Contour Plot Algorithm Documentation

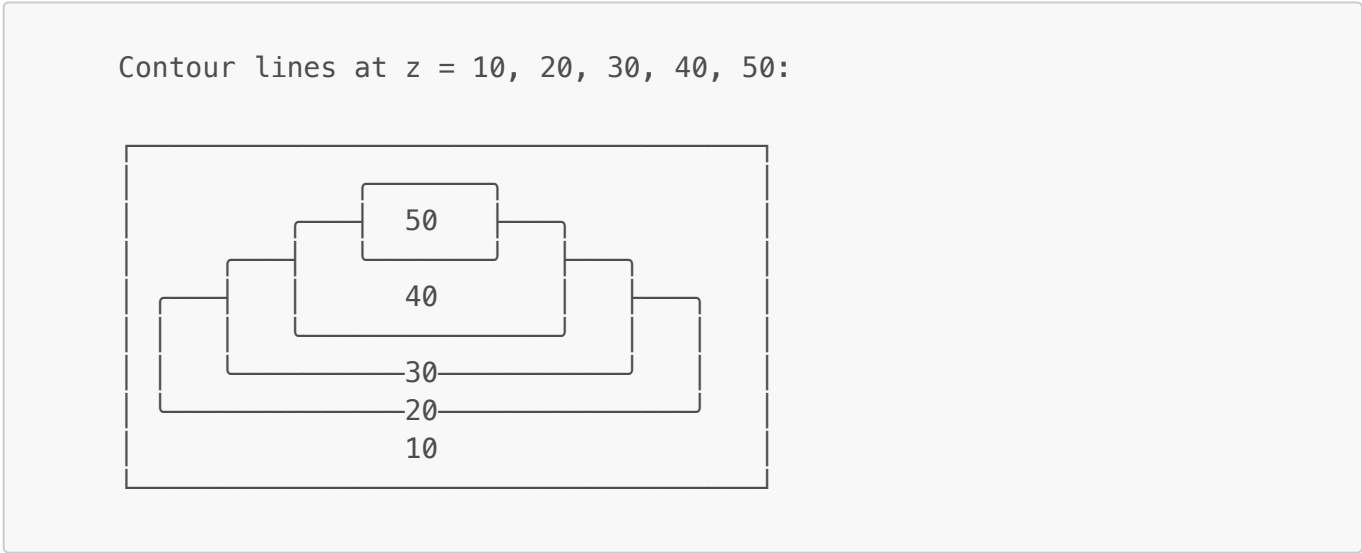
> **Implementation Reference** — This document provides a comprehensive description of the contour plotting algorithms used in JpGraph v3.1.6p. The goal is to enable complete re-implementation in any programming language.

Table of Contents

Overview and Terminology

What is a Contour Plot?

A **contour plot** visualizes a 3D surface $z = f(x, y)$ on a 2D plane by drawing curves of constant z values. These curves are called **isobars**, **isolines**, or **contour lines**. Each isobar connects points where the function has the same value.



Key Terminology

Term	Definition
Isobar/Isoline	A curve connecting points of equal value
Contour level	The specific z value for an isobar
Cell	A rectangular region bounded by four adjacent data points
Edge	A line segment between two adjacent vertices

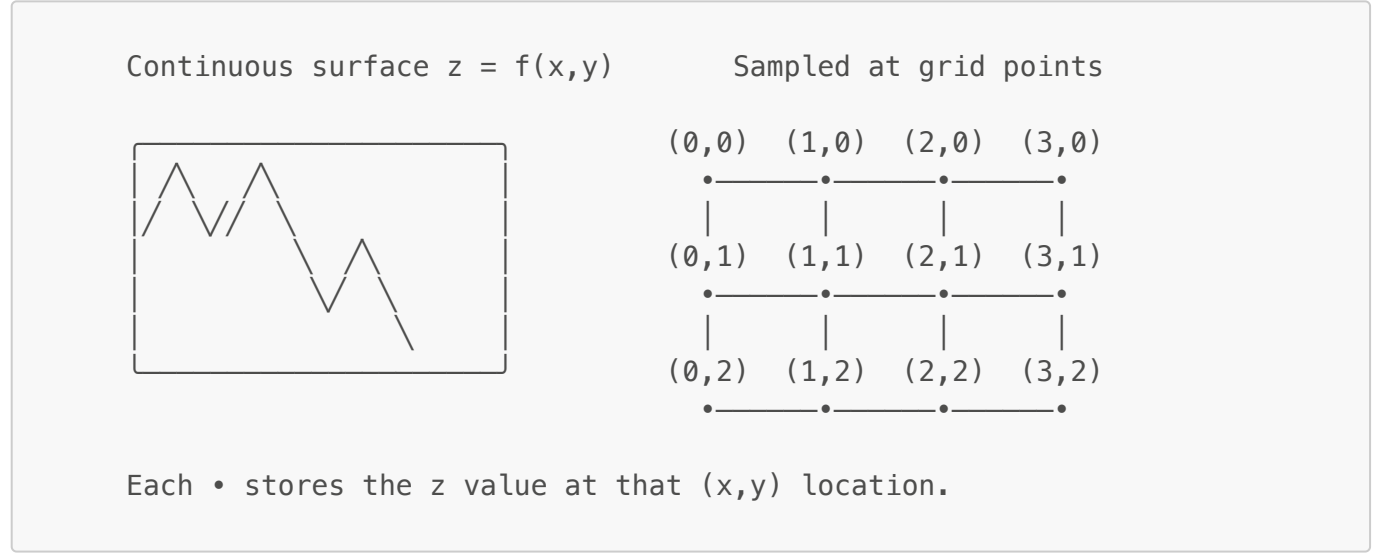
Term	Definition
Saddle point	A cell where the isobar crosses all four edges (ambiguous case)
Contour band	The region between two adjacent contour levels

Input Data Model

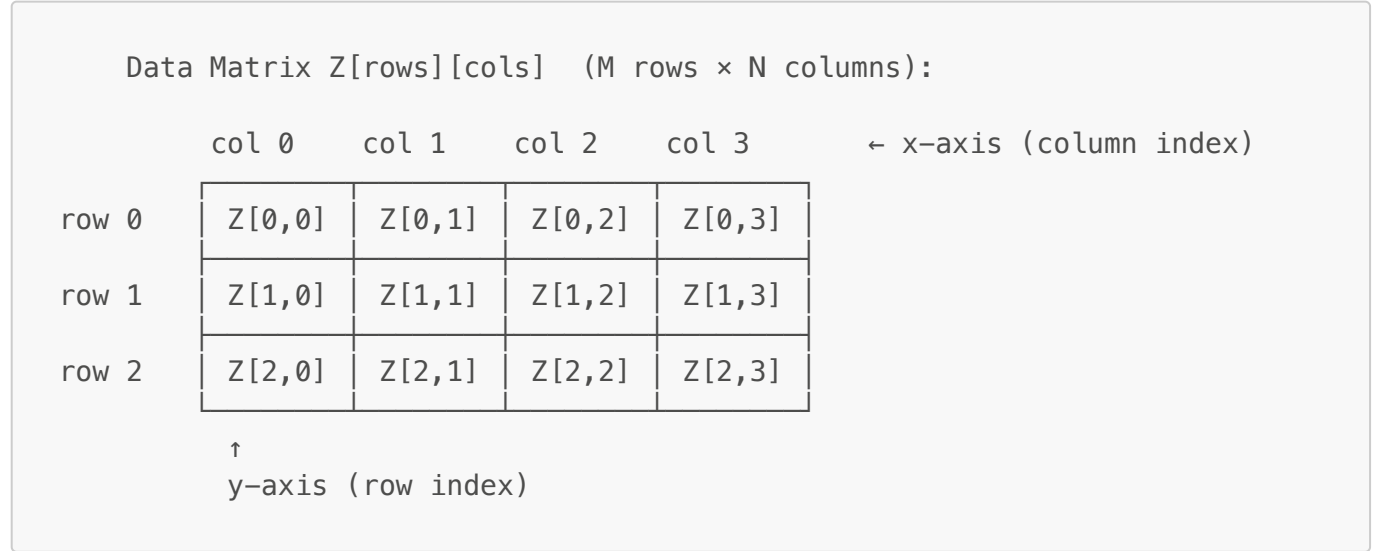
Both algorithms assume the input is a **2D matrix** of scalar values representing samples of a surface $z = f(x, y)$ on a **regular equi-spaced grid**. Each matrix entry $Z[r][c]$ stores the height (or intensity) of the surface at one sample point. The algorithms do not use explicit (x, y) coordinates — instead, the row and column indices **are** the coordinates. Row index r maps to the y-axis and column index c maps to the x-axis, with unit spacing between adjacent samples.

