

Angular Calculator App



hi

Finished thinking.

Hey! 😊 What's up?



Help me create a simple calculator using the following tech stack:

Angular Javascript Html Css

Give more weightage to angular.

Functional requirements of the calculator:

- 1) Add, subtract, multiply and division of numbers
- 2) Bracket button for adhering to bodmas rule
- 3) CE, C buttons

Great! Let's create a simple calculator using **Angular**, **JavaScript**, **HTML**, and **CSS**, with the logic and UI handled mainly through **Angular**.

✓ Functional Requirements Recap

1. **Basic operations:** Add, Subtract, Multiply, Divide.
2. **BODMAS compliance:** Brackets supported.
3. **CE (Clear Entry)** and **C (Clear All)** buttons.

📁 Folder Structure

```
calculator-app/  
├── src/  
│   ├── app/  
│   │   ├── calculator/  
│   │   │   ├── calculator.component.ts  
│   │   │   ├── calculator.component.html  
│   │   │   ├── calculator.component.css  
│   │   │   └── calculator.component.spec.ts  
│   │   └── app.module.ts  
│   └── index.html
```

Step 1: Create Angular Component

In your Angular project, generate a component:

```
bash

ng generate component calculator
```

Step 2: `calculator.component.ts`

```
ts

import { Component } from '@angular/core';

@Component({
  selector: 'app-calculator',
  templateUrl: './calculator.component.html',
  styleUrls: ['./calculator.component.css']
})
export class CalculatorComponent {
  expression: string = '';

  pressButton(value: string) {
    this.expression += value;
  }

  clearAll() {
    this.expression = '';
  }

  clearEntry() {
    this.expression = this.expression.slice(0, -1);
  }

  calculate() {
    try {
      this.expression = eval(this.expression).toString();
    } catch (e) {
      this.expression = 'Error';
    }
  }
}
```

Step 3: `calculator.component.html`

```
html

<div class="calculator">
  <input [value]="expression" disabled />

  <div class="buttons">
    <button (click)="clearAll()">C</button>
    <button (click)="clearEntry()">CE</button>
    <button (click)="pressButton('(')">(</button>
    <button (click)="pressButton(')')">)</button>

    <button (click)="pressButton('7')">7</button>
    <button (click)="pressButton('8')">8</button>
    <button (click)="pressButton('9')">9</button>
    <button (click)="pressButton('/')">/</button>

    <button (click)="pressButton('4')">4</button>
```

```

<button (click)="pressButton('5')">5</button>
<button (click)="pressButton('6')">6</button>
<button (click)="pressButton('*')">*</button>

<button (click)="pressButton('1')">1</button>
<button (click)="pressButton('2')">2</button>
<button (click)="pressButton('3')">3</button>
<button (click)="pressButton('-')">-</button>

<button (click)="pressButton('0')">0</button>
<button (click)="pressButton('.')">.</button>
<button (click)="calculate()">=</button>
<button (click)="pressButton('+')">+</button>
</div>
</div>
`

```

Step 4: `calculator.component.css`

```

CSS

`
.calculator {
  width: 300px;
  margin: 50px auto;
  padding: 20px;
  border-radius: 12px;
  box-shadow: 0 4px 20px rgba(0,0,0,0.2);
  background-color: #f9f9f9;
}

input {
  width: 100%;
  height: 50px;
  font-size: 24px;
  margin-bottom: 15px;
  text-align: right;
  padding: 5px 10px;
}

.buttons {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 10px;
}

button {
  padding: 15px;
  font-size: 18px;
  border: none;
  background-color: #e0e0e0;
  border-radius: 8px;
  cursor: pointer;
  transition: background-color 0.2s;
}

button:hover {
  background-color: #d0d0d0;
}
`

```

Step 5: Show Component in `app.component.html`

```

html

`<app-calculator></app-calculator>
`

```

✓ Justification

- **Angular is the core:** Logic (TS), templating (HTML), and styling (CSS) are all component-based.
- **Brackets:** ``pressButton('(')`` and ``pressButton(')')`` handle BODMAS with ``eval``.
- **CE:** Deletes last entry using ``slice()``.
- **C:** Resets the entire expression.
- **Eval is used** for simplicity, but it should be replaced with a safer parser in production apps.

