Survey of Scientific Computing

(SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 10
Matrix Algebra,
Number Theory

Session Goals

- Learn how to declare and define a **2D** matrix in C++
- Implement a function to perform **matrix multiplication**
- Develop **recursive** code to calculate the **determinant** of a 2D matrix of any size
- Implement **Cramer's rule** to solve a system of linear solutions
- Verify **Goldbach's Conjecture** within a given range by developing a vector of primes

Matrices

- A “matrix” in C++ is represented as a **vector of vectors**
  - A matrix can hold any number of elements (in each dimension) but **all** elements must be of the <u>same</u> data type, such as **int** or **double**
  - Matrix elements are accessed by their index numbers, which starts at **zero** and correspond to each dimension
- Matrix nomenclature
  - In mathematics, a matrix size is written as **(Rows x Columns)**
  - In ***legacy*** C++ code a matrix (aka an **array**) is written using **commas** instead of multiplication crosses, and using **square brackets** instead of parenthesis: **[Row][Column]**
  - Example: the C++ array **[5][7]** has **5 rows** and **7 columns**

Matrix Multiplication

- When rendering computer graphics, we often need to find the **product** of two matrices, e.g.

$$\begin{pmatrix} \boxed{4 \quad 5 \quad 8} \\ 1 \quad 9 \quad 7 \end{pmatrix} \cdot \begin{pmatrix} 2 & 4 \\ 6 & 1 \\ 5 & 9 \end{pmatrix} = ?$$

- There is a standard algorithm to do this multiplication
- It involves multiplying each element in the **rows** of first matrix by each element in the **columns** of the second matrix

Matrix Multiplication

- In order to multiply two matrices together, the number of *columns* in matrix **A** **must equal** the number of *rows* in matrix **B** (Cols **A** = Rows **B**)
- The resulting matrix will have as many rows as there were rows in matrix **A**, and as many columns as there were columns in matrix **B** (Rows **A** x Cols **B**)

Matrix Multiplication

- Example
  - Matrix **A** has dimension (2 x 3) = 2 rows, 3 columns
  - Matrix **B** has dimension (3 x 2) = 3 rows, 2 columns

The <u>inner</u> values must match!

$$\begin{pmatrix} 4 & 5 & 8 \\ 1 & 9 & 7 \end{pmatrix} \cdot \begin{pmatrix} 2 & 4 \\ 6 & 1 \\ 5 & 9 \end{pmatrix} = ?$$

$$(2 \text{ x } 3) \bullet (3 \text{ x } 2)$$

The <u>outer</u> values will be the dimension of the final matrix product!

Matrix Multiplication

- *Matrix* multiplication is **not** commutative!
  - If we multiple **A** x **B** we will get a different matrix than if we multiply **B** x **A**

    (3 x 2) • **(2 x 3)** = result is a (3 x 3) matrix

    (2 x 3) • **(3 x 2)** = result is a (2 x 2) matrix

- Welcome to the world of **non-commutative algebra**
  - This is very strange – it catches even great physicists by surprise!
  - This asymmetry is the foundation of the matrix formulation of **quantum mechanics**

Matrix Multiplication

- The algorithm is simple but tedious for **A** x **B** = **C**
- **Sum** the **product** of every element in each row of matrix **A** and the corresponding element in each column of matrix **B**
- That *sum* becomes just ***one element*** in new matrix **C**
- Continue this process for all rows in matrix **A**
  - Every row in **A** gets multiplied by every column in **B**
  - The resulting matrix will have dimensions (Rows **A** x Cols **B**)

Matrix Multiplication

Matrix A
(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B
(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**
Product Cell (**1,2**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**
Product Cell (**2,1**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**
Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C
(2 rows x 2 cols)

78	**93**
91	**76**

Matrix Multiplication

Matrix A
(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B
(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**
Product Cell (**1,2**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**
Product Cell (**2,1**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**
Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C
(2 rows x 2 cols)

78	**93**
91	**76**

Matrix Multiplication

Matrix A
(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B
(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**
Product Cell (**1,2**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**
Product Cell (**2,1**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**
Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C
(2 rows x 2 cols)

78	**93**
91	**76**

Matrix Multiplication

Matrix A

(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B

(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**

Product Cell (**1,2**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**

Product Cell (**2,1**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**

Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C

(2 rows x 2 cols)

78	**93**
91	**76**

Matrix Multiplication

Matrix A
(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B
(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**
Product Cell (**1,2**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**
Product Cell (**2,1**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**
Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C
(2 rows x 2 cols)

78	**93**
91	**76**

Open Lab 1 – Matrix Multiply

- Review the key functions **main**(), **DisplayMatrix**(), and **MultiplyMatrices**()
- The code will display the product matrix in row, col format

```
int main()
{
    matrix A{{4,5,8},{1,9,7}};
    matrix B{{2,4},{6,1},{5,9}};
    matrix C = MultiplyMatrices(A, B);

    cout << "Matrix A = " << endl;
    DisplayMatrix(A);

    cout << "Matrix B = " << endl;
    DisplayMatrix(B);

    cout << "Matrix C = " << endl;
    DisplayMatrix(C);

    return 0;
}
```

We can define matrices using the consistent C++ value initialization syntax

Passing Objects To/From Functions in C++

- In C++ inbound and outbound (**return**) function parameters are passed **by value**, even for object types
  - However, for maximum speed, it is always best to pass large objects by <u>reference</u> between functions than to pass them by *value*
  - This is because passing by *value* forces the computer to make a complete **copy** of the object between caller ↔ callee
- However, passing **by reference** may allow the called function to unexpectedly modify the caller's object
  - Therefore if your called function does not make any changes to the object, then prefix the reference type with the keyword **const**
  - Specifying **const** will get the compiler to verify the promise to the caller that the called function **will not modify** the passed object

Lab 1 – Matrix Multiply

```
void DisplayMatrix(const matrix& A)
{
    const size_t rowsA = A.size();
    const size_t colsA = A.at(0).size();

    for (size_t row{}; row < rowsA; row++)
    {
        for (size_t col{}; col < colsA; col++)
            cout << setw(5) << A.at(row).at(col);
        cout << endl;
    }
    cout << endl;
}
```

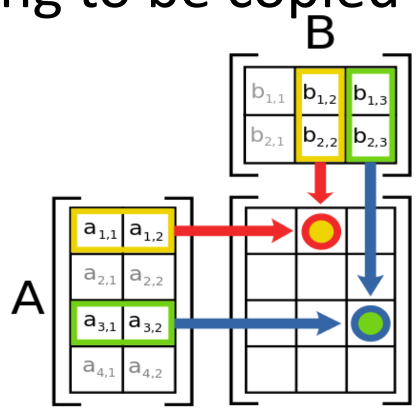
Matrix A is received as a constant reference (const matrix&)

Declaring function parameters as a **const &** is a promise your function will **not** modify the variable passed to it

