

# JpGraph — Architecture Overview

---

⚠ **Historic document.** This describes the architecture of JpGraph v3.1.6p (January 2010), written for PHP 5.x.

## Table of Contents

- [JpGraph — Architecture Overview](#)
  - [Table of Contents](#)
  - [Design Philosophy](#)
  - [Layered Architecture](#)
    - [Layer 1 — GD Extension](#)
    - [Layer 2 — Image Abstraction \(`gd\_image.inc.php`\)](#)
    - [Layer 3 — Graph Framework \(`jpgraph.php`\)](#)
    - [Layer 4 — Plot Modules](#)
  - [The Rendering Pipeline](#)
  - [Class Hierarchy](#)
  - [The Auto-Scaling Algorithm](#)
  - [The Contour Plot Algorithm](#)
    - [Input](#)
    - [Algorithm Steps](#)
      - [Step 1: Perturbation](#)
      - [Step 2: Edge Crossing Detection](#)
      - [Step 3: Cell Analysis](#)
      - [Step 4: Coordinate Interpolation](#)
      - [Output](#)
    - [Color Assignment](#)
  - [Client-Side Image Maps \(CSIM\)](#)
  - [Image Caching](#)
  - [Error Handling as Image Rendering](#)
  - [Notable Design Details](#)
    - [The Self-Documenting Architecture Diagram](#)
    - [500+ Named Colors](#)
    - [The 90° Rotation Trick](#)
    - [Reed-Solomon Error Correction](#)
    - [The QR Specification Error](#)
    - [Gantt Chart Complexity](#)
    - [Anti-Spam CAPTCHA Generator](#)
    - [Wind Rose — A Non-Standard Coordinate System](#)

---

## Design Philosophy

JpGraph was designed around several principles that were ahead of their time for a PHP library circa 2001:

1. **Object-oriented from the start.** At a time when most PHP code was procedural, JpGraph used a deep class hierarchy with inheritance, polymorphism, and the template method pattern.
2. **"Charts are images."** Every chart is rendered server-side to a raster image (PNG/JPEG/GIF) via PHP's GD extension. There is no JavaScript, no SVG, no client dependency. The output is an `<img>` tag or a direct image stream.
3. **Plugin architecture.** The core module (`jpggraph.php`) defines the framework — `Graph`, `Scale`, `Axis`, `Plot`. Each chart type is a separate file that extends `Plot` with its own rendering logic. Users only `require_once` the modules they need.
4. **Convention over configuration.** Reasonable defaults for everything — colors, margins, tick density, legend placement — so a simple chart can be created in 5 lines of code. But every aspect is configurable.

---

## Layered Architecture

The library is organized in five layers, from low-level graphics to user-facing chart objects:

Layer 5: User Script <code>\$graph = new Graph(400,300); \$graph-&gt;Add(new LinePlot(\$y)); \$graph-&gt;Stroke();</code>
Layer 4: Chart-Type Modules (Plot subclasses) <code>LinePlot</code> , <code>BarPlot</code> , <code>PiePlot</code> , <code>GanttBar</code> , <code>RadarPlot</code> , <code>ContourPlot</code> , <code>FilledContourPlot</code> , <code>MatrixPlot</code> , ...
Layer 3: Graph Framework <code>Graph</code> , <code>Axis</code> , <code>LinearScale/LogScale</code> , <code>LinearTicks</code> , <code>Legend</code> , <code>Text</code> , <code>Footer</code> , <code>PlotBand</code> , <code>PlotLine</code> , <code>DisplayValue</code> Specialized: <code>GanttGraph</code> , <code>PieGraph</code> , <code>PolarGraph</code> , <code>WindroseGraph</code>
Layer 2: Image Abstraction <code>Image</code> → <code>RotImage</code> (transparent rotation), <code>RGB</code> (named colors), <code>TTF</code> (font management), <code>Gradient</code> (11 fill styles), <code>ImgTrans</code> (3D perspective), <code>ImgStreamCache</code>
Layer 1: PHP GD Extension <code>imagecreatetruecolor</code> , <code>imageline</code> , <code>imagefilledpolygon</code> , <code>imagettftext</code> , <code>imagepng</code> , <code>imagejpeg</code> , ...

