**Chapter 12 Client-side Geoprocessing**

**12.1 Introduction**

In Chapter [11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1), we used spatial SQL queries to load a subset of features from a PostGIS database on a web map. In other words, we performed a geoprocessing operation—finding the *n*-nearest points—using SQL. Notably, the operation was performed on the **server** (the CARTO platform), and the result was returned to us via HTTP. Sometimes, it is more convenient to perform geoprocessing operations on the **client**, rather than on the server, for various reasons.

One reason to prefer client-side geoprocessing is that we may need the web page to be *instantaneously* responsive, such as a web map that is continuously updated in response to user input (Figure [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-04)). This is very difficult to achieve when communicating with a server: even if the internet connection is very fast, there is usually going to be a noticeable lag due the time interval between sending the request and receiving the response. Another reason may be the relative simplicity and flexibility of using various algorithms and methods through JavaScript libraries, compared to the cost of setting up and managing a server with the same functionality.

The main disadvantage of client-side geoprocessing is that we are limited by the hardware constraints of the client. While the hardware of the server is (potentially) very powerful, and in any case it is under our control, the different clients that connect to our web page may have widely varying hardware. To make our website responsive, we should therefore limit client-side geoprocessing to relatively light-weight operations. Another constraint in client-side geoprocessing is that we are limited to JavaScript libraries and cannot rely on any other software or programming language (such as a PostGIS database).

In this chapter we will see several examples of **client-side geoprocessing**:

* Calculating Great Circle lines (Section [12.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#great-circle-line))
* Drawing a continuously-updated TIN (Section [12.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#continuously-updated-tin))
* Detecting spatial clusters (Section [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#clustering))
* Drawing heatmaps (Section [12.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#heatmaps-with-leafletheat))

All of these will be done in the browser, using JavaScript code, without requiring a dedicated server. In the examples, we will use two different JavaScript libraries:

* **Turf.js** (Sections [12.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#geoprocessing-with-turfjs)–[12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#clustering))—A general-purpose geoprocessing library, with a comprehensive set of functions for various geoprocessing tasks
* **Leaflet.heat** (Section [12.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#heatmaps-with-leafletheat))—A single-purpose plugin for Leaflet, dedicated to generating point-density heatmaps

**12.2 Geoprocessing with Turf.js**

**12.2.1 Overview.**[**Turf.js**](http://turfjs.org/) is a JavaScript library for client-side geoprocessing. It includes a comprehensive set of functions, covering a wide range of geoprocessing tasks. The Turf.js library also has excellent [documentation](http://turfjs.org/docs/), including a small visual example for each of its functions.

As we will see shortly, the basic data type that Turf.js works with is GeoJSON. This is very convenient, because it means that any geoprocessing product can be immediately loaded on a Leaflet map with the L.geoJSON function, which we have been using in Chapters [7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-1)–[11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1). As for the other way around, we will also see that any existing Leaflet layer can be converted back to GeoJSON using the .toGeoJSON method, so that it can passed to a Turf.js function for processing. Therefore, when working with Turf.js we typically have to go back and forth between Leaflet layer objects, which the Leaflet library understands, and GeoJSON objects, which the Turf.js library understands (Section [12.4.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#continuous-updating)).

In this chapter, we will use seven different functions from the Turf.js library (Table [12.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#tab:turf-functions)). There are dozens of other functions which we will not use; you are invited to browse through the documentation to explore what kind of other tasks can be done with Turf.js.

| TABLE 12.1: Turf.js functions used in Chapter [12](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#client-side-geoprocessing) | |
| --- | --- |
| **Function** | **Description** |
| turf.point | Convert point coordinates to GeoJSON of type "Point" |
| turf.greatCircle | Calculate a Great Circle line |
| turf.randomPoint | Calculate random points |
| turf.tin | Calculate a Triangulated Irregular Network (TIN) |
| turf.clusterEach | Apply function on each subset of GeoJSON features |
| turf.convex | Calculate a Convex Hull polygon |
| turf.clustersDbscan | Detect clusters using the DBSCAN method |

**12.2.2 Including the Turf.js library.**To use Turf.js in our web page, we first need to include it, using a <script> element. As always, the JavaScript file can be included from a local copy, or from a CDN (Section [6.5.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#loading-a-remote-script)) such as the following one:

[https://npmcdn.com/@turf/turf/turf.min.js](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\npmcdn.com\%40turf\turf%406.5.0\turf.min.js)

We will use a local copy stored in the js folder on our server. Therefore, we add the following <script> element inside the <head>:

**<script** src="js/turf.js"**></script>**

You can download the Turf.js file from the URL specified at the beginning of this section, or from the online version of the book (Section [0.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\index.html#online-version)), and place it in a local folder if you want to include a local copy in your web page.

**12.3 Great Circle line.**As a first experiment with Turf.js, we will use the library to calculate a **Great Circle** line, then display it on a Leaflet web map. A Great Circle line is the shortest path between two points on the earth surface, taking the curvature of the earth into account (Figure [12.10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:solution-12)). Through the Great Circle example, we will demonstrate the way that Turf.js functions operate on GeoJSON objects.

All functions from the Turf.js package are loaded as methods of a global object named turf, thus sharing the turf. prefix. This is much like Leaflet functions start with L (Section [6.5.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#creating-map-object)).

As an example, let’s open the documentation of the turf.greatCircle [function](http://turfjs.org/docs/#greatCircle). Here is a slightly modified code of the example given for the turf.greatCircle function in the documentation:

**let** start = turf.point([-122, 48]);

**let** end = turf.point([-77, 39]);

**let** greatCircle = turf.greatCircle(start, end);

* Open the HTML document of the basic map example-06-02.html from Section [6.5.11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-a-tile-layer).
* Add a <script> element in the <head> for including the Turf.js library.
* Open the console and execute the above three expressions, then examine the resulting objects as described below.

What does this code do? The first two expressions are using a convenience function named [turf.point](http://turfjs.org/docs/#point) to convert pairs of coordinates into a GeoJSON object of type "Feature", comprising one feature of type "Point" (Section [7.3.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#single-part-geometries)). The coordinates passed to turf.point are assumed to be of the [lon, lat] form, same as in GeoJSON. Typing JSON.stringify(start) reveals the resulting GeoJSON:

{

"type": "Feature",

"properties": {},

"geometry": {

"type": "Point",

"coordinates": [-122, 48]

}

}

Note that the [-122, 48] coordinates passed to turf.point are now in the coordinates property of the GeoJSON. The output of typing JSON.stringify(end) would be identical, except for the coordinates, which will be [-77, 39] instead of [-122, 48]. (Type JSON.stringify(end) in the console to see for yourself.)

The third expression uses the geoprocessing function turf.greatCircle to calculate the Great Circle line between the two points. The result is assigned to a variable named greatCircle. Here is the printout of JSON.stringify(greatCircle)[106](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fn106):

{

"type": "Feature",

"properties": {},

"geometry": {

"type": "LineString",

"coordinates": [

[-122, 48],

[-121.49670597260395, 48.006695584053034],

[-120.9933027193627, 48.011192319649155],

**...**,

[-77, 38.99999999999999]

]

}

}

This is a GeoJSON Feature of type "LineString" (Section [7.3.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#single-part-geometries)) representing a Great Circle line.

* How many [lon, lat] coordinate pairs is the greatCircle line composed of?
* Type the appropriate expression in the console to find out[107](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fn107).

When the above code section is executed in a web page with a Leaflet map object named map, the following expression can be used to draw the Great Circle line we just calculated on the map. We are using L.geoJSON to go from a GeoJSON object to a Leaflet layer, then adding the layer on the map. Note that you need to zoom-in on the U.S. to see the line, since the line goes from Seattle to Washington, D.C.

L.geoJSON(greatCircle).addTo(map);

Similarly, we can draw markers at the start and end points, as follows:

L.geoJSON(start).addTo(map);

L.geoJSON(end).addTo(map);

* Type these expressions in the console, in a Leaflet web map where Turf.js was loaded and start, end and greatCircle were calculated.
* You should see the Great Circle line and the markers on the map.

This is the general principle of working with most functions in Turf.js, in a nutshell: reshaping our data with Turf.js convenience functions (such as turf.point), then passing GeoJSON objects to Turf.js geoprocessing functions (such as turf.greatCircle) and getting new GeoJSON objects in return. We are now ready for two extended examples demonstrating the workflow of using Turf.js with Leaflet:

* In the first example (Section [12.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#continuously-updated-tin)), we build a web map where a continuously updated **TIN** layer is generated, while the user can drag any of the underlying points (Figure [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-04)).
* In the second example (Section [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#clustering)), we perform spatial **clustering** to detect and display distinct groups of rare species observations (Figure [12.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-07)).

**12.4 Continuously updated TIN**

**12.4.1 Overview.**As our first extended use case of Turf.js, we are going to build a web map with a dynamic demonstration of TIN layers. The map will display a TIN layer generated from a set of randomly placed points. The user will be able to drag any of the points, observing how the TIN layer is being updated in real time (Figure [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-04)). We will go through four steps to accomplish this task:

* In example-12-01.html, we are to going to generate **random points** and add them on a Leaflet map (Section [12.4.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#generating-random-points)).
* In example-12-02.html, we will generate a **TIN** layer on top of the points (Section [12.4.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#adding-a-tin-layer)).
* In example-12-03.html, we will learn how to make a Leaflet marker **draggable** (Section [12.4.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#draggable-circle-markers)).
* In example-12-04.html, we will make all of our random points draggable and binded to an event listener that **updates** the TIN layer in response to any of the points being dragged (Section [12.4.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#continuous-updating)).

**12.4.2 Generating random points.**The turf.randomPoint [function](http://turfjs.org/docs/#randomPoint) from the Turf.js library can be used to generate GeoJSON with randomly placed points. When using the turf.randomPoint function, we need to specify:

* The **number** of points to generate
* A **bounding box**, defined using an array of the form [lon0, lat0, lon1, lat1]

Given these two arguments, the turf.randomPoint function randomly places the specified number of points within the given bounding box, and returns the resulting point layer as GeoJSON. For example, in the following code section, the *first* expression defines a bounding box array named bounds, using the [lon0, lat0, lon1, lat1] structure that turf.randomPoint expects. In this case, we are using the coordinates of the bounding box of Israel (Figure [12.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-01)). The *second* expression then generates 20 random points placed within the bounding box, and returns a GeoJSON object which we assign to a variable named points. Note that the bounding box array needs to be passed as the bbox property inside the options object in turf.randomPoint. If we omit the bbox option, the random points will be generated in the default global extent, i.e., [-180, -90, 180, 90]. The *third* expression transforms the returned GeoJSON points to a Leaflet layer, then adds the layer on the map:

**let** bounds = [34.26801, 29.49708, 35.90094, 33.36403];

**let** points = turf.randomPoint(20, {bbox: bounds});

L.geoJSON(points).addTo(map);

To view the points, it is convenient to set the initial map extent to the same bounding box where the points were generated. In the following expression, we initialize the Leaflet map using a view-setting method called .fitBounds. The .fitBounds method automatically detects the appropriate map center and zoom level so that the given extent fits the map bounds. The .fitBounds method is sometimes more convenient than .setView, which was introduced in Section [7.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#adding-geojson-to-leaflet-map), where the extent is set based on map center and zoom level.

Confusingly, the .fitBounds method from Leaflet and the turf.randomPoint function from Turf.js require two different forms of bounding box specifications. Specifically, like all other Leaflet functions, .fitBounds uses the [lat, lon] ordering instead of [lon, lat]. Also, .fitBounds needs the two bounding box “corners” to be separated in two internal arrays. Therefore:

* **Leaflet** needs a [[lat0, lon0], [lat1, lon1]] bounding box
* **Turf.js** needs a [lon0, lat0, lon1, lat1] bounding box

This is the reason that the bounds array is “rearranged” in the following call to .fitBounds, when initializing our Leaflet map to the same extent where the random points are[108](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fn108):

**let** map = L.map("map")

.fitBounds([[bounds[1], bounds[0]], [bounds[3], bounds[2]]]);

L.tileLayer(

"https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",

{attribution: '&copy; <a href="https://www.openstreetmap.org/[...]</a>'}

).addTo(map);

Including the latter five expressions in our script displays 20 randomly placed point markers (example-12-01.html), as shown in Figure [12.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-01).

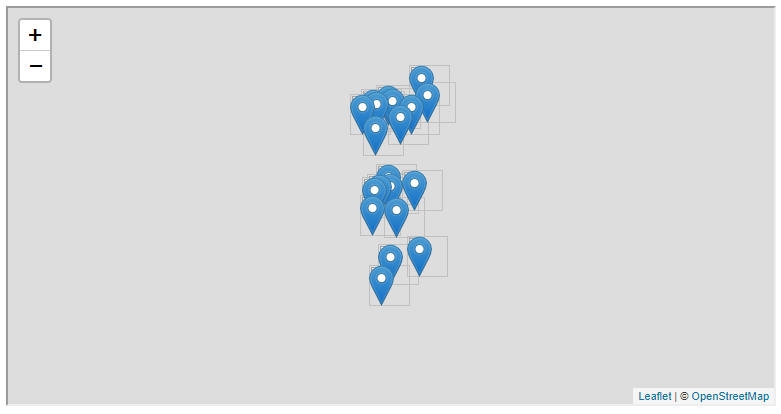


FIGURE 12.1: example-12-01.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-12-01.html)

* Open example-12-01.html in the browser.
* Refresh the page several times—you should see the 20 points being randomly placed in different locations each time.

**12.4.3 Adding a TIN layer.**A [**Triangulated Irregular Network** (**TIN**)](https://en.wikipedia.org/wiki/Triangulated_irregular_network) is a geometrical method of connecting a set of points in such a way that the entire surface is covered with triangles. TIN is frequently used in 3D modeling, as this method can be used to construct a continuous surface out of a set of 3D points. In Turf.js, the turf.tin [function](http://turfjs.org/docs/#tin) can be used to calculate a TIN layer given a set of points.

Like with the turf.greatCircle function shown previously (Section [12.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#great-circle-line)), both input and output of turf.tin are GeoJSON objects. In the case of turf.tin, the input should be a point layer, and the output is a polygonal layer with the resulting TIN.

To draw a TIN layer based on the GeoJSON object named points, we can add the following expressions at the bottom of the <script> in example-12-01.html. You can also run the expressions in the console of example-12-01.html to see the TIN layer being added in “real-time”:

**let** tin = turf.tin(points);

L.geoJSON(tin).addTo(map);

As shown in Section [12.4.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#generating-random-points), points is a GeoJSON object representing a set of 20 random points. In the first expression, we calculate the TIN layer GeoJSON with turf.tin(points) and assign it to a variable named tin. In the second expression, tin is added on the map with L.geoJSON, the same way we added the points.

The resulting map example-12-02.html, now with the calculated TIN, is shown in Figure [12.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-02).

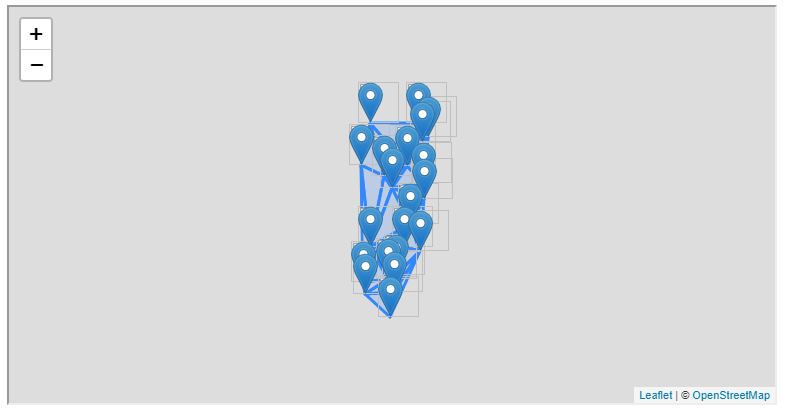


FIGURE 12.2: example-12-02.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-12-02.html)

* Open example-12-02.html in the browser and refresh the page several times.
* You should see a different TIN layer each time, according to the random placement of the markers.

**12.4.4 Draggable circle markers.**So far, in example-12-02.html (Figure [12.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-02)), we created a web map with randomly generated points and their associated TIN layer. Refreshing the web page in this example gives a nice demonstration of the TIN algorithm—each time the page is refreshed, the points are randomly rearranged and the TIN layer is updated accordingly. An even nicer demonstration, though, would be if we could *drag* any of the points wherever we want, and watch the TIN layer being updated in *real-time*. Perhaps you are already familiar with this type of behavior from [**Google Maps**](https://www.google.com/maps/), or other web applications for routing, where the user can drag the marker denoting an origin or a destination, and the calculated route is continuously updated in response (Figure [12.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:google-maps-directions-drag-marker)).

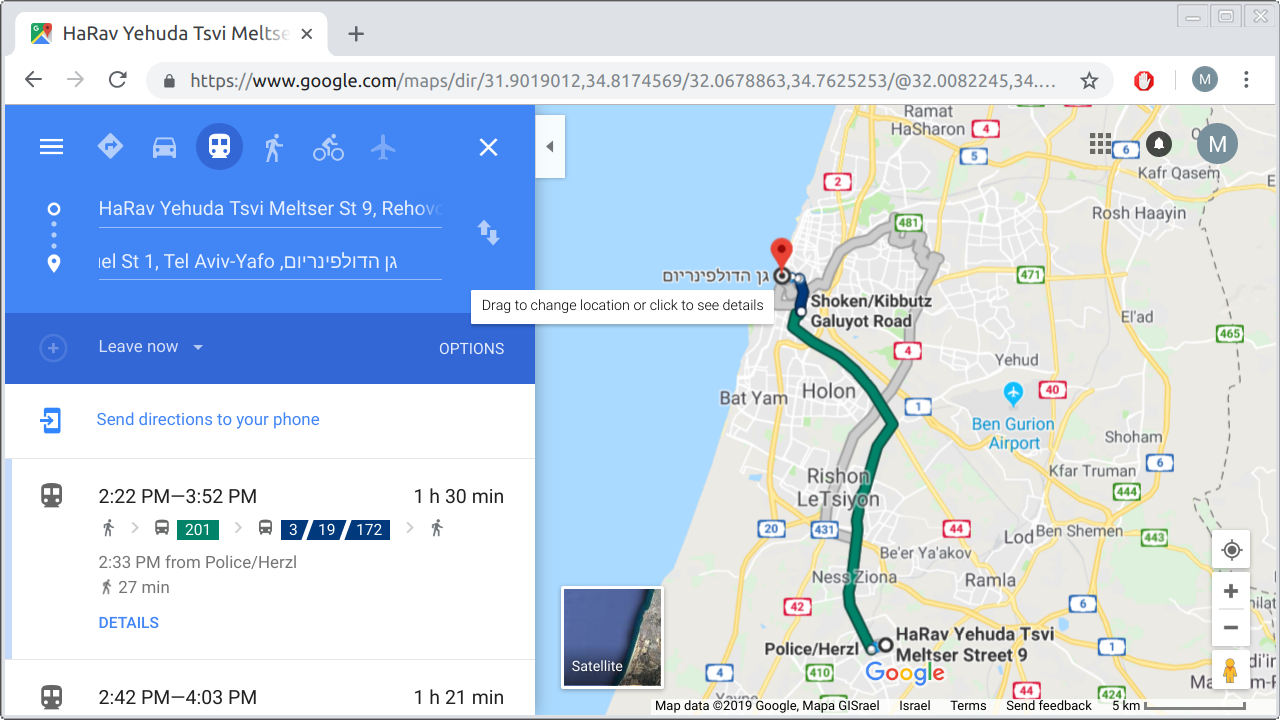


FIGURE 12.3: Draggable marker in the Google Maps Directions web application

The first thing missing to make our continuously-updated TIN is to make the 20 random markers actually *draggable*. This is exactly what we learn how to do in this section. Going back to the basic map example-06-02.html from Section [6.5.11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-a-tile-layer), adding the following code displays a draggable marker on the map. Note that the marker is placed in a manually defined location [31.262218, 34.801472] (of the form [lat, lon]), and has a popup message. Making the marker draggable requires setting its styling option draggable to true when creating it with L.marker:

**let** marker =

L.marker([31.262218, 34.801472], {draggable: **true**})

.bindPopup("This marker is draggable! Move it around...")

.addTo(map)

.openPopup();

On its own, the draggable marker is not very useful, because the TIN layer will not follow it once dragged. What we would like to do is make the TIN layer respond to every change in marker locations, by continuously updating itself. To do that, we can add an event listener of type "drag" on each and every one of our 20 random markers. The event listener function we define will be executed each time any of the markers is moved around.

For now, we have just one draggable marker. To experiment with the "drag" event listener we can use a function that simply prints the current marker position. The latter can be accessed through a property named .\_latlng that the marker has. The .\_latlng property is an object with properties .lat and .lng, similar to the .latlng event object property (Sections [6.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#introducing-map-events) and [11.2.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#getting-click-coordinates)):

marker.addEventListener("drag", **function**() {

console.log(marker.\_latlng);

});

The resulting map example-12-03.html, displaying one draggable marker, is shown in Figure [12.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-03).

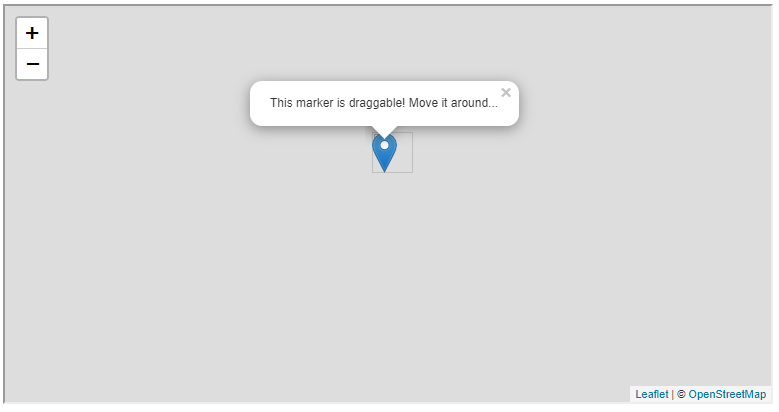


FIGURE 12.4: example-12-03.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-12-03.html)

* Open example-12-03.html and try dragging the marker around.
* Open the console to see the current location being printed each time the marker is dragged.

**12.4.5 Continuous updating.**Now, let’s combine the TIN example (Figure [12.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-02)) with the draggable marker event listener example (Figure [12.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-03)) to create a continuously-updated TIN. Basically, we are going to set {draggable: true} for all of the 20 randomly generated markers, then make the TIN layer *update* in response to any of those markers being dragged. The final result example-12-04 is shown in Figure [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-04).

To achieve this result, we will use a slightly modified approach for adding the points as well as the TIN layer. Instead of simply adding the points and the tin GeoJSON objects on the map right away, like we did in example-12-02.html:

**let** points = turf.randomPoint(20, {bbox: bounds});

**let** tin = turf.tin(points);

L.geoJSON(points).addTo(map);

L.geoJSON(tin).addTo(map);

we will set up two empty layer groups for the points and TIN layers, named pnt\_layer and tin\_layer, respectively. These layer groups will be referred to later on in the code. We are using L.layerGroup (Section [7.6.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#using-layer-groups)) to create the layer groups:

**let** pnt\_layer = L.layerGroup().addTo(map);

**let** tin\_layer = L.layerGroup().addTo(map);

Next, we calculate the 20 random points:

**let** points = turf.randomPoint(20, {bbox: bounds});

This time, we do not want to convert all points to a GeoJSON Leaflet layer with the default settings of L.geoJSON. Instead, we want use L.geoJSON options to set the following:

* Make each of the markers draggable
* Add a "drag" event listener to each marker, triggering the update of the TIN layer whenever the marker is dragged

This is why instead of using L.geoJSON with the default options, like we did in example-12-02.html:

L.geoJSON(points).addTo(map);

we are now setting both the onEachFeature and pointToLayer options of L.geoJSON:

L.geoJSON(points, {

onEachFeature: **function**(feature, layer) {

layer.addEventListener("drag", drawTIN);

},

pointToLayer: **function**(geoJsonPoint, latlng) {

**return** L.marker(latlng, {draggable: **true**});

}

}).addTo(pnt\_layer);

We are already well familiar with using the onEachFeature option for adding event listeners to each GeoJSON feature. For example, in Chapter [8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#symbology-and-interactivity) we used this option to add "mouseover" and "mouseout" event listeners to highlight the town polygons on mouse hover (Section [8.7.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#styling-in-response-to-events)). Here, we are adding a "drag" event listener. The function named drawTIN, which will be executed on marker drag, is yet to be defined.

The other option, pointToLayer, was only briefly mentioned in Chapter [8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#symbology-and-interactivity) (Section [8.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#l-geojson-options)). The pointToLayer option is used to specify the way that GeoJSON points are to be translated to Leaflet layers, in case we want to override the default L.marker. In this case, we just want to draw a marker with the {draggable: true} option, rather than the default marker.

Moving on, we now need to define the drawTIN function, which is being passed to the event listener. Each time a marker is *dragged*, the drawTIN function will be executed to calculate the new GeoJSON of the TIN layer, and replace the old TIN with the new one. Importantly, the new TIN layer is calculated based on the up-to-date pnt\_layer to keep the points and the TIN *in sync*. Here is the drawTIN function definition:

**function** drawTIN() {

tin\_layer.clearLayers();

points = pnt\_layer.toGeoJSON();

tin = turf.tin(points);

tin = L.geoJSON(tin);

tin.addTo(tin\_layer);

}

Essentially, each time a marker is dragged, the drawTIN function does the following things:

* **Clears** the tin\_layer layer group, to remove the old TIN layer
* **Calculates** the new TIN layer based on the current points in pnt\_layer
* **Adds** the new TIN layer to the tin\_layer layer group, thus drawing it on the map

These three things are accomplished in the 1st, 3rd, and 5th expressions in the drawTIN code body, respectively. What are the 2nd and 4th expressions for, then? The latter are necessary because of the above-mentioned fact (Section [12.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#great-circle-line)) that Turf.js functions only accept GeoJSON, and thus cannot directly operate on Leaflet layers. Therefore, we need to translate the Leaflet marker layer to GeoJSON in the 2nd expression, then translate the TIN GeoJSON back to a Leaflet layer in the 4th expression. The following code section repeats the internal code of drawTIN, this time with comments specifying the purpose of each expression:

tin\_layer.clearLayers(); *// (1) Clear old TIN*

points = pnt\_layer.toGeoJSON(); *// (2) Layer -> GeoJSON*

tin = turf.tin(points); *// (3) Calculate new TIN*

tin = L.geoJSON(tin); *// (4) GeoJSON -> Layer*

tin.addTo(tin\_layer); *// (5) Display new TIN*

Finally, we need to execute the drawTIN function one time outside of the event listener. That way, the *initial* TIN layer will be displayed on page load[109](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fn109), even if the user has not dragged any of the markers yet. Without this initial function call, the TIN layer will only appear after any of the markers is dragged, but not on initial page load:

drawTIN();

The resulting map example-12-04.html is shown in Figure [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-04). The screenshot shows how one of the points was dragged to the left, i.e., towards the west, and the TIN layer was extended accordingly.

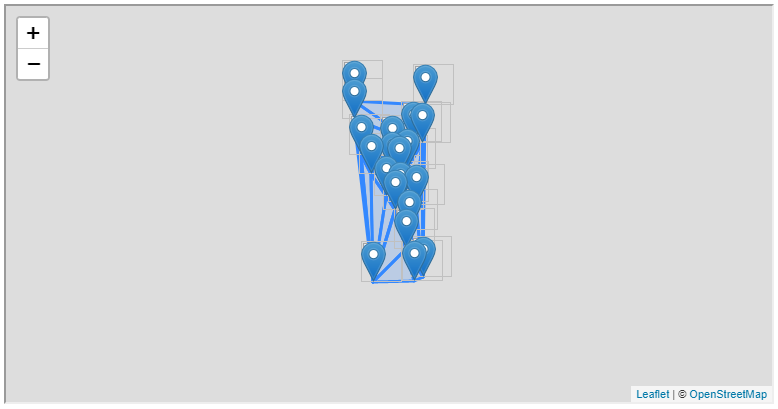


FIGURE 12.5: example-12-04.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-12-04.html)

**12.5 Clustering**

**12.5.1 Overview.**[**Clustering**](https://en.wikipedia.org/wiki/Cluster_analysis) is the process of classifying a set of observations into groups, so that objects within the same group (called a cluster) are more similar to each other than to objects in other groups. With spatial clustering, similarity usually means geographical proximity, so that the aim of clustering is to group mutually proximate observations. In our second example with Turf.js, the final goal is to create a web map where clusters of nearby plant observations per species (i.e., populations) are detected using a clustering method called DBSCAN (Section [12.5.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#dbscan-clustering)) and displayed on a web map (Figure [12.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-07)). We will approach this task in three steps:

* In example-12-05.html, we will learn to automatically apply the same function on **subsets** of GeoJSON features sharing the same value of a given property, such as observations of the same species (Section [12.5.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#processing-sets-of-features)).
* In example-12-06.html, we will learn to emphasize the spatial extent that a group of points occupies, by drawing **Convex Hull** polygons around the group (Section [12.5.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#adding-a-convex-hull)).
* In example-12-07.html, we will apply the **DBSCAN** clustering algorithm to detect spatial clusters in the observation points per species, and draw a Convex Hull polygon around each cluster (Section [12.5.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#dbscan-clustering)).

**12.5.2 Processing sets of features.**In the first step of our clustering example, we are going to load observations of four *Iris* species from the plants table on CARTO, which we are familiar with from Chapters [9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\databases.html#databases)–[11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#spatial-queries-1). We are not doing any clustering just yet. The purpose of this example is to get familiar with the turf.clusterEach [function](http://turfjs.org/docs/#clusterEach). The turf.clusterEach function is used to apply a function on each group of GeoJSON features, whereas a group is defined through common values in one of the GeoJSON properties.

The script starts with constructing the URL for the SQL API, then defining a color selection function iris\_color and styling function iris\_style (Section [8.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#varying-style)), for loading and setting the color of four *Iris* species observations, respectively:

*// Set base URL*

**let** url = "https://michaeldorman.carto.com/api/v2/sql?format=GeoJSON&q=";

*// Set SQL Query*

**let** sqlQuery = "SELECT name\_lat, the\_geom " +

"FROM plants WHERE " +

"name\_lat='Iris atrofusca' OR " +

"name\_lat='Iris atropurpurea' OR " +

"name\_lat='Iris mariae' OR " +

"name\_lat='Iris petrana'";

*// Color function*

**function** iris\_color(species) {

**if**(species === "Iris mariae") **return** "yellow";

**if**(species === "Iris petrana") **return** "brown";

**if**(species === "Iris atrofusca") **return** "black";

**if**(species === "Iris atropurpurea") **return** "orange";

}

*// Style function*

**function** iris\_style(feature) {

**return** {

fillColor: iris\_color(feature.properties.name\_lat),

weight: 1,

opacity: 1,

color: "black",

fillOpacity: 0.5

};

}

What comes next is the important part of the script, where our GeoJSON is loaded and added on the map:

fetch(url + sqlQuery)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

turf.clusterEach(

data,

"name\_lat",

**function**(cluster, clusterValue, currentIndex) {

L.geoJSON(cluster, {

onEachFeature: **function**(feature, layer) {

layer.bindPopup("<i>" + clusterValue + "</i>");

},

pointToLayer: **function**(geoJsonPoint, latlng) {

**return** L.circleMarker(latlng);

},

style: iris\_style

}).addTo(map);

}

);

});

This is quite a long and complicated expression, which we will now explain in detail. First, note that the function passed to the second .then is not using L.geoJSON right away, like we did before. Instead, it contains an internal function call of the turf.clusterEach function, of the following form:

turf.clusterEach(

data,

"name\_lat",

**function**(cluster, clusterValue, currentIndex) {

...

}

);

The turf.clusterEach function is a convenience function from Turf.js. It is used for iterating over groups of GeoJSON features sharing the same property values. The turf.clusterEach function accepts three arguments:

* The **GeoJSON** to iterate on, in our case it is the data object passed from fetch, i.e., the GeoJSON with the observations of four *Iris* species
* The **property** name used for grouping, in our case it is "name\_lat", i.e., the Latin species name
* A **function** to be applied on each set of features, with function parameters being: the current set of features (cluster), the current property value (clusterValue) and the current cluster index (currentIndex)

In our case, the internal function passed to turf.clusterEach creates and draws a GeoJSON layer, using L.geoJSON. The function code uses the cluster and clusterValue parameters, which refer to the current set of GeoJSON *features* sharing the same value in the name\_lat property, and the current *value* of the name\_lat property, respectively:

fetch(url + sqlQuery)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

turf.clusterEach(

data,

"name\_lat",

**function**(cluster, clusterValue, currentIndex) {

L.geoJSON(cluster, {

... *// What to do with each set of features*

}).addTo(map);

}

);

});

Eventually, the code loads all observations of four *Iris* species (see SQL query above), then iterates over the four groups sharing the same species name. The iteration sequentially adds the observations of each species on the map. For example, note that when adding popups we are referring to the clusterValue, which refers to the name\_lat of the current group of features in each step of the iteration:

layer.bindPopup("<i>" + clusterValue + "</i>");

You may wonder why do we need to split the GeoJSON with turf.clusterEach in the first place, rather than just add the entire GeoJSON with all four species at once, like we used to do in previous Chapters:

fetch(url + sqlQuery)

.then(**function**(response) {

**return** response.json();

})

.then(**function**(data) {

L.geoJSON(data, {

... *// What to do with the entire GeoJSON*

}).addTo(map);

});

In this example, the latter approach is indeed preferable since it is shorter and simpler, while giving the same result. For example, it does not matter whether we apply the styling function iris\_style, which sets the marker color, on GeoJSON subsets or on the entire object at once: the styling of a given species observation depends on its name\_lat value either way (see Chapter [8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#symbology-and-interactivity)). However, the turf.clusterEach approach is just a preparation for what we do in the next two sections, where it is indeed required to treat each species separately, since we will be drawing a **Convex Hull** polygon around each species (Figure [12.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-06)) or detecting populations within each species using the **DBSCAN** clustering algorithm (Figure [12.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-07)).

Additionally, note that when loading the GeoJSON, we are using the pointToLayer option. As shown in Section [12.4.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#continuous-updating), the pointToLayer option controls the way that GeoJSON points are translated to Leaflet layers. Recall that in the previous example (example-12-04.html), we used pointToLayer to make our markers draggable (Figure [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-04)). In the present example, the purpose of using pointToLayer is to override the default marker display, using *circle markers* (Sections [6.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-markers) and [8.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#exercise-08)) instead of ordinary markers. The advantage of circle markers is that they can be colored according the clusterValue, which refers to the name\_lat property of the current group, so that observations of the different species are distinguished by their color:

...

pointToLayer: **function**(geoJsonPoint, latlng) {

**return** L.circleMarker(latlng);

},

...

The resulting map example-12-05.html shows the observations of four *Iris* species. Each species is displayed with differently colored circle markers (Figure [12.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-05)).

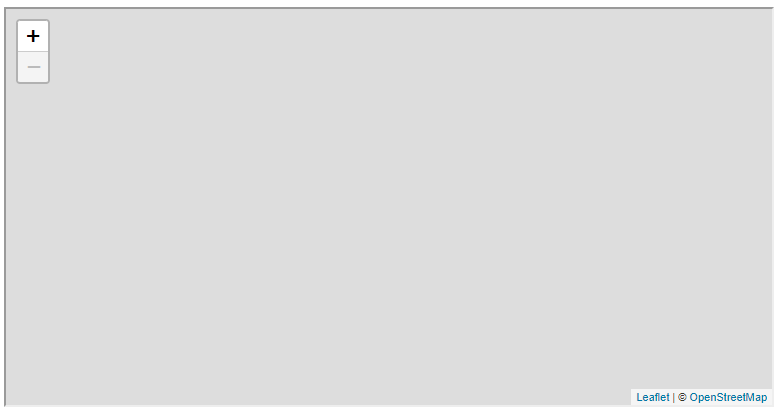


FIGURE 12.6: example-12-05.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-12-05.html)

**12.5.3 Adding a Convex Hull.**A [**Convex Hull**](https://en.wikipedia.org/wiki/Convex_hull) is the smallest convex geometrical shape, typically a polygon, enclosing a given set of points. One of the uses of a Convex Hull in data visualization, in general, and in geographical mapping, in particular, is to highlight the geographical extent that a group of observations occupies, usually to distinguish between extents occupied by several groups. In other words, Convex Hull polygons are a useful and convenient way of highlighting the division of individual observations into *groups*.

In our second step of the clustering example (this section), we will experiment with drawing Convex Hull polygons to highlight the extent that each *Iris* species occupies in space (Figure [12.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-06)). In the third and final step (Section [12.5.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#dbscan-clustering), below), we are going to delineate internal clusters *within* each species, and draw Convex Hull polygons around each cluster within each species (Figure [12.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-07)).

To draw a Convex Hull polygon around all observations of each species, all we need to do is add the following code within the turf.clusterEach iteration from example-12-05.html:

**let** ch = turf.convex(cluster);

ch.properties.name\_lat = clusterValue;

L.geoJSON(ch, {style: iris\_style}).addTo(map);

In the first expression we are taking the current set of points (cluster) and calculating their Convex Hull polygon GeoJSON using the turf.convex [function](http://turfjs.org/docs/#convex). The second expression sets the name\_lat property of the Convex Hull according to the currently processed clusterValue. This step is necessary because turf.convex does not “carry over” the point properties from cluster to the resulting polygon ch. The name\_lat property is needed, though, for setting the Convex Hull polygon color with iris\_style. That way, the convex hull polygons are colored the same way as the underlying circle markers (Figure [12.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-06)). Finally, in the third expression, we add the polygon on the map using L.geoJSON and .addTo.

Remember that cluster is consecutively assigned with a different set of features for each species, as part of the turf.clusterEach iteration. Since the code for calculating the Convex Hull and drawing it on the map is placed *inside* the iteration, a separate Convex Hull polygon is calculated per species, highlighting the separation in species distribution ranges (Figure [12.7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-06)).

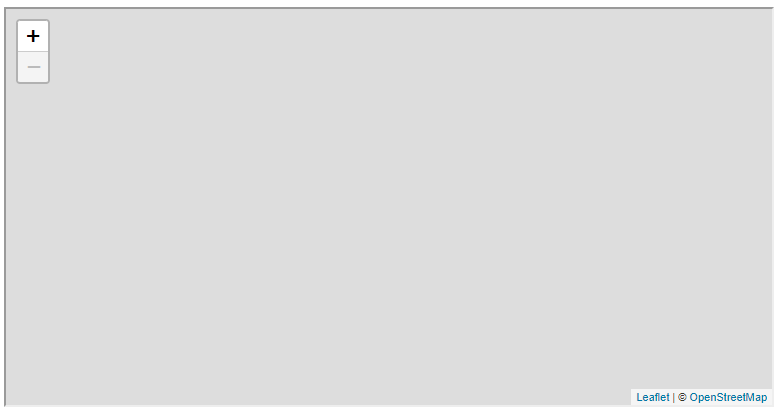


FIGURE 12.7: example-12-06.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-12-06.html)

**12.5.4 DBSCAN clustering.**In spatial clustering, we aim to detect groups of nearby observations, which are close to each other but further away from other observations. In the previous *Iris* example, we may wish to detect spatial clusters representing separate populations *within* each species. To do that, we will experiment with a clustering technique called **Density-Based Spatial Clustering of Applications with Noise** or [**DBSCAN**](https://en.wikipedia.org/wiki/DBSCAN).

DBSCAN is a density-based clustering method, which means that points that are closely packed together are assigned into the same cluster and given the same *ID*. The DBSCAN algorithm has two parameters, which the user needs to specify:

* \(\varepsilon\)—The maximal **distance** between points to be considered within the same cluster
* \(minPts\)—The **minimal number** of points required to form a cluster

In short, all groups of at least \(minPts\) points, where each point is within \(\varepsilon\) or less from at least one other point in the group, are considered to be separate clusters and assigned with unique IDs. All other points are considered “noise” and are not assigned with an ID.

The turf.clustersDbscan [function](http://turfjs.org/docs/#clustersDbscan) is an implementation of the DBSCAN algorithm in Turf.js. The function accepts an input point layer and returns the same layer with a new property named cluster. The cluster property contains the cluster ID assigned using DBSCAN. Noise points are not assigned with an ID and thus are not given a cluster property. The first parameter of the turf.clustersDbscan function is maxDistance (\(\varepsilon\)), which determines what is the maximal distance between two points to be considered within the same cluster, in kilometers. In our example (see below), we will use a maximal distance of 10 kilometers. The minPoints (\(minPts\)) parameter has a default value of 3, which we will not override.

To experiment with turf.clustersDbscan, you can execute the following expression in the console of example-12-01.html (Figure [12.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-01)). This will apply DBSCAN with a maxDistance value of 50 kilometers on the 20 randomly generated points:

turf.clustersDbscan(points, 50);

* Examine the result of the above expression to see which of the 20 random points were assigned to a cluster and given a cluster property with a numeric value (i.e., the ID).
* Try increasing or decreasing the maxDistance value of 50 and executing the expression again.
* As maxDistance gets larger, all points will tend to aggregate into the same single cluster. Conversely, as maxDistance gets smaller, it will be more “difficult” for clusters to be formed, which means that more and more points will be classified as noise.

Back to our *Iris* species example. When applying the DBSCAN algorithm on the observations of a given species, the non-noise observations are going to be given a cluster property with a unique ID per cluster. We can then iterate over the clusters, to draw a Convex Hull polygon around each population. Practically, this can be accomplished by adding another, *internal*, turf.clusterEach iteration through all of the features sharing the same cluster attribute value *within* a given species.

To achieve the latter, we replace the three expressions we used for adding Convex Hull polygons *per species* in example-12-06.html (Section [12.5.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#adding-a-convex-hull)), with the following code section which adds Convex Hull polygons based on *DBSCAN clustering* per species:

clustered = turf.clustersDbscan(cluster, 10);

turf.clusterEach(

clustered,

"cluster",

**function**(cluster2, clusterValue2, currentIndex2) {

**let** ch = turf.convex(cluster2);

ch.properties.name\_lat = clusterValue;

L.geoJSON(ch, {style: iris\_style}).addTo(map);

}

);

We leave it as an exercise for the reader to go over the code, as we are essentially combining the methods already covered in the previous two examples. The final result (example-12-07.html) is shown in Figure [12.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-07). It is now evident, for instance, how *Iris petrana* (the south-eastern species shown in brown) observations are all clustered in one place, while the distribution ranges of the other three species are divided into two or three distinct populations, separated by at least 10 kilometers from each other (Figure [12.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-07)).

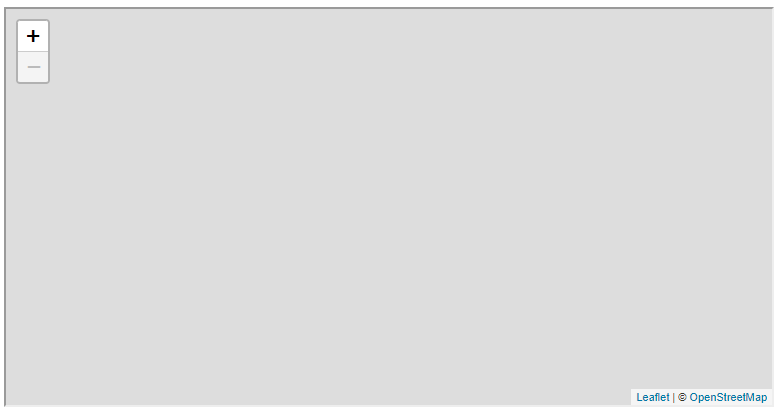


FIGURE 12.8: example-12-07.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-12-07.html)

* Expand example-12-07.html, adding popups for the Convex Hull polygons to display the species name and the cluster ID when clicked.
* Modify the maximal distance parameter in turf.clustersDbscan (currently set to 10) and check out how the clustering result changes. What is the reason for the console error when maxDistance is very small? How can we address the problem?

**12.6 Heatmaps with Leaflet.heat.**Turf.js is one of the most comprehensive client-side geoprocessing JavaScript libraries, but it is not the only one. The [**Leaflet.heat**](https://github.com/Leaflet/Leaflet.heat) library, for example, is a geoprocessing plugin for the Leaflet library. The term “plugin,” in this context, means that Leaflet.heat is an extension of the Leaflet library and thus cannot be used without Leaflet, unlike Turf.js, which is an independent library that can be used with Leaflet or with any other library, or even on its own. Also, unlike Turf.js, which is a general-purpose geoprocessing library, Leaflet.heat does just one thing—drawing a heatmap to display the density of point data. The Leaflet.heat plugin is one of many created to extend the core functionality of Leaflet[110](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fn110).

Drawing individual points can be visually overwhelming, slow, and hard to comprehend when there are too many of them. **Heatmaps** are a convenient alternative way of conveying the distribution of large amounts of point data. A heatmap is basically a way to summarize and display point density, instead of showing each and every individual point. Technically speaking, a heatmap uses two-dimensional [**Kernel Density Estimation (KDE)**](https://en.wikipedia.org/wiki/Kernel_density_estimation) to calculate a smooth surface representing point density.

The following example (example-12-08.html) draws a heatmap of all rare plant observations from the plants table on CARTO. Overall, there are 23,827 points in the plants table (Section [10.4.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#how-many-species-do-we-have)). Drawing markers, or even the simpler circle markers, for such a large amount of points is usually not a good idea. First, it may cause the browser to become unresponsive due to computational intensity. Second, it is usually difficult to pick the right styling (size, opacity, etc.) for each point so that dense locations are clearly distinguished on the various map zoom levels the user may choose. This is where the density-based heatmaps become very useful. Particularly, the Leaflet.heat library automatically re-calculates the heatmap for each zoom level, allowing the user to conveniently explore point density on both large and small scales.

To use the Leaflet.heat library we first need to load it. Again, we will use a local copy:

**<script** src="js/leaflet.heat.js"**></script>**

which can be downloaded, or directly referenced, using the following URL:

[https://leaflet.github.io/Leaflet.heat/dist/leaflet-heat.js](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\leaflet.github.io\Leaflet.heat\dist\leaflet-heat.js)

Next, when loading the plant observations GeoJSON we need to convert it to an array of [lat, lon, intensity] arrays, which is the data structure that the Leaflet.heat library expects. The intensity value can be used to give different *weights* to each point:

[

[lat, lon, intensity],

[lat, lon, intensity],

[lat, lon, intensity],

...

]

There are many possible ways to go from a GeoJSON of type "FeatureCollection", composed of "Point" features, to the array structure shown above. The code section below shows one way to do it, using the .forEach array iteration method (Section [3.10.3.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#the-foreach-array-method)). The .forEach iteration goes over the features of a GeoJSON object named data, such as the result of a CARTO SQL API request obtained with fetch. For each of the features, the latitude and longitude are extracted into an array of the form [lat, lon], along with the constant intensity value of 0.5 which gives all points equal weights when calculating density. When the iteration ends, we have an array named locations, containing 23,827 triplets of the form [lat, lon, intensity].

**let** locations = [];

data.features.forEach(**function**(element) {

**let** coords = element.geometry.coordinates;

**let** location = [coords[1], coords[0], 0.5];

locations.push(location);

});

Once the locations array is ready, we can run the L.heatLayer function on the array to create the heatmap layer, then add it on our map. The additional parameters define the radius of each point on the map (i.e., the smoothing kernel width) and the minimal heatmap opacity:

L.heatLayer(locations, {radius: 20, minOpacity: 0.5}).addTo(map);

The resulting map example-12-08.html, with the heatmap, is shown in Figure [12.9](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-08).

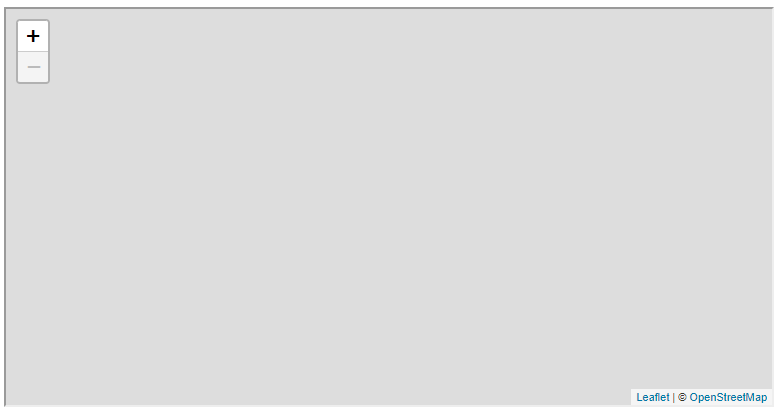


FIGURE 12.9: example-12-08.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\example-12-08.html)

**12.7 Exercise**

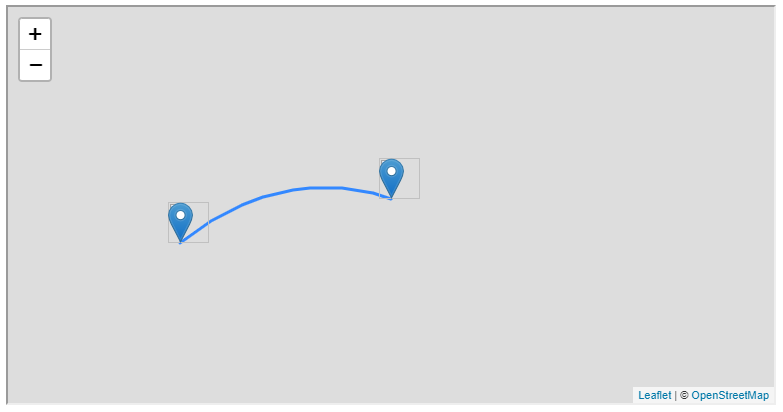
* Create a web map with two markers and a Great Circle line between them, where the Great Circle line is continuously updated whenever the markers are dragged (Figure [12.10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:solution-12)).
* Use example-12-04.html (Figure [12.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fig:example-12-04)) for guidance on how to make the Great Circle line continuously updated.
* 

FIGURE 12.10: solution-12.html [(Click to view this example on its own)](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\examples\solution-12.html)

1. The line coordinates in the greatCircle object, except for the first three and last one, were replaced with ... to save space.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fnref106)
2. The answer is 100.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fnref107)
3. Note that part of the attribution was replaced with [...] to save space.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fnref108)
4. We used the same technique to display the initially selected species on the map in example example-10-05.html (Section [10.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\non-spatial-queries-1.html#rearranging-the-code)).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fnref109)
5. In Chapter [13](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\collaborative-mapping.html#collaborative-mapping), we are going to work with another Leaflet plugin, called **Leaflet.draw**. See the *Leaflet Plugins* section in the Leaflet documentation (<https://leafletjs.com/plugins.html>) for the complete list of Leaflet plugins.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\client-side-geoprocessing.html#fnref110)