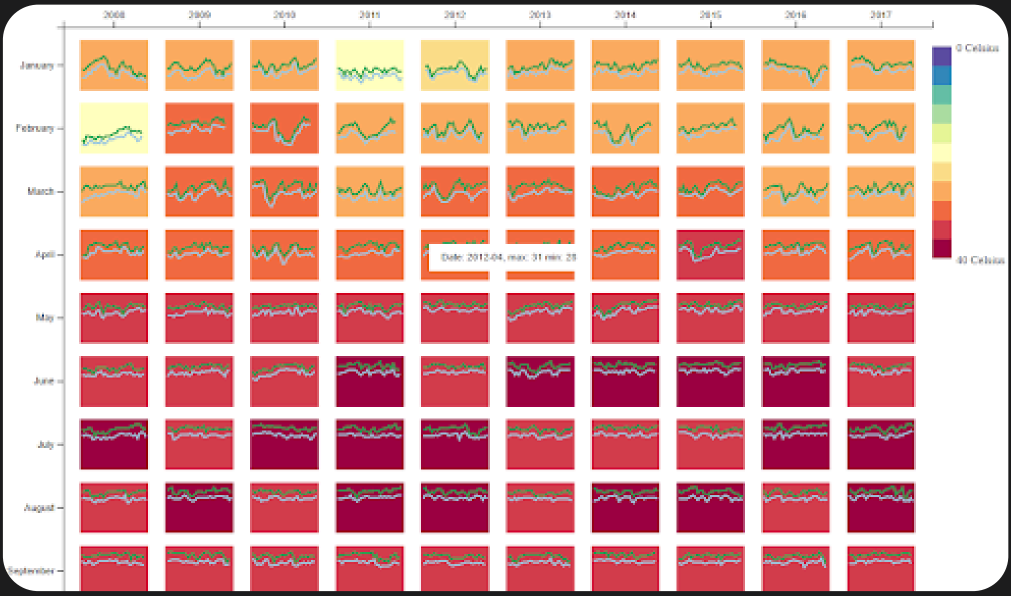
Visualization Project Help



I want my output visualization to look like the example that we were given, image attached. How do I setup D3.js for this project.

To get started with your D3.js matrix visualization, you'll need to set up a solid foundation consisting of an HTML file to load the library and hold your visual elements, a CSS section for styling (especially for things like the hover tooltip), and a JavaScript file where the D3 logic will live.

Here is the best way to structure your initial setup:

1. The HTML Skeleton (`index.html`)

You will want to pull in D3.js (version 7 is a great choice) via a CDN. You also need specific `<div>` containers where D3 will inject the SVG matrix and the tooltip.

HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-wi
  <title>Monthly Temperature - Matrix View</titl

  <script src="https://d3js.org/d3.v7.min.js"></

  <style>
    /* We will add specific styles here next */
  </style>
</head>
<body>
  <h1>Hong Kong Monthly Temperature (last 10 yea
  <p id="instruction">Click the matrix to switch

  <div id="container">
    <div id="matrix-wrap"></div>
  </div>

  <div id="tooltip" class="tooltip"></div>

  <script src="main.js"></script>
</body>
</html>
```

2. Essential CSS Styling

Inside the `<style>` tags of your HTML (or in a separate CSS file), you need to define how the container and the tooltip behave. The tooltip is especially important because it needs absolute positioning to follow the user's mouse.

CSS

```css
body {
  font-family: system-ui, sans-serif;
  margin: 1rem 2rem;
  background: #f8f9fa;
```

```
}

#container {
  background: #fff;
  padding: 1rem;
  border-radius: 8px;
  box-shadow: 0 1px 3px rgba(0,0,0,0.08);
}

/* Tooltip setup */
.tooltip {
  position: absolute;
  padding: 6px 10px;
  background: rgba(0,0,0,0.85);
  color: #fff;
  font-size: 12px;
  border-radius: 4px;
  pointer-events: none; /* Prevents the tooltip f
  opacity: 0;
  transition: opacity 0.15s;
}