Lab 1 – Matrix Multiply

- C++ uses **copy elision** (rhymes with decision) to avoid duplicating an entire object when returning to the caller
- One elision example is **NRVO** (named return value optimization) where *locally* scoped variables can be returned **directly** to the caller without having to be copied

```
matrix MultiplyMatrices(const matrix& A, const matrix& B)
{
    const size_t rowsA = A.size();
    const size_t colsA = A.at(0).size();
    const size_t colsB = B.at(0).size();

    matrix c(rowsA, vector<int>(colsB, 0));

    for (size_t i{}; i < rowsA; ++i)
        for (size_t j{}; j < colsB; ++j)
            for (size_t k{}; k < colsA; ++k)
                c.at(i).at(j) += A.at(i).at(k) * B.at(k).at(j);

    return c;
}
```



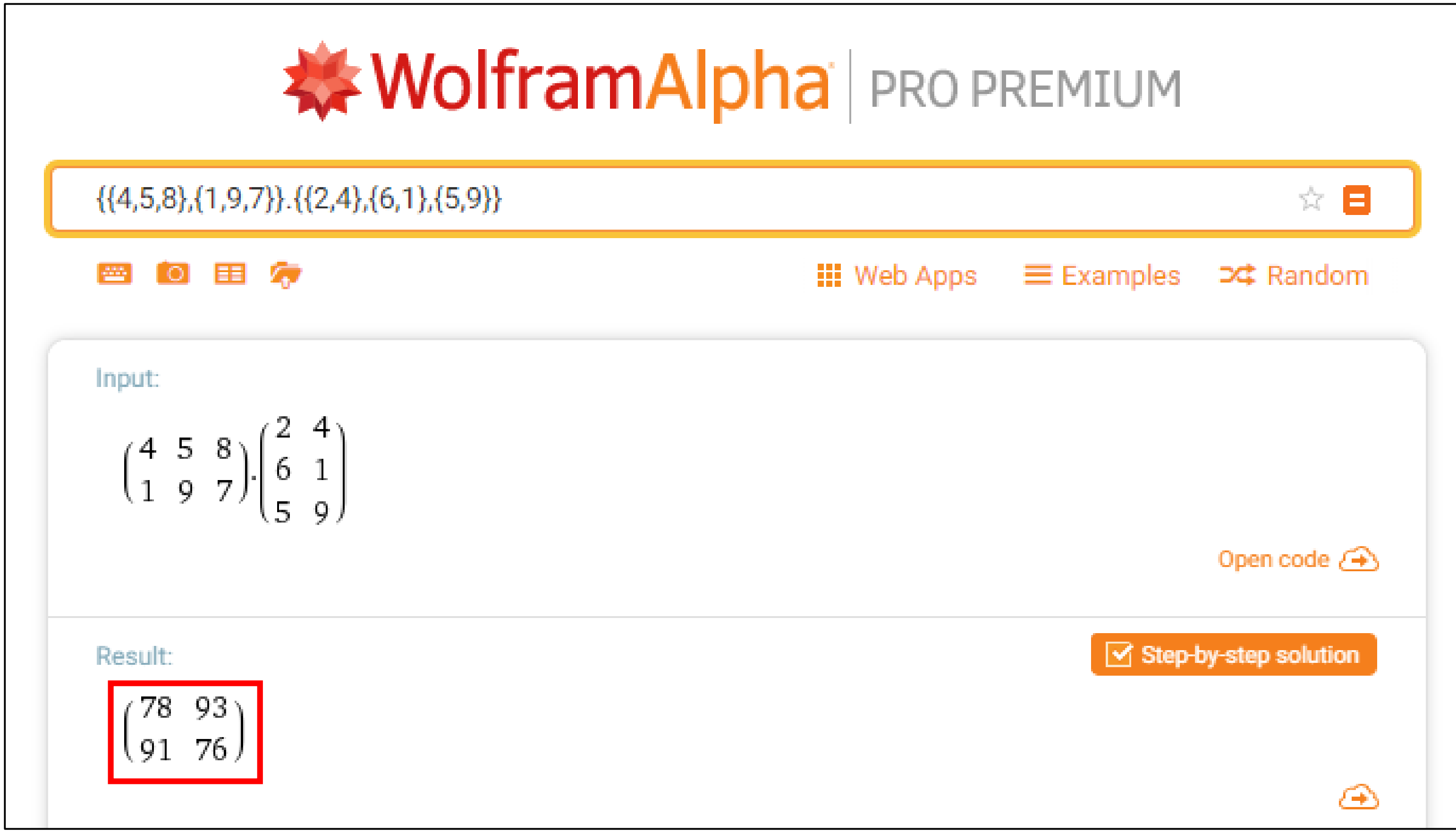
Run Lab 1 – Matrix Multiply

matrix-multiply

File Edit View Terminal Tabs Help

```
Matrix A =
    4    5    8
    1    9    7

Matrix B =
    2    4
    6    1
    5    9

Matrix C =
   78   93
   91   76


Process returned 0 (0x0)   execution time : 0.019 s
Press ENTER to continue.
```

Check Lab 1 - Matrix Multiply



Edit Lab 1 – Matrix Multiply

- Change the code to perform $B \times A$ – what is the output?

```
int main()
{
    matrix A{{4,5,8},{1,9,7}};
    matrix B{{2,4},{6,1},{5,9}};
    matrix C = MultiplyMatrices(B, A);
```

$A \times B$

```
Matrix A =
    4     5     8
    1     9     7

Matrix B =
    2     4
    6     1
    5     9

Matrix C =
   78    93
   91    76
```

$$A \times B \neq B \times A$$

$B \times A$

```
Matrix A =
    4     5     8
    1     9     7

Matrix B =
    2     4
    6     1
    5     9

Matrix C =
   12    46    44
   25    39    55
   29   106   103
```