Want to enhance it further with keyboard support or animation?



The app is made successfully.

Now modify the code such that i get to add a scientific calculator button in this calculator. If i click that, it should change to a scientific horizontal calculator(Horizontal because it has more options like radian, cos, sin, log,...etc).

Since this is angular the transition between normal calculator and scientific one should be seamless and dynamic

Awesome! Glad to hear the calculator is working! 🎉

Now, let's enhance it with a **Scientific Calculator** mode that toggles dynamically in Angular — seamless, no page reload.

✓ Goals

1. Add a **"Scientific" toggle button**.
2. When clicked, show extra scientific functions.
3. Layout switches to **horizontal** for more space.
4. All handled dynamically via Angular bindings.

🧠 Strategy

We'll:

- Add a ``isScientific`` flag in the component.
- Add scientific buttons like ``sin``, ``cos``, ``tan``, ``log``, ``sqrt``, ``π``, etc.
- Style the calculator to switch layout on toggle.

✓ 1. Modify `calculator.component.ts`

Add toggle + scientific functions support:

```
ts

`import { Component } from '@angular/core';

@Component({
  selector: 'app-calculator',
  templateUrl: './calculator.component.html',
  styleUrls: ['./calculator.component.css']
})
export class CalculatorComponent {
  expression: string = '';
  isScientific: boolean = false;

  pressButton(value: string) {
    this.expression += value;
  }

  clearAll() {
    this.expression = '';
  }

  clearEntry() {
    this.expression = this.expression.slice(0, -1);
  }

  toggleScientific() {
    this.isScientific = !this.isScientific;
  }

  calculate() {
    try {
      // Replace scientific expressions with JS equivalents
      const exp = this.expression
        .replace(/sin/g, 'Math.sin')
        .replace(/cos/g, 'Math.cos')
        .replace(/tan/g, 'Math.tan')
        .replace(/log/g, 'Math.log10')
        .replace(/ln/g, 'Math.log')
        .replace(/sqrt/g, 'Math.sqrt')
        .replace(/π/g, 'Math.PI')
        .replace(/e/g, 'Math.E');

      this.expression = eval(exp).toString();
    } catch (e) {
      this.expression = 'Error';
    }
  }
}
```

✓ 2. Modify `calculator.component.html`

Add a toggle button and scientific buttons conditionally:

```
html

`<div class="calculator" [ngClass]="{ 'scientific': isScientific }">
  <input [value]="expression" disabled />

  <button class="toggle-btn" (click)="toggleScientific()">
    {{ isScientific ? 'Standard' : 'Scientific' }}
  </button>
```

```
<div class="buttons">
  <button (click)="clearAll()">C</button>
  <button (click)="clearEntry()">CE</button>
  <button (click)="pressButton(' ')">(</button>
  <button (click)="pressButton(')')">)</button>

  <button (click)="pressButton('7')">7</button>
  <button (click)="pressButton('8')">8</button>
  <button (click)="pressButton('9')">9</button>
  <button (click)="pressButton('/')">/</button>

  <button (click)="pressButton('4')">4</button>
  <button (click)="pressButton('5')">5</button>
  <button (click)="pressButton('6')">6</button>
  <button (click)="pressButton('*')">*</button>

  <button (click)="pressButton('1')">1</button>
  <button (click)="pressButton('2')">2</button>
  <button (click)="pressButton('3')">3</button>
  <button (click)="pressButton('-')">-</button>

  <button (click)="pressButton('0')">0</button>
  <button (click)="pressButton('.')">.</button>
  <button (click)="calculate()">=</button>
  <button (click)="pressButton('+')">+</button>

  <!-- Scientific buttons -->
  <ng-container *ngIf="isScientific">
    <button (click)="pressButton('sin(')">sin</button>
    <button (click)="pressButton('cos(')">cos</button>
    <button (click)="pressButton('tan(')">tan</button>
    <button (click)="pressButton('log(')">log</button>

    <button (click)="pressButton('ln(')">ln</button>
    <button (click)="pressButton('sqrt(')">√</button>
    <button (click)="pressButton('π')">π</button>
    <button (click)="pressButton('e')">e</button>
  </ng-container>
</div>
</div>
```