.tooltip.visible {
  opacity: 1;
}

/* Base styles for the mini line charts inside th
.cell-line {
  fill: none;
  stroke-width: 1.5;
  vector-effect: non-scaling-stroke;
}
.cell-line-max { stroke: #2e7d32; } /* Green for
.cell-line-min { stroke: rgba(255,255,255,0.85);
```

3. The JavaScript Entry Point (`main.js`)

In your JavaScript file, it's good practice to wrap your code in an Immediately Invoked Function Expression (IIFE) to avoid polluting the global namespace. Then, set up your basic layout constants and load the data.

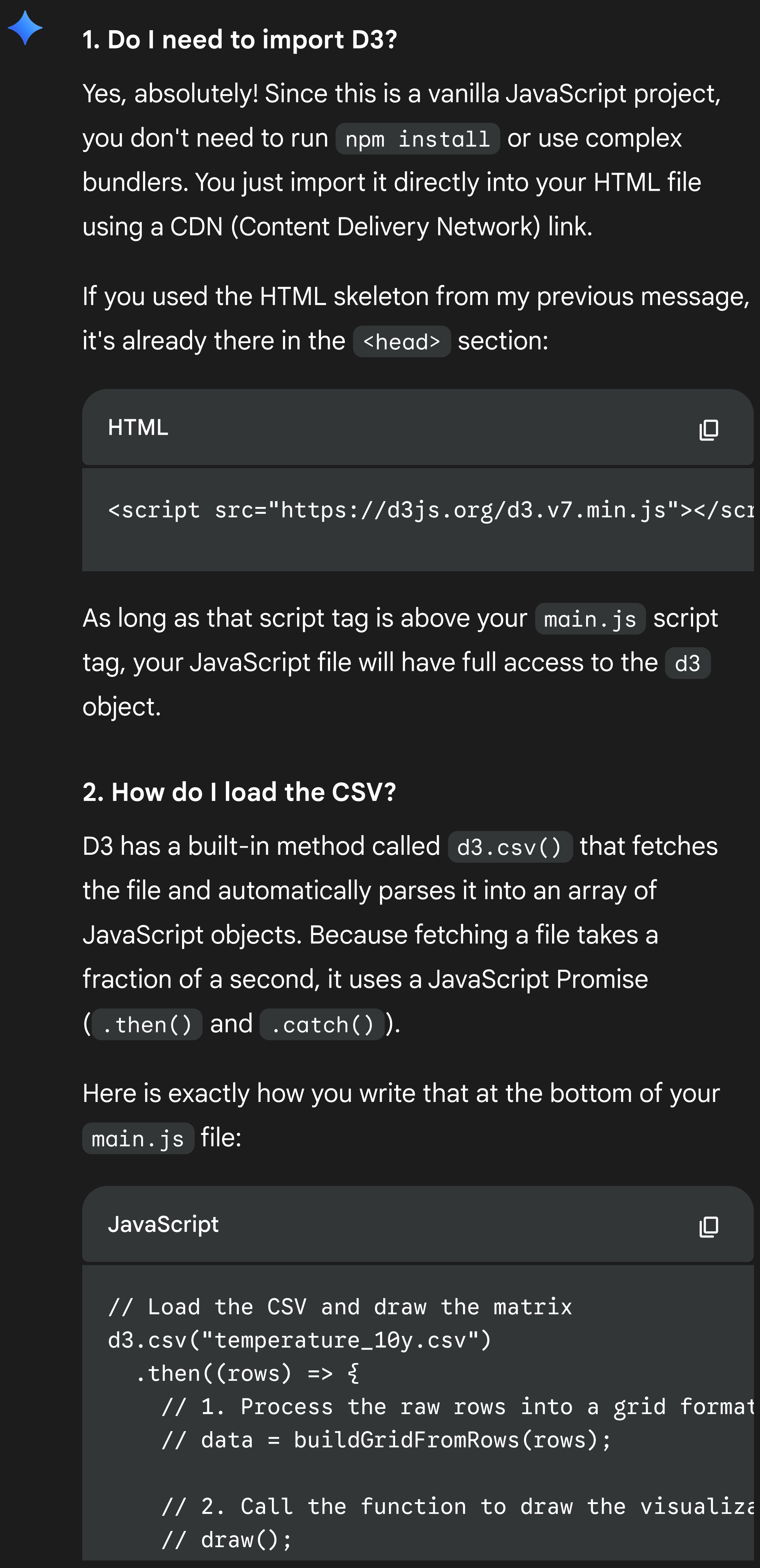
JavaScript

```
(function () {
  "use strict";

