How to Multiply a 3×3 Matrix by a 3×2 Matrix: A Step-by-Step Guide

Authored by

stats writer

March 1, 2026

Introduction to the Principles of Matrix Multiplication

In the vast and intricate field of **linear algebra**, the operation of **matrix multiplication** serves as a fundamental building block for a wide variety of advanced mathematical concepts and computational applications. At its core, multiplying a **matrix** by another involves a systematic process of combining **row vectors** and **column vectors** to produce an entirely new set of data. This specific tutorial focuses on the multiplication of a 3×3 square matrix by a 3×2 rectangular matrix, a scenario that frequently arises in fields such as **computer graphics**, **physics** simulations, and **data science**.

To successfully perform this operation, one must deeply understand the **dimension** requirements that govern the interaction between two matrices. A matrix is defined by its number of rows (m) and columns (n), often denoted as m×n. For the **product** of two matrices to be defined, the number of columns in the first matrix must strictly match the number of rows in the second matrix. In our specific case, the first matrix is 3×3 (3 columns) and the second is 3×2 (3 rows), making them perfectly compatible for multiplication. This compatibility ensures that every **element** in a row has a corresponding partner in the target column during the calculation phase.

The resulting matrix from this operation will inherit its vertical dimension from the first matrix and its horizontal dimension from the second matrix. Therefore, when we multiply a 3×3 matrix by a 3×2 matrix, the output will invariably be a 3×2 matrix. This transformation represents a **linear transformation** where information from a three-dimensional input space is mapped into a two-dimensional output space, or vice versa, depending on the context of the **vector** basis being used. Understanding this structural outcome is crucial for verifying the accuracy of your results before diving into the individual arithmetic steps.

Throughout this comprehensive guide, we will break down the mechanics of the **dot product**, explore the symbolic representations of these operations, and provide several detailed examples using real numbers. By the end of this article, you will possess a robust understanding of how to navigate these **algorithms** with precision. Whether you are a student of **calculus** or a professional working with complex datasets, mastering this skill is essential for your mathematical repertoire.

Dimensional Analysis and Structural Requirements

The first step in any matrix operation is performing a rigorous dimensional check to ensure the **conformability** of the operands. In **linear algebra**, matrix multiplication is not **commutative**, meaning the order in which you multiply matters significantly. To multiply matrix A (the multiplier) by matrix B (the multiplicand), the inner dimensions must be identical. For a 3×3 matrix A and a 3×2 matrix B, the "inner" numbers are both 3. This alignment allows the **dot product** of the rows and columns to be calculated without any leftover or missing values.

If the dimensions were not aligned--for instance, if we attempted to multiply a 3×2 matrix by a 3×3 matrix--the operation would be mathematically undefined. This is because the first matrix would have only two elements per row, while the second matrix would require three elements per column to complete the calculation. This structural rule is a safeguard that maintains the integrity of **transformation matrices**. By adhering to these rules, we ensure that the resulting **matrix** correctly represents the combined linear operations of the two original components.

Once the compatibility is confirmed, we can predict the size of the resulting matrix. The "outer" dimensions of the pair provide this information. For a (3×3) × (3×2) multiplication, the outer dimensions are 3 and 2. Thus, the product matrix will consist of 3 rows and 2 columns, containing a total of 6 individual **elements**. Each of these six positions represents a unique intersection between the horizontal data of the first matrix and the vertical data of the second, acting as a summary of their interaction.

The Procedural Logic of Row-by-Column Multiplication

The actual calculation of each element in the product matrix follows the "row-by-column" rule. To find the element in the first row and first column of the result, you must take the first **row vector** from the first matrix and calculate its **dot product** with the first **column vector** of the second matrix. This involves multiplying the first elements together, the second elements together, and the third elements together, then summing those three **products**. This summation provides a single **scalar** value for that specific position.

This process is repeated systematically for every row and column combination. Specifically, for our 3×2 output, we will perform this operation six times: once for each of the three rows interacting with each of the two columns. It is helpful to visualize this as the first matrix "scanning" across the second matrix. Each row of the 3×3 matrix effectively generates a full row of the 3×2 result by interacting with all available columns in the second **matrix**. This repetitive nature makes the process an ideal candidate for **parallel computing** and **GPU** acceleration in modern technology.

Precise record-keeping is vital during these calculations, as a single error in **arithmetic** or sign placement will invalidate the entire result. Many practitioners find it useful to write out the individual multiplication steps before summing them, especially when dealing with negative numbers or large **integers**. This methodical approach reduces cognitive load and ensures that each **scalar** in the final matrix accurately reflects the **linear map** intended by the original matrices.

