

# RMS and Standard deviation

Gemini AI Pro 2.5

## Q

*Root mean square. What is the usefulness of this metric?*

## A

The Root Mean Square (RMS) is a statistical measure used to find the **effective value or magnitude** of a varying quantity. Unlike a simple average, it's especially useful for sets of numbers that include negative values or fluctuate, such as waves.

Its primary usefulness comes from its method: 1. **Square:** Square each value in the set. This makes all values non-negative (e.g., -10 becomes 100) and gives greater weight to larger values. 2. **Mean:** Calculate the average (mean) of these squared values. 3. **Root:** Take the square root of that mean. This returns the metric to the original unit of measurement (e.g., from Volts<sup>2</sup> back to Volts).

This “squaring” step is key. A simple average of a fluctuating signal like an AC (alternating current) sine wave would be zero, which is not useful. The RMS value, however, gives a meaningful measure of its strength.

---

## Usefulness in Physics & Engineering

The most common application is in **electrical engineering** to describe alternating current (AC) voltage or current.

- **AC Power:** An AC voltage (like in your wall outlet) constantly fluctuates between a positive and negative peak.
- **The Problem:** A simple average would be zero, but the current is clearly doing work (e.g., lighting a bulb).
- **RMS Solution:** The RMS value tells you the **equivalent DC (direct current) value** that would deliver the **same amount of power** to a resistor.

**Example:** A 120V AC outlet in the US means its **RMS voltage is 120 Volts**. This provides the same power and heating effect as

a steady 120V DC battery. The *peak* voltage of that AC signal is actually much higher (about 170V).

This is why RMS is used for rating electrical appliances, fuses, and circuit breakers—it provides a standard, meaningful way to compare the power of AC and DC.

---

### Usefulness in Statistics & Data Science

In statistics, the concept is most famously used as the **Root Mean Square Error (RMSE)**. This metric is a standard way to measure the accuracy of a model's predictions.

- **The Problem:** When building a model (e.g., to predict house prices), you need to know how “wrong” its predictions are. Your model will be off by different amounts for different houses (the “errors” or “residuals”).
- **RMSE Solution:** The RMSE aggregates these errors into a single number.
  - It squares each error (so +\$10k and -\$10k errors are treated equally).
  - It averages these squared errors.
  - It takes the square root to show the error in the original units (e.g., dollars).

**Example:** If a house price prediction model has an RMSE of \$15,000, it means the model's predictions are, *on average*, about \$15,000 away from the actual sale prices.

Because it squares the errors, the RMSE **penalizes large errors more heavily** than smaller ones. This makes it particularly useful when large mistakes are highly undesirable.

---

This video provides a good explanation of how RMS values are calculated for different waveforms. \* Average and RMS Calculations

[http://googleusercontent.com/youtube\\_content/0](http://googleusercontent.com/youtube_content/0) <https://www.youtube.com/watch?v=TDgTYsgz4UI>

### Q

*What is the link between RMS and Standard Deviation?*

### A

Of course. The link is very direct: **Standard Deviation is a specific type of Root Mean Square.**

To be precise, the Standard Deviation is the RMS of the *deviations from the mean*. The RMS you asked about previously is the RMS of the *actual values* themselves.

Let's break that down.

## 1. The Core Difference

- **Root Mean Square (RMS):** Measures the magnitude of a set of values, centered around **zero**.
- **Standard Deviation (SD):** Measures the “spread” or “dispersion” of a set of values, centered around their own **mean (average)**.

**The critical link:** If the mean of your dataset is exactly zero, the RMS and the Standard Deviation will be identical.

---

## 2. A Look at the Formulas

This is where the connection becomes obvious. Let's look at a set of values  $x_1, x_2, \dots, x_n$  and their mean,  $\bar{x}$ .

**Root Mean Square (RMS)** You square the **values**, find their mean, and take the root.

$$RMS = \sqrt{\frac{x_1^2 + x_2^2 + \dots + x_n^2}{n}}$$

**Standard Deviation ( $\sigma$ )** You first find the *deviation* (error) of each value from the mean ( $x_i - \bar{x}$ ). Then, you square those **deviations**, find their mean, and take the root.

$$\sigma = \sqrt{\frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \dots + (x_n - \bar{x})^2}{n}}$$

See the pattern? Both are a “Root” of a “Mean” of “Squares.” The only difference is *what* you are squaring: \* **RMS:** Squares the **values**. \* **SD:** Squares the **distance from the mean**.

---

## 3. The Mathematical Relationship

There is a formal equation that ties RMS, Standard Deviation ( $\sigma$ ), and the Mean ( $\bar{x}$ ) together:

$$RMS^2 = \sigma^2 + \bar{x}^2$$

Or, written out:

$$(\text{RMS of values})^2 = (\text{Standard Deviation})^2 + (\text{Mean of values})^2$$

This formula perfectly illustrates their relationship:

- **If the Mean ( $\bar{x}$ ) is 0:** The equation becomes  $RMS^2 = \sigma^2$ , so  $RMS = \sigma$ . This is why they are identical for signals with no DC offset (like a pure AC sine wave).
- **In all other cases:** The RMS value will always be **larger** than the Standard Deviation because it also accounts for the energy or magnitude of the mean.

