decompositions
  - Multiple Regression
  - Fund Tracking
- Cholesky Decomposition (Correlated random std normal generator)
  - Basket options valuation
  - Portfolio risk management
- Principal Components Decomposition
  - Yield Curve Dynamics
  - Measuring feature importance in a trading strategy

Eigen Decompositions

- Target fund return y
- Three fund returns to predict target fund performance x_1, x_2, x_3
- n days of data (observations)
- Seek best fit estimates $\widehat{\beta_1}, \widehat{\beta_2}, \widehat{\beta_3}$: $\hat{y} = \widehat{\beta_1} x_1 + \widehat{\beta_2} x_2 + \widehat{\beta_3} x_3$
- Drop intercept term
- Number of observations $n \gg p =$ number of funds (features) $= 3$
- Use the Householder QR Decomposition

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

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

- And the matrix $\mathbf{X}$ containing fund style index returns in each column:

$$\mathbf{X} = \begin{bmatrix} -0.044700 & -0.019003 & -0.030629 \\ -0.007888 & 0.026037 & 0.024919 \\ \vdots & \vdots & \vdots \\ 0.001440 & 0.052195 & -0.004396 \end{bmatrix}$$

```
// 3 funds, 30 days of data
MatrixXd X
{
    {-0.044700, -0.007888, . . ., 0.001440},
    {-0.019003,  0.026037, . . ., 0.052195},
    {-0.030629,  0.024919, . . ., -0.004396}
};

X.transposeInPlace();   // Design matrix X column-major

// Target return data over 30 days:
VectorXd Y
{
    {-0.039891,  0.001787,  . . ., 0.011249}
};

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

```
(Beta estimates:)
 0.352343
-0.089911
 0.391251
```

2024: stdBLAS and the Eigen Library

ChatGPT 3.5

```
    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/

COMPILER EXPLORER Add... More Templates

C++ source #1

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

```

using std::vector, std::size_t, std::cout, std::format;
namespace stdex = std::experimental;

cout << "***\ncol_major_mtx_vec_mult() col major overload\n\n ***";
std::vector col_maj_data{ 10.1, 10.3, 10.5, 10.2, 10.4, 10.6 };
std::vector col_vec_data{ 1.0, 1.0 };
std::vector prod_result_init{ 0.0, 0.0, 0.0 };  // For the resulting

// Create mdspan objects
stdex::layout_left::mapping col_major{ stdex::extents{ 3, 2 } };
stdex::mdspan col_maj{ col_maj_data.data(), col_major };
cout << "Col-major matrix:\n";
print_dynamic_mdspan(col_maj);

stdex::mdspan col_vec{ col_vec_data.data(), stdex::extents{ 2 } };
cout << "Column vector to be multiplied:\n";
print_dynamic_mdspan(col_vec);

stdex::mdspan prod_result{prod_result_init.data(), stdex::extents{ 3
stdex::linalg::matrix_vector_product(col_maj, col_vec, prod_result);
cout << "Column vector result:\n";
print_dynamic_mdspan(prod_result);

std::transform(col_maj_data.begin(), col_maj_data.end(), col_maj_data.begin(), [](double x) {return 2 * x; });
stdex::linalg::matrix_vector_product(col_maj, col_vec, prod_result);
cout << "Column vector result after modifying original data vector:\n";
print_dynamic_mdspan(prod_result);


```

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 $\alpha\boldsymbol{Ax} + \beta\boldsymbol{y}$, 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 $\alpha\boldsymbol{AB} + \beta\boldsymbol{C}$, for matrices $\boldsymbol{A}$, $\boldsymbol{B}$, & $\boldsymbol{C}$, and scalars α & β

Eigen and StdBLAS

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

Portfolio variance:

```
using std::vector;
namespace stdex = std::experimental;
// Data originates somewhere in contiguous storage (ex: std::vector)
vector cov_mtx_data        // Cov Matrix (Sigma)
{
          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 weights{ 0.25, -0.25, 0.50, 0.50 };  // Vector of asset weights (omega)

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

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

Portfolio variance (cont'd):

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

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

Next task: Cholesky Decomposition

- Covariance matrix $\mathbf{\Sigma}$, symmetric, positive definite

- Then there is a lower triangular matrix $\mathbf{L}$ such that $\mathbf{\Sigma} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$

Eigen and StdBLAS

Cholesky Decomposition:

```cpp
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> chol{ cov_mtx_map };

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

// Results could be stored as view in new mdspan (but. . .):
stdex::mdspan md_chol_mtx{ chol_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

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 limited in C++98

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

- Eigen
  - Header only/expression templates
  - Popular in quant finance, pharmaceutical applications, statistics, applied physics, data science
  - Basic Linear algebra, linear solvers, matrix decomposition
  - Unary expression function **`unaryExpr(.)`**, and other features

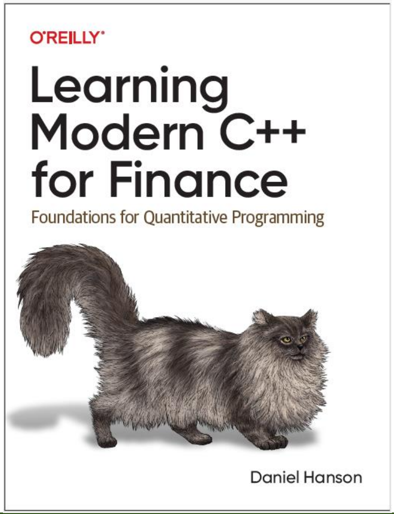
- BLAS interface proposal 1673 accepted for C++26 Standard Library
  - Matrix and vector addition, subtraction, multiplication, etc.
  - Can use **`Eigen::Map`** as view of BLAS matrix data in decompositions, solvers and other Eigen functions

References

- Eigen documentation: https://eigen.tuxfamily.org/
- Eigen in quantitative finance (Quantstart article): https://www.quantstart.com/articles/Eigen-Library-for-Matrix-Algebra-in-C
- Eigen chosen for ATLAS Experiment (CERN) tracking software: https://iopscience.iop.org/article/10.1088/1742-6596/608/1/012047/pdf
- 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
- Shameless plug: Hanson, Ch 7,

  Learning Modern C++ for Finance (O'Reilly)

  (publication later this year)



Contact/ Questions

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

- Thank You!

- Questions?