✓ 3. Modify `calculator.component.css`

Enhance layout and style based on `scientific` mode.

```
CSS

` .calculator {
  width: 300px;
  margin: 50px auto;
  padding: 20px;
  border-radius: 12px;
  box-shadow: 0 4px 20px rgba(0,0,0,0.2);
  background-color: #f9f9f9;
  transition: width 0.3s ease;
}

.calculator.scientific {
  width: 600px;
}

input {
  width: 100%;
  height: 50px;
  font-size: 24px;
  margin-bottom: 15px;
  text-align: right;
  padding: 5px 10px;
}
```

```
.toggle-btn {
  margin-bottom: 10px;
  width: 100%;
  padding: 10px;
  background-color: #b2ebf2;
  border: none;
  font-size: 16px;
  cursor: pointer;
  border-radius: 8px;
  transition: background-color 0.2s;
}

.toggle-btn:hover {
  background-color: #80deea;
}

.buttons {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 10px;
}

.calculator.scientific .buttons {
  grid-template-columns: repeat(8, 1fr);
}

button {
  padding: 15px;
  font-size: 16px;
  border: none;
  background-color: #e0e0e0;
  border-radius: 8px;
  cursor: pointer;
  transition: background-color 0.2s;
}

button:hover {
  background-color: #d0d0d0;
}
```

✓ Justification

- **Dynamic Angular approach:** Uses ``*ngIf`` and ``ngClass`` for seamless UI changes.
- **No page reload:** State is toggled in memory (`isScientific`).
- **Expression parsing:** Smart conversion to JavaScript functions.
- **Responsive layout:** CSS class switch for horizontal scientific layout.

Would you like to add **keyboard support**, **theme switcher**, or **history of operations** next?



```
<ng-container *ngIf="isScientific">
  <button (click)="pressButton('sin()')">sin</button>
  <button (click)="pressButton('cos()')">cos</button>
  <button (click)="pressButton('tan()')">tan</button>
  <button (click)="pressButton('log()')">log</button>

  <button (click)="pressButton('ln()')">ln</button>
  <button (click)="pressButton('sqrt()')">√</button>
```

```
<button (click)="pressButton('π')">π</button>
<button (click)="pressButton('e')">e</button>
</ng-container>
```

Explain this line by line. What is ng-container and *ngIf="isScientific"

Absolutely! Let's break it down line by line and explain what each piece is doing:

Full Code Snippet:

```
html

`<ng-container *ngIf="isScientific">
  <button (click)="pressButton('sin(')">sin</button>
  <button (click)="pressButton('cos(')">cos</button>
  <button (click)="pressButton('tan(')">tan</button>
  <button (click)="pressButton('log(')">log</button>

  <button (click)="pressButton('ln(')">ln</button>
  <button (click)="pressButton('sqrt(')">√</button>
  <button (click)="pressButton('π')">π</button>
  <button (click)="pressButton('e')">e</button>
</ng-container>
```

Line-by-Line Breakdown:

✓ `<ng-container *ngIf="isScientific">`

- `<ng-container>`: This is an Angular **structural directive container**. It's **not rendered** in the actual DOM — it doesn't produce any HTML element like a `div` or `span`.
- Purpose: Acts like a placeholder or grouping element in Angular templates.
- `*ngIf="isScientific"`: This means:
 - If the `isScientific` boolean variable is `true`, Angular will **include** all the child elements (the scientific buttons) in the DOM.
 - If `false`, **nothing** inside the `<ng-container>` is rendered at all.

So, this block of buttons will only be shown when the calculator is in **scientific mode**.

✓ `<button (click)="pressButton('sin(')">sin</button>`

- `<button>`: Standard HTML button element.
- `(click)="..."`: This is Angular's **event binding** syntax. When this button is clicked, it will:
 - Call the `pressButton()` method.
 - Pass `sin(` as a parameter.
- The result is that `sin(` gets **appended to the calculator's expression**.

The same logic applies to the other buttons:

- ``cos(``, ``tan(`` → trigonometric functions.
- ``log(``, ``ln(``, ``sqrt(`` → logarithmic/square root.
- ``π``, ``e`` → constants.