In engineering terms, RMS represents the **total power** of a signal, while the Standard Deviation ( $\sigma$ ) represents the power of the **fluctuating (AC) part** of the signal, and the Mean ( $\bar{x}$ ) represents the power of the **steady (DC) part**.

---

This video discusses the calculation of RMS for different types of signals, which relates to these concepts. \* RMS Value Calculations

[http://googleusercontent.com/youtube\\_content/1](http://googleusercontent.com/youtube_content/1) <https://www.youtube.com/watch?v=TDgTYsgz4UI>

## Q

*This code works on a vector. Please convert it to work on a matrix, and explain carefully how the column slicing operations work.*

```
function mystd(x)
    n = size(x,1)
    mu = mymean(x)
    sigma = sqrt(1/n * sum((x.-mu).^2))
    return sigma
end;
```

## A

Here is the code converted to work on a matrix, calculating the standard deviation for each column.

### The Converted Code

The main change is adding `dims=1` to the `sum` function and updating `mymean` to also work column-wise. This assumes you want the standard deviation **of each column**, which is the standard convention in statistics.

I've included the necessary `mymean` function for clarity.

```
"""
Calculates the mean of each column of a matrix x.
Returns a 1xk row vector of means.
"""

function mymean(x::Matrix)
    # sum(x, dims=1) sums all rows for each column,
    # producing a 1xk row vector.
```

```

# size(x, 1) is the number of rows (n).
return sum(x, dims=1) / size(x, 1)
end

"""
Calculates the population standard deviation of each column of a matrix x.
Returns a 1xk row vector of standard deviations.
"""

function mystd(x::Matrix)
    n = size(x, 1) # n = number of rows (observations)
    mu = mymean(x) # mu = 1xk row vector of column means

    # See explanation below for this line
    squared_diffs = (x .- mu).^2

    # Sum the squared differences *down the columns* (dims=1)
    # and divide by n to get the variance for each column.
    variance = sum(squared_diffs, dims=1) / n

    # Take the element-wise square root
    sigma = sqrt.(variance)

    return sigma # Returns a 1xk row vector
end

```

You could also write `mystd` as a one-liner, just like your original:

```

function mystd_oneliner(x::Matrix)
    n = size(x, 1)
    mu = mymean(x)
    # The .^ and .- are element-wise. sum(..., dims=1) is column-wise.
    sigma = sqrt.( (1/n) * sum((x .- mu).^2, dims=1) )
    return sigma
end

```

---

### Explanation of Column-Wise Operations

Your original code used “dot operators” (`.-`, `.^`) for element-wise operations on vectors. When moving to a matrix, we combine this with two other concepts: **broadcasting** and **dimensional aggregation**.

This is more efficient than “column slicing” (e.g., writing a `for` loop over `x[:, i]`).

Let’s break down the key line: `sum((x .- mu).^2, dims=1)`

Assume `x` is a **10x3** matrix (10 rows, 3 columns).

- `n = size(x, 1)` will be **10**.

**1. `mu = mymean(x)`** First, `mymean(x)` calculates the mean of each column.

- `sum(x, dims=1)` sums the 10 numbers in each column, producing a **1x3** row vector.
- It divides this by `n` (10) to get the average for each column.
- `mu` is now a **1x3** row vector: `[mu_col1, mu_col2, mu_col3]`

**2. `(x .- mu)` — Broadcasting (The “Implicit Slicing”)** This is the most important step. You are subtracting a **1x3** row vector (`mu`) from a **10x3** matrix (`x`).

Julia’s **broadcasting** feature handles this automatically. It “stretches” (or *broadcasts*) the **1x3** row vector `mu` to match the **10x3** matrix.

Operationally, it subtracts:

- `mu_col1` from every element in the **1st column** of `x`.
- `mu_col2` from every element in the **2nd column** of `x`.
- `mu_col3` from every element in the **3rd column** of `x`.

The result is a new **10x3** matrix of “deviations from the mean.” This single operation achieves the same result as a `for` loop over the columns, but is much faster.

**3. `(...)^2`** This is a simple element-wise operation. It squares every deviation in the **10x3** matrix, resulting in a new **10x3** matrix of squared differences.

**4. `sum(..., dims=1)` — Dimensional Aggregation** This is the final step in getting the variance.

- `sum(...)` adds up the numbers.
- `dims=1` tells Julia *which dimension to sum along*. Dimension 1 is **rows**.
- So, `sum(..., dims=1)` “collapses” the rows by summing them up. It sums all 10 squared differences in each of the 3 columns.
- The result is a **1x3** row vector containing the total sum of squared differences for each column.

The rest of the function (`/ n` and `sqrt.`) then scales this **1x3** vector to get the final standard deviations.

## Q

*Please explain the inside of the parenthesis: function mymean(x::Matrix)*

## A

This part (`x::Matrix`) defines the function's *arguments* (also called *parameters*).

