OpenStreetMap.jl Documentation

Release 0.3.0

Ted Steiner

July 14, 2016

Contents

OpenStreetMap.jl is a Julia package that provides basic functionality for parsing, viewing, and working with OpenStreetMap map data. The package is intended mainly for researchers who want to incorporate this rich, global data into their work, and has been designed with both speed and simplicity in mind, especially for those who might be new to Julia.

CHAPTER 1

Contents

1.1 Overview

This package is provided to give researchers quick and convenient access to OpenStreetMap data in Julia. It provides means for extracting and classifying map data, basic route planning, and convenient data visualization.

I found comparable tools for Matlab to be painfully slow, and therefore decided to write a new set of functions from scratch in Julia. Julia provides an excellent platform for quickly and easily working with very large datasets. With the exception of the plotting tools, the functions in this Julia package run significantly faster than comparable tools available in Matlab.

1.1.1 Features

The following features are provided:

- Parse an OpenStreetMap XML datafile (OSM files)
- Crop maps to specified boundaries
- Convert maps between LLA, ECEF, and ENU coordinates
- Extract highways, buildings, and tagged features from OSM data
- Filter data by various classes
  - Ways suitable for driving, walking, or cycling
  - Freeways, major city streets, residential streets, etc.
  - Accomodations, shops, industry, etc.
- Draw detailed maps using Julia's Winston graphics package with a variety of options
- Compute shortest or fastest driving, cycling, and walking routes using Julia's Graphs package

1.1.2 Package Status

All the functionality that I personally need for my work is now implemented in this package. Therefore, future updates will depend on GitHub issues (bug reports or feature requests) created by users. Pull requests for additional functionality are very welcome, as well.

1.2 Data Types

This page gives an overview of the main data types used by OpenStreetMap.jl.

1.2.1 Map Data

These types pertain directly to map elements.

Highway

All roads and paths in OpenStreetMap are generically called "highways." These types must include a list of nodes that comprises the path of the highway. All other fields are optional, and are empty strings when missing from the OSM database.

When a highway is labeled as "oneway," the road or path is only legally traversable in the order in which the nodes are listed.

```
type Highway
    class::String       # Type of highway
    lanes::Int          # Number of lanes (1 if unspecified)
    oneway::Bool        # True if road is one-way
    sidewalk::String    # Sidewalk classifier, if available
    cycleway::String    # Cycleway classifier, if available
    bicycle::String     # Bicycle classifier, if available
    name::String        # Name, if available
    nodes::Vector{Int}  # List of nodes
end
```

"Segments" represent a subset of a highway, and can be used for faster route planning. They begin and end at highway intersections (see below). Segments can be extracted from a list of roads and intersections using "extractSegments()."

```
type Segment
    node0::Int          # Source node ID
    node1::Int          # Target node ID
    nodes::Vector{Int}  # List of nodes falling within node0 and node1
    class::Int          # Class of the segment
    parent::Int         # ID of parent highway
    oneway::Bool        # True if road is one-way
end
```

Feature

"Features" are nodes tagged with additional data. OpenStreetMap.jl currently ignores some of these tags (e.g., crosswalks), but

- amentity
- shop
- building
- craft
- historic
- sport

- tourism

Many of these features also have a specified name and class detail (e.g., shop:restaurant). Nodes with no tags are never made into features.

```
type Feature
    class::String       # Shop, amenity, crossing, etc.
    detail::String      # Class qualifier
    name::String        # Name
end
```

Building

Buildings in OpenStreetMap may optionally have a name and class (though typically buildings are unlabeled). Like highways, they include a list of nodes.

```
type Building
    class::String       # Building type (usually "yes")
    name::String        # Building name (usually unavailable)
    nodes::Vector{Int}  # List of nodes
end
```

Intersection

OpenStreetMap.jl includes an intersection detector. An intersection is a node which is included in at least two highways' lists of nodes. The intersection object maintains a `Set` (no duplicates allowed) of highway ids that use that node.

```
type Intersection
    highways::Set{Int} # Set of highway IDs
end
```

Region Boundaries

Region boundaries include the minimum and maximum latitude and longitude of a region. While Bounds targets the LLA coordinate system, Bounds{ENU} can be used with ENU coordinates. Bounds will not work well with ECEF coordinates.

```
type Bounds
    min_y::Float64    # min_lat or min_north
    max_y::Float64    # max_lat or max_north
    min_x::Float64    # min_lon or min_east
    max_x::Float64    # max_lon or max_east
end
```

1.2.2 Point Types

These types give alternative representations for point locations in OpenStreetMap.jl.

Latitude-Longitude-Altitude (LLA) Coordinates

Used to store node data in OpenStreetMap XML files.

```
type LLA
    lat::Float64
    lon::Float64
    alt::Float64
end
```

Because OpenStreetMap typically does not store altitude data, the following alias is available for convenience: `LLA(lat, lon) = LLA(lat, lon, 0.0)`

Earth-Centered-Earth-Fixed (ECEF) Coordinates

Global cartesian coordinate system rotating with the Earth.

```
type ECEF
    x::Float64
    y::Float64
    z::Float64
end
```

East-North-Up (ENU) Coordinates

Local cartesian coordinate system, centered on a reference point.

```
type ENU
    east::Float64
    north::Float64
    up::Float64
end
```

1.2.3 Additional Types

Transportation Network

The Network type is used to represent a street transportation network as a graph. This type nicely encapsulates the graph data from the user, simplifying the use of Graphs.jl for route planning. Most users will not need to interact with the internals of these objects.

```
type Network
    g        # Incidence graph of streets
    v        # Dictionary of vertices indexed by their OSM node IDs
    w        # Edge weights
    class    # Edge classification
end
```

Plot Styles

The `Style` type is used to define custom plot elements. More information on its usage can be found on the Plots page.

```
type Style
    color::Uint32   # Line color
    width::Real     # Line width
    spec::String    # Line type
```

```
end

style(x, y) = style(x, y, "-")
```

1.3 Reading OSM Data

OpenStreetMap data is available in a variety of formats. However, the easiest and most common to work with is the OSM XML format. OpenStreetMap.jl makes reading data from these files easy and straightforward:

`getOSMData` (*filename::String*)

Inputs:

- Required:
  - `filename` [`String`]: Filename of OSM datafile.

Outputs:

- `nodes` [`false` or `Dict{Int,LLA}`]: Dictionary of node locations
- `highways` [`false` or `Dict{Int,Highway}`]: Dictionary of highways
- `buildings` [`false` or `Dict{Int,Building}`]: Dictionary of buildings
- `features` [`false` or `Dict{Int,Feature}`]: Dictionary of features

These four outputs store all data from the file. `highways`, `buildings`, and `features` are dictionaries indexed by their OSM ID number, and contain an object of their respective type at each index. "Features" actually represent tags attached to specific `nodes`, so their ID numbers are the node numbers. The `Highway` and `Building` types both contain lists of `nodes` within them.

Example Usage:

```
nodes, hwys, builds, feats = getOSMData(MAP_FILENAME)
```

1.3.1 Extracting Intersections

A simple function is provided to find all highway ends and intersections:

`findIntersections` (*highways::Dict{Int, Highway}*)

The only required input is the highway dictionary returned by "getOSMData()." A dictionary of "Intersection" types is returned, intexed by the node ID of the intersection.

In some cases, such as boulevards and other divided roads, OpenStreetMap represents a street as two one-way highways. This can result in multiple "intersections" detected per true intersection. If desired, these intersections can be "clustered," replacing these multiple intersection-lets with a single node. This gives a better estimate of the total number of highway intersections in a region.

To do this, we first "cluster" the highways, by gathering all highways with a common name (note that this is dependent on the quality of street name tags in your source data). We then search for proximal instances of these highway sets crossing one another. Flag max_dist can be used to change the required proximity of the nodes to be considered an intersection (the default is 15 meters). Note that this proximity is the maximum distance the node can be from the centroid of all nodes in the intersection at the time the node is added. If an intersection involves the same highways as an existing cluster during the search but is further away than max_dist, a new cluster will be formed, initialized at that point.

The code to accomplish this is as follows:

```
highway_sets = findHighwaySets(highways)
intersection_mapping = findIntersectionClusters(nodes,intersections,highway_sets,max_dist=15)
replaceHighwayNodes!(highways,intersection_mapping)
cluster_node_ids = unique(collect(values(intersection_mapping)))
```

The optional flag "max_dist" is in the units of your "nodes" object.

1.3.2 Working with Segments

Usually routing can be simplified to the problem of starting and ending at a specified intersection, rather than any node in a highway. In these cases, we can use "Segments" rather than "Highways" to greatly reduce the computation required to compute routes. Segments are subsets of highways that begin and end on nodes, keep track of their parent highway, and hold all intermediate nodes in storage (allowing them to be converted back to Highway types for plotting or cropping). The following functions extract Segments from Highways and convert Segments to Highways, respectively:

`segmentHighways`(*highways*, *intersections*, *classes*, *levels=Set(1:10)*)

`highwaySegments`(*segments::Vector{Segment}*)

Note: By default, segmentHighways() converts only the first 10 levels into segments. If you wish to exclude certain road classes, you should do so here prior to routing. By default, OpenStreetMap.jl uses only 8 road classes, but only classes 1-6 represent roads used for typical routing (levels 7 and 8 are service and pedestrian roads, such as parking lot entrances and driveways). In the United States, roads with class 8 should not be used by cars.

1.4 Downloading OSM Data

Downloading OpenStreetMap data in the form of a .osm file is very easy for a simple square region. However, OpenStreetMap.org provides so many options that it is sometimes a little hard to understand the simple tasks.

For a simple region, you want to use the "OpenStreetMap Overpass API." There are a few mirrors available, but I have had the best luck with the server in Denmark, hence its usage in the example below.

There are a few ways to access the API. Here are a few of them.

1.4.1 OpenStreetMap Interface

On OpenStreetMap.org, there is a big "Export" button at the top. For very small regions, this is the best option, because the region boundary will be embedded in the file for you (so you don't have to record it). Just drag the box around your region and click export. Easy!

If your region is too large, you will usually just get a blank page in your browser without any error messages. If this happens, there is a link below the "Export" button that says "Overpass API." This will very conveniently send your region to the API for an automatic download through that system. Unforunately, this .osm file will not include the boundary information, so you will not be able to use OpenStreetMap.jl's convenient `getBounds` function. Otherwise, as far as I can tell, it's the same as clicking the "Export" button.

1.4.2 Overpass API Interface

If you're not the type to like easy interfaces like dragging a box around your desired region and clicking a button, then this is the option for you! There are two ways to interact with the API. The syntax is confusing, so we will just download a simple rectangular region and do everything else happily within Julia.

The easist way to access the API is just directly through the web. The syntax is as follows:

```
http://overpass-api.de/api/map?bbox=minLon,minLat,maxLon,maxLat
```

Be sure to replace minLon, etc., with the decimal latitude and longitudes of your bounding box. This will download the file for you, but it is missing the ".osm" extension (you can add this yourself, if you'd like). You can use this to script downloads, but please don't overload the OpenStreetMap servers, which are donation