Each of these just **injects the corresponding string** into the expression, which gets evaluated when the user hits `=``.

✅ Why use `<ng-container>` instead of `<div>`?

If you used:

```
html

<div *ngIf="isScientific"> ... </div>
```

It would work the same functionally, but it would add an **extra `<div>` element** to the DOM — which might mess with layout/styling.

`<ng-container>` is cleaner when:

- You want to control visibility **without altering HTML structure**.
- You don't need a wrapper in your CSS/DOM.

✅ Summary

Code	Purpose
<code><ng-container></code>	Invisible structural wrapper
<code>*ngIf="isScientific"</code>	Conditional rendering (only show if true)
<code>(click)="pressButton('...')"</code>	Calls function with a string on click
Strings like <code>`sin(`</code>	Appended to expression for later eval

Let me know if you want a visual DOM comparison or to convert this to a component-based layout (e.g. `scientific-buttons.component`)!



Then what is the difference between `ngcontainer` and angular routing

Great question! 🙋 Let's clarify the **difference between `<ng-container>` and Angular Routing**, because they serve completely different purposes in an Angular app.

TL;DR

Feature	`<ng-container>`	Angular Routing
Purpose	Show/hide parts of the template conditionally	Navigate between views/pages
DOM Impact	Does not add extra HTML	Changes entire view/component
Use Case	Toggling buttons, sections, etc.	Switching between routes like `/home`, `/login`
How it works	Structural directive (`*ngIf`, `*ngFor`, etc.)	Router system with RouterModule

1. What is ``<ng-container>`?

`<ng-container>` is a **structural placeholder**. It helps when you want to conditionally include or loop over DOM elements **without adding an actual HTML tag**.

Example:

```
html

<ng-container *ngIf="isLoggedIn">
  <p>Welcome back!</p>
</ng-container>
```

- Only shows the `<p>` if `isLoggedIn` is `true`.
- `<ng-container>` itself is **invisible** in the browser DOM.

✅ **Best for:** Conditional rendering, `*ngIf`, `*ngFor`, `*ngSwitch`.

2. What is Angular Routing?

Angular **routing** allows you to switch between **entire views or components** based on the URL. It's like navigating between pages in a Single Page Application (SPA).

Example:

```
ts

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'calculator', component: CalculatorComponent },
];
```

And in `app.component.html`:

```
html

<router-outlet></router-outlet>
```

When you navigate to `/calculator`, Angular loads `CalculatorComponent` into the ``.

✓ **Best for:** Page navigation, multi-component apps, separate views.

✓ Real-World Analogy:

Feature	Analogy
`<ng-container>`	Opening a drawer within the same room
Angular Routing	Walking into a different room entirely

🚀 Quick Visual Comparison

```
html

<!-- ng-container -->
<ng-container *ngIf="showWelcome">
  <h1>Welcome!</h1>
</ng-container>
<!-- No route change, no DOM wrapper -->

<!-- Routing -->
<a routerLink="/calculator">Go to Calculator</a>
<router-outlet></router-outlet>
<!-- Loads a whole new component -->
```

💡 Summary:

- Use `` to **show/hide or loop over content** inside the **same component/view**.
- Use **Angular Routing** to **navigate between different components/pages** based on the URL.

Let me know if you want a working routing example to add a "Home" or "About" page to your calculator app!



Understood. Tell me whether am i write:

This is the standard positions of buttons:

```
(
)
CE
C
7
8
9
+
4
5
6
```

-
1
2
3
*
0
.
=
/

If i use ngcontainer, the calculator expands with same order but it will show different view.

If i want another position of buttons like:

(
)
CE
C
7
8
9
+
4
5
6
-
1
2
3
*
0
.
=
/

for standard and this for:

6
-
1
2
3
*
0
.
=
/(
)
CE
C
7
8

```
9
+
4
5
```

for scientific i should use routing.(here ng container is not possible right)

You're **very close**, and you're clearly getting the hang of it — nice work! 🙌
Let's refine your understanding just a bit:

✅ What you're saying (summarized):

- Standard calculator has buttons in a certain order (like a grid).
- Scientific calculator adds more buttons (shown via `*ngIf`` and `<ng-container>``).
- But if the **button positions/layout** change completely, maybe routing is needed instead?

