GGR472 LAB 4: Incorporating GIS Analysis into web maps using Turf.js (6.25%)

Lab objective

Turf.js is an open-source JavaScript library with predefined functions for performing GIS analysis in the browser. For example, you can measure things like the length of linear features, compare distances, and extract points that intersect polygons.

The aim of this lab is to deepen your understanding and experience of working with Turf.js. You will use a range of Turf functions in conjunction to create a hexgrid web map based on point features which locate road collisions for pedestrians and cyclists in Toronto. Hexgrid (or hexbin) maps divide geographic areas into uniform hexagons with each polygon representing count data. Colour gradients are typically used to display the data range. Hexgrid maps can help to reduce bias produced by usual choropleth maps by presenting each geographic region equally. However, it can become difficult to recognise locations and so it's important that the map developer considers ways to incorporate layers and interactivity to improve the web map's readability.

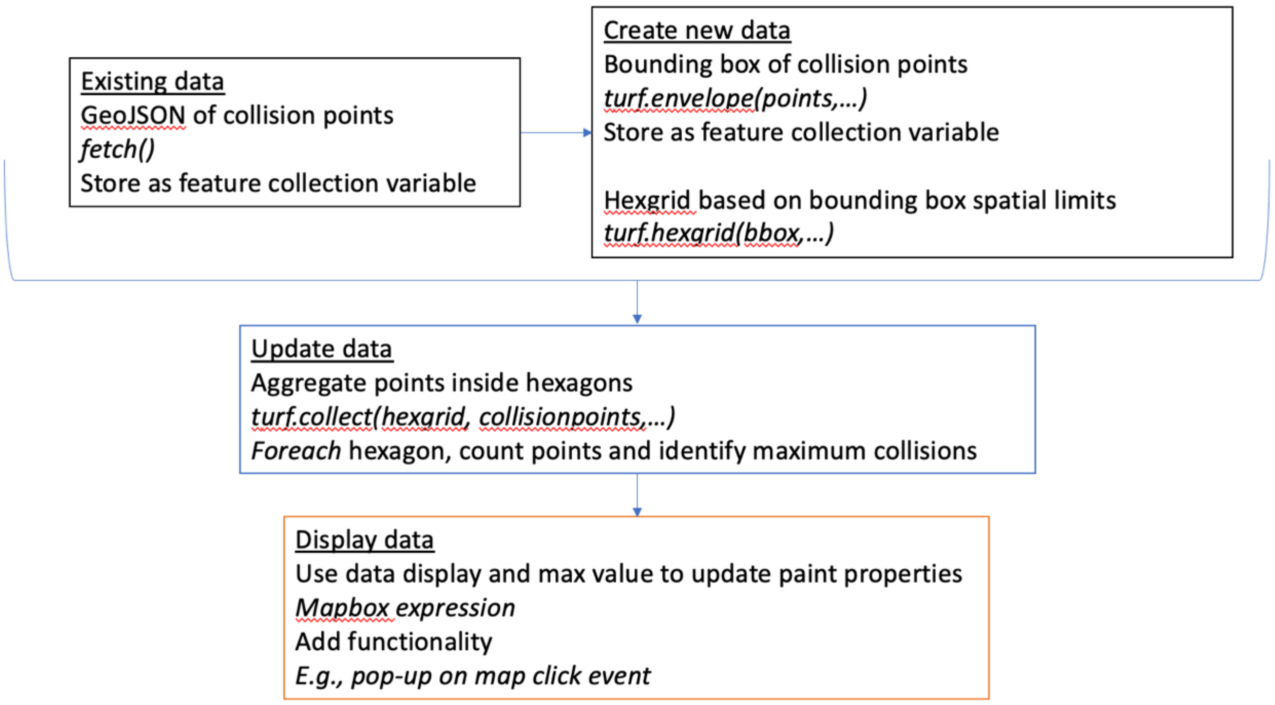
The lab is worth 6.25% of your final grade, however, keep in mind that the primary purpose of the labs is to develop knowledge and skills which may be applied in your group project. You may use the exercises covered in class as guidance.

Due Tuesday 18 March, 5pm

Create a hexgrid web map

In this lab you will create a hexgrid web map of traffic collisions involving pedestrians and cyclists in Toronto. Your data should be displayed using an appropriate colour gradient and, as a minimum, your map should include a **legend** and **pop-up windows**.

The lab will follow the stepped approach outlined below:


Existing data
GeoJSON of collision points
fetch()
Store as feature collection variable
Create new data
Bounding box of collision points
turf.envelope(points,...)
Store as feature collection variable
Hexgrid based on bounding box spatial limits
turf.hexgrid(bbox,...)
Update data
Aggregate points inside hexagons
turf.collect(hexgrid, collisionpoints,...)
Foreach hexagon, count points and identify maximum collisions
Display data
Use data display and max value to update paint properties
Mapbox expression
Add functionality
E.g., pop-up on map click event

Step 1: Set up your repository and data source

Download the ggr472-lab4 repository, including data folder, and save to your local machine.