Edit Lab 1 – Matrix Multiply

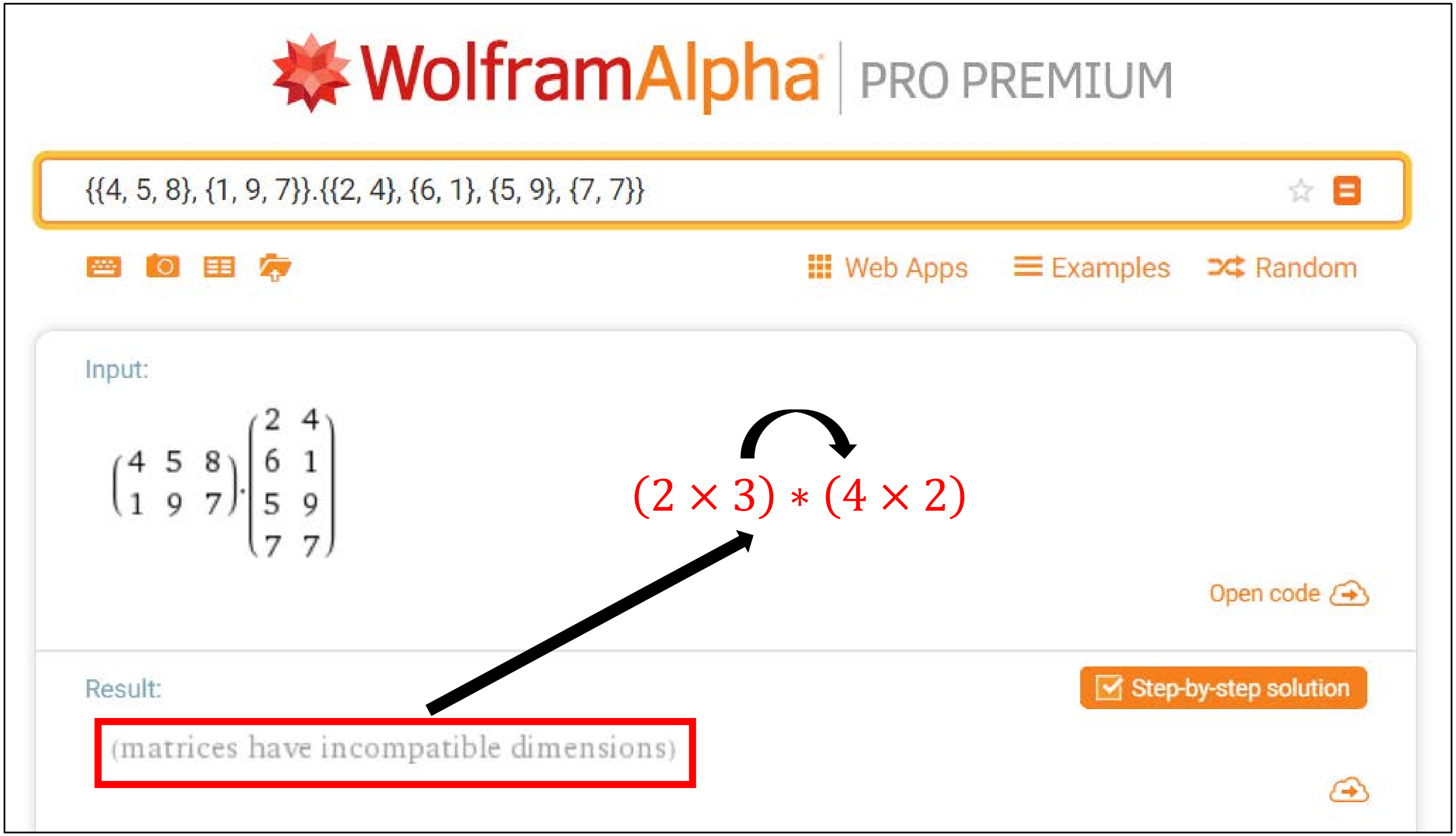
- Switch line #41 back so we return to finding $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$
- Then change **matrix B** so it is declared with an extra row of the values {7,7} – what is the output?

```
37  int main()
38  {
39      matrix A{{4,5,8},{1,9,7}};
40      matrix B{{2,4},{6,1},{5,9},{7,7}};
41      matrix C = MultiplyMatrices(A, B);
```

```
Matrix A =
    4     5     8
    1     9     7

Matrix B =
    2     4
    6     1
    5     9
    7     7

Matrix C =
   78    93
   91    76
```

Is this correct?

Check Lab 1 – Matrix Multiply



Edit Lab 1 – Matrix Multiply

- Switch line #41 back so we return to finding $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$
- Then change **matrix B** so it is declared with an extra row of the values {7,7} – what is the output?

```
37 int main()
38 {
39     matrix A{{4,5,8},{1,9,7}};
40     matrix B{{2,4},{6,1},{5,9},{7,7}};
41     matrix C = MultiplyMatrices(A, B);
```

```
Matrix A =
    4     5     8
    1     9     7

Matrix B =
    2     4
    6     1
    5     9
    7     7

Matrix C =
   78    93
   91    76
```

Is this correct?

$\color{red}{Cols_A \neq Rows_B \rightarrow Not\ valid!}$

$$(2 \times 3) \cdot (4 \times 2)$$

How could we ***strengthen*** the code to prevent such anomalies?

Determinant of a Matrix

- In linear algebra, the **determinant** is a value that can be computed from the elements of a **square matrix**
- The determinant can be used:
  - To solve a system of linear equations when those equations are represented by a matrix
  - To find the Jacobian of the matrix of all first-order partial derivatives of a vector-valued function
  - To calculate the characteristic polynomial of a matrix which is essential for eigenvalue problems
  - To express the signed *n*-dimensional volumes of *n*-dimensional parallelepipeds in analytic geometry

Determinant of a Matrix

Calculating the Determinant of a **2 x 2** Matrix

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \rightarrow \det \boldsymbol{A} = |\boldsymbol{A}| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

$$det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \boxed{a_{11} \cdot a_{22}} \boxed{-} a_{12} \cdot a_{21}$$

$$det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11} \cdot a_{22} - \boxed{a_{12} \cdot a_{21}}$$

Determinant of a Matrix

Calculating the Determinant of a **2 x 2** Matrix

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \rightarrow \det \boldsymbol{A} = |\boldsymbol{A}| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

$$det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21}$$

$$det \begin{bmatrix} 8 & 3 \\ 4 & 2 \end{bmatrix} = 8 \cdot 2 \boxed{-} 3 \cdot 4 = 16 - 12 = 4$$

Determinant of a Matrix

Calculating the Determinant of a **3 x 3** Matrix

$$\det \boldsymbol{A} = |\boldsymbol{A}| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$|\boldsymbol{A}| = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{21} \begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} + a_{31} \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix}$$

Determinant of a Matrix

Calculating the Determinant of a **3 x 3** Matrix

$$\det \boldsymbol{A} = |\boldsymbol{A}| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$|\boldsymbol{A}| = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{21} \begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} + a_{31} \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix}$$

Determinant of a Matrix

Calculating the Determinant of a **3 x 3** Matrix

$$\det \boldsymbol{A} = |\boldsymbol{A}| = \begin{vmatrix} a_{11} & \boxed{a_{12} \quad a_{13}} \\ a_{21} & \cancel{a_{22}} \quad \cancel{a_{23}} \\ a_{31} & \boxed{a_{32} \quad a_{33}} \end{vmatrix}$$

$$|\boldsymbol{A}| = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{21} \begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} + a_{31} \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix}$$

Determinant of a Matrix

Calculating the Determinant of a **3 x 3** Matrix

$$\det \boldsymbol{A} = |\boldsymbol{A}| = \begin{vmatrix} a_{11} & \boxed{\begin{matrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{matrix}} \\ a_{21} & \\ a_{31} & a_{32} \quad a_{33} \end{vmatrix}$$

$$|\boldsymbol{A}| = a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{21}\begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} + a_{31}\begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix}$$