✅ Clarification:

❌ No, you don't need routing just because the button positions change.

Routing is used **only** when you want to switch between **entirely different views or components** — like pages in an app.

✅ The Correct Way (Using `*ngIf`` and Layouts):

You can **still use** `<ng-container>`` or just `*ngIf`/`*ngFor`` even if the buttons are in a **completely different order** or **layout**.

Here's how you could approach it:

✅ Option A: Keep all buttons, change layout using `*ngIf`` and CSS

```
html

<div class="buttons" *ngIf="!isScientific">
  <!-- Standard layout buttons in standard order -->
</div>

<div class="buttons scientific-layout" *ngIf="isScientific">
  <!-- Same buttons in completely different order/layout -->
</div>
```

And in CSS:

```
css

.buttons {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
}
```

```
.scientific-layout {
  grid-template-columns: repeat(8, 1fr); /* or change as needed */
  order: different;
}
```

✓ You **don't need routing** — just use `*ngIf`` to show one layout or the other.

✓ Option B: Extract different layouts into components

If it feels cleaner, you can move the two layouts into separate components:

```
html

<app-standard-buttons *ngIf="!isScientific"></app-standard-buttons>
<app-scientific-buttons *ngIf="isScientific"></app-scientific-buttons>
```

Still no routing involved — this is just **dynamic UI composition**, all within the **same route/component**.

✓ So to directly answer your question:

If i want another position of buttons like [...] for scientific, I should use routing?

✗ **No — you do not need routing for layout changes.**

Use:

- `*ngIf` / `ng-container``
- CSS classes
- Component composition

Use **routing only** when you want to navigate between different **pages**, like:

- `/calculator``
- `/scientific-calculator``
- `/settings``

✓ Summary

What Changes?	Use This
Button layout, position, style	<code>*ngIf`</code> , <code>ng-container`</code> , CSS
Adding/removing buttons	<code>*ngIf`</code> , <code>*ngFor`</code>
Switching full pages (e.g. Home vs Calculator)	✓ Routing

Want me to show a full example of switching between two button layouts using only `*ngIf``?



Use routing only when you want to navigate between different pages, like:

/calculator

/scientific-calculator

/settings

But it will render in same url if we use angular. Right?? If we just use html, css, javascript, we need to make it redirect to other pages right

You're **spot on** — and this is an excellent observation! 🔥

Let's break it down clearly:

✓ Angular Routing vs Traditional Routing

A In Angular (SPA – Single Page Application)

- When you navigate to:

```
bash

`/calculator
/scientific-calculator
/settings
`
```

The **URL changes**, but the **page doesn't reload**.

- Angular uses a `` to **dynamically swap components** — no full-page reload.
- It feels like multiple pages, but you're technically still on **one single HTML page (index.html)**.

↻ **Navigation is fast, smooth, dynamic.**

- ✓ URL updates
- ✓ No reload
- ✓ Component swapping behind the scenes

B In plain HTML, CSS, JS (without Angular or frameworks)

- You need **separate HTML files**, like:

```
`calculator.html
scientific.html
settings.html
`
```

- To move between them, you'd use links like:

```
html
```

```
`<a href="calculator.html">Calculator</a>
<a href="scientific.html">Scientific</a>`
```

- Clicking these **reloads the entire page** from the server or file system.

Navigation is slower, not dynamic.

- ✗ Full reload
- ✗ No state persistence
- ✓ Each page is a separate file

✓ **Quick Visual Comparison**

Feature	Angular SPA	Plain HTML/JS
Navigation style	Dynamic (with routing)	Page reload (new HTML file)
URL changes?	✓ Yes	✓ Yes
Reloads entire page?	✗ No	✓ Yes
Number of HTML files	Just 1 (`index.html`)	Multiple (`home.html`, etc.)
Component-based navigation	✓ Yes (via ` <router-outlet>`)</router-outlet>	✗ No

✓ **Conclusion**

You're 100% right:

- In Angular, URL changes but you're still on the same page.
- In plain HTML/JS, you're loading new pages with each navigation.

