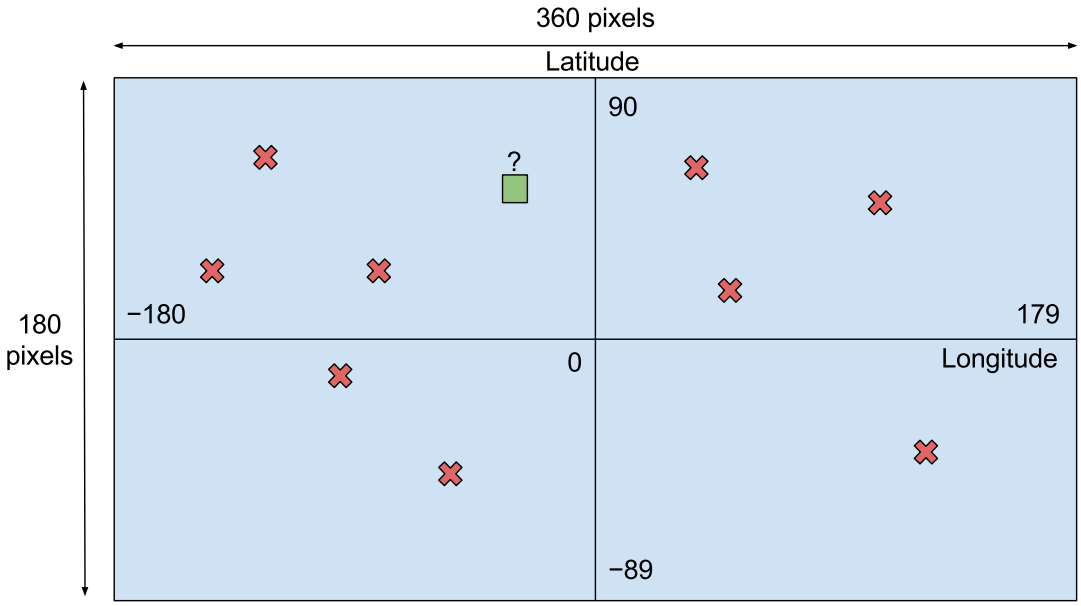
Milestone overview

This milestone consists in producing images showing the content of the temperature records. You will have to complete the file **Visualization.scala**. But first, remember to update the grading milestone number:

```
1  val milestone: Int = 2
```

Your records contain the average temperature over a year, for each station's location. Your work consists in building an image of 360×180 pixels, where each pixel shows the temperature at its location. The point at latitude 0 and longitude 0 (the intersection between the Greenwich meridian and the equator) will be at the center of the image:



In this figure, the red crosses represent the weather stations. As you can see, you will have to *spatially interpolate* the data in order to guess the temperature corresponding to the location of each pixel (such a pixel is represented by a green square in the picture). Then you will have to convert this temperature value into a pixel color based on a color scale:

This color scale means that a temperature of 60°C or above should be represented in white, a temperature of 32°C should be represented in red, a temperatures of 12°C should be represented in yellow, and so on. For temperatures between thresholds, say, between 12°C and 32°C, you will have to compute a *linear interpolation*

between the yellow and red colors.

Here are the RGB values of these colors:

Temperature (°C)	Red	Green	Blue
60	255	255	255
32	255	0	0
12	255	255	0
0	0	255	255
-15	0	0	255
-27	255	0	255
-50	33	0	107
-60	0	0	0

You can monitor your progress by submitting your work at any time during the development of this milestone. Your submission token and the list of your graded submissions is available on this page.

Spatial interpolation

You will have to implement the following method:

```
def predictTemperature(
  temperatures: Iterable[(Location, Temperature)],
  location: Location
): Temperature
```

This method takes a sequence of known temperatures at the given locations, and a location where we want to guess the temperature, and returns an estimate based on the inverse distance weighting algorithm (you can use any *p* value greater or equal to 2; try and use whatever works best for you!). To approximate the distance between two locations, we suggest you to use the great-circle distance formula.

Note that the great-circle distance formula is known to have rounding errors for short distances (a few meters), but that's not a problem for us because we don't need such a high degree of precision. Thus, you can use the first formula given on the Wikipedia page, expanded to cover some edge cases like equal locations and antipodes:

$$\Delta\sigma = \begin{cases} 0 & \text{for equal points} \\ \pi & \text{for antipodes} \\ \arccos\left(\sin\phi_1 \cdot \sin\phi_2 + \cos\phi_1 \cdot \cos\phi_2 \cdot \cos\Delta\lambda\right) & \text{otherwise} \end{cases}$$

$$d = r \cdot \Delta\sigma$$

However, running the inverse distance weighting algorithm with small distances will result in huge numbers (since we divide by the distance raised to the power of p), which can be a problem. A solution to this problem is to directly use the known temperature of the close (less than 1 km) location as a prediction.

Linear interpolation

We're providing you with a simple case class for representing color; you can see **Color**'s documentation in **models.scala** for more information.

```
case class Color(red: Int, green: Int, blue: Int)
```

You will have to implement the following method:

```
def interpolateColor(points: Iterable[(Temperature, Color)], value: Temperature
  ): Color
```

This method takes a sequence of reference temperature values and their associated color, and a temperature value, and returns an estimate of the color corresponding to the given value, by applying a linear interpolation algorithm.

Note that the given points are not sorted in a particular order.

Visualization

Once you have completed the above steps you can implement the visualize method to build an image (using the scrimage library) where each pixel shows the temperature corresponding to its location.

```
def visualize(
  temperatures: Iterable[(Location, Temperature)],
  colors: Iterable[(Temperature, Color)]
): Image
```

Note that the (x,y) coordinates of the top-left pixel is (0,0) and then the x axis grows to the right and the y axis grows to the bottom, whereas the latitude and longitude origin, (0,0), is at the center of the image, and the top-left pixel has GPS coordinates (90, -180).

Appendix: scrimage cheat sheet

Here is a description of scrimage's API parts that are relevant for your work.

- **Image** type and companion object.
- A simple way to construct an image is to use constructors that take an **Array[Pixel]** as parameter. In such a case, the array must contain exactly width × height elements, in the following order: the first element is the top-left pixel, followed by all the pixels of the top row, followed by the other rows.
- A simple way to construct a pixel from RGB values is to use this constructor.
- To write an image into a PNG file, use the **output** method. For instance: **myImage.output(new java.io.File("target/some-image.png"))**.
- To check that some predicate holds for all the pixels of an image, use the **forall** method.
- Also, note that scrimage defines a **Color** type, which could be ambiguous with our **Color** definition. Beware to not import scrimage's **Color**.

Mark as completed