  // --- Constants & Layout Setup ---
  const cellWidth = 72;
  const cellHeight = 52;
  const paddingLeft = 80;
  const paddingTop = 40;
  const paddingRight = 20;
  const paddingBottom = 30;

  // State variables
  let useMax = true; // Tracks whether we are sho
  let data = null;

  const tooltip = d3.select("#tooltip");

  // Load the CSV data
  d3.csv("temperature_10y.csv").then((rows) => {
    console.log("Data loaded successfully!", rows
    // Next steps: process 'rows' and draw the gr
  }).catch((error) => {
    console.error("Error loading the CSV file:",
  });

})();
```

Once you have these files set up in the same folder as your `temperature_10y.csv` file, opening `index.html` in your browser will give you the basic layout and log the first row of your data to the console.

Would you like to move on to processing that CSV data into the format we need for the grid cells, or would you prefer to set up the SVG canvas and scales first?

I already manually proceessed the csv data by removing all data we dont need. Its read to go. Do I need to import D3? How do I load the csv?



1. Do I need to import D3?

Yes, absolutely! Since this is a vanilla JavaScript project, you don't need to run `npm install` or use complex bundlers. You just import it directly into your HTML file using a CDN (Content Delivery Network) link.

If you used the HTML skeleton from my previous message, it's already there in the `<head>` section:

HTML

```html
<script src="https://d3js.org/d3.v7.min.js"></scr
```

As long as that script tag is above your `main.js` script tag, your JavaScript file will have full access to the `d3` object.

2. How do I load the CSV?

D3 has a built-in method called `d3.csv()` that fetches the file and automatically parses it into an array of JavaScript objects. Because fetching a file takes a fraction of a second, it uses a JavaScript Promise (`.then()` and `.catch()`).

Here is exactly how you write that at the bottom of your `main.js` file:

JavaScript

```javascript
// Load the CSV and draw the matrix
d3.csv("temperature_10y.csv")
  .then((rows) => {
    // 1. Process the raw rows into a grid format
    // data = buildGridFromRows(rows);

    // 2. Call the function to draw the visualiza
    // draw();
```

```
      console.log("Data loaded!", rows);
    })
    .catch((error) => {
      // Fallback if the CSV is missing or fails tc
      document.getElementById("matrix-wrap").innerH
        "<p>Error loading temperature_10y.csv. Make
    });
```

What happens next?

Even though you've cleaned up the CSV, D3 loads it as a flat list of daily records (e.g., `[{date: "2008-01-01", max_temperature: "17", min_temperature: "10"}, ...]`).

For a matrix visualization, we need to group this flat list by **Year** (for the x-axis) and **Month** (for the y-axis), and calculate the highest and lowest temperatures for each month to color the cells.

Would you like me to show you how to write the `buildGridFromRows` function to group the data perfectly for our matrix?

provide the skeleton code I need for all functions of matrix visualization in main.js

Here is the skeleton for your `main.js` file. This structure sets up all the constants, state variables, and empty functions you will need to build the complete matrix visualization, including the mini-line charts and the interactive legend.

You can copy and paste this directly into your `main.js` file to serve as your blueprint.

JavaScript

```
/**
 * Hong Kong Monthly Temperature matrix (last 10
 * Click a cell to toggle max/min; hover for tool
 */
(function () {
  "use strict";

  // --- 1. Constants & Layout Setup ---
  // We explicitly define the years and months t
  const YEARS = [2008, 2009, 2010, 2011, 2012, 20
  const MONTHS = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  const MONTH_NAMES = [
    "January", "February", "March", "April", "Ma
    "July", "August", "September", "October", "N
  ];

  // Sizing for individual cells and the overall
  const cellWidth = 72;
  const cellHeight = 52;
  const linePadding = 4;
  const paddingLeft = 80;
  const paddingTop = 40;
  const paddingRight = 20;
  const paddingBottom = 30;
  const legendBarWidth = 24;
  const legendGap = 16;

  // --- 2. State Variables ---
  let useMax = true; // Tracks whether to color
  let data = null;   // Will hold our processed
  let matrixG = null; // Will hold the main SVG

  // D3 selections and scales
  const tooltip = d3.select("#tooltip");

  // A sequential color scale (Yellow to Orange
  const colorScale = d3.scaleSequential(d3.interp

