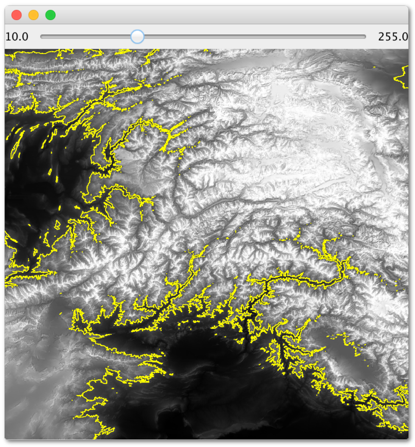
CS 330 Homework
Iso

1 Overview

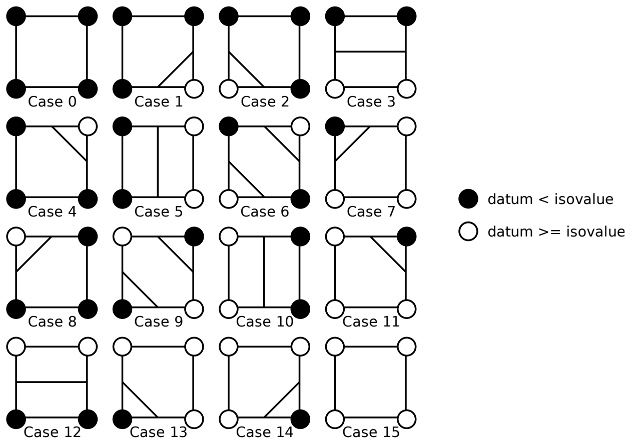
Your responsibility in this homework is to combine object-oriented, imperative, and functional programming into an application that extracts contour lines from elevation datasets. You will learn about the marching squares algorithm, an embarrassingly parallel routine for computing contour lines. You will also use the Scala language. Like Ruby, Scala combines all three programming paradigms, but it retains static typing and runs off the Java Virtual Machine.

1.1 Marching Squares Algorithm

The marching squares algorithm extracts contour lines from a two-dimensional grid of data. The lines follow a selected contour or *isovalue*. You've likely seen such lines on topographical maps (showing elevation thresholds, as in Figure 1a) or weather maps (showing temperature boundaries). The algorithm literally involves marching through each 2×2 cell in the grid and seeing how the values at the four corners relate to the isovalue. Each corner's datum is either greater than or equal to the isovalue or less than it. This is true for all four corners of a square, so in total, there are 16 possible relationships that a cell may have with an isovalue, as shown in Figure 1b. A black circle indicates a corner is less than the isovalue and white indicates otherwise. If an edge has one corner black and one corner white, then we know the contour line must pass through it somewhere. Convince yourself of this before moving on.



(a) An elevation map of Afghanistan with a highlighted contour line.



(b) The 16 possible relations a 2x2 cell may have with an isovalue.

By examining the configuration of all corners, we can determine how the contour line passes through the square. If all corners' elevations are less than the isovalue (case 0), the contour does not pass through the square. Likewise for case 15, where all elevations are greater than the isovalue. In some cases (6 and 9), two line segments pass through. Roughly, the marching squares algorithm is:

```
for each 2x2 cell in the grid
  determine the cell's configuration
  if case 0
    do nothing
  else if case 1
    run segment from bottom to right
  else if case 2
    run segment from bottom to left
  ...
```

After visiting every square and extracting the line segments which pass through them, the individual segments can be plotted to form a coherent contour line.

For each intersected edge, we must determine the coordinates at which the intersection occurs. To do this, you can interpolate the intersection point based on how the data changes between the two corners. Suppose you have an edge from one grid point (50, 13)—which we'll call the *fore* point—to the *aft* grid point to the right (51, 13). The fore point has value 10 and the aft has value 20. The isovalue chosen by the user is 19. Where should the contour line pass through the edge?

Your intuition should tell you the contour line should pass pretty close to the aft point since 19 is much closer to 20 than 10. The exact formula you must use for interpolating is:

$$\text{intersection position} = \text{fore position} + \frac{\text{isovalue} - \text{fore value}}{\text{aft value} - \text{fore value}}$$

Using this formula, the intersection point is (50.9, 13). For this intersection, we only needed to interpolate between the x-coordinates. The edge is on $y = 13$, so the intersection point must have y-coordinate 13.

Consider the figure above once again. Are there really 16 cases? Notice how cases 8–15 mirror cases 0–7. In your implementation, you can save yourself some work by coalescing cases 0 and 15, 1 and 14, 2 and 13, 3 and 12, etc.

2 Requirements

In order to complete this homework, you must satisfy these requirements:

1. Place all code in directory `<YOUR-REPOSITORY>/iso`.
2. All code must run on `thing-0[456]`.
3. Generate an elevation map in the `FloatGrid` format, described below. You can generate this from an image, the National Elevation Dataset, an algorithm, etc. How you generate it is not specified. Share on Piazza both your generated elevation dataset and a screenshot of it rendered with contour lines using `IsoFrame`. Use folder `iso`.