Data represent locations of road collisions involving pedestrians or cyclists in Toronto between 2006 and 2021. The data were derived from the following source and have been edited for use in the lab.

https://open.toronto.ca/dataset/motor-vehicle-collisions-involving-killed-or-seriously-injured-persons/

Take time to explore the original and updated data (*pedcyc_collision_06-21.geojson)*. You may like to view the spatial distribution and attributes of the data points using geojson.io or GIS desktop software such as QGIS. As a minimum, take a look at the GeoJSON in the browser so that you have an understanding of the range of properties available.

Once familiar with the data, open the starter code in **Visual Studio Code**.

In the *script.js* file, add your Mapbox access token and a map style. You may use a style created by Mapbox or yourself. Feel free to update or add additional elements to your new map object.

Check the map style loads in the browser as expected then save your changes.

In **GitHub Desktop**, create a new repository for your working directory then push your new repository online.

Step 2: Store data points as a variable and view on map

Below the code that initializes the map, create a new empty variable. The variable will ultimately store the data points from the *pedcyc_collision_06-21.geojson* file.

Use the fetch function to access the GeoJSON data from your online GitHub repository and assign to the collision data variable created in Step 1. See Week 9 lecture and exercise material as a reminder of how to do this. For now, you may use the URL which references the raw data stored in your online repository (which will look something like the link shown below). Once you publish your website, you can update this to the link created by GitHub pages (see Week 5 Lecture content and code snippet).

```
fetch("https://raw.githubusercontent.com/smith-lg/ggr472-lab4/refs/heads/main/data/pedcyc-collision-18_23.geojson")
  // Add code here to 1) convert response to JSON format and 2) assign to variable
```

To check the data are stored as expected, you may like to view the new variable using console.log(). Better still, use the addSource() and addLayer() methods inside a map load event handler to view your data on the map. Remember the addSource() data source should point to the name of your collision data variable.

Step 3: Create a bounding box and hexgrid from the collision point data

If you did not create a map load event handler in Step 2, create one now. All your code to create and view a hexgrid will go inside the event handler.

To create a hexgrid, we will use the **hexGrid** function from Turf.js

https://turfjs.org/docs/api/hexGrid

Using the documentation linked above, take time to explore the hexGrid function's required arguments. Notice that it requires a bounding box as an array of coordinates representing the extent of the bounding rectangle i.e., [minX, minY, maxX, maxY].

There are a range of Turf functions we may use to create a bounding box and extract coordinates as an array, including:

https://turfjs.org/docs/api/bboxPolygon

https://turfjs.org/docs/api/bbox

https://turfjs.org/docs/api/envelope

Explore the documentation for each function above, noting the required arguments returned types.

Understanding return types and how to access required values

For now, let's explore the envelope function to understand return types and how to access required values for the bounding coordinates.

Return a feature object

Inside the map load event handler, use turf.envelope() to create a bounding box from the collision data and assign to a variable. For example:

```
// create a bounding box around the collision points and store as a variable
let envresult = turf.envelope(collisionData);
```

Use console.log() to view the output in the console. The output will be a GeoJSON **feature** object which includes a property called bbox (an array of coordinates for our bounding box).

Return a property

To access the **array property**, you can simply update your console.log to take variablename.propertyname e.g., (envresult.bbox).

Return a value

To further access a **value**, you can use the index e.g., (envresult.bbox[0]). This corresponds to the first value in the bbox array which is the minX of our bounding box.

Return output as a feature collection

At the start of the course, we learnt that a GeoJSON **feature collection** holds multiple feature objects. Should you want to include a feature (such as the result of our envelope function) in a FeatureCollection, it can be wrapped as follows. This may be useful if you require FeatureCollection as an argument, such as in the addSource method. Take care with your property names here – these are case sensitive.

```
// create feature collection holding the bbox feature
bboxgeojson = {
  "type": "Feature Collection",
  "features": [envresult]
}
```

Again, use the console log to view the output. See if you can work out what value the following code is accessing.

```
console.log(bboxgeojson.features[0].geometry.coordinates[0][0][1]);
```

Bounding box

Now we have an understanding of return types and how to access values, use one of the recommended functions to create a bounding box of the collision data. The return type must be an array of coordinates in the following order [minX, minY, maxX, maxY].

Previously we used the envelope function but of course you may find that a function such as bbox provides a valid alternative – I will leave it to you to experiment!

HexGrid

You will use the bounding coordinates of the bbox variable as an input to the hexGrid function. You may like to store the array as its own variable.

Use the **Turf.js hexGrid function** to create a grid of 0.5km hexagons inside the spatial limits of the *bbox* coordinates. Use the Turf.js documentation for guidance

Once created, view the output on the map using addSource() and addLayer() methods. Remember the resulting hexgrid variable from the Turf function will be the data source in your addSource() method.