  // --- 3. Core Functions ---
  /** * Function 1: Data Processing
   * Takes the flat array of CSV rows and groups
   * Calculates the absolute max and min for the
   * Returns an array of cell objects ready for D
   */
  function buildGridFromRows(rows) {
    // TODO: Parse dates, roll up the data by yea
  }

  /** * Function 2: Path Generator for Mini-Line
   * Generates the SVG "d" attribute string for c
   * Scales the values to fit perfectly within th
   */
  function linePath(values, tempMin, tempMax) {
    // TODO: Create scale functions for X and Y i
  }

  /** * Function 3: The Main Draw Routine
   * Sets up the SVG, draws the axes, creates the
   * appends the mini-lines, and builds the color
   */
  function draw() {
    // TODO: Calculate total dimensions, append S
    // TODO: Create X (Years) and Y (Months) scal
    // TODO: Draw rectangles for cells with click
    // TODO: Draw min/max mini-lines inside each
    // TODO: Draw the vertical color legend
  }

  // --- 4. Execution: Load CSV and trigger the b
  d3.csv("temperature_10y.csv")
    .then((rows) => {
      // Step 1: Process the raw rows into our gr
      data = buildGridFromRows(rows);

      // Step 2: Draw the visualization
      draw();
    })
    .catch(() => {
      // Fallback message if the CSV is missing
      document.getElementById("matrix-wrap").inne
        "<p>Run prepare_data.py to generate tempe
    });
})();
```

Would you like me to walk you through the logic for `buildGridFromRows` so we can get your flat CSV data grouped into the correct matrix structure, or would you rather start by setting up the SVG canvas inside the `draw` function?

Walk me through all of the functions

Absolutely. Let's walk through the exact logic needed for each of the three core functions. We will start with the data processing, move to the line generator, and finish with the main drawing routine.

1. `buildGridFromRows(rows)`

The Goal: D3 loads CSV data as a flat list of days. We need to group these days by **Year** (columns) and **Month** (rows). For each month, we need to find the absolute maximum and minimum temperatures, and keep a sorted list of daily temperatures so we can draw the mini-line charts.

The Logic:

1. **Parse Dates:** Convert the string dates (e.g., "2008-01-01") into JavaScript Date objects using `d3.timeParse("%Y-%m-%d")`.
2. **Group Data:** Use `d3.rollup` to group the rows first by the year, then by the month.
3. **Calculate Min/Max:** Inside the rollup, sort the days in chronological order, then use `d3.max` and `d3.min` to find the highest and lowest temperatures for that specific month.
4. **Format into Cells:** Loop through our `YEARS` and `MONTHS` arrays to create a clean, flat array of "cell"

objects. Each cell knows its exact grid coordinates (`xi` and `yi`).

Here is how you write it:

JavaScript

```
function buildGridFromRows(rows) {
  const parseDate = d3.timeParse("%Y-%m-%d");

  // 1. Convert strings to numbers and parse date
  const raw = rows.map((r) => ({
    date: parseDate(r.date),
    max: +r.max_temperature,
    min: +r.min_temperature,
  })).filter((d) => d.date);

  // 2 & 3. Group by Year -> Month, and calculate
  const byYearMonth = d3.rollup(
    raw,
    (group) => {
      // Sort days chronologically
      const sorted = group.sort((a, b) => a.date.
      return {
        month_max: d3.max(sorted, (d) => d.max),
        month_min: d3.min(sorted, (d) => d.min),
        daily: sorted.map((d) => ({ max: d.max, m
      };
    },
    (d) => d.date.getFullYear(),
    (d) => d.date.getMonth() + 1
  );

  // 4. Build the final array of cell objects
  const cells = [];
  YEARS.forEach((year, xi) => {
    MONTHS.forEach((month, yi) => {
      const value = byYearMonth.get(year)?.get(mo
      if (value) {
        // We save xi and yi so we know exactly w
        cells.push({ year, month, xi, yi, ...valu
      }
    });
  });
  return cells;
}
```

---

2. `linePath(values, tempMin, tempMax)`

The Goal: Inside each rectangle, we want a tiny line chart showing the daily temperatures. This function takes an array of temperatures and calculates the exact `x` and `y` pixel coordinates to draw the SVG line within the cell.

The Logic:

- Calculate the drawing width and height by taking the cell dimensions and subtracting `linePadding` so the line doesn't touch the edges.
- The `scaleX` distributes the points evenly from left to right based on their index.
- The `scaleY` maps the temperature to the height of the cell (remember, in SVG, `y=0` is at the top, so higher temperatures need a lower `y` value).
- Return a `d3.line()` string.

Here is the implementation:

JavaScript