Detailed Walkthrough of the Mathematical Formula

To provide a clear conceptual framework, let us examine the symbolic representation of this process. Suppose we have a **3×3** matrix **A**, which contains nine distinct variables arranged in three rows and three columns. This matrix acts as our primary operator in the multiplication sequence.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.super_short2 {
max-width: 250px;
margin: 5px auto;
color: blue;
}
```


A =

	A11	A12	A13	
A21	A22	A23		
A31	A32	A33		

Next, we introduce a **3×2** matrix **B**, which consists of six elements. Although it has the same number of rows as matrix A, its reduced column count will define the final width of our resulting **product**.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.short2 {
max-width: 250px;
margin: 5px auto;
color: red;
}
```


B =

	B11	B12	
B21	B22		
B31	B32		

The formula for multiplying matrix A by matrix B involves the summation of products across each row and column intersection. The following table illustrates the complete symbolic expansion of the resulting 3×2 matrix. Note how each **element** is a composite of three separate multiplications.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.long{
margin: 5px auto;
color: #000000;
}
.red {
color: red;
}
.blue {
color: blue;
}
```


A x B =				
		A11*B11+A12*B21+A13*B31	A11*B12+A12*B22+A13*B32	
	A21*B11+A22*B21+A23*B31	A21*B12+A22*B22+A23*B32		
	A31*B11+A32*B21+A33*B31	A31*B12+A32*B22+A33*B32		

As demonstrated, the final output is a **3×2 matrix**. Each row of this output corresponds to the weighted sum of the columns of matrix B, where the weights are provided by the rows of matrix A. This relationship is a fundamental aspect of **linear combinations** in vector spaces.

Practical Application: Worked Example 1

To transition from theory to practice, let us consider a concrete example using specific numerical values. Suppose we define a **3×3** matrix **C** as follows. This matrix contains a mix of positive and negative integers, which will require careful attention during the summation process.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.ex3_3 {
max-width: 250px;
margin: 5px auto;
color: #000000;
}
```

C =	

	-3	5	4	
1	2	3		
-1	0	2		

We will multiply this by a **3×2** matrix **D**. In this matrix, the values are relatively small, which will help us clearly see the **arithmetic** progression without becoming overwhelmed by large magnitudes.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.short {
max-width: 250px;
margin: 5px auto;
color: #000000;
}
```

D =	

	2	1	
5	1		
0	-1		

The multiplication of matrix C by matrix D is performed by executing the **dot product** for each of the six resulting positions. Observe how the negative values in matrix C interact with the positive and negative values in matrix D.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.medium {
max-width: 500px;
margin: 5px auto;
color: #000000;
}
```

.red {
color: red;
}
.blue {
color: blue;
}

C x D =				
		-3*2 + 5*5 + 4*0	-3*1 + 5*1 + 4*-1	
	1*2 + 2*5 + 3*0	1*1 + 2*1 + 3*-1		
	-1*2 + 0*5 + 2*0	-1*1 + 0*1 + 2*-1		

After performing the addition and multiplication for each cell, we arrive at the final 3×2 **matrix**. Notice how the zero in the middle row of the third column in matrix C simplifies several calculations, effectively "ignoring" the corresponding values in matrix D.

table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;

}
.short {
max-width: 250px;
margin: 5px auto;
color: #000000;
}

C x D =				
		19	-2	
	12	0		
	-2	-3		

Practical Application: Worked Example 2

In our second example, we will explore a scenario where the matrices contain larger integers and a higher frequency of negative signs. This requires a **formal** and rigorous approach to ensure that the final **scalars** are calculated correctly. Consider the 3×3 matrix **E**:

table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;

```
width: 5px;
padding: 0;
}
.ex3_3 {
max-width: 250px;
margin: 5px auto;
color: #000000;
}
```

E =					
	2	8	1	
3	3	0		
0	1	2		

Next, we identify the 3×2 matrix **F**, which will serve as our second operand. Multiplying these together will demonstrate how zero elements can significantly simplify the **product** calculation by nullifying specific terms.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
```