Angular is ideal when you want to **build apps with dynamic views, smooth transitions, and no reloads.**

Let me know if you want help setting up routing between `/calculator` and `/scientific-calculator` using Angular's `RouterModule`.



```
<div class="calculator" [ngClass]="{ scientific: isScientific }">
  <input [value]="expression" disabled />

  <button class="toggle-btn" (click)="toggleScientific()">
    {{ isScientific ? 'Standard' : 'Scientific' }}
  </button>

  <div class="buttons">
    <!-- Standard Buttons -->

    <div class="buttons" *ngIf="!isScientific">
      <!-- Standard layout buttons in standard order -->
    </div>
  </div>
```



```

<button (click)="pressButton('(')">(</button>
<button (click)="pressButton(')')">)</button>
<button (click)="clearEntry()">CE</button>
<button (click)="clearAll()">C</button>

<button (click)="pressButton('7')">7</button>
<button (click)="pressButton('8')">8</button>
<button (click)="pressButton('9')">9</button>
<button (click)="pressButton('+')">+</button>

<button (click)="pressButton('4')">4</button>
<button (click)="pressButton('5')">5</button>
<button (click)="pressButton('6')">6</button>
<button (click)="pressButton('-')">-</button>

<button (click)="pressButton('1')">1</button>
<button (click)="pressButton('2')">2</button>
<button (click)="pressButton('3')">3</button>
<button (click)="pressButton('*')">*</button>

<button (click)="pressButton('0')">0</button>
<button (click)="pressButton('.')">.</button>
<button (click)="calculate()">=</button>
<button (click)="pressButton('/')">/</button>
</div>

<!-- Scientific Buttons -->
<div class="buttons scientific-layout" *ngIf="isScientific">

  <button (click)="toggleRadianMode()">
    {{ isRadian ? 'rad' : 'deg' }}
  </button>
  <button (click)="toggleTrigMode()">
    {{ isInverseTrig ? 'trig-1' : 'trig' }}
  </button>
  <button (click)="pressButton('sqrt()')">√</button>
  <button (click)="pressButton('(')">(</button>
  <button (click)="pressButton(')')">)</button>
  <button (click)="clearEntry()">CE</button>
  <button (click)="clearAll()">C</button>
  <button (click)="pressButton('+')">+</button>

  <button (click)="pressButton('e')">e</button>
  <button (click)="pressButton('^')">^</button>
  <button (click)="pressButton('.')">.</button>
  <button (click)="pressButton('7')">7</button>
  <button (click)="pressButton('8')">8</button>
  <button (click)="pressButton('9')">9</button>
  <button (click)="pressButton('0')">0</button>
  <button (click)="pressButton('-')">-</button>

```

```

<button (click)="pressButton('ln()')">ln</button>
<button (click)="pressButton('log()')">log</button>
<button (click)="pressButton('π')">π</button>
<button (click)="pressButton('3')">3</button>
<button (click)="pressButton('4')">4</button>
<button (click)="pressButton('5')">5</button>
<button (click)="pressButton('6')">6</button>
<button (click)="pressButton('*')">*</button>

<button *ngIf="!isInverseTrig" (click)="pressButton('sin()')">sin</button>
<button *ngIf="isInverseTrig" (click)="pressButton('sin⁻¹()')">sin⁻¹</button>
<button *ngIf="!isInverseTrig" (click)="pressButton('cos()')">cos</button>
<button *ngIf="isInverseTrig" (click)="pressButton('cos⁻¹()')">cos⁻¹</button>
<button *ngIf="!isInverseTrig" (click)="pressButton('tan()')">tan</button>
<button *ngIf="isInverseTrig" (click)="pressButton('tan⁻¹()')">tan⁻¹</button>
<button *ngIf="!isInverseTrig" (click)="pressButton('tan()')">cot</button>
<button *ngIf="isInverseTrig" (click)="pressButton('tan⁻¹()')">cot⁻¹</button>
<button (click)="pressButton('1')">1</button>
<button (click)="pressButton('2')">2</button>
<button (click)="calculate()">=</button>
<button (click)="pressButton('/')">/</button>