```javascript
function linePath(values, tempMin, tempMax) {
  if (values.length < 2) return null;

  const w = cellWidth - 2 * linePadding;
  const h = cellHeight - 2 * linePadding;
  const range = tempMax - tempMin || 1; // Prever

  const scaleY = (v) => h - ((v - tempMin) / rang
  const scaleX = (i) => (i / (values.length - 1))

  return d3.line()
    .x((_, i) => linePadding + scaleX(i))
    .y((v) => linePadding + scaleY(v))(values);
}
```

---

3. `draw()`

The Goal: This is the heavy lifter. It sets up the SVG canvas, draws the labels, loops through our cell data to draw the rectangles and lines, and handles interactions (clicks and hover tooltips).

The Logic Breakdown:

1. **Canvas Setup:** Calculate total width and height (grid size + padding + legend space). Append an `<svg>` to `#matrix-wrap`.
2. **Scales:** Create `d3.scaleBand()` for the X axis (Years) and Y axis (Months).
3. **Labels:** Append text elements for the top (Years) and left (Months).
4. **Cells (The Matrix):** * Bind our cell data to `<g>` (group) elements and translate them into position using their `xi` and `yi` coordinates.
    - Append a `<rect>` to each group. Color it using our `colorScale`.
    - Add `.on("click")` to toggle the `useMax` variable and recolor all rectangles.
    - Add `.on("mouseover")` to update the tooltip's HTML and make it visible.
5. **Mini-lines:** Loop through each cell's `daily` data, figure out the absolute min and max, and append two `<path>` elements using our `linePath` function.
6. **Legend:** Draw a rectangle with a linear gradient on the right side and add a Y-axis to it.

Here is the code to make it happen:

JavaScript

```javascript
function draw() {
  const cells = data;
  const gridWidth = YEARS.length * cellWidth;
  const gridHeight = MONTHS.length * cellHeight;

  // Calculate total canvas size
  const totalWidth = paddingLeft + gridWidth + le
  const totalHeight = paddingTop + gridHeight + p

  // Clear any existing drawing, then append the
  d3.select("#matrix-wrap").selectAll("*").remove
  const svg = d3.select("#matrix-wrap")
    .append("svg")
    .attr("width", totalWidth)
    .attr("height", totalHeight);

  // Band scales for exact grid positioning
  const xScale = d3.scaleBand().domain(YEARS.map(
  const yScale = d3.scaleBand().domain(MONTHS.map

  // --- Draw X Axis (Years) ---
  const yearLabels = svg.append("g")
    .attr("transform", `translate(${paddingLeft},
  yearLabels.selectAll("text").data(YEARS).join('
    .attr("x", (y) => xScale(String(y)) + xScale.
    .attr("y", 0)
    .attr("text-anchor", "middle")
    .attr("font-size", 10)
    .text((y) => y);

  // --- Draw Y Axis (Months) ---
  const monthLabels = svg.append("g")
    .attr("transform", `translate(${paddingLeft -
  monthLabels.selectAll("text").data(MONTHS).join
    .attr("x", 0)
    .attr("y", (m) => yScale(String(m)) + yScale.
    .attr("text-anchor", "end")
    .attr("dominant-baseline", "middle")
    .attr("font-size", 10)
    .text((m) => MONTH_NAMES[m - 1]);
```

```
// --- Draw Matrix Cells ---
matrixG = svg.append("g")
  .attr("transform", `translate(${paddingLeft},

const cell = matrixG.selectAll(".cell").data(ce
  .attr("class", "cell")
  .attr("transform", (d) => `translate(${d.xi *

// The actual colored rectangles
cell.append("rect")
  .attr("class", "cell-rect")
  .attr("width", cellWidth)
  .attr("height", cellHeight)
  .attr("fill", (d) => colorScale(useMax ? d.mo
  .attr("stroke", "rgba(0,0,0,0.12)")
  .attr("stroke-width", 1)
  .style("cursor", "pointer")
  .on("click", () => {
    useMax = !useMax; // Toggle state
    matrixG.selectAll(".cell-rect") // Recolor
      .attr("fill", (d) => colorScale(useMax ?
  })
  .on("mouseover", function (event, d) {
    const dateStr = `${d.year}-${String(d.month
    tooltip
      .style("left", event.pageX + 10 + "px")
      .style("top", event.pageY + 10 + "px")
      .html(`Date: ${dateStr}, max: ${d.month_m
      .classed("visible", true);
  })
  .on("mouseout", () => tooltip.classed("visibl