When you view the hexgrid on the map, you may notice that not all the points at the edge of the bounding box are covered by hexagons. To fix this, you may use the Turf transformScale method on the bbox variable to increase its size by 10%, for example. Note that the transformScale function takes a feature or featurecollection object, not an array.

Take time here to format and reorder your code. For example, your code may read more clearly if you group the addSource and addLayer methods together and move all the code to make the bounding box and hexgrid to the top of the event handler.

Step 4: Aggregate collision data by hexgrid

So far we have created and viewed the hexgrid layer. Next we will associate data with it so that the number of collisions may be categorised and viewed with a colour gradient. The code will go inside the map load event handler below the code written in Steps 2 and 3.

To aggregate the point data, use the Turf collect method to count all the unique IDs from the collision data that are inside each polygon

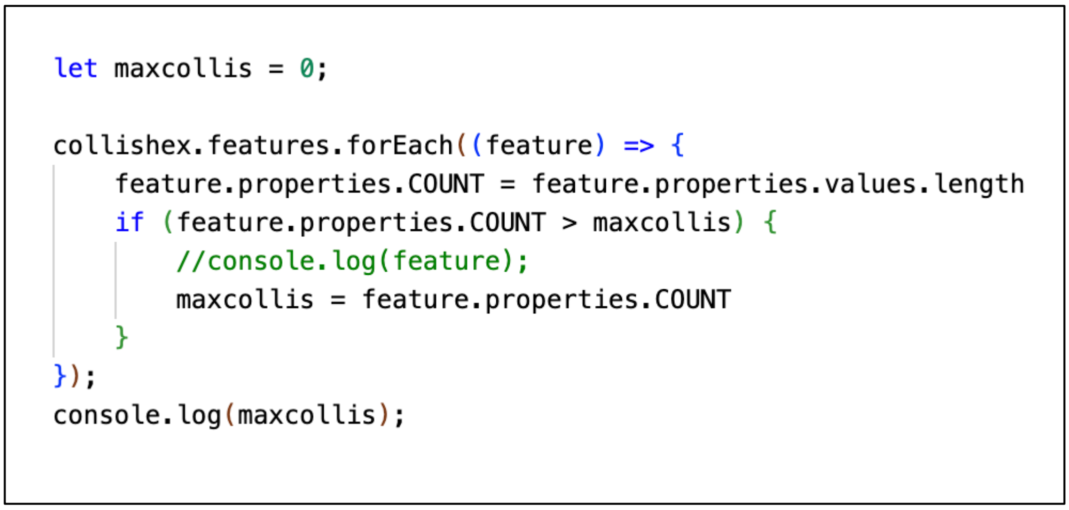
https://turfjs.org/docs/api/collect

For example:

```
let collishex = turf.collect(/* your-hexgrid-variable */, /*your-step1-collision-variable*/, '_id', 'values');
```

View the collect output in the console. Notice that where there are no points intersecting hexagons, the values array property will be empty.

Using the following code as an example, create a foreach loop which iterates through the hexagons to i) add a point count (COUNT) and ii) identify the maximum number of collisions in a polygon. Check your outputs using the console log.



```
let maxcollis = 0;

collishex.features.forEach((feature) => {
    feature.properties.COUNT = feature.properties.values.length
    if (feature.properties.COUNT > maxcollis) {
        //console.log(feature);
        maxcollis = feature.properties.COUNT
    }
});
console.log(maxcollis);
```

Add comments to the foreach loop to explain what each line does

Step 5: Finalize your web map

You now have all the data needed to create a hexgrid web map. The final part of the lab involves updating the display of the data and including different interactivity that will improve usability and readability.

Paint properties
Use the range of data based on the maximum collision value and the COUNT attribute to update the paint properties. Explore the possible ways to use expressions to do this.

Review the output of your hexgrid. How may you improve the display of the data? Perhaps by changing the size of the hexagons, changing the opacity, filtering hexagons without counts, or adding/removing other layers for clarity?

Legend
Add a legend to your map. See Week 8 lecture content for guidance and explore mapbox examples. Think about how the legend can help to explain what data are shown and whether a categorical or continuous colour scale is most appropriate.

Functionality
Add functionality to your web map. As a minimum, this should include pop-up windows but aim to incorporate additional HTML elements and events that help explain the data. For example, you may like to include the option to toggle additional layers on and off.

Step 6: Finalize your web map

Once complete, refine your code and remove any redundant lines that are not used. Add comments throughout your code to demonstrate your understanding and update the README file associated with your repository. The README file is a markdown document and can be updated within Visual Studio code. The README should provide details about the project such as its purpose and data source.

Update the files in your online repository and deploy your website using GitHub pages.

Submit a link to your GitHub repository AND a link to your website via Quercus