GPH 573: Assignment #5

Jay Laura

Problem 1

For this assignment we were required to read a text file containing 1024 Polylines in some coordinate system and visualize them using `TKinter`. Each polyline is defined by n vertices in the format

$$(x_a, y_a, x_{a+1}, y_{a+1}, x_{a+2}, y_{a+2}, ...x_{a+m}, y_{a+m}) \quad (1)$$

For this implementation I open the input text file in read mode, create an empty list into which we will load `NumPy` arrays and iterate through the input lines, reading the polyline vertces as an array. This implementation utilizes regular expressions, available via the `regex` module. While the python documentation and numerous technical books provide a more in depth presentation of writing and using regular expressions, the following describes this particular implementation.

I utilize `re.findall(r"[-+]?\d*\.\d+|\d+",line)`, to find all instances of floating point numbers in a line. `r` indicates that the notation is in raw form. This is necessary to keep the line succinct. Without alerting Python that raw input is being used it would be necessary to pepper the line with '\ escape characters. We then utilize `[ ]` to indicate that we are generating a set and `-+` to indicate that the set will be greedy, that is it will repeat through the line. `?` then indicates that the previous greedy extraction will repeat only once, i.e. we will extract all numbers prior to a decimal point only once. `\d` locates a decimal digit, `*`, matches this n times, `.` checks all characters, `+` is greedy, and `|` indicates that we test all branches until a match occurs. For example, when examining a line, we iteratively add a character until either we have extracted a single floating point number without additional text, or we hit a line end. The `findall` function then iterates over the line again, discounting those floating points numbers it has already identified.

Note that we pack the individual polylines, as `NumPy` arrays, into a list. This is inefficient and we would be better served trying to pack each polyline into an array of arrays. Since `NumPy` does not support ragged arrays, this would require that we find the total length of the longest polyline and generate a matrix with dimensions len(polyline) by len(longest polyline). We could pack null vertices with `NaN` or `np.infinity`. This section of code is implemented as:

Listing 1: Code to read floating point coordinates from a text file using regular expressions

```
f = open(sys.argv[1])
geometries = []
for line in f:
    fl = re.findall(r"[-+]?\d*\.\d+|\d+",line)
    fl =  fl[1:]
    float_pts = [float(x) for x in fl]
    float_arr = np.asarray(float_pts)
    if float_pts:
        geometries.append(float_arr)
```

Next we setup to perform a geotransformation to convert from coordinate space to monitor space. The key to this code snippet is the use of fancy indexing. This is why I choose to pack the coordinates into `NumPy` arrays. We are able to iterate over ever other coordinate, i.e. either the x or y coordinate by indicating that we start at index 0 or index 1 and touch every other coordinate. `[0::2]` indicates that the code will iterate from index 0 to the final index in twos. `[1::2]` indicates that the code iterates in twos, starting at the first index. This is implemented as:

Listing 2: Code to compute scaling values (minimum and maximum coordinate values)

```
#Geotransform
xmax = None
ymax = None
xmin = float('inf')
ymin = float('inf')
for geometry in geometries:
    if geometry[0::2].max() > xmax:
        xmax = geometry[0::2].max()
    if geometry[0::2].min() < xmin:
        xmin = geometry[0::2].min()
    if geometry[1::2].max() > ymax:
        ymax = geometry[1::2].max()
    if geometry[1::2].min() < ymin:
        ymin = geometry[1::2].min()
```

We then compute the scaling ratio to convert between coordinate space and monitor space. It is possible to detect the monitor extent or allow the user to supply the desired screen dimensions. The implementation is robust enough that hard coded values can be replaced with variables. This is implemented as:

Listing 3: Code to compute scaling values (minimum and maximum coordinate values)

```
umax = 800
vmax = 600
ratioX = (umax - 0)/(xmax-xmin)
ratioY = (vmax - 0)/(ymax-ymin)
ratio = min(ratioX, ratioY)
```

Here we apply the scaling factor to each polyline. Again this uses `NumPy` fancy indexing and vectorized computation to avoid the use of nested `for` loops. This implementation is significantly more efficient and is implemented as:

Listing 4: Code to scale each coordinate

```
for line in geometries:
    line[0::2] = ratio * (line[0::2] - xmin)
    line[1::2] = vmax + ((-1 * ratio) * (line[1::2] - ymin))
```

Finally, we visualize the data in a TK window. Note that it is necessary to unpack the `NumPy` array into a list for visualization. We also have provided the option to visualize in various colors using the `choice` function in the `random` module. This is implemented as:

Listing 5: Code to visualize the polylines

```
#TK window
root = Tk()
can = Canvas(root, width=umax, height=vmax)
colors = ['red']#,'green','blue','yellow','orange','purple']
```

```
for line in geometries:
    line = line.tolist()
    color = choice(colors)
    color == 'black'
    can.create_line(line, fill=color)
can.pack()
root.mainloop()
```

Figure 1: Script output in the default color.

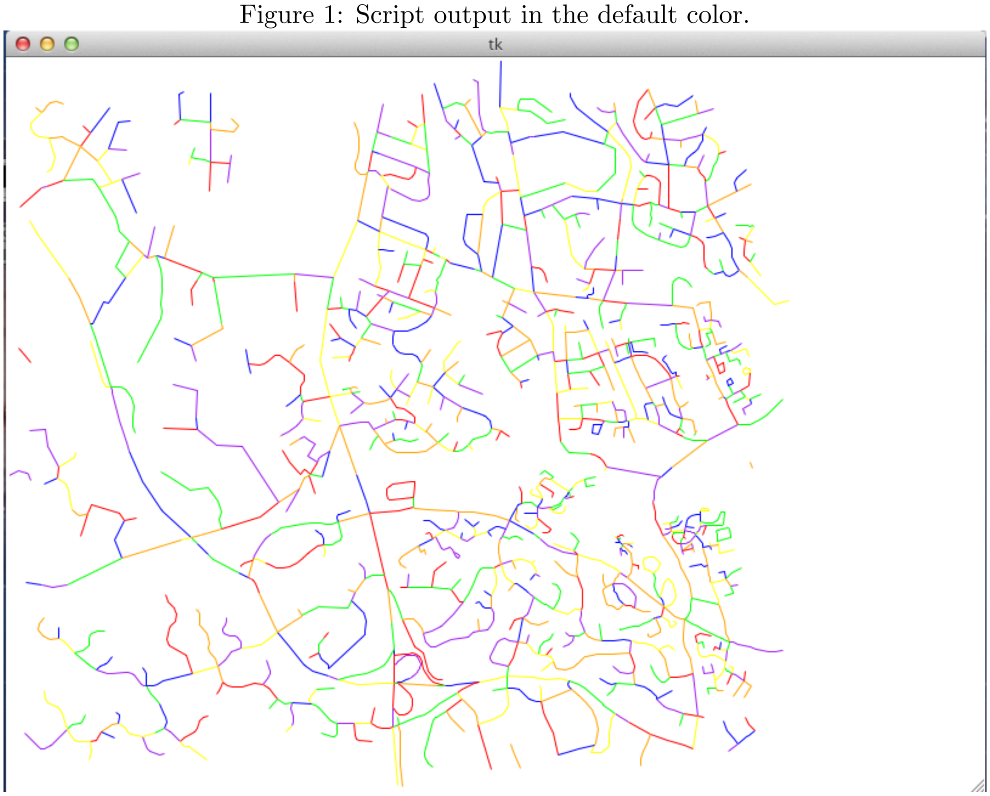

Figure 2: Script output with each polyline randomly colored.