// --- Draw Mini Lines in each cell ---
cell.each(function (d) {
  const daily = d.daily || [];
  if (daily.length < 2) return;

  const maxTemps = daily.map((x) => x.max);
  const minTemps = daily.map((x) => x.min);
  const tempMin = Math.min(...minTemps);
  const tempMax = Math.max(...maxTemps);

  const g = d3.select(this);
  // Max line (green)
  g.append("path")
    .attr("class", "cell-line cell-line-max")
```

```
      .attr("d", linePath(maxTemps, tempMin, temp
    // Min line (white)
    g.append("path")
      .attr("class", "cell-line cell-line-min")
      .attr("d", linePath(minTemps, tempMin, temp
  });

  // --- Draw Color Legend ---
  const legendG = svg.append("g")
    .attr("transform", `translate(${paddingLeft

  const gradient = legendG.append("defs").append(
    .attr("id", "legend-gradient")
    .attr("x1", "0%").attr("x2", "0%")
    .attr("y1", "0%").attr("y2", "100%");

  // Create 6 color stops from 0 to 40 degrees
  for (let i = 0; i <= 5; i++) {
    gradient.append("stop")
      .attr("offset", (i / 5) * 100 + "%")
      .attr("stop-color", colorScale(i * 8));
  }

  legendG.append("rect")
    .attr("width", legendBarWidth)
    .attr("height", gridHeight)
    .attr("fill", "url(#legend-gradient)")
    .attr("rx", 3);

  const legendAxis = d3.scaleLinear().domain([0,
  legendG.append("g")
    .attr("transform", `translate(${legendBarWidt
    .call(d3.axisRight(legendAxis).ticks(5).tickS
}
```

With these functions filled in, your skeleton becomes a fully working visualizer! Try plugging these into `main.js`. Do you have any questions about how the D3 scales (`scaleBand`, `scaleLinear`) work, or how the tooltips track the mo

how can I add some more custom colors to the legend

Changing the colors is one of the most impactful ways to customize your visualization. Right now, you are using `d3.interpolateYlOrRd`, which is a built-in preset (Yellow-Orange-Red).

To use your own custom colors, you should switch from `d3.scaleSequential` to `d3.scaleLinear`. This allows you to define specific "stops" where a temperature maps to an exact color, and D3 will automatically blend (interpolate) the colors between them.

Here is how to modify your `main.js`:

1. Find your current `colorScale`

Look for this line near the top of your file:

JavaScript

```javascript
const colorScale = d3.scaleSequential(d3.interpol
```

2. Replace it with a custom linear scale

You can pass an array of numbers to `.domain()` and a matching array of colors to `.range()`.

For example, if you want a "Cold to Hot" look (Blue → White → Red):

JavaScript

```javascript
// Map 0°C to Blue, 20°C to White, and 40°C to Re
const colorScale = d3.scaleLinear()
  .domain([0, 20, 40])
  .range(["#2c7bb6", "#ffffbf", "#d7191c"]);
```

Or, if you want more "steps" for richer detail (Deep Blue → Light Blue → Yellow → Orange → Red):

JavaScript

```
const colorScale = d3.scaleLinear()
  .domain([0, 10, 20, 30, 40])
  .range(["#053061", "#4393c3", "#f4a582", "#d66
```

3. Do I need to update the legend code?

No, you don't! That's the beauty of how we set up the `draw()` function.

Your legend generation code looks like this:

JavaScript

```
for (let i = 0; i <= 5; i++) {
  gradient.append("stop")
    .attr("offset", (i / 5) * 100 + "%")
    .attr("stop-color", colorScale(i * 8)); // Sa
}
```

Because it asks `colorScale` for the color at specific temperatures (0, 8, 16, etc.), it will automatically pick up whatever new colors you defined in Step 2.

Summary

Just swap that one `const colorScale` definition, and both your matrix cells and your legend will instantly update to the new theme. Try experimenting with hex codes to find a palette you like!