What the Values Represent

Imagine a physical surface — a terrain elevation map, a temperature field, or a pressure distribution. We cannot store every point on the continuous surface, so we **sample** it at evenly spaced locations and record the z value at each location:



These sampled values are stored in a 2D array (a matrix), where the row index selects the y position and the column index selects the x position:



For example, in the JpGraph PHP source, the data is passed as a plain nested array:

```
$data = array(
    array(0.5, 1.1, 1.5, 1.0, 2.0),   // row 0 — five sample values
    array(1.0, 1.5, 3.0, 5.0, 6.0),   // row 1
    array(0.9, 2.0, 2.1, 3.0, 6.0),   // row 2
    array(1.0, 1.5, 3.0, 4.0, 6.0),   // row 3
);
// This is a 4×5 matrix (M=4 rows, N=5 columns)
```

The number of rows is `count($data)` and the number of columns is `count($data[0])`. No explicit (x, y) coordinates are supplied — the position of each sample point is implied by its array indices.

From Sample Points to Cells

The sample points form the **vertices** (corners) of a grid. The contour algorithm works not on individual vertices but on the **cells** formed between them. A cell is the rectangular region bounded by four neighboring sample points:

```
    Four adjacent sample points define one cell:

    Z[r][c] ──────────────── Z[r][c+1]       The cell sits in the
         │                       │            space BETWEEN the four
         │                       │            corner sample points.
         │     Cell[r][c]        │
         │                       │            Its corners carry the
         │                       │            known z values; the
    Z[r+1][c] ──────────────── Z[r+1][c+1]   algorithm interpolates
                                              along the edges.
```

Because each cell requires a neighbor to its right (column $c+1$) and below (row $r+1$), the last row and last column of sample points serve only as the right/bottom boundaries of cells — they do not start new cells. This is why an $M \times N$ matrix of sample points produces $(M{-}1) \times (N{-}1)$ cells:

```
    3×4 data matrix (M=3, N=4) → 2×3 grid of cells:

    Sample points (vertices):          Cells:

    •─────────•─────────•─────────•─────────•
    │         │         │         │         │
    │         │         │         │         │
    •─────────•─────────•─────────•─────────•
┌─────────┬─────────┬─────────┐
    │         │         │         │         │     │           │           │
│
    │         │         │         │         │     │ Cell[0,0] │ Cell[0,1] │
Cell[0,2]│
```

```
    •─────────•─────────•─────────•─────────•
├─────────┼─────────┼─────────┤
    |         |         |         |         |     |           |           |
|
    |         |         |         |         |     | Cell[1,0]| Cell[1,1]|
Cell[1,2]|
    •─────────•─────────•─────────•─────────•
└─────────┴─────────┴─────────┘

    12 sample points                              (3−1)×(4−1) = 2×3 = 6 cells
    (3 rows × 4 columns)
```

Why Cells Matter

The contour algorithm scans **every cell** in the grid and asks: *does the isobar (contour line at value c) cross any edges of this cell?* Each cell edge connects two sample points whose z values are known. If the isobar value c lies between those two endpoint values, the contour line must cross that edge somewhere. The algorithm then uses linear interpolation along the edge to pinpoint where, and connects the crossing points within each cell to form contour line segments.

In other words:

- **Vertices** (sample points) provide the known z values.
- **Edges** (lines between adjacent vertices) are where the algorithm detects crossings.
- **Cells** (rectangles between four vertices) are the units of work — the algorithm processes one cell at a time to find and connect contour crossings.

```
    How a contour line emerges from cells:

     0.5 ──────── 1.1 ──────── 1.5 ──────── 3.0
      |           |           |           |
      |  Cell A   |  Cell B   |   Cell C  |    isobar c = 1.0
      |  (no      |           |           |
      |  crossing)|           ×           |     × = crossing point
      |           |         / |  \        |
     1.0 ──────── 1.5 ─────×─ 0.9 ─×─────── 2.0
      ↑
    vertex ON the isobar
    (handled by perturbation, Step 1)

    Cell A: all corners either ≤1.0 or ≥1.0 after perturbation → no
crossing
    Cell B: isobar crosses right edge (1.5→0.9) and bottom edge (1.5→0.9)
→ segment
    Cell C: isobar crosses left edge (1.5→0.9) and bottom  edge (0.9→2.0)?
```

> **Summary:** The data matrix is a grid of sampled z values. Column indices are x coordinates, row indices are y coordinates, and the values are z heights. Adjacent sample points form cells — the

> fundamental processing units of the contour algorithm. An $M \times N$ matrix yields $(M{-}1) \times (N{-}1)$ cells because each cell needs four corner vertices.

Part I: Line-Only Contour Algorithm (Marching-Edges)

Source File: `jpgraph_contour.php` — Class `Contour` (~380 lines)

This algorithm finds contour **line segments** for each isobar value. It is a variant of the classic Marching Squares algorithm, operating on edges rather than cell configurations.

Algorithm Overview

```
┌─────────────────────────────────────────────────────────────┐
│                  MARCHING-EDGES ALGORITHM                   │
├─────────────────────────────────────────────────────────────┤
│  INPUT:                                                     │
│    • Data matrix Z[rows][cols]                              │
│    • Isobar values {c₀, c₁, ..., cₙ}                        │
│                                                             │
│  OUTPUT:                                                    │
│    • For each isobar: list of line segments [(x1,y1)→(x2,y2)]  │
│                                                             │
│  STEPS:                                                     │
│    1. Perturbation    – Avoid vertex-on-isobar degeneracy   │
│    2. Edge Detection  – Mark which edges each isobar crosses│
│    3. Cell Analysis   – Connect edge crossings within cells │
│    4. Interpolation   – Calculate precise crossing coordinates │
└─────────────────────────────────────────────────────────────┘
```

Step 1: Data Perturbation

The Problem

When a data point's value **exactly equals** an isobar value, the contour line passes directly through that vertex. This creates ambiguity:

```
Ambiguous case: Z[1,1] = 50.0, isobar = 50.0

     45 ──────────── 55
      │              │
      │      ?       │      Does the isobar cross the
      │              │      left edge? Top edge? Both?
     48 ──────────── 50 ← vertex ON the isobar
```

The crossing test $(c - v_1)(c - v_2) < 0$ returns `false` when either vertex equals the isobar (since one factor is zero).

The Solution

Perturb any vertex value that equals an isobar by a tiny amount (0.1%):

```
PERTURBATION FACTOR = 0.001

for each isobar value c in isobar_values:
    for each vertex (row, col):
        if |Z[row][col] - c| < 0.0001:
            Z[row][col] = Z[row][col] * (1 + PERTURBATION_FACTOR)
```

Pseudocode:

```
function adjustDataPointValues(Z, isobarValues):
    for k = 0 to length(isobarValues) - 1:
        ib = isobarValues[k]
        for row = 0 to nRows - 1:
            for col = 0 to nCols - 1:
                if abs(Z[row][col] - ib) < 0.0001:
                    Z[row][col] = Z[row][col] + Z[row][col] * 0.001
```

> ⚠ **KEY INSIGHT**: This perturbation is applied to the working copy of the data, not the original. The visual difference is imperceptible (0.1% shift) but eliminates an entire class of edge cases.