### Layer 1 — GD Extension

The raw PHP GD2 library. JpGraph requires `imagecreatetruecolor` (true-color images) and optionally `imageantialias` and TrueType font support via `imagettftext`.

### Layer 2 — Image Abstraction (`gd_image.inc.php`)

The **Image** class (~2 000 lines) wraps every GD call:

- **Drawing primitives** — **Line**, **Rectangle**, **FilledRectangle**, **Polygon**, **FilledPolygon**, **Circle**, **FilledCircle**, **Arc**, **FilledArc**, rounded rectangles, dashed/dotted line styles.
- **Color management** — Named color resolution (500+ names via the **RGB** class), alpha transparency (**color@0.5** syntax), color stack (**PushColor/PopColor**).
- **Text rendering** — TTF and built-in GD fonts, with bounding box calculation, paragraph alignment, and word wrap.
- **Image I/O** — Stream to browser with correct **Content-Type** headers, save to file, or return a GD handle.

The **RotImage** subclass extends **Image** with **transparent rotation** — the entire coordinate system can be rotated by an arbitrary angle (commonly 90° for horizontal bar charts). Rotation is applied to all drawing calls via a 2D affine transformation around a configurable center point.

### Layer 3 — Graph Framework (**jpggraph.php**)

The central **Graph** class (~2 000 lines in the 5 400-line core module) orchestrates the rendering pipeline:

- **Scale system** — **LinearScale** and **LogScale** map data coordinates to pixel positions. Scales handle auto-ranging (finding appropriate min/max from data), tick calculation, and the **Translate()** method that converts a world value to an absolute pixel position.
- **Axis system** — The **Axis** class renders the axis line, tick marks (major/minor), and labels. Supports text scales (categorical), linear, logarithmic, date, and manual tick placement.
- **Plot management** — The **Graph** maintains arrays of **Plot** objects for Y1, Y2, and additional Y-axes (**ynplots**). The **Add()** method uses **instanceof** checks to auto-route objects to the correct collection (plots, text annotations, plot bands, icons, or tables).
- **Dual and multiple Y-axes** — Full support for a second Y-axis (**y2axis**) and arbitrary additional Y-axes (**ynaxis**), each with independent scales and plot collections.
- **Specialized graph types** — **PieGraph**, **GanttGraph**, **PolarGraph**, **WindroseGraph**, **OdoGraph**, **MatrixGraph**, and **CanvasGraph** each subclass **Graph** to provide coordinate systems and rendering pipelines specific to their chart type.

### Layer 4 — Plot Modules

Each chart type implements the abstract **Stroke(\$img, \$xscale, \$yscale)** method:

- **LinePlot** — Draws line segments between data points, with optional filled area, gradient fills, step-style, and bar-centering.
- **BarPlot** — Renders bars with gradient fills, pattern fills, shadows, and value labels. **GroupBarPlot** and **AccBarPlot** compose multiple **BarPlot** instances for grouped and stacked bar charts.
- **ScatterPlot** — Places plot marks at arbitrary (x,y) positions; supports impulse (stem) mode and linked points.
- **ContourPlot / FilledContourPlot** — See the dedicated sections below.