Determinant of a Matrix

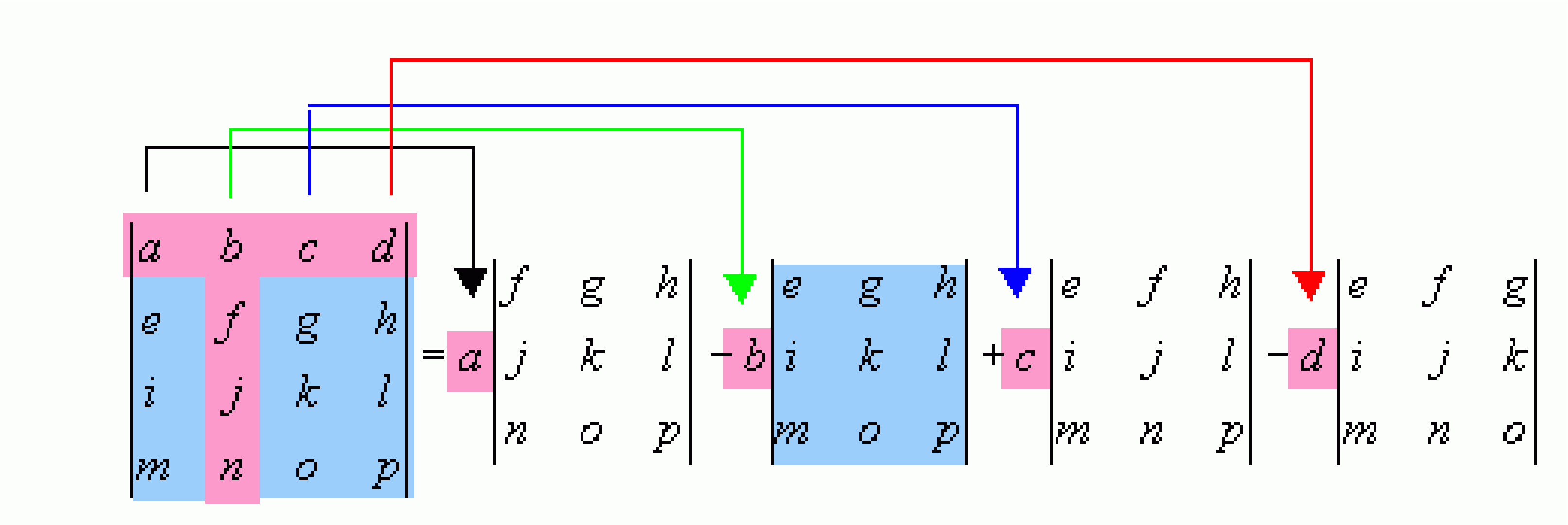
Calculating the Determinant of a **4 x 4** Matrix

$$\begin{vmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{vmatrix} = a\begin{vmatrix} f & g & h \\ j & k & l \\ n & o & p \end{vmatrix} - b\begin{vmatrix} e & g & h \\ i & k & l \\ m & o & p \end{vmatrix} + c\begin{vmatrix} e & f & h \\ i & j & l \\ m & n & p \end{vmatrix} - d\begin{vmatrix} e & f & g \\ i & j & k \\ m & n & o \end{vmatrix}$$

Notice the definition of determinant is inherently **recursive**:
The determinant of a $\boldsymbol{n} \; x \; \boldsymbol{n}$ matrix is calculated from the determinants of $\boldsymbol{n}$ **reduced matrices** of size $(\boldsymbol{n}-\mathbf{1}) \times (\boldsymbol{n}-\mathbf{1})$,
and so on and so on, getting down to simple 2 x 2 matrices

Determinant of a Matrix

Calculating the Determinant of a **4 x 4** Matrix



$$\begin{vmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{vmatrix} = a\begin{vmatrix} f & g & h \\ j & k & l \\ n & o & p \end{vmatrix} - b\begin{vmatrix} e & g & h \\ i & k & l \\ m & o & p \end{vmatrix} + c\begin{vmatrix} e & f & h \\ i & j & l \\ m & n & p \end{vmatrix} - d\begin{vmatrix} e & f & g \\ i & j & k \\ m & n & o \end{vmatrix}$$

These smaller matrices, where a row and column has been removed, are called the **cofactors** of the original (larger) matrix

Notice the sign in front of each cofactor alternates from positive (+1) to negative (-1)

Open Lab 2 – Matrix Determinant

- Your scientist wants you to write a program to calculate the determinant of a **10 x 10** matrix
- Each element in the matrix should be a **uniform** random **integer** between **-10** and **10** inclusively
- Implement the matrix as a C++ **vector of vectors**

These type aliases declare the same structure

```
typedef vector<double> matrix1D;
typedef vector<matrix1D> matrix2D;
```

```
typedef vector<vector<double>> matrix;
```

View Lab 2 – Matrix Determinant

```
int main()
{
    matrix2D A = CreateRandomMatrix(10,10);
    DisplayMatrix(A);

    double det{};
    CalcDeterminant(A, det);

    cout.imbue(std::locale(""));
    cout << fixed << setprecision(4);
    cout << "det = " << det << endl;

    return 0;
}
```

```
matrix2D CreateRandomMatrix(size_t rows, size_t cols)
{
    seed_seq seed{ 2016 };
    default_random_engine generator{ seed };
    uniform_int_distribution<> distribution(-10, 10);

    matrix2D A;
    A.resize(rows, matrix1D(cols));

    for (size_t row{}; row < rows; row++)
        for (size_t col{}; col < cols; col++)
            A.at(row).at(col) = distribution(generator);

    return A;
}
```

Remember a matrix in C++ is a vector of **rows** where each element is itself a vector of **columns**

View Lab 2 – Matrix Determinant

We pass the accumulating value of the determinant

```cpp
void CalcDeterminant(const matrix2D& A, double& det, double f = 1)
{
    size_t rowsA = A.size();
    size_t colsA = A[0].size();

    if (rowsA == 2 && colsA == 2)
        det += f * (A[0][0] * A[1][1] - A[0][1] * A[1][0]);
    else
    {
        for (size_t rowA{}; rowA < rowsA; rowA++)
        {
            matrix2D B = CreateReducedMatrix(A, rowA, 0);

            double f2 = A[rowA][0];
            if (rowA % 2 == 1) f2 *= -1;
            CalcDeterminant(B, det, f * f2);
        }
    }
}
```