It can be broken into two parts:

1. `x`: This is the **name of the argument**. It's a local variable name. When you call the function, like `mymean(my_data)`, the data `my_data` will be assigned to the variable `x` *inside* the function.
  2. `::Matrix`: This is a **type annotation**.
    - The `::` (double colon) operator is used to “assert” or “declare” the type of a variable.
    - `Matrix` is a built-in Julia type that means “a 2-dimensional array.”
- 

### What it means together

`function mymean(x::Matrix)` creates a “contract” or “signature” for this function. It means:

“I am defining a function named `mymean` that must be called with **one argument**, and that argument **must be a 2D matrix**.”

### Why this is important in Julia

This syntax is the key to Julia's main feature: **multiple dispatch**.

You can (and often do) create multiple functions with the *same name* but *different type annotations*:

- `function mymean(x::Matrix)`: Calculates the mean for a 2D matrix (e.g., column-wise).
- `function mymean(x::Vector)`: Calculates the mean for a 1D vector (e.g., a simple average).
- `function mymean(x::Number)`: Just returns the number itself.

When you call `mymean(my_data)`, Julia looks at the **type** of `my_data` at runtime and automatically chooses (or *dispatches to*) the correct, specialized version of the function for you.

This makes your code: \* **Fast**: Each version is optimized for its specific data type. \* **Clear**: It's obvious what kind of data the function expects. \* **Safe**: If you call `mymean("hello")`, Julia will immediately throw a `MethodError` because you haven't defined a version of `mymean` that accepts a `String`.

## Q

*So this function cannot be called with a vector?*

## A

That's exactly right. **No, that specific function cannot be called with a vector.**

The signature `function mymean(x::Matrix)` strictly defines this version of `mymean` to *only* accept arguments of type `Matrix`.

If you try to call it with a vector, like `mymean([1, 2, 3])`, Julia will stop and give you a `MethodError`. This error basically means, “I looked for a function named `mymean` that accepts a `Vector`, but I couldn’t find one.”

### The Solution: Multiple Dispatch

This isn’t a problem; it’s the intended way Julia works! The solution is to simply define *another* function (a new “method”) with the same name but for the `Vector` type.

You would add this code:

```
"""
Calculates the mean of a 1D vector.
"""

function mymean(x::Vector)
    # The logic for a vector is simpler
    return sum(x) / length(x)
end
```

Now, your code has **two** methods for `mymean`:

1. `function mymean(x::Matrix)`
2. `function mymean(x::Vector)`

When you call `mymean(my_data)`, Julia will automatically check the type of `my_data` and call the correct version for you. This is **multiple dispatch**, and it’s what makes Julia code so clear and efficient.

## Q

*So, this means I have to make two function definitions?*

## A

Yes, if you want them to do two different things.

This isn’t a limitation; it’s the central design feature of Julia. You are defining two **methods** for the *same* generic function `mymean`.

### Why You Want Two Definitions

The two functions have fundamentally different behaviors and return types:

1. `mymean(x::Vector)`
  - **Input:** A 1D vector (e.g., 10 elements).
  - **Action:** Calculates the mean of all 10 elements.
  - **Output:** One single number (a scalar).
  
2. `mymean(x::Matrix)`
  - **Input:** A 2D matrix (e.g., 10 rows by 3 columns).
  - **Action:** Calculates the mean of *each column* separately.
  - **Output:** A **1x3 row vector** (one mean for each column).

Because the behavior (total mean vs. column-wise mean) is different, you provide a separate, optimized definition for each type.

### The “Julia Way”: Multiple Dispatch

This is called **multiple dispatch**. You are not creating two unrelated functions. You are adding two specific *methods* to the *single concept* of `mymean`.

When you call `mymean(data)`, Julia will:

1. Look at the **type** of `data`.
2. If `data` is a `Vector`, it dispatches to your vector version.
3. If `data` is a `Matrix`, it dispatches to your matrix version.

Your code would look like this:

```
# Method 1: For Vectors
function mymean(x::Vector)
    return sum(x) / length(x)
end

# Method 2: For Matrices (column-wise)
function mymean(x::Matrix)
    return sum(x, dims=1) / size(x, 1)
end

# --- Now you can use both! ---
v = [1, 2, 3, 4]
m = [1 10;
     2 20]

mymean(v)  # Calls Method 1, returns 2.5
mymean(m)  # Calls Method 2, returns [1.5 15.0]
```

---

### The Alternative (If You Wanted The *Same* Behavior)

What if you wanted the mean of *all* elements, whether it's a vector or a matrix?

Then you could write **one** function using **AbstractArray** (which is the parent type of both **Vector** and **Matrix**):

```
# This one function works for both
function my_total_mean(x::AbstractArray)
    return sum(x) / length(x)
end

my_total_mean(v) # Returns 2.5
my_total_mean(m) # Returns (1+2+10+20)/4 = 8.25
```

But since your goal was to get the **column-wise** mean for a matrix, you need the two separate definitions.