---

## The Rendering Pipeline

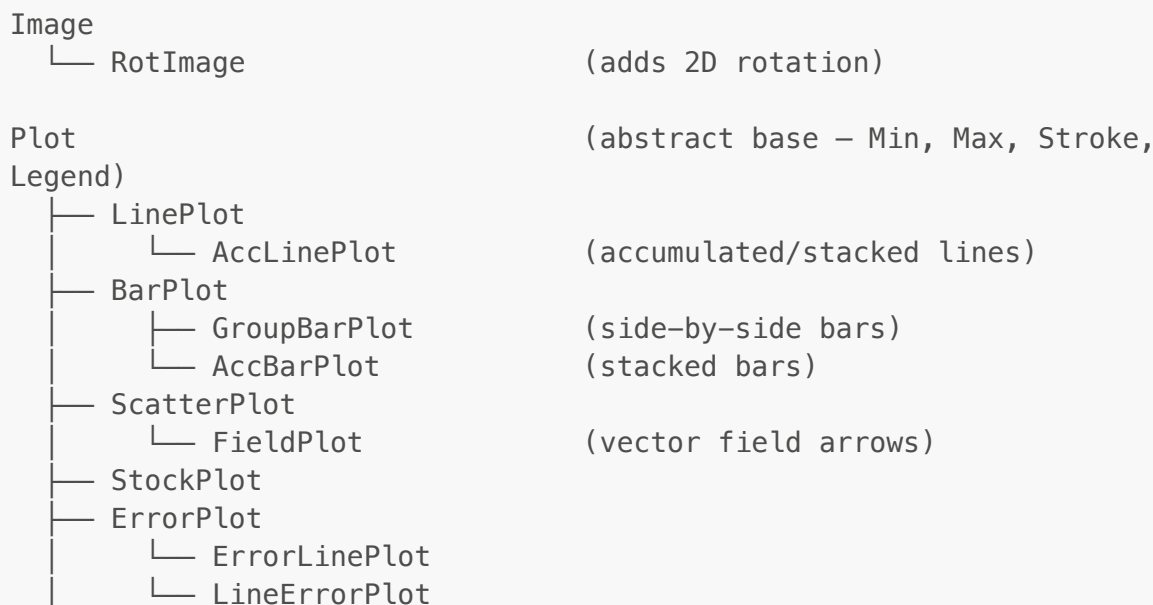
When **\$graph->Stroke()** is called, the following sequence executes:

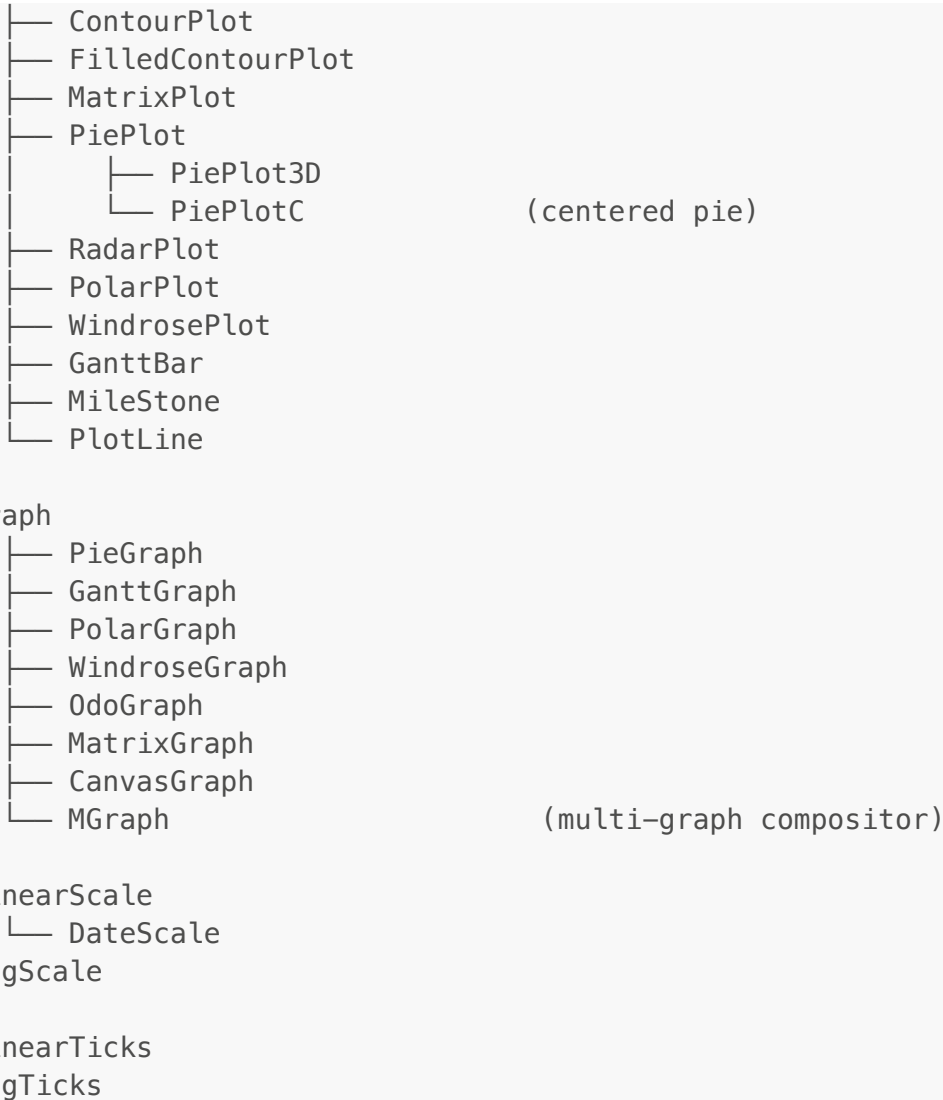
1. Check cache → if valid cached image exists, stream it and exit
2. For each Plot: call `PreScaleSetup()` → plots can adjust scale ranges
3. Auto-scale: determine min/max from data → calculate tick intervals
4. For each Plot: call `PreStrokeAdjust()` → final plot adjustments
5. Render background (solid color, gradient, or background image)
6. Render background country flag (if set)
7. Render plot bands (DEPTH\_BACK)
8. Render grid lines (if `grid_depth == DEPTH_BACK`)
9. Render icons (DEPTH\_BACK)
10. For each Plot: call `Stroke($img, $xscale, $yscale)`
11. Render icons (DEPTH\_FRONT)
12. Render grid lines (if `grid_depth == DEPTH_FRONT`)
13. Render axes, tick marks, and labels
14. Render Y2-axis plots (second Y-axis)
15. Render additional Yn-axis plots
16. Render text annotations
17. Render plot lines and plot bands (DEPTH\_FRONT)
18. Render legend
19. Render title, subtitle, sub-subtitle
20. Render footer (with optional generation timer)
21. Render frame and optional bevel effect
22. Apply 3D perspective transformation (if enabled)
23. Apply image adjustments (brightness, contrast, saturation)
24. Stream image to browser / save to file / save to cache
25. Generate CSIM HTML if requested

The `PreScaleSetup()` hook is particularly important for contour and matrix plots, which need to override the auto-scaling to match their grid dimensions.

## Class Hierarchy

The key inheritance relationships:





## The Auto-Scaling Algorithm

One of JpGraph's most important features is automatic scale determination. Given a set of data values, the library must choose axis min/max values and tick intervals that are "nice" round numbers.

The algorithm (implemented in `LinearScale::CalcTicks()`) works as follows:

1. Compute `diff = max - min` of the data range.
2. Compute `ld = floor(log10(diff))` — the order of magnitude.
3. If `min` is close to zero (positive, but less than  $10^{\text{ld}}$ ), snap it to zero — a common-sense heuristic that makes charts more readable.
4. Try candidate major tick steps:  $\frac{10^{\text{ld}}}{a}$  where  $a \in \{1, 2, 5\}$  produces multiples of 10, 5, and 2.
5. Adjust `min` and `max` to align with tick boundaries: `adjmin = floor(min / minstep) * minstep`.
6. Count the resulting number of major ticks. If too many for the available pixel width, increase the step by bumping `ld`.
7. Minor ticks are set to `majstep / b` where `b` divides each major interval.

The method `CalcTicks` returns `[$numsteps, $adjmin, $adjmax, $minstep, $majstep]`.

Three variants exist:

- `CalcTicks` — adjusts both min and max to align with major tick boundaries.
- `CalcTicksFreeze` — keeps user-specified min/max unchanged.
- `IntCalcTicks` — integer-aware variant for discrete data.