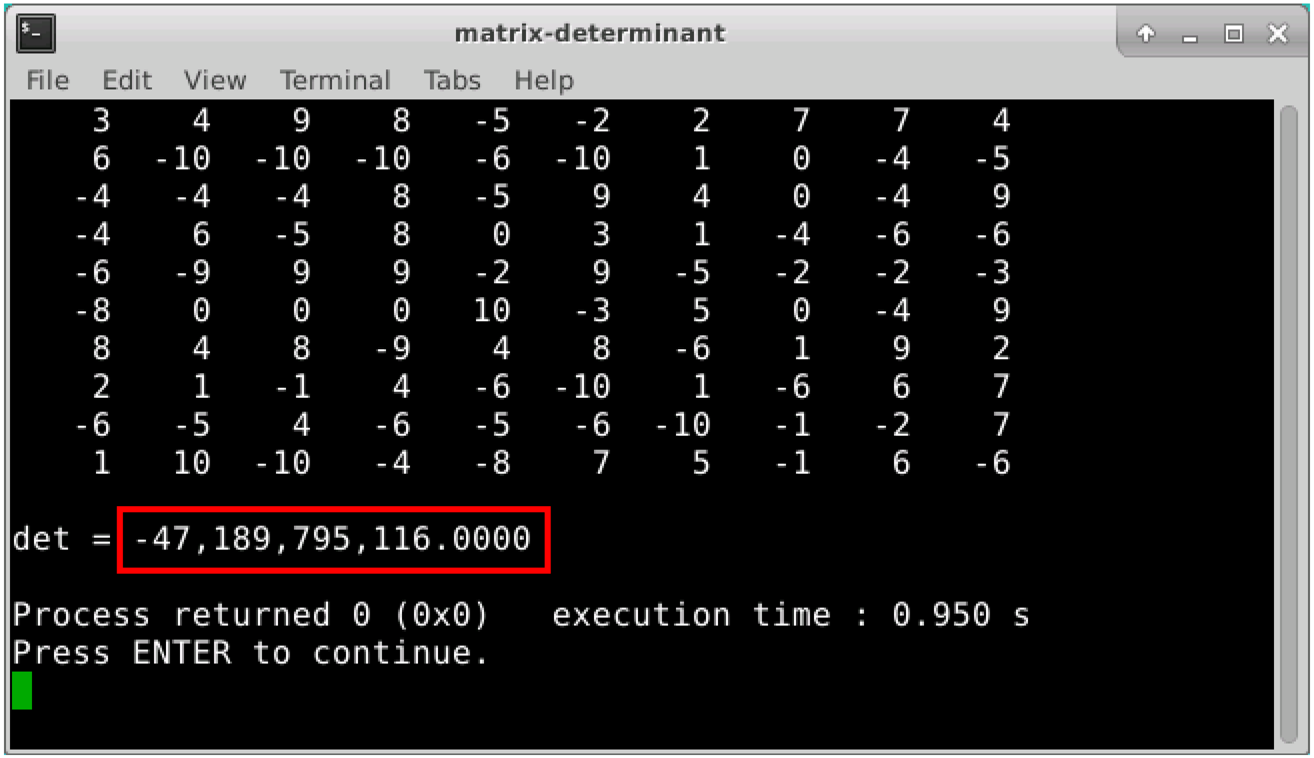
This is the base case because we know how to find the determinant of a 2 × 2 matrix <u>directly</u>

```cpp
matrix2D CreateReducedMatrix(const matrix2D& A, size_t skipRow, size_t skipCol)
{
    size_t rowsA = A.size();
    size_t colsA = A.at(0).size();

    matrix2D B(rowsA - 1, matrix1D(colsA - 1, 0));

    size_t rowB{};
    for (size_t rowA{}; rowA < rowsA; rowA++)
    {
        if (rowA == skipRow)
            continue;
        size_t colB{};
        for (size_t colA{}; colA < colsA; colA++)
        {
            if (colA == skipCol)
                continue;
            B[rowB][colB] = A[rowA][colA];
            colB++;
        }
        rowB++;
    }

    return B;
}
```

This is the recursive case as the function calls itself

We use the mod operator (%) to determine the sign of each cofactor matrix

Run Lab 2 – Matrix Determinant



```
matrix-determinant
File  Edit  View  Terminal  Tabs  Help
   3    4    9    8   -5   -2    2    7    7    4
   6  -10  -10  -10   -6  -10    1    0   -4   -5
  -4   -4   -4    8   -5    9    4    0   -4    9
  -4    6   -5    8    0    3    1   -4   -6   -6
  -6   -9    9    9   -2    9   -5   -2   -2   -3
  -8    0    0    0   10   -3    5    0   -4    9
   8    4    8   -9    4    8   -6    1    9    2
   2    1   -1    4   -6  -10    1   -6    6    7
  -6   -5    4   -6   -5   -6  -10   -1   -2    7
   1   10  -10   -4   -8    7    5   -1    6   -6

det = -47,189,795,116.0000

Process returned 0 (0x0)   execution time : 0.950 s
Press ENTER to continue.
```

Check Lab 2 – Matrix Determinant

```
In[5]:= A = {{3, 4, 9, 8, -5, -2, 2, 7, 7, 4},
    {6, -10, -10, -10, -6, -10, 1, 0, -4, -5},
    {-4, -4, -4, 8, -5, 9, 4, 0, -4, 9},
    {-4, 6, -5, 8, 0, 3, 1, -4, -6, -6},
    {-6, -9, 9, 9, -2, 9, -5, -2, -2, -3},
    {-8, 0, 0, 0, 10, -3, 5, 0, -4, 9},
    {8, 4, 8, -9, 4, 8, -6, 1, 9, 2},
    {2, 1, -1, 4, -6, -10, 1, -6, 6, 7},
    {-6, -5, 4, -6, -5, -6, -10, -1, -2, 7},
    {1, 10, -10, -4, -8, 7, 5, -1, 6, -6}};
Det[A]

Out[6]= -47 189 795 116
```

Goal: Solve a System of Linear Equations

Two Equations and Two Unknowns

$$ax + by = c$$
$$dx + ey = f$$

$$ax = c - by$$

$$x = \frac{c - by}{a}$$

$$\frac{d(c - by)}{a} + ey = f$$

$$cd - bdy + aey = af$$

$$y(ae - bd) = af - cd$$

$$y = \boxed{\frac{af - cd}{ae - bd}}$$

Goal: Solve a System of Linear Equations

Two Equations and Two Unknowns

$$ax + by = c$$
$$dx + ey = f$$

$$by = c - ax$$

$$y = \frac{c - ax}{b}$$

$$dx + \frac{e(c - ax)}{b} = f$$

$$bdx + ce - aex = bf$$

$$x(bd - ae) = bf - ce$$

$$x = \frac{bf - ce}{bd - ae}\left(\frac{-1}{-1}\right) = \boxed{\frac{ce - bf}{ae - bd}}$$