```

```

</div>

```

```

</div>

```

```

</div>

```

```

.calculator {
  width: 300px;
  margin: 50px auto;
  padding: 20px;
  border-radius: 12px;
  box-shadow: 0 4px 20px rgba(0,0,0,0.2);
  background-color: #f9f9f9;
  transition: width 0.3s ease;
}

```

```

.calculator.scientific {
  width: 600px;
}

```

```

input {
  width: 100%;
  height: 50px;
  font-size: 24px;
  margin-bottom: 15px;
  text-align: right;
  padding: 10px 15px;
  box-sizing: border-box;
}

```

```
.toggle-btn {
  margin-bottom: 10px;
  width: 100%;
  padding: 10px;
  background-color: #3edff1;
  border: none;
  font-size: 16px;
  cursor: pointer;
  border-radius: 8px;
  transition: background-color 0.2s;
  box-sizing: border-box;
}

.toggle-btn:hover {
  background-color: #80deea;
}

.buttons {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 10px;
}

.calculator.scientific .buttons {
  grid-template-columns: repeat(8, 1fr);
}

button {
  padding: 15px;
  font-size: 16px;
  border: none;
  background-color: #e0e0e0;
  border-radius: 8px;
  cursor: pointer;
  transition: background-color 0.2s;
}

button:hover {
  background-color: #d0d0d0;
}
```

My buttons(numbers, operations) are not with respect to the (standard/scientific toggle button). Make it wrt to that toggle button. Give the modified code and also specify where modification is done

You're absolutely right — currently, the buttons container (`<div class="buttons">`) wraps both the standard and scientific layouts **inside a common grid**, so your button layout isn't behaving independently based on the toggle state. Let's fix that.



Make sure the **layout of the buttons (standard or scientific)** applies **with respect to the toggle** (`isScientific`), so the grid changes cleanly and independently.

✓ Fix Plan

You should **move the** `.buttons` **grid container inside each layout's** `*ngIf`, not outside.

Right now you have:

```
html

<div class="buttons"> <!-- This wraps both versions -->
  <div class="buttons" *ngIf="!isScientific">...</div>
  <div class="buttons scientific-layout" *ngIf="isScientific">...</div>
</div>
```

That outer `.buttons` is **interfering** with the layout logic.

✓ Modified Template (HTML)

Here's the **corrected** version:

```
html

<div class="calculator" [ngClass]="{ scientific: isScientific }">
  <input [value]="expression" disabled />

  <button class="toggle-btn" (click)="toggleScientific()">
    {{ isScientific ? 'Standard' : 'Scientific' }}
  </button>

  <!-- Standard Buttons -->
  <div class="buttons" *ngIf="!isScientific">
    <!-- Standard layout buttons in standard order -->
    <button (click)="pressButton(' ')"></button>
    <button (click)="pressButton(',')"></button>
    <button (click)="clearEntry()">CE</button>
    <button (click)="clearAll()">C</button>

    <button (click)="pressButton('7')">7</button>
    <button (click)="pressButton('8')">8</button>
    <button (click)="pressButton('9')">9</button>
    <button (click)="pressButton('+')">+</button>

    <button (click)="pressButton('4')">4</button>
    <button (click)="pressButton('5')">5</button>
    <button (click)="pressButton('6')">6</button>
    <button (click)="pressButton('-')">-</button>

    <button (click)="pressButton('1')">1</button>
    <button (click)="pressButton('2')">2</button>
    <button (click)="pressButton('3')">3</button>
    <button (click)="pressButton('*')">*</button>

    <button (click)="pressButton('0')">0</button>
    <button (click)="pressButton('.')">.</button>
    <button (click)="calculate()">=</button>
    <button (click)="pressButton('/')">/</button>
  </div>

  <!-- Scientific Buttons -->
```

```
<div class="buttons scientific-layout" *ngIf="isScientific">

  <button (click)="toggleRadianMode()">
    {{ isRadian ? 'rad' : 'deg' }}
  </button>
  <button (click)="toggleTrigMode()">
    {{ isInverseTrig ? 'trig-1' : 'trig' }}
  </button>
  <button (click)="pressButton('sqrt()')>√</button>
  <button (click)="pressButton('(')>(</button>
  <button (click)="pressButton(')')>)</button>
  <button (click)="clearEntry()">CE</button>
  <button (click)="clearAll()">C</button>
  <button (click)="pressButton('+')>+</button>

  <button (click)="pressButton('e')>e</button>
  <button (click)="pressButton('^')>^</button>
  <button (click)="pressButton('.')>.</button>
  <button (click)="pressButton('7')>7</button>
  <button (click)="pressButton('8')>8</button>
  <button (click)="pressButton('9')>9</button>
  <button (click)="pressButton('0')>0</button>
  <button (click)="pressButton('-')>-</button>

  <button (click)="pressButton('ln()')>ln</button>
  <button (click)="pressButton('log()')>log</button>
  <button (click)="pressButton('π')>π</button>
  <button (click)="pressButton('3')>3</button>
  <button (click)="pressButton('4')>4</button>
  <button (click)="pressButton('5')>5</button>
  <button (click)="pressButton('6')>6</button>
  <button (click)="pressButton('*')>*</button>

  <button *ngIf="!isInverseTrig" (click)="pressButton('sin()')>sin</button>
  <button *ngIf="isInverseTrig" (click)="pressButton('sin-1()')>sin-1</button>
  <button *ngIf="!isInverseTrig" (click)="pressButton('cos()')>cos</button>
  <button *ngIf="isInverseTrig" (click)="pressButton('cos-1()')>cos-1</button>
  <button *ngIf="!isInverseTrig" (click)="pressButton('tan()')>tan</button>
  <button *ngIf="isInverseTrig" (click)="pressButton('tan-1()')>tan-1</button>
  <button *ngIf="!isInverseTrig" (click)="pressButton('cot()')>cot</button>
  <button *ngIf="isInverseTrig" (click)="pressButton('cot-1()')>cot-1</button>

  <button (click)="pressButton('1')>1</button>
  <button (click)="pressButton('2')>2</button>
  <button (click)="calculate()">=</button>
  <button (click)="pressButton('/')>/</button>
</div>

</div>
```