```
width: 5px;
padding: 0;
}
.short {
max-width: 250px;
margin: 5px auto;
color: #000000;
}
```

F =	

	-2	-2	
3	1		
4	10		

By applying the **matrix multiplication** rule, we multiply each row of E by each column of F. The presence of zeros in matrix E acts as a filter, removing the influence of certain elements in matrix F during the **dot product** phase.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
```

```
width: 5px;
padding: 0;
}
.medium {
max-width: 500px;
margin: 5px auto;
color: #000000;
}
.red {
color: red;
}
.blue {
color: blue;
}
```

E x F =				
		2*-2 + 8*3 + 1*4	2*-2 + 8*1 + 1*10	
	3*-2 + 3*3 + 0*4	3*-2 + 3*1 + 0*10		
	0*-2 + 1*3 + 2*4	0*-2 + 1*1 + 2*10		

The resulting 3×2 matrix is shown below. This output represents the **linear transformation** of the space defined by matrix F through the operator matrix E. Such transformations are common in **linear algebra** applications involving coordinate systems.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
```

```
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.short {
max-width: 250px;
margin: 5px auto;
color: #000000;
}
```

E x F =				
		24	14	
	3	-3		
	11	21		

Practical Application: Worked Example 3

For our final example, we will look at a matrix that resembles a **triangular matrix**. Matrix G has several zero entries, which will facilitate a very efficient calculation process. This example is particularly useful for understanding how **sparse matrices** behave in computational environments.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
```

padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.ex3_3 {
max-width: 250px;
margin: 5px auto;
color: #000000;
}

G =	

	-1	0	0	
7	1	0		
2	4	6		

Matrix H is our final 3×2 **matrix**. When multiplying G by H, the first row of G, which consists primarily of zeros, will isolate only the first element of each column in H, significantly speeding up the derivation of the first row of our result.

table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {

```
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.short {
max-width: 250px;
margin: 5px auto;
color: #000000;
}
```

H =

	4	5	
9	2		
0	1		

The calculation for G x H is detailed below. Follow the **dot product** steps to see how the lower-triangular structure of G interacts with the values in H to create the final **product**.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
```

```
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.medium {
max-width: 500px;
margin: 5px auto;
color: #000000;
}
.red {
color: red;
}
.blue {
color: blue;
}
```

G x H =				
		-1*4 + 0*9 + 0*0	-1*5 + 0*2 + 0*1	
	7*4 + 1*9 + 0*0	7*5 + 1*2 + 0*1		
	2*4 + 4*9 + 6*0	2*5 + 4*2 + 6*1		

The final result is a 3×2 matrix that demonstrates the combined effect of these **linear algebra** operations. This completes our series of practical examples, illustrating the versatility of the multiplication process.

```
table {
border-collapse: collapse;
border-spacing: 0;
padding: 0;
}
td.tdleft {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
```

```
border-left: solid 1px #000;
width: 5px;
padding: 0;
}
td.tdreg {
padding: 2px 1px;
text-align: center;
border-bottom: solid 1px #fff;
}
td.tdright {
border-top: solid 1px #000;
border-bottom: solid 1px #000;
border-right: solid 1px #000;
width: 5px;
padding: 0;
}
.short {
max-width: 250px;
margin: 5px auto;
color: #000000;
}
```

G x H =

	-4	-5	
37	37		
44	24		

Real-World Utility and Algorithmic Importance

While calculating **matrix multiplication** by hand is an excellent way to internalize the underlying principles, most modern applications rely on software to handle these tasks. Programs like **MATLAB**, **NumPy**, and various **machine learning** libraries use highly optimized **BLAS** (Basic Linear Algebra Subprograms) to perform these operations at lightning speed. Understanding the 3×3 by 3×2 interaction helps developers debug data pipelines where dimension mismatches are a common source of **software bugs**.

In the realm of **computer graphics**, these matrices are used to transform 3D coordinates into 2D screen space. A 3×3 matrix might represent a rotation or scaling in three dimensions, while the multiplication with a 3×2 matrix could be part of a **projection** or a coordinate mapping to a specific

interface element. The efficiency of these calculations is what allows modern video games and visual effects to render millions of **polygons** per second with fluid motion.

Furthermore, in **artificial intelligence**, particularly in **neural networks**, matrix multiplication is the primary engine for forward and backward propagation. Weights are often stored in matrices, and the input data is processed through these layers using the very same row-by-column logic we have discussed. Mastery of these concepts is therefore not just a mathematical exercise, but a gateway to understanding the technology that powers the modern digital world.

The following tutorials explain how to perform other common matrix multiplications:

Identity Matrix Multiplication
Matrix Transposition and Products
Inverting Matrices for Division