The actual tick density adapts to the available pixel space. The `xtick_factor` and `ytick_factor` properties (set by `SetTickDensity()`) control the tradeoff between readability and label crowding.

## The Contour Plot Algorithm

**File:** `jpggraph_contour.php` — The `Contour` class (~320 lines)

This implements a **marching-edges algorithm** for finding isobar (contour line) paths through a 2D data grid. The approach is related to — but distinct from — the well-known Marching Squares algorithm.

### Input

- A 2D matrix of scalar values  $Z[i][j]$  representing values on an equi-spaced X-Y mesh.
- A set of isobar values  $\{c_0, c_1, \dots, c_n\}$  (either user-specified or auto-determined as equi-spaced between  $Z_{\min}$  and  $Z_{\max}$ ).

### Algorithm Steps

#### Step 1: Perturbation

Before processing, any data point whose value exactly equals an isobar value is perturbed by 0.1%. This avoids the degenerate case where a contour passes exactly through a vertex, which would create ambiguous edge crossings:

```
if |Z[i][j] - isobar_k| < 0.0001:
    Z[i][j] += Z[i][j] * 0.001
```

This is a pragmatic engineering decision that has no visible effect on the output but eliminates an entire class of edge cases.

#### Step 2: Edge Crossing Detection

For each isobar value  $c_k$ , the algorithm examines every horizontal and vertical edge in the grid. An isobar crosses an edge between vertices  $v_1$  and  $v_2$  if and only if:

$$(c_k - v_1)(c_k - v_2) < 0$$

This elegant product test catches exactly the case where the isobar value lies strictly between the two vertex values. The results are stored in two boolean matrices: `edges[HORIZ_EDGE][i][j]` and `edges[VERT_EDGE][i][j]`.

#### Step 3: Cell Analysis

Each grid cell (bounded by four edges) is examined. A cell can have 0, 2, or 4 edge crossings:

- **0 crossings** — The isobar does not pass through this cell. Skip.
- **2 crossings** — The isobar enters on one edge and exits on another. Connect them with a line segment.
- **4 crossings** — A **saddle point**. The isobar crosses all four edges, creating ambiguity about how to connect them. The algorithm resolves this by computing the average of the four corner values (a virtual "center point") and comparing it with the top-left corner:
  - If center and top-left are on the **same side** of the isobar → connect as `\` (NW-SE orientation)
  - If they are on **opposite sides** → connect as `/` (NE-SW orientation)
  - If center equals the isobar exactly → connect as `+`

#### Step 4: Coordinate Interpolation

For each crossing, the exact position along the edge is determined by **linear interpolation**:

$$x_{\text{cross}} = \text{col} + \frac{c_k - Z[\text{row}][\text{col}]}{Z[\text{row}][\text{col}] - Z[\text{row}][\text{col}+1]}$$

A minimum denominator check ( $> 0.001$ ) prevents division by near-zero when adjacent vertices have nearly equal values.

#### Output

The result is an array of line segments per isobar: `isobarCoord[k][ ]` where each entry is a pair of  $(x, y)$  coordinate pairs defining one segment of the contour line. These segments can then be rendered as individual lines by the `ContourPlot::Stroke()` method.

#### Color Assignment

Colors are assigned per isobar using either:

- A user-specified color array.
- Automatic **spectral coloring** via `RGB::GetSpectrum($v)` where  $v \in [0, 1]$  maps to a blue→cyan→green→yellow→red color ramp.
- **High-contrast mode** — a linear blue-to-red gradient, or pure black.

## Client-Side Image Maps (CSIM)

JpGraph can generate HTML image maps alongside chart images, enabling clickable regions on bars, pie slices, data points, and legend entries. The `GetHTMLCSIM()` method renders the chart and simultaneously collects `<area>` tags for each interactive element.

The CSIM system uses a special internal filename `_csim_special_` to signal a "dry run" `Stroke()` that calculates map coordinates without streaming the image. A separate request with `_jpg_csimd=1` then generates the actual image.

## Image Caching

The `ImgStreamCache` class provides filesystem caching of rendered chart images:

- Charts are cached by a user-specified name (or auto-generated from the script filename and query string).
  - Cache validity is controlled by a configurable timeout.
  - On cache hit, the image is streamed directly from disk, bypassing all rendering logic.
  - File permissions and group ownership are configurable for shared hosting environments.
- 

## Error Handling as Image Rendering

One of JpGraph's cleverest features: **error messages are rendered as images**.

When a chart script fails (invalid data, misconfiguration, missing fonts), the error handler creates a new GD image, renders the error message as text on a white background with a red header, and streams it back with the correct `Content-Type: image/png` header.

This means a broken `` tag shows the error message *inside the image area* instead of a broken image icon or a wall of PHP error text that gets swallowed by the browser's image parser.

The error messages themselves are localized via message catalogs in the `lang/` directory (English, German, and a production mode that hides internal details).

---

## Notable Design Details

### The Self-Documenting Architecture Diagram

The file `misc/jpgarch.php` is a JpGraph script that uses the `CanvasGraph` and `CanvasRectangleText` classes to draw a visual architecture diagram of JpGraph's own internal structure — a chart about the charting library, made with the charting library. It uses manual coordinate positioning to lay out labeled boxes representing the GD layer, image primitives, axis/scaling, and the various plot types.

### 500+ Named Colors

The `RGB` class contains a hand-curated lookup table of over 500 named colors drawn from the X11/CSS color name standard, including shade variants (`snow1` through `snow4`, `seashell1` through `seashell4`, etc.). Colors can be specified anywhere as strings: `'darkgoldenrod3'`, `'cadetblue'`, or as `[R, G, B]` arrays. Alpha transparency is specified with an `@` suffix: `'blue@0.5'`.

### The 90° Rotation Trick

The `Set90AndMargin()` method on `Graph` rotates the entire coordinate system by 90° to produce horizontal bar charts. Rather than implementing horizontal bars as a separate chart type, the existing vertical bar rendering code is reused with a coordinate transformation. The `RotImage` class handles all the trigonometry.

### Reed-Solomon Error Correction

The QR Code, Data Matrix, and PDF417 barcode modules each include their own Reed-Solomon error correction implementation. The QR module's `reed-solomon.inc.php` performs Galois field arithmetic ( $\text{GF}(2^8)$ ) for generating error correction codewords — the same mathematical foundation used in CDs, DVDs, and deep-space communication.

## The QR Specification Error

During development of the QR module, Johan Persson discovered minor errors in the official QR barcode specification (ISO/IEC 18004). The paper in [QR-paper/](#) documents these discrepancies, discovered through systematic testing of the encoder against the specification's own worked examples.

## Gantt Chart Complexity

At ~3 950 lines, the Gantt chart module (`jpggraph_gantt.php`) is the largest single plot type. It implements its own time-based coordinate system with multi-granularity headers (minutes, hours, days, weeks, months, years), constraint arrows between tasks, progress indicators, milestone diamonds, and icon embedding. The `GanttGraph` class maintains its own scale system independent of the standard linear/log scales.

## Anti-Spam CAPTCHA Generator

`jpggraph_antispam.php` is a complete CAPTCHA system that generates images of hand-drawn digits and letters. The digit images are stored as base64-encoded JPEG data directly in the PHP source file — a self-contained solution that needs no external image files. The `HandDigits` class contains a multi-dimensional array of pre-rendered character variants with intentional imperfections to defeat OCR.

## Wind Rose — A Non-Standard Coordinate System

The wind rose plot (`jpggraph_windrose.php`, ~1 570 lines) implements a circular frequency chart with compass-direction axes (N, NE, E, SE, ...). It supports 4, 8, or 16 compass directions plus free-form arbitrary angles. The rendering uses polar-to-cartesian transformation for each directional bar segment, with percentage-based concentric grid rings.

---

*This architecture overview was written from the JpGraph v3.1.6p source code (January 2010). The library was created by Johan Persson at Aditus Consulting.*