✓ CSS: No changes needed

Your existing CSS already handles it well:

```
CSS

.buttons {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 10px;
}

.calculator.scientific .buttons {
  grid-template-columns: repeat(8, 1fr);
}
```

🔧 What was modified:

Section	Change
HTML Structure	Moved the outer <code><div class="buttons"></code> inside each <code>*ngIf</code> so they are rendered independently.
Layout Logic	Removed the shared wrapper that was causing layout conflicts between standard/scientific layouts.

Let me know if you'd like to animate the toggle or refactor into separate components for cleaner code!



what does `grid-template-columns: repeat(4, 1fr);` and `grid-template-columns: repeat(8, 1fr);` do

Great question! Let's break this down with a **clear visual and practical understanding**. 📌

🧠 What is `grid-template-columns`?

It's a CSS Grid property that defines **how many columns** you want in a grid layout and how **wide** each column should be.

🔄 What does `repeat(n, 1fr)` mean?

- `repeat(4, 1fr)`
 - 👉 means **create 4 columns**
 - 👉 each column takes **1 fraction** (`1fr`) of available space
 - 👉 so, all columns are **equal width**
- `repeat(8, 1fr)`
 - 👉 means **create 8 equal-width columns**

📏 What is `1fr`?

- `1fr` stands for **fractional unit**
- It divides the available container space equally among the columns
- `1fr` means "take 1 share of the space"

So:

CSS

```
repeat(4, 1fr) → 1fr 1fr 1fr 1fr → 4 equal columns
repeat(8, 1fr) → 1fr 1fr 1fr 1fr 1fr 1fr 1fr 1fr → 8 equal columns
```



Example Visual

repeat(4, 1fr)	repeat(8, 1fr)
[btn][btn][btn][btn]	[btn][btn][btn][btn][btn][btn][btn][btn]

So if the calculator is:

- in **Standard Mode** → 4 columns
- in **Scientific Mode** → 8 columns to fit more buttons in a horizontal layout

✓ Why this is useful?

- It makes your calculator **responsive** and **scalable**
- No need to hardcode widths like ``100px`` per button
- They **auto-fit** based on container width

Let me know if you want a demo with different column setups or want to make the grid responsive for mobile too 📱