Goal: Solve a System of Linear Equations

Two Equations and Two Unknowns

$$\begin{aligned} \boxed{a}x + \boxed{b}y &= c \\ \boxed{d}x + \boxed{e}y &= f \end{aligned}$$

$$x = \frac{ce - bf}{ae - bd}$$

$$y = \frac{af - cd}{ae - bd}$$

$$\det \boldsymbol{C} = \begin{vmatrix} a & b \\ d & e \end{vmatrix} = ae - bd$$

Goal: Solve a System of Linear Equations

Two Equations and Two Unknowns

$$ax + by = c$$
$$dx + ey = f$$

$$x = \frac{ce - bf}{ae - bd}$$

$$\det \boldsymbol{A} = \begin{vmatrix} c & b \\ f & e \end{vmatrix} = ce - bf$$

$$y = \frac{af - cd}{ae - bd}$$

$$\det \boldsymbol{C} = \begin{vmatrix} a & b \\ d & e \end{vmatrix} = ae - bd$$

Goal: Solve a System of Linear Equations

Two Equations and Two Unknowns

$$\boxed{\begin{matrix} a \\ d \end{matrix}}\begin{matrix} x + by = \\ x + ey = \end{matrix}\boxed{\begin{matrix} c \\ f \end{matrix}}$$

$$x = \frac{ce - bf}{ae - bd}$$

$$y = \frac{af - cd}{ae - bd}$$

$$\det \boldsymbol{A} = \begin{vmatrix} c & b \\ f & e \end{vmatrix} = ce - bf$$

$$\det \boldsymbol{B} = \begin{vmatrix} a & c \\ d & f \end{vmatrix} = af - cd$$

$$\det \boldsymbol{C} = \begin{vmatrix} a & b \\ d & e \end{vmatrix} = ae - bd$$

Goal: Solve a System of Linear Equations

Two Equations and Two Unknowns

$$ax + by = c$$
$$dx + ey = f$$

$$x = \frac{ce - bf}{ae - bd}$$

$$y = \frac{af - cd}{ae - bd}$$

$$\det \boldsymbol{A} = \begin{vmatrix} c & b \\ f & e \end{vmatrix} = ce - bf$$

$$\det \boldsymbol{B} = \begin{vmatrix} a & c \\ d & f \end{vmatrix} = af - cd$$

$$\det \boldsymbol{C} = \begin{vmatrix} a & b \\ d & e \end{vmatrix} = ae - bd$$

$$\boxed{x = \frac{|\boldsymbol{A}|}{|\boldsymbol{C}|}}$$

$$\boxed{y = \frac{|\boldsymbol{B}|}{|\boldsymbol{C}|}}$$

Goal: Solve a System of Linear Equations

Three Equations and Three Unknowns

$$
\begin{aligned}
4x + 5y - 2z &= -14 \\
7x - \ \ y + 2z &= 42 \\
3x + \ \ y + 4z &= 28
\end{aligned}
$$

Open Lab 3 – Cramer's Rule

- Develop a console mode C++ application that uses Cramer's Rule to solve a system of three linear equations with three unknowns
  - The code encodes the system of equations as a 2D coefficient matrix and 1D value vector
  - The code already handles **linearly dependent** systems
- The code displays the equations using standard algebraic formatting rules (aka "pretty print")
  - No coefficients of 1, such as **1**x + 5y + 32z = 49
  - No plus signed followed by a negative sign, such as 2x **+ -**5y + 9 = -13
  - Don't display an unknown if that term's coefficient is **0**

Create a Coefficient Matrix & Value Vector

Coefficients

Values

$$\begin{aligned} 4x + 5y - 2z &= -14 \\ 7x - y + 2z &= 42 \\ 3x + y + 4z &= 28 \end{aligned}$$

NOTE: Using [] braces is the **legacy** way of declaring 2D arrays

```
int main()
{
    double coeffMatrix[3][3]
    {
        { 4,5,-2 },
        { 7,-1,2 },
        { 3,1,4 }
    };

    double valueVector[3] { -14,42,28 };

```

Cramer's Rule

Given the system:

We create one **overlay** matrix for *each* unknown

$$\begin{aligned} a_1x + b_1y + c_1z &= \boxed{d_1} \\ a_2x + b_2y + c_2z &= \boxed{d_2} \\ a_3x + b_3y + c_3z &= \boxed{d_3} \end{aligned}$$

with

$$D = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} \neq 0 \quad D_x = \begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix} \quad D_y = \begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix} \quad D_z = \begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}$$

then the solution of this system is:

$$x = \frac{D_x}{D} \qquad y = \frac{D_y}{D} \qquad z = \frac{D_z}{D}$$

Cramer's Rule

Calculating the Determinant of a 3 x 3 Matrix

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$= a(ei - fh) - b(di - fg) + c(dh - eg)$$

$$= aei - afh - bdi + bfg + cdh - ceg$$

$$= (aei + bfg + cdh) - (afh + bdi + ceg)$$

```
double Determinant(double matrix[3][3])
{
    double a = matrix[0][0];
    double b = matrix[0][1];
    double c = matrix[0][2];
    double d = matrix[1][0];
    double e = matrix[1][1];
    double f = matrix[1][2];
    double g = matrix[2][0];
    double h = matrix[2][1];
    double i = matrix[2][2];

    double det = a * (e * i - f * h)
        - (b * (d * i - f * g))
        + c * (d * h - e * g);

    return det;
}
```

Coefficients

Values

$$\begin{aligned} 4x + 5y - 2z &= -14 \\ 7x - \ y + 2z &= 42 \\ 3x + \ y + 4z &= 28 \end{aligned}$$

Cramer's Rule

$$\begin{aligned} a_1x + b_1y + c_1z &= \boxed{d_1} \\ a_2x + b_2y + c_2z &= \boxed{d_2} \\ a_3x + b_3y + c_3z &= \boxed{d_3} \end{aligned}$$

$$D_x = \begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix} \quad D_y = \begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix} \quad D_z = \begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}$$

Overlay the valueVector onto a given column in the coeffMatrix

```
void OverlayValues(double coeffMatrix[3][3], double valueVector[3],
    int col, double newMatrix[3][3])
{
    // Copy existing coeffMatrix to newMatrix
    for (int i{}; i < 3; ++i)
        for (int j{}; j < 3; ++j)
            newMatrix[i][j] = coeffMatrix[i][j];

    // Overlay the valueVector on the specified column
    for (int i{}; i < 3; ++i)
        newMatrix[i][col] = valueVector[i];
}
```