4. Write classes `Point2`, `LineSegment`, and `FloatGrid`, described below. Each is written in a file of the same name with a `.scala` extension.

2.1 Point2

This class represents a location in two-dimensional space. It has the following `public` interface:

- A constructor accepting the point's x- and y-coordinates as `floats`. The class must have getters for `x` and `y`, but no setters. The point's coordinates are immutable.
- A `toString` method that returns the point's coordinates separated by a space and using the default precision that comes from concatenating `floats` and `Strings`. For example, `new Point2(5.6, 3).toString` → `"5.6␣3.0"`.

2.2 LineSegment

This class represents a line segment in two-dimensional space. It has the following `public` interface:

- A constructor accepting the segment's endpoints `a` and `b` as `Point2`s. The class must have getters for `a` and `b`, but no setters. The endpoints are immutable.
- A `toString` method that returns the segments's endpoints separated by `"␣to␣"` and using the default precision that comes from concatenating `floats` and `Strings`. For example, `new LineSegment(new Point2(5.6, 3), new Point2(0, 1)).toString` → `"5.6␣3.0␣to␣0.0␣1.0"`.

2.3 FloatGrid

The `FloatGrid` class represents a 2D grid of `float` data—elevations, temperatures, or some other measured quantity. A measure may be any legal `float` value. It has the following `public` interface:

- A constructor taking a `String` argument for the path to a file containing the 2D data in little-endian binary format. The resolution of the data is encoded in the file's name, which has the form `basename.WIDTH.HEIGHT.bin`, where `WIDTH` and `HEIGHT` are integers. The `floats` are enumerated within the file in row major order. For example, if the file is named `canyon.4.3.bin`, then the measurements in the file are laid out in the following manner:

0, 0	1, 0	2, 0	3, 0	0, 1	1, 1	2, 1	3, 1	0, 2	1, 2	2, 2	3, 2

  Scala's primary constructors are not very self-contained. If you introduce any helper variables, they become instance variables. To prevent this, wrap up their declarations inside a block.

- Getters for the grid's `width`, `height`, `min`, and `max`, but no setters. All values are immutable.
- An `apply` method that returns the `float` measurement at a given (column, row) coordinate. The column and row are `int`s. This method affords clients of `FloatGrid` a means of subscripting into the grid:

  ```
  val grid : new FloatGrid("crater.512.1024.bin")
  for (r <- 0 until grid.height; c <- 0 until grid.width) {
    println(grid(c, r))
  }
  ```

- A method `each` that accepts a function as a parameter and invokes it on each measurement in the grid. The function expects three parameters: a column index, a row index, and the `float` measurement. It returns nothing—i.e., `Unit`. This method can be used to simplify the example above:

  ```
  val grid : new FloatGrid("crater.512.1024.bin")
  grid.each((c, r, intensity) => {
    println(grid(c, r))
  })
  ```

- A method `toImage` that returns a `BufferedImage` representation of this grid. The image has the same width and height as the grid. Each pixel is an RGB triplet where each channel is the measurement range-mapped to [0, 255]. Range-mapping is done by:

  1. Normalizing the measurement by determining its proportion within the value range of the grid. The minimum maps to 0, and the maximum to 1. A value halfway between the minimum and maximum maps to 0.5.
  2. Applying the proportion by multiplying the normalized value by 255.

  Since all channels of the color will have the same range-mapped value, the image will appear gray. However, use `BufferedImage.TYPE_INT_RGB` for the image type; other values may cause colorspace conversions.

- A generic method `mapOverCells` that accepts a function as a parameter and returns a `List` of `T`. It invokes the function in parallel on each 2x2 neighborhood in the grid. The function expects six parameters: the column and row indices of the neighborhood's lower-left corner and the measurements of the four corners in this order: bottom-left, bottom-right, top-left, and top-right. The function returns a `T`. The function can be applied in parallel by constructing `ParRange`s for the cell indices over which you iterate. The following code demonstrates how a serial `Range` can be turned in a `ParRange`:

  ```
  scala> for (i <- 0 to 9) { print(i) }
  0123456789
  scala> for (i <- (0 to 9).par) { print(i) }
  8750341962
  ```

The result of `for` on a parallel collection is a `ParSeq[T]`. Convert this to a `List[T]` using its `toList` method.

2.4 Contourer

This class was written together in lecture and is provided in your `iso` directory. It contains methods for determining if and where contour lines pass through a 2x2 cell.

2.5 IsoFrame

This class is provided in your `iso` directory. It accepts a path to a `FloatGrid` as a command-line parameter and uses your `FloatGrid.toImage` method to display the grid in a GUI. A `JSlider` lets the user extract contour values between the grid's minimum and maximum values, calling `FloatGrid.mapOverCells` on each change. Display the frame with:

```
scala -cp . IsoFrame trench.768.512.bin
```

3 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you are resubmitting.