Step 2: Edge Crossing Detection

Edge Types

Each cell has four edges. The algorithm recognizes two types of edges:

```
    Horizontal edges (HORIZ_EDGE = 0):

    v[row][col] ———————————— v[row][col+1]
                 H[row][col]

    Vertical edges (VERT_EDGE = 1):

    v[row][col]
         |
         | V[row][col]
         |
    v[row+1][col]
```

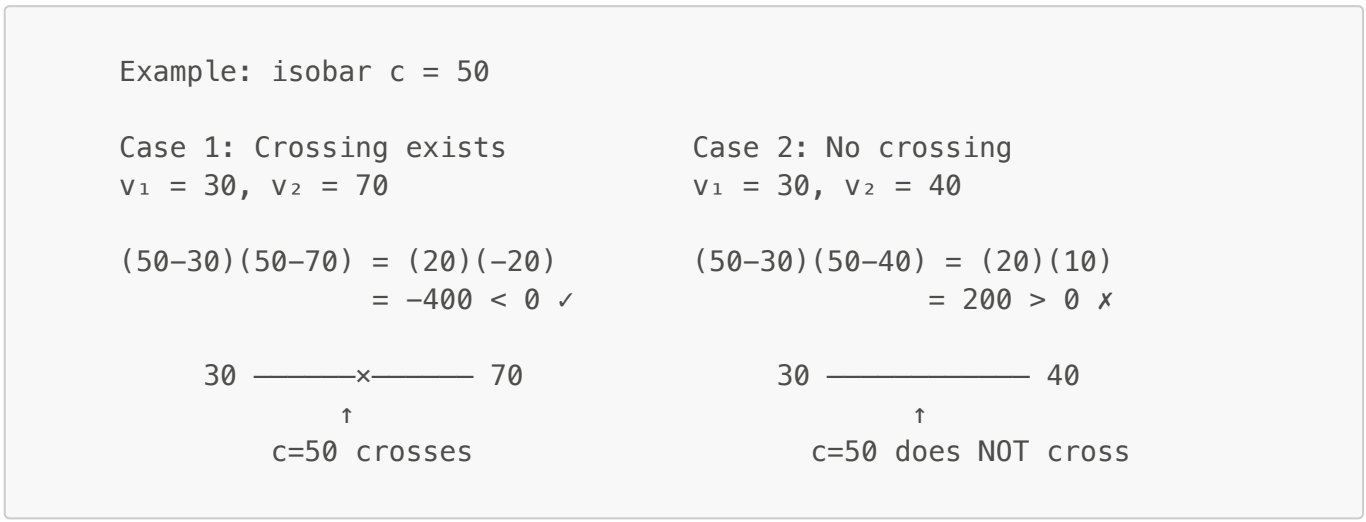
The Sign-Change Test

An isobar with value c crosses an edge between vertices v_1 and v_2 **if and only if**:

$$ \boxed{(c - v_1)(c - v_2) < 0} $$

Why this works:

- If $v_1 < c < v_2$ (or $v_2 < c < v_1$), one factor is positive and one is negative → product is negative.
- If both vertices are on the same side of c, both factors have the same sign → product is positive.
- If either vertex equals c (after perturbation, this won't happen), product is zero.



Building the Edge Matrices

For each isobar, create two boolean matrices tracking crossings of vertical and horizontal edges:

```
function determineIsobarEdgeCrossings(isobarIndex, Z, isobarValues):
    ib = isobarValues[isobarIndex]

    // Initialize edges array: edges[edgeType][row][col]

    // Check interior horizontal and vertical edges
    for row = 0 to nRows - 2:
        for col = 0 to nCols - 2:
            edges[HORIZ_EDGE][row][col] = isobarHCrossing(row, col, ib)
            edges[VERT_EDGE][row][col] = isobarVCrossing(row, col, ib)

    // Check rightmost vertical edges (column = nCols-1)
    for row = 0 to nRows - 2:
        edges[VERT_EDGE][row][nCols-1] = isobarVCrossing(row, nCols-1, ib)

    // Check bottom horizontal edges (row = nRows-1)
    for col = 0 to nCols - 2:
        edges[HORIZ_EDGE][nRows-1][col] = isobarHCrossing(nRows-1, col,
ib)

function isobarHCrossing(row, col, isobarValue):
    v1 = Z[row][col]
    v2 = Z[row][col + 1]
    return (isobarValue - v1) * (isobarValue - v2) < 0
```

```
function isobarVCrossing(row, col, isobarValue):
    v1 = Z[row][col]
    v2 = Z[row + 1][col]
    return (isobarValue − v1) ∗ (isobarValue − v2) < 0
```

Edge Matrix Visualization

```
    For a 3×4 data matrix with isobar c = 50:

    Data:           Horizontal Edges:       Vertical Edges:

    30  45  55  70      ✓   ×   ✓            ✓   ×   ×   ✓
     |   |   |   |      |   |   |            |   |   |   |
    35  48  52  65  →   ×   ✓   ×     +      ✓   ×   ×   ✓
     |   |   |   |      |   |   |            |   |   |   |
    40  55  58  60      ×   ×   ×           (boundary edges)

    ✓ = isobar crosses this edge
    × = isobar does NOT cross this edge
```

Step 3: Cell Analysis — Connecting the Crossings

Cell Edge Configuration

For each cell, examine which of its four edges are crossed:

```
    Cell at (row, col):

                    TOP edge
           H[row][col]
    v[row][col] ──────────────── v[row][col+1]
          |                          |
    LEFT  | V[row][col]  V[row][col+1] | RIGHT
          |                          |
    v[row+1][col] ──────────────── v[row+1][col+1]
                    H[row+1][col]
                   BOTTOM edge
```

Collecting crossings:

```
function analyzeCell(row, col):
    crossings = []

    // Check each of the four edges in order:
    // TOP, BOTTOM, LEFT, RIGHT

    if edges[HORIZ_EDGE][row][col]:        // TOP
```

```
        crossings.append((row, col, HORIZ_EDGE))

    if edges[HORIZ_EDGE][row+1][col]:      // BOTTOM
        crossings.append((row+1, col, HORIZ_EDGE))

    if edges[VERT_EDGE][row][col]:         // LEFT
        crossings.append((row, col, VERT_EDGE))

    if edges[VERT_EDGE][row][col+1]:       // RIGHT
        crossings.append((row, col+1, VERT_EDGE))

    return crossings
```

Case Analysis: Number of Crossings

A cell can have **0, 2, or 4** edge crossings (never 1 or 3 — a contour line cannot enter without exiting).

Case 0: No Crossings

The isobar does not pass through this cell. Skip it.

```
    All vertices on the same side of isobar:

    30 ──────── 35
     │          │
     │   (no    │     isobar = 50
     │  isobar) │     All vertices < 50
     │          │
    28 ──────── 32
```

Case 2: Simple Crossing (Two Edges)

The isobar enters through one edge and exits through another. Connect them with a line segment.

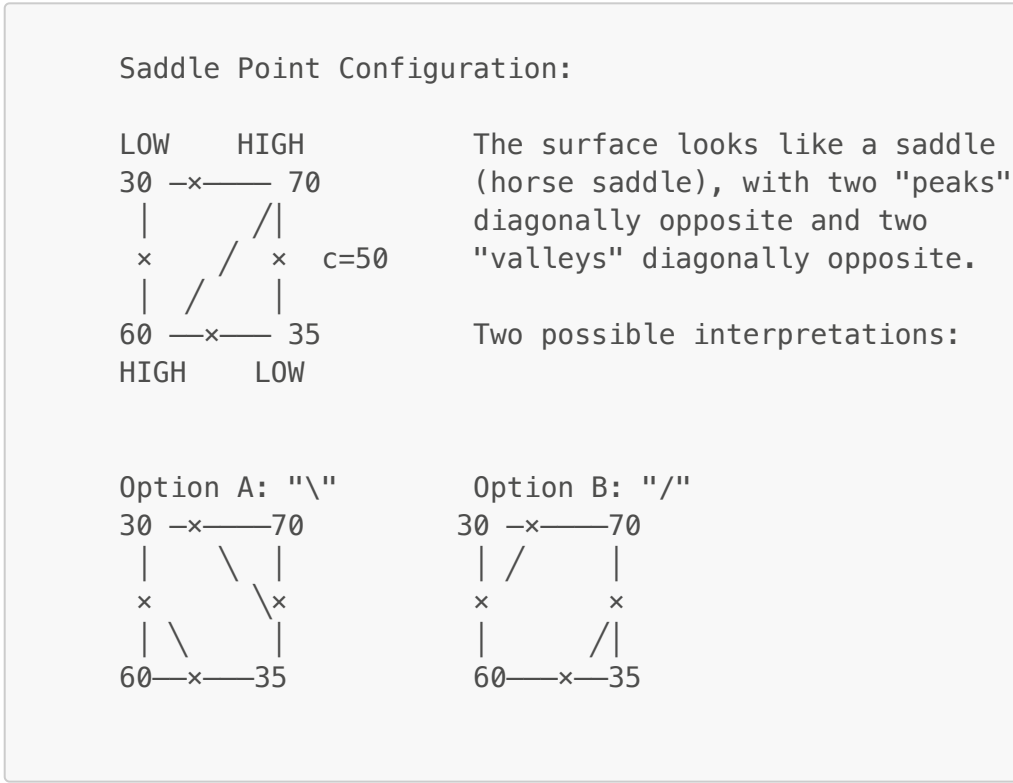
```
    Six possible configurations for 2-crossing case (The exact crossing
point is determined through interpolation):


    TOP-BOTTOM    TOP-LEFT      TOP-RIGHT
    ────×────     ──×──────     ──────×──
        │           ╱                 ╲
        │          ╱                   ╲
        │         ×                     ×
    ────×────     ──────────    ──────────

    BOTTOM-LEFT   BOTTOM-RIGHT   LEFT-RIGHT
    ──────────    ──────────     ──────────
    ×                    ×
```

Implementation for 2-crossing case:

```
if length(crossings) == 2:
    // Simply connect crossing 0 to crossing 1
    (x1, y1) = getCrossingCoord(crossings[0])
    (x2, y2) = getCrossingCoord(crossings[1])
    addLineSegment(isobarIndex, (x1, y1), (x2, y2))
```

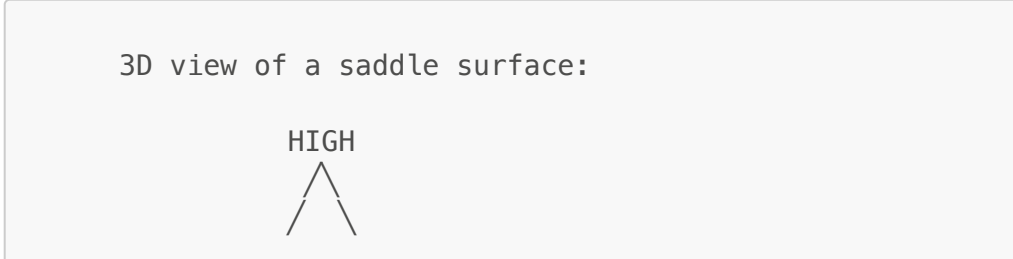
Case 4: Saddle Point (All Four Edges)

This is the critical ambiguous case. When all four edges are crossed, we must decide how to pair them.



Understanding Saddle Points

A **saddle point** occurs when the data surface has a configuration like a horse saddle or a mountain pass. At the center, the surface is a local maximum in one direction and a local minimum in the perpendicular direction.



```
      LOW ──────/──────\────── LOW
                 \    /
                  \  /
                   \/
                  HIGH

      Looking down from above, the isobar can connect:
      - The two LOWs (creating "/" pattern)
      - The two HIGHs (creating "\" pattern)
```

Saddle Point Resolution Algorithm

To determine the correct pairing, compute the **average value** at the cell center and compare it with the **top-left corner**:

$$ v_{center} = \frac{v_1 + v_2 + v_3 + v_4}{4} $$

```
      Cell corners (v1 is top-left, clockwise):

      v1 ──────────── v2
        │             │
        │   v_center  │
        │             │
      v4 ──────────── v3
```

Decision Logic:

```
function resolveSaddleOrientation(v1, v2, v3, v4, isobarValue):
    vc = (v1 + v2 + v3 + v4) / 4    // Virtual center value

    if vc == isobarValue:
        // Extremely rare after perturbation
        return "+" shaped crossing (all four meet at center)

    // Check if center and v1 are on the SAME side of the isobar
    center_above = (vc > isobarValue)
    v1_above = (v1 > isobarValue)

    if center_above == v1_above:
        // Same side → "\" orientation (NW-SE)
        // Connect TOP to RIGHT, and LEFT to BOTTOM
        return BACKSLASH
    else:
        // Opposite sides → "/" orientation (NE-SW)
        // Connect TOP to LEFT, and RIGHT to BOTTOM
        return FORWARDSLASH
```

Visual explanation of the logic:

```
    Case: vc and v1 are BOTH ABOVE the isobar (BACKSLASH "\")

    v1=70    isobar=50    v2=30
     \   |       ↓        /
      \  |             /
       ╳... vc=55
      /  |   (above)    \
     /   |               \
    v4=30              v3=65

    The "ridge" runs from v1 to v3 (upper-left to lower-right)
    Contour lines run perpendicular to ridges
    So contours connect: TOP↔RIGHT, LEFT↔BOTTOM


    Case: vc and v1 are on OPPOSITE sides (FORWARDSLASH "/")

    v1=30    isobar=50    v2=70
     /   |       ↓        \
      /  |             \
       ╳... vc=55
      \  |   (above)    /
     \   |               /
    v4=60              v3=40

    v1=30 is BELOW 50, vc=55 is ABOVE 50
    The "ridge" runs from v2 to v4 (upper-right to lower-left)
    So contours connect: TOP↔LEFT, RIGHT↔BOTTOM
```

Complete saddle point handling:

```
if length(crossings) == 4:
    // crossings array contains edges in order:
    // [0]=TOP, [1]=BOTTOM, [2]=LEFT, [3]=RIGHT

    v1 = Z[row][col]         // top-left
    v2 = Z[row][col+1]       // top-right
    v3 = Z[row+1][col+1]     // bottom-right
    v4 = Z[row+1][col]       // bottom-left

    midval = (v1 + v2 + v3 + v4) / 4

    if midval == isobarValue:
        // "+" pattern: connect TOP↔BOTTOM, LEFT↔RIGHT
        n1=0; n2=1; n3=2; n4=3

    else if (midval > isobarValue) == (v1 > isobarValue):
        // "\" pattern: connect TOP↔RIGHT, LEFT↔BOTTOM
        n1=0; n2=3; n3=2; n4=1
```

```
    else:
        // "/" pattern: connect TOP↔LEFT, BOTTOM↔RIGHT
        n1=0; n2=2; n3=3; n4=1

    // Generate two line segments
    addLineSegment(getCrossingCoord(crossings[n1]),
                   getCrossingCoord(crossings[n2]))
    addLineSegment(getCrossingCoord(crossings[n3]),
                   getCrossingCoord(crossings[n4]))
```

Step 4: Coordinate Interpolation

Once we know which edges are crossed, we need the **exact coordinates** of each crossing point.

Linear Interpolation Formula

For a horizontal edge from (col, row) to $(col+1, row)$:

$$ x_{cross} = col + \frac{|c - Z[row][col]|}{|Z[row][col] - Z[row][col+1]|} $$ $$ y_{cross} = row $$

For a vertical edge from (col, row) to $(col, row+1)$:

$$ x_{cross} = col $$ $$ y_{cross} = row + \frac{|c - Z[row][col]|}{|Z[row][col] - Z[row+1][col]|} $$

Visual example:

```
    Horizontal edge interpolation:

    Z[0][0]=30                    Z[0][1]=70
        ├─────────────────×─────────────────┤
        0             0.5                 1
                       ↑
                   isobar=50

    x = 0 + |50-30|/|30-70| = 0 + 20/40 = 0.5
    y = 0
    Crossing at (0.5, 0)
```

Pseudocode with numerical stability:

```
function getCrossingCoord(row, col, edgeDir, isobarValue):
    EPSILON = 0.001    // Minimum valid denominator

    if edgeDir == HORIZ_EDGE:
        v1 = Z[row][col]
        v2 = Z[row][col + 1]
        d = abs(v1 - v2)

        if d > EPSILON:
```

```
            xcoord = col + abs(isobarValue - v1) / d
        else:
            xcoord = col    // Vertices nearly equal, use left vertex

        ycoord = row

    else:  // VERT_EDGE
        v1 = Z[row][col]
        v2 = Z[row + 1][col]
        d = abs(v1 - v2)

        if d > EPSILON:
            ycoord = row + abs(isobarValue - v1) / d
        else:
            ycoord = row    // Vertices nearly equal, use top vertex

        xcoord = col

    // Optional: flip Y coordinate if using inverted coordinate system
    if invertY:
        ycoord = (nRows - 1) - ycoord

    return (xcoord, ycoord)
```

> ⚠ **NUMERICAL STABILITY**: The `d > EPSILON` check prevents division by very small numbers when adjacent vertices have nearly equal values. This is essential for robust implementation.

Complete Algorithm Pseudocode

```
function getIsobars(Z, isobarValues):
    // Step 1: Perturbation
    adjustDataPointValues(Z, isobarValues)

    result = {}  // Map from isobar index to list of line segments

    for isobar = 0 to length(isobarValues) - 1:
        isobarValue = isobarValues[isobar]
        result[isobar] = []

        // Step 2: Build edge crossing matrices
        resetEdgeMatrices()
        determineIsobarEdgeCrossings(isobar, Z, isobarValues)

        // Step 3 & 4: Analyze cells and interpolate coordinates
        for row = 0 to nRows - 2:
            for col = 0 to nCols - 2:
                crossings = analyzeCell(row, col, edges)
                n = length(crossings)

                if n == 0:
                    continue  // No isobar in this cell
```

```
                else if n == 2:
                    // Simple case: connect the two crossings
                    p1 = getCrossingCoord(crossings[0], isobarValue)
                    p2 = getCrossingCoord(crossings[1], isobarValue)
                    result[isobar].append((p1, p2))

                else if n == 4:
                    // Saddle point: determine orientation
                    v1 = Z[row][col]
                    v2 = Z[row][col+1]
                    v3 = Z[row+1][col+1]
                    v4 = Z[row+1][col]
                    midval = (v1 + v2 + v3 + v4) / 4

                    if midval == isobarValue:
                        // "+" shape (very rare after perturbation)
                        pairs = [(0,1), (2,3)]
                    else if (midval > isobarValue) == (v1 > isobarValue):
                        // "\" shape
                        pairs = [(0,3), (2,1)]
                    else:
                        // "/" shape
                        pairs = [(0,2), (3,1)]

                    for (i, j) in pairs:
                        p1 = getCrossingCoord(crossings[i], isobarValue)
                        p2 = getCrossingCoord(crossings[j], isobarValue)
                        result[isobar].append((p1, p2))

    return result
```

Color Assignment for Contour Lines

Colors can be assigned in three ways:

1. User-Specified Colors

User provides an array of colors, one per isobar level.

2. Spectral Coloring (Default)

Map each isobar to a position on a color spectrum:

```
function calculateSpectralColors(nIsobars):
    colors = []
    step = 1.0 / (nIsobars - 1)
    v = 0

    for i = 0 to nIsobars - 1:
        colors.append(getSpectrum(v))  // Blue → Cyan → Green → Yellow →
```

```
Red
        v = v + step

    return colors

function getSpectrum(v):  // v in [0, 1]
    // Maps v to rainbow colors: 0=blue, 0.5=green, 1=red
    // Implementation varies; typical approach uses HSV color space
```

3. High-Contrast Mode

Linear interpolation between blue and red:

```
function calculateHighContrastColors(nIsobars):
    colors = []
    step = 255 / (nIsobars - 1)

    for i = 0 to nIsobars - 1:
        r = round(i * step)
        g = 50
        b = round(255 - i * step)
        colors.append((r, g, b))

    return colors
```

Part II: Filled Contour Algorithm (Adaptive Recursive Subdivision)

Source File: `jpgraph_contourf.php` — Class `ContourWorker` (~1000 lines)

The filled contour algorithm is substantially more complex than the line-only version. It must not only find contour lines but also **fill the regions between them** with appropriate colors.

Algorithm Overview

```
┌─────────────────────────────────────────────────────────────────┐
│           ADAPTIVE RECURSIVE SUBDIVISION ALGORITHM              │
├─────────────────────────────────────────────────────────────────┤
│  INPUT:                                                         │
│    • Data matrix Z[rows][cols]                                  │
│    • Contour values {c₀, c₁, ..., cₙ}                          │
│    • Contour colors {color₀, color₁, ..., colorₙ₊₁}            │
│    • Maximum recursion depth (default: 6)                       │
│    • Subdivision method: 'rect' or 'tri'                        │
│                                                                 │
│  OUTPUT:                                                        │
```

```
│    • Filled polygons colored by contour band                    │
│    • Optional contour lines                                     │
│    • Optional contour labels                                    │
│                                                                 │
│  CORE IDEA:                                                     │
│    Recursively subdivide cells until either:                    │
│    (a) Max depth reached → fill with average color              │
│    (b) At most one contour crosses each edge → fill directly    │
└─────────────────────────────────────────────────────────────────┘
```

Key Concepts

Contour Bands and Color Indexing

With n contour levels, there are $n+1$ **bands** (regions between contours):

```
Contour levels: c₀=20, c₁=40, c₂=60, c₃=80

VALUE RANGE          BAND INDEX    COLOR
──────────────────────────────────────────
v < 20               0             color[0]
20 ≤ v < 40          1             color[1]
40 ≤ v < 60          2             color[2]
60 ≤ v < 80          3             color[3]
v ≥ 80               4             color[4]

Total: n=4 contour levels → n+1=5 colors needed
```

Key function — GetNextHigherContourIdx:

```
function GetNextHigherContourIdx(value, contourValues):
    // Returns the index of the first contour level > value
    // This is also the color band index for this value

    for i = 0 to length(contourValues) - 1:
        if value < contourValues[i]:
            return i

    return length(contourValues)  // Value is above all contours
```

> ⚠ **KEY INSIGHT**: The band index for a value directly determines its fill color. Values in the same band get the same color.

Termination Conditions

The recursion terminates when **either** condition is met:

Condition A: Maximum Depth Reached

```
if depth > maxDepth:
    avgValue = (v1 + v2 + v3 + v4) / 4   // For rectangles
    color = GetColor(avgValue)
    fillPolygon(vertices, color)
    return
```

Condition B: Simple Configuration

- For rectangles: All four corners in the same band, OR exactly 2 edge crossings with the same contour, OR exactly 4 crossings (saddle point).
- For triangles: All three corners in the same band, OR exactly 2 edge crossings with the same contour.

Subdivision Methods

Method 1: Rectangular Subdivision

Each cell is recursively split into four sub-rectangles:

```
    Initial rectangle:

    v1 ──────────────────────────── v2
      │                             │
      │                             │
      │            vc               │
      │                             │
      │                             │
      │                             │
    v4 ──────────────────────────── v3

    After one subdivision:

    v1 ─────────v12──────────────── v2
      │           │                 │
      │   NW      │      NE         │
      │           │                 │
    v14─────────vc──────────────── v23
      │           │                 │
      │   SW      │      SE         │
      │           │                 │
    v4 ─────────v34──────────────── v3

    New vertex values (averages):
    v12 = (v1 + v2) / 2
    v23 = (v2 + v3) / 2
    v34 = (v3 + v4) / 2
    v14 = (v1 + v4) / 2
    vc  = (v1 + v2 + v3 + v4) / 4

    Coordinates:
    vc is at ((x1+x4)/2, (y1+y2)/2) = center of rectangle
```

Recursive calls:

```
// After computing vc, v12, v23, v34, v14 and their coordinates:

// Northwest (top-left)
RectFill(v12, v2, v23, vc,
         x1,yc,  x2,y2,  xc,y2,  xc,yc,  depth+1)

// Northeast (top-right)
RectFill(vc, v23, v3, v34,
         xc,yc,  xc,y2,  x3,y3,  x3,yc,  depth+1)

// Southwest (bottom-left)
RectFill(v1, v12, vc, v14,
         x1,y1,  x1,yc,  xc,yc,  xc,y4,  depth+1)

// Southeast (bottom-right)
RectFill(v14, vc, v34, v4,
         xc,y1,  xc,yc,  x3,yc,  x4,y4,  depth+1)
```

Method 2: Triangular Subdivision

Each cell is first split into two triangles, then each triangle is recursively subdivided into four smaller triangles:

```
    Initial cell split into two triangles:

    v1 ──────────────────── v2
     | \                    |
     |   \     Upper        |
     |     \                |
     |       \              |
     |         \            |
     |   Lower   \          |
     |             \        |
     |               \      |
    v4 ──────────────────── v3

    Upper triangle: (v4, v1, v2)  →  TriFill(v4,v1,v2, x4,y4, x1,y1,
x2,y2, 0)
    Lower triangle: (v4, v2, v3)  →  TriFill(v4,v2,v3, x4,y4, x2,y2,
x3,y3, 0)


    Triangle subdivision (into 4 sub-triangles):

           v1
          /  \
         /    \
      v12------v13
      / \      / \
     /   \    /   \
```

```
 v2——v23———v3

Where:
v12 = (v1 + v2) / 2, at midpoint of edge (v1, v2)
v23 = (v2 + v3) / 2, at midpoint of edge (v2, v3)
v13 = (v1 + v3) / 2, at midpoint of edge (v1, v3)

Four sub-triangles:
1. (v1, v12, v13)   - top corner
2. (v12, v2, v23)   - left corner
3. (v13, v12, v23)  - center (inverted)
4. (v13, v23, v3)   - right corner
```

> ⚠ **WHY TRIANGLES?**: Triangular subdivision often produces smoother results because triangles have only three edges, making contour crossing analysis simpler. There's no saddle point ambiguity with triangles.

RectFill Algorithm — Complete Detail

Step 1: Check Max Depth

```
function RectFill(v1, v2, v3, v4, x1, y1, x2, y2, x3, y3, x4, y4, depth):
    // v1,v2,v3,v4 = values at corners (counter-clockwise from top-left)
    // (x1,y1)...(x4,y4) = coordinates of corners

    if depth > maxdepth:
        // Terminate: fill with average color
        color = GetColor((v1 + v2 + v3 + v4) / 4)
        fillPolygon([x1,y1, x2,y2, x3,y3, x4,y4], color)
        return
```

Step 2: Perturbation

```
    // Avoid vertex-on-contour degeneracy
    Perturbate(v1, v2, v3, v4)  // Modifies values in-place
```

Perturbation function:

```
function Perturbate(v1, v2, v3, v4):
    PERT = 0.9999
    for each contourValue in contourValues:
        if v1 == contourValue: v1 = v1 * PERT
        if v2 == contourValue: v2 = v2 * PERT
        if v3 == contourValue: v3 = v3 * PERT
        if v4 == contourValue: v4 = v4 * PERT
```

Step 3: Classify Corners by Band

```
vv1 = GetNextHigherContourIdx(v1)
vv2 = GetNextHigherContourIdx(v2)
vv3 = GetNextHigherContourIdx(v3)
vv4 = GetNextHigherContourIdx(v4)
```

Step 4: Check for Uniform Fill

```
if vv1 == vv2 == vv3 == vv4:
    // All corners in same band → single color fill
    color = GetColor(v1)
    fillPolygon([x1,y1, x2,y2, x3,y3, x4,y4], color)
    return
```

Step 5: Analyze Edge Crossings

```
// Compute difference in band index across each edge
dv1 = abs(vv1 - vv2)  // Top edge (v1 to v2)
dv2 = abs(vv2 - vv3)  // Right edge (v2 to v3)
dv3 = abs(vv3 - vv4)  // Bottom edge (v3 to v4)
dv4 = abs(vv1 - vv4)  // Left edge (v1 to v4)

// dv == 1 means exactly ONE contour crosses this edge
// dv == 0 means NO contour crosses this edge
// dv > 1 means MULTIPLE contours cross this edge → must subdivide
```

Step 6: Handle Simple Cases

Case: fcnt=2, totdv=2 (exactly two single crossings)

This means one contour line crosses exactly two edges. Fill the two resulting polygons.

```
// Interpolate crossing points
fcnt = 0
if dv1 == 1:
    (x1p, y1p, v1p) = interp2(x1,y1, x2,y2, v1,v2)
    fcnt++
if dv2 == 1:
    (x2p, y2p, v2p) = interp2(x2,y2, x3,y3, v2,v3)
    fcnt++
if dv3 == 1:
    (x3p, y3p, v3p) = interp2(x3,y3, x4,y4, v3,v4)
    fcnt++
if dv4 == 1:
```

```
        (x4p, y4p, v4p) = interp2(x4,y4, x1,y1, v4,v1)
        fcnt++

    totdv = dv1 + dv2 + dv3 + dv4

    if fcnt == 2 and totdv == 2:
        // Handle each of the six edge combinations...
```

All six 2-crossing configurations:

```
Configuration 1: TOP & RIGHT (dv1=1, dv2=1)

    v1 ──────×p1──────── v2
     |      /            |
     |    /              ×p2
     |  /                |
     |                   |
    v4 ──────────────────v3

Polygon 1: (p1, v2, p2)         → color of v2
Polygon 2: (v1, p1, p2, v3, v4) → color of v4


Configuration 2: TOP & BOTTOM (dv1=1, dv3=1)

    v1 ──────×p1──────── v2
     |        |          |
     |        |          |
     |        |          |
     |        |          |
     |        |          |
    v4 ──────×p3──────── v3

Polygon 1: (p1, v2, v3, p3)     → color of v2
Polygon 2: (v1, p1, p3, v4)     → color of v4


Configuration 3: TOP & LEFT (dv1=1, dv4=1)

    v1 ──────×p1──────── v2
     |        \          |
    ×p4        \         |
     |          \        |
     |           \       |
     |            \      |
    v4 ──────────────────v3

Polygon 1: (v1, p1, p4)         → color of v1
Polygon 2: (p1, v2, v3, v4, p4) → color of v3


Configuration 4: LEFT & RIGHT (dv2=1, dv4=1)

    v1 ──────────────────── v2
```

```
     |                     |
    ×p4———————————————————×p2
     |                     |
     |                     |
     |                     |
    v4 ——————————————————— v3

Polygon 1: (v1, v2, p2, p4)     → color of v1
Polygon 2: (p4, p2, v3, v4)     → color of v3


Configuration 5: RIGHT & BOTTOM (dv2=1, dv3=1)

    v1 ——————————————————— v2
     |                     |
     |                    ×p2
     |                  /  |
     |                /    |
     |              /      |
    v4 ———×p3———————————— v3

Polygon 1: (v1, v2, p2, p3, v4) → color of v1
Polygon 2: (p2, v3, p3)         → color of v3


Configuration 6: LEFT & BOTTOM (dv3=1, dv4=1)

    v1 ——————————————————— v2
     |                     |
    ×p4                    |
     | \                   |
     |   \                 |
     |     \               |
    v4 ———×p3———————————— v3

Polygon 1: (v1, v2, v3, p3, p4) → color of v1
Polygon 2: (p4, p3, v4)         → color of v4
```

Step 7: Handle Saddle Point (fcnt=4, totdv=4)

```
    if fcnt == 4 and totdv == 4:
        vc = (v1 + v2 + v3 + v4) / 4

        // Check if all four crossings are the SAME contour
        if v1p == v2p == v3p == v4p:
            // Saddle with single contour
            if GetNextHigherContourIdx(vc) == GetNextHigherContourIdx(v1):
                // "\" orientation
                orientation = BACKSLASH
            else:
                // "/" orientation
                orientation = FORWARDSLASH
        else:
            // Two different contours crossing
```

```
            if v1p == v2p:
                orientation = FORWARDSLASH
            else:
                orientation = BACKSLASH
```

Saddle point fill patterns:

```
BACKSLASH "\" orientation:

    v1 ———×p1——————— v2
     | P1 \           |
    ×p4———— \ ————————×p2
     |       \   P2   |
     |        \       |
    v4 ———×p3——————— v3

    Polygon 1 (P1): (v1, p1, p4)           → color of v1
    Polygon 2 (P2): (p1, v2, p2, p3, v4, p4) → color of v2
    Polygon 3 (P3): (p2, v3, p3)           → color of v1 (same as P1)

    Contour lines: p1↔p4 and p2↔p3


FORWARDSLASH "/" orientation:

    v1 ———×p1——————— v2
     |      /   P1   |
    ×p4———— / ————————×p2
     |   /    P2     |
     | P3            |
    v4 ———×p3——————— v3

    Polygon 1 (P1): (p1, v2, p2)           → color of v2
    Polygon 2 (P2): (p1, p2, v3, p3, p4, v1) → color of v3
    Polygon 3 (P3): (p4, p3, v4)           → color of v2 (same as P1)

    Contour lines: p1↔p2 and p4↔p3
```

Step 8: Recursive Subdivision

If none of the simple cases apply, subdivide and recurse:

```
    else:
        // Need to subdivide
        vc = (v1 + v2 + v3 + v4) / 4
        xc = (x1 + x4) / 2
        yc = (y1 + y2) / 2

        // Also compute edge midpoint values
```

```
        v12 = (v1 + v2) / 2
        v23 = (v2 + v3) / 2
        v34 = (v3 + v4) / 2
        v14 = (v1 + v4) / 2

        // Recurse on four sub-rectangles
        RectFill(v12, v2, v23, vc, ...)    // Top-right
        RectFill(vc, v23, v3, v34, ...)    // Bottom-right
        RectFill(v1, v12, vc, v14, ...)    // Top-left
        RectFill(v14, vc, v34, v4, ...)    // Bottom-left
```

TriFill Algorithm — Complete Detail

The triangular version is simpler because triangles have no saddle point ambiguity.

Termination Case: Max Depth

```
function TriFill(v1, v2, v3, x1, y1, x2, y2, x3, y3, depth):
    if depth > maxdepth:
        color = GetColor((v1 + v2 + v3) / 3)
        fillPolygon([x1,y1, x2,y2, x3,y3], color)
        return
```

Perturbation

```
    dummy = 0
    Pertubate(v1, v2, v3, dummy)  // Fourth param unused
```

Classify Corners

```
    vv1 = GetNextHigherContourIdx(v1)
    vv2 = GetNextHigherContourIdx(v2)
    vv3 = GetNextHigherContourIdx(v3)
```

Uniform Fill

```
    if vv1 == vv2 and vv2 == vv3:
        color = GetColor(v1)
        fillPolygon([x1,y1, x2,y2, x3,y3], color)
        return
```

Edge Analysis

```
dv1 = abs(vv1 - vv2)  // Edge v1-v2
dv2 = abs(vv2 - vv3)  // Edge v2-v3
dv3 = abs(vv1 - vv3)  // Edge v1-v3
```

Compute Midpoints

```
// Always compute midpoints (used for either termination or recursion)

if dv1 == 1:
    (x1p, y1p, v1p) = interp2(x1,y1, x2,y2, v1,v2)
    fcnt++
else:
    x1p = (x1 + x2) / 2
    y1p = (y1 + y2) / 2
    v1p = (v1 + v2) / 2

// Similarly for edges 2 and 3...
```

Simple Case: Two Crossings

```
EPSILON = 0.0001

if fcnt == 2:
    // Check which two edges have the crossing
    if abs(v1p - v2p) < EPSILON and dv1 == 1 and dv2 == 1:
        // Contour crosses edges v1-v2 and v2-v3

        Quad: (v1, p1, p2, v3)     → fill with color of v1
        Tri:  (p1, v2, p2)         → fill with color of v2
        Line: p1 ↔ p2             → contour line

    else if abs(v1p - v3p) < EPSILON and dv1 == 1 and dv3 == 1:
        // Contour crosses edges v1-v2 and v1-v3

        Quad: (p1, v2, v3, p3)     → fill with color of v2
        Tri:  (v1, p1, p3)         → fill with color of v1
        Line: p1 ↔ p3             → contour line

    else if abs(v2p - v3p) < EPSILON and dv2 == 1 and dv3 == 1:
        // Contour crosses edges v2-v3 and v1-v3

        Quad: (v1, v2, p2, p3)     → fill with color of v2
        Tri:  (p3, p2, v3)         → fill with color of v3
        Line: p3 ↔ p2             → contour line
```

Visual representation of triangle crossing cases:

```
Case 1: Edges v1–v2 and v2–v3 crossed

          v1
         /× p1
        /  /
       /  /
      /  /
     /  /
    v2×————————v3
       p2

    Quad (v1, p1, p2, v3): color of v1
    Tri (p1, v2, p2):      color of v2


Case 2: Edges v1–v2 and v1–v3 crossed

          v1
        ×/ \×
      p1/   \p3
       /     \
      /       \
     /         \
    v2—————————v3

    Tri (v1, p1, p3):      color of v1
    Quad (p1, v2, v3, p3): color of v2


Case 3: Edges v2–v3 and v1–v3 crossed

          v1
         /  \
        /    \
       /      \
      /       ×p3
     /       /
    v2———×——v3
         p2

    Quad (v1, v2, p2, p3): color of v2
    Tri (p3, p2, v3):      color of v3
```

Recursive Subdivision

```
    else:
        // Must subdivide into four sub-triangles
        //
        //         v1
        //        /  \
        //       / T1 \
        //   p1 ———————— p3
        //     /\  T3  /\
```

```
        //   /T2\    /T4\
        //  v2——p2——v3
        //
        TriFill(v1,  v1p, v3p, x1, y1, x1p, y1p, x3p, y3p, depth+1)
        TriFill(v1p, v2,  v2p, x1p, y1p, x2, y2, x2p, y2p, depth+1)
        TriFill(v3p, v1p, v2p, x3p, y3p, x1p, y1p, x2p, y2p, depth+1)
        TriFill(v3p, v2p, v3,  x3p, y3p, x2p, y2p, x3, y3, depth+1)
```

Linear Interpolation Function

The `interp2` function finds where a contour crosses an edge:

```
function interp2(x1, y1, x2, y2, v1, v2):
    // Find the contour value between v1 and v2
    cv = GetContVal(min(v1, v2))

    // Compute interpolation factor
    alpha = (v1 - cv) / (v1 - v2)

    // Interpolate coordinates
    xp = x1 * (1 - alpha) + x2 * alpha
    yp = y1 * (1 - alpha) + y2 * alpha

    // Also return the contour value (for saddle point checks)
    vp = v1 + alpha * (v2 - v1)

    return (xp, yp, vp)
```

Label Placement Algorithm

Contour labels are placed at midpoints of contour line segments, with collision avoidance:

```
function PutLabel(x, y, x2, y2, contourValue):
    // Calculate angle for label rotation
    if x2 - x != 0:
        gradient = (y2 - y) / (x2 - x)
        angle = -atan(gradient) * 180 / PI
    else:
        angle = 0

    // Check for nearby labels
    if not LabelProx(x, y, contourValue):
        // No collision, add label
        labels[GetNextHigherContourIdx(contourValue)].append(
            (x, y, contourValue, angle)
        )

function LabelProx(x, y, value):
    // Check if too close to plot edges
```

```
    if x < 30 or x > width - 20:
        return true
    if y < 20 or y > height - 20:
        return true

    // Get existing labels for this contour
    idx = GetNextHigherContourIdx(value)
    existingLabels = labels[idx]

    // Compute minimum distance to any existing label
    minDist = INFINITY
    for each (lx, ly, lv, la) in existingLabels:
        dist = (x - lx)² + (y - ly)²
        minDist = min(minDist, dist)

    // Distance threshold based on plot size
    limit = max(min(width * height / 9, 10000), 3000)

    return minDist < limit / 2
```

Part III: Mesh Interpolation

Source File: `jpgraph_meshinterpolate.inc.php`

Both contour algorithms support **data upsampling** to produce smoother results from coarse input data.

Algorithm: Recursive Bilinear Interpolation

Size Calculation

For an $m \times n$ input matrix with interpolation factor f:

$$ \text{output rows} = (m-1) \cdot 2^{f-1} + 1 $$ $$ \text{output cols} = (n-1) \cdot 2^{f-1} + 1 $$

Example calculations:

Input	Factor	Output
5×5	1	5×5 (no change)
5×5	2	9×9
5×5	3	17×17
5×5	4	33×33
10×10	3	37×37
10×10	5	145×145

Algorithm Steps

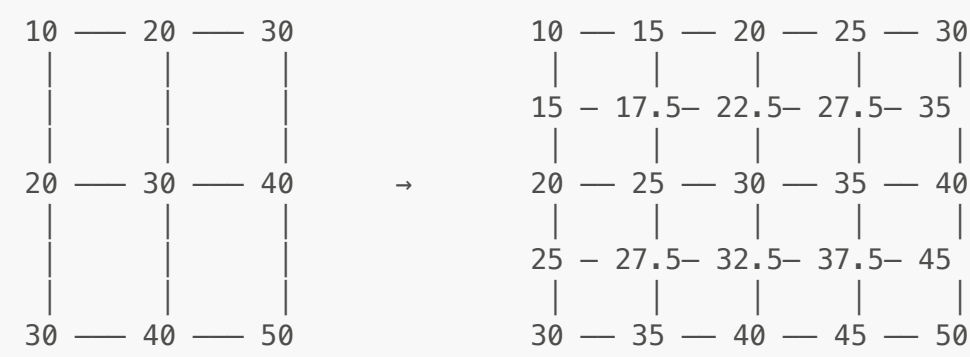
```
function LinearInterpolate(inputData, factor):
    step = 2^(factor - 1)

    inputRows = rows(inputData)
    inputCols = cols(inputData)

    outputRows = (inputRows - 1) * step + 1
    outputCols = (inputCols - 1) * step + 1

    // Create output matrix
    output = new Matrix(outputRows, outputCols, initialValue=0)

    // Step 1: Place original values at stride positions
    for i = 0 to outputRows - 1:
        for j = 0 to outputCols - 1:
            if (i mod step == 0) and (j mod step == 0):
                output[i][j] = inputData[i / step][j / step]

    // Step 2: Recursively interpolate each cell
    for i = 0 to outputRows - 1 by step:
        for j = 0 to outputCols - 1 by step:
            IntSquare(i, j, factor, output)

    return output
```

IntSquare: Recursive Cell Interpolation

```
function IntSquare(row, col, factor, output):
    if factor <= 1:
        return  // Base case: nothing to interpolate

    step = 2^(factor - 1)

    // Get the four corner values
    v0 = output[row][col]                       // Top-left
    v1 = output[row][col + step]                // Top-right
    v2 = output[row + step][col]                // Bottom-left
    v3 = output[row + step][col + step]         // Bottom-right

    // Compute and place the five interpolated values
    halfStep = step / 2

    output[row][col + halfStep]             = (v0 + v1) / 2  // Top edge
midpoint
    output[row + halfStep][col]             = (v0 + v2) / 2  // Left edge
midpoint
    output[row + step][col + halfStep]      = (v2 + v3) / 2  // Bottom
edge midpoint
    output[row + halfStep][col + step]      = (v1 + v3) / 2  // Right
edge midpoint
```

```
    output[row + halfStep][col + halfStep]   = (v0 + v1 + v2 + v3) / 4  //
Center

    // Recurse on four sub-squares
    IntSquare(row, col, factor - 1, output)
    IntSquare(row, col + halfStep, factor - 1, output)
    IntSquare(row + halfStep, col, factor - 1, output)
    IntSquare(row + halfStep, col + halfStep, factor - 1, output)
```

Visual Example



Implementation Notes and Common Pitfalls

Critical Implementation Details

1. Perturbation is Essential

> ⚠ **Never skip perturbation.** Without it, contour lines passing exactly through vertices create undefined behavior and visual artifacts.

Both algorithms use perturbation:

- Line-only: Perturb by factor of `1.001` (add 0.1%)
- Filled: Perturb by factor of `0.9999` (subtract 0.01%)

2. Color Array Size for Filled Contours

> ⚠ **For filled contours with n levels, you need $n+1$ colors.**

```
Contour levels:  [20, 40, 60, 80]  // 4 levels
Colors needed:   [c0, c1, c2, c3, c4]  // 5 colors
```

```
c0 = color for values < 20
c1 = color for values 20 ≤ v < 40
c2 = color for values 40 ≤ v < 60
c3 = color for values 60 ≤ v < 80
c4 = color for values ≥ 80
```

3. Coordinate System Orientation

The algorithms support two coordinate orientations:

```
Default (invert=false):          Inverted (invert=true):
Y increases upward               Y increases downward

    ↑ Y                              Row 0 ──────── (top)
    │                                Row 1
Row 2│                               Row 2 ──────── (bottom)
Row 1│
Row 0├─────→ X                           ↓ Y
```

4. Numerical Stability

Always include epsilon checks when dividing:

```
BAD:
x = (c - v1) / (v1 - v2)  // Crashes if v1 ≈ v2

GOOD:
d = abs(v1 - v2)
if d > 0.001:
    x = (c - v1) / d
else:
    x = 0  // Or handle degeneracy
```

5. Polygon Winding Order

When filling polygons, maintain consistent **counter-clockwise winding** to ensure correct rendering with graphics APIs that distinguish front/back faces.

6. Edge Index Consistency

The line-only algorithm uses specific edge ordering:

```
When analyzing cell (row, col), crossings are collected as:
  [0] = TOP edge    (horizontal at row)
  [1] = BOTTOM edge (horizontal at row+1)
  [2] = LEFT edge   (vertical at col)
  [3] = RIGHT edge  (vertical at col+1)
```

```
Saddle point pairing uses these indices:
  "\" pattern: connect 0↔3, 2↔1 (TOP↔RIGHT, LEFT↔BOTTOM)
  "/" pattern: connect 0↔2, 3↔1 (TOP↔LEFT, RIGHT↔BOTTOM)
```

Performance Considerations

1. Recursion Depth

Maximum depth of 6 means up to 64 subdivisions per cell. For a 100×100 grid:

- Worst case: 99×99×64 = 627,264 recursive calls
- Use iterative approaches for very large datasets

2. Memory Usage

Interpolation factor 5 expands memory by ~256×:

- 100×100 input → 6497×6497 output ≈ 42 million values

3. Caching Edge Calculations

For multiple isobars, edge crossing detection is repeated. Consider caching for performance-critical applications.

Testing Recommendations

Test Cases for Line-Only Algorithm

1. **Uniform data** — All values same. Expect: no contour lines.
2. **Linear gradient** — Values increase uniformly. Expect: parallel straight lines.
3. **Single peak** — Local maximum in center. Expect: concentric closed curves.
4. **Saddle surface** — $z = x^2 - y^2$. Expect: hyperbolic crossing pattern.
5. **Vertex on isobar** — Force $Z[i][j] = c_k$ exactly. Verify perturbation handles it.

Test Cases for Filled Algorithm

1. **Two-value data** — Only values above/below single contour. Expect: two colors, clean boundary.
2. **Checkerboard pattern** — Alternating high/low. Expect: complex saddle handling.
3. **All corners same band** — Verify uniform fill without contour lines.
4. **Maximum depth reached** — Use very coarse data with high contour count.

Common Bugs and Their Symptoms

Symptom	Likely Cause
Missing contour segments	Edge matrix indexing error
Contours crossing incorrectly at saddles	Wrong saddle orientation logic
"Jagged" filled contours	Insufficient recursion depth

Symptom	Likely Cause
Colors off by one	Band index calculation error
Crash on specific data	Missing perturbation; division by zero
Inverted image	Y-coordinate orientation mismatch

Summary

The JpGraph contour algorithms implement two complementary approaches:

1. **Line-Only (Marching-Edges)**: Efficient cell-by-cell edge analysis producing disconnected line segments. Simple to implement but requires separate handling of saddle points.
2. **Filled Contour (Adaptive Recursive Subdivision)**: Sophisticated recursive algorithm that subdivides cells until contours can be cleanly filled. Handles complexity through recursion, with smooth triangular subdivision as the preferred method.

Both algorithms share critical features:

- **Perturbation** to avoid vertex-on-contour degeneracy
- **Linear interpolation** for precise crossing coordinates
- **Band-based color assignment** for consistent visual output

The key insight enabling both algorithms is the **sign-change test** $(c - v_1)(c - v_2) < 0$ for detecting edge crossings, combined with careful handling of the **saddle point** ambiguity through center-value comparison.