Cramer's Rule

Create new matrices mX, mY, mZ

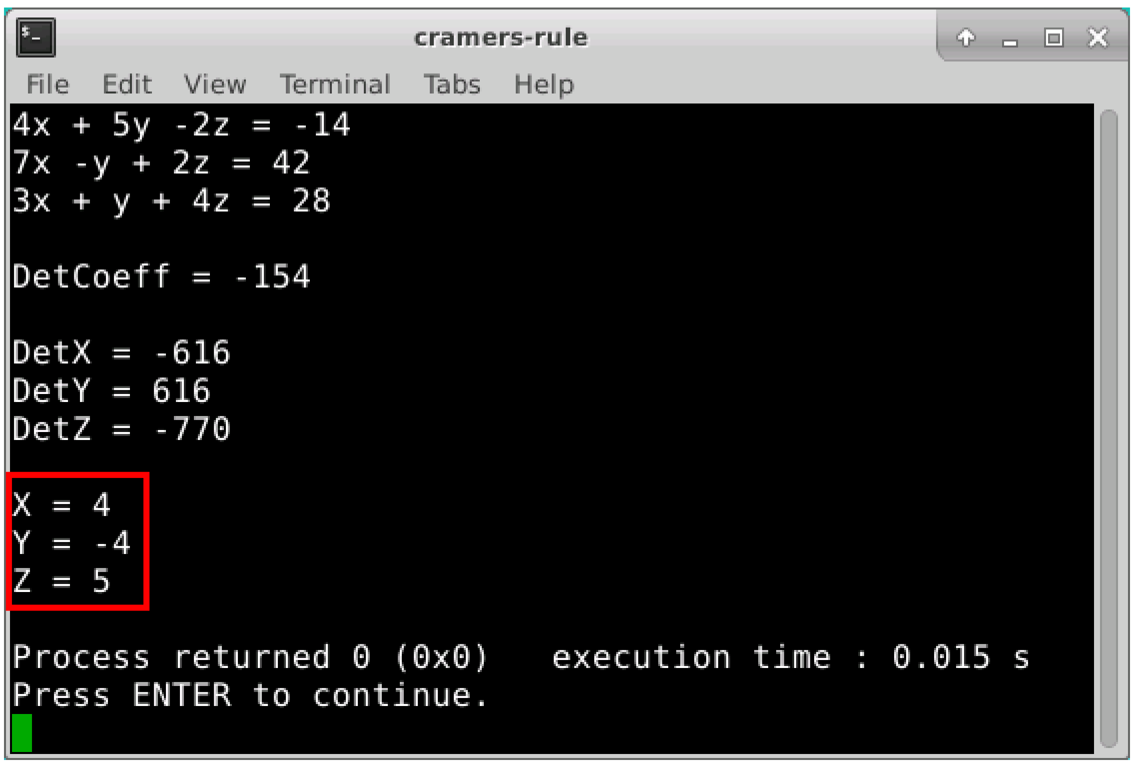
```
double mX[3][3];
OverlayValues(coeffMatrix, valueVector, 0, mX);
double detX = Determinant(mX);

double mY[3][3];
OverlayValues(coeffMatrix, valueVector, 1, mY);
double detY = Determinant(mY);

double mZ[3][3];
OverlayValues(coeffMatrix, valueVector, 2, mZ);
double detZ = Determinant(mZ);

cout << "DetCoeff = " << detCoeff << endl;
cout << endl;

cout << "DetX = " << detX << endl;
cout << "DetY = " << detY << endl;
cout << "DetZ = " << detZ << endl;
cout << endl;

cout << "X = " << detX / detCoeff << endl;
cout << "Y = " << detY / detCoeff << endl;
cout << "Z = " << detZ / detCoeff << endl;
```

All of these calculations could be done in parallel to find each unknown *simultaneously*

Run Lab 3 - Cramer’s Rule

$$x = \frac{D_x}{D} \qquad y = \frac{D_y}{D} \qquad z = \frac{D_z}{D}$$

What does it mean if D = 0?



Edit Lab 3 – Cramer's Rule

- Update the **main**() function to solve these two systems of linear equations:

$$
\begin{aligned}
-6r + 5s + 2t &= -11 \\
-2r + s + 4t &= -9 \\
4r - 5s + 5t &= -4
\end{aligned}
$$

$$
\begin{aligned}
-3a - b - 3c &= -8 \\
-5a + 3b + 6c &= -4 \\
-6a - 4b + c &= -20
\end{aligned}
$$

```
int main()
{
    double coeffMatrix[3][3]
    {
        { 4,5,-2 },
        { 7,-1,2 },
        { 3,1,4 }
    };

    double valueVector[3] { -14,42,28 };

```

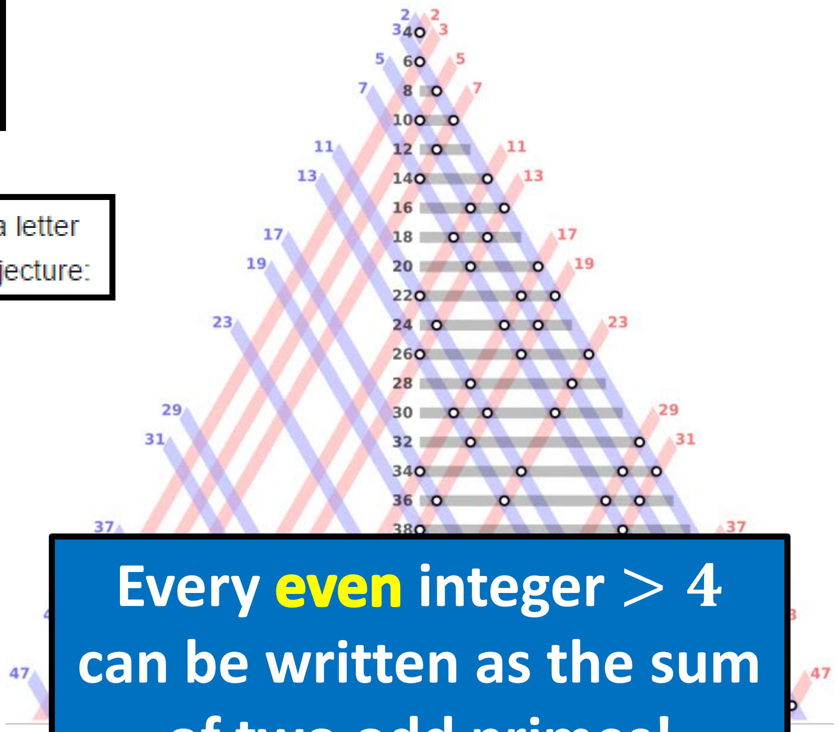
Goldbach’s Conjecture

Goldbach's conjecture

From Wikipedia, the free encyclopedia

Goldbach's conjecture is one of the oldest and best-known unsolved problems in number theory and all of mathematics.

On 7 June 1742, the German mathematician Christian Goldbach wrote a letter to Leonhard Euler (letter XLIII)[6] in which he proposed the following conjecture:



Every even integer > 4 can be written as the sum of two odd primes!

Open Lab 4 – Goldbach's Conjecture

- Goldbach's Conjecture (1742): **All even integers > 4 are the sum of just two odd primes**
- Develop a console mode C++ application that verifies Goldbach's Conjecture for all even integers $n \leq 458$
  1. Create a vector of the first **50 odd primes**
  2. Mark every integer that is the sum of any two elements in this **primes** vector
  3. Display on screen **any violations** of the conjecture
- Despite its simplicity, this remains a **conjecture** as it has been neither proven nor disproven… we still don't know!

Lab 4
Goldbach's Conjecture

```
vector<int> GeneratePrimes(size_t numPrimes)
{
    vector<int> primes;

    primes.push_back(2);

    int n = 3;
    while (primes.size() < numPrimes)
    {
        int r = 0;
        for (int p : primes)
        {
            r = n % p;
            if (r == 0) break;
        }
        if (r > 0)
            primes.push_back(n);
        n = n + 2;
    }

    return primes;
}
```

primes	**0**	**1**	**2**	**3**	4
.at()	2				

n	3	
p	2	
r	1	

primes	**0**	**1**	**2**	**3**	4
.at()	2	3			

n	5	
p	2	3
r	1	2

primes	**0**	**1**	**2**	**3**	4
.at()	2	3	5		

primes	**0**	**1**	**2**	**3**	4
.at()	2	3	5	7	

n	9	
p	2	3
r	1	0

n 11

primes	**0**	**1**	**2**	**3**	4
.at()	2	3	5	7	11

Lab 4
Goldbach's Conjecture

```
int main()
{
    vector<int> primes = GeneratePrimes(50);

    // Remove the first (only even) prime which is 2
    primes.erase(primes.begin());

    cout << "For primes " << primes.front()
         << " to " << primes.back()
         << ": " << endl;

    vector<bool> goldbachNumbers(primes.back() * 2 + 1);

    // Pair up each prime with itself, and then to each successive prime.
    // Sum these two numbers, and use this sum as an index into a boolean
    // vector that records if a sum has occurred at least once
    for (size_t i{0}; i < primes.size(); i++)
        for (size_t j{i}; j < primes.size(); j++)
            goldbachNumbers.at(primes.at(i) + primes.at(j)) = true;

    // Verify all evens #s > 4 are the sum of two odd primes
    for (size_t k{6}; k < goldbachNumbers.size(); k += 2)
        if (goldbachNumbers.at(k) == false)
            cout << "Goldbach violation at " << k << endl;

    return 0;
}
```

Every even integer > 4 can be written as the sum of two odd primes!

primes	0	1	2	3	4	5	6	7
.at()	2	3	5	7	11	13	17	19

primes	0	1	2	3	4	5	6	7
.at()	3	5	7	11	13	17	19	23

goldbachNumbers	0	1	2	3	4	5	6	7
.at()	false	false	false	false	false	false	false	false

i 0
j 0
primes.at(i) 3
primes.at(j) 3

goldbachNumbers	6	7	8	9	10	11	12	13
.at()	TRUE	false	false	false	false	false	false	false

i 0
j 1
primes.at(i) 3
primes.at(j) 5

goldbachNumbers	6	7	8	9	10	11	12	13
.at()	TRUE	false	TRUE	false	false	false	false	false

i 2
j 2
primes.at(i) 5
primes.at(j) 5

goldbachNumbers	6	7	8	9	10	11	12	13
.at()	TRUE	false	TRUE	false	TRUE	false	false	false

Run Lab 4 – Goldbach's Conjecture

goldbach-conjecture

File Edit View Terminal Tabs Help

```
For primes 3 to 233:
Goldbach violation at 436
Goldbach violation at 442
Goldbach violation at 448
Goldbach violation at 464

Process returned 0 (0x0)   execution time : 0.029 s
Press ENTER to continue.
```

How can this be? Where did we go wrong?

List of first 100 prime numbers

2	3	5	7	11
13				
31				
53				
73				
101				
127				
151				
179				
199	211	[illegible]	227	229
233	239	[illegible]	251	257
263	269	[illegible]	277	281
283	293	[illegible]	311	313
317	331	[illegible]	347	349
353	359	[illegible]	373	379
383	389	[illegible]	401	409
419	421	431	433	439
443	449	457	461	463
467	479	487	491	499
503	509	521	523	541

We need to include the first 87 *odd* primes to cover the range $n \leq 458$

WolframAlpha PRO PREMIUM

what two primes sum to 436?

Input interpretation:

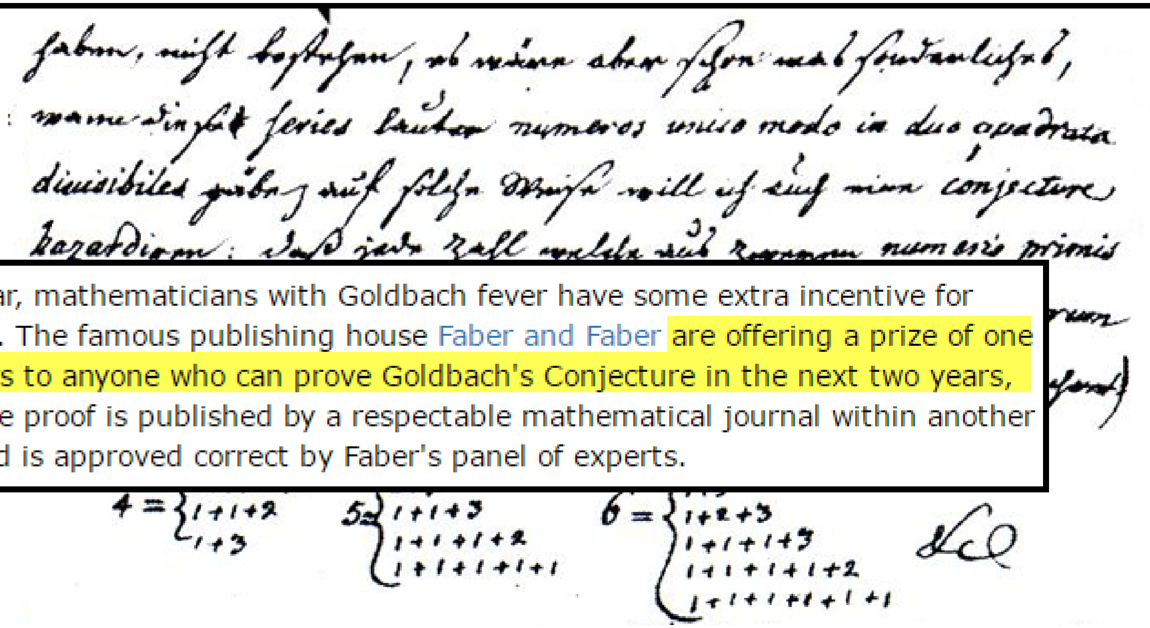
436	as the sum of exactly	2	primes

Result:

$433 + 3 = 436$

Verify for all even integers ≤ 458

Goldbach's Conjecture



As of this year, mathematicians with Goldbach fever have some extra incentive for their labours. The famous publishing house Faber and Faber are offering a prize of one million dollars to anyone who can prove Goldbach's Conjecture in the next two years, as long as the proof is published by a respectable mathematical journal within another two years and is approved correct by Faber's panel of experts.

PRIME NUMBERS: THE 271 YEAR OLD PUZZLE RESOLVED

STORY BY ARTEM KAZNATCHEEV

Published: May 13, 2013

The odd Goldbach conjecture, a two-hundred and seventy-one year open problem of mathematics, has been resolved. Earlier today, H.A. Helfgott proved that any odd number greater than 5 can be written as the sum of 3 primes.

Now you know…

- To multiply two matrices, the **# of columns** in the first matrix must match the **# of rows** in the second matrix
  - The **product matrix** will have the same # of rows as the first matrix, and the same # of columns as the second matrix
  - Matrix multiplication is the sum of the element by element products of the rows in the first matrix and the columns in the second matrix
- **Cramer's Rule** is a step-by-step algorithmic way to solve systems of linear equations ***without*** the tedious algebra of back substitution – it uses **determinants**
  - This method is easy to run in parallel as each CPU core can calculate a separate unknown at the same time
- **Goldbach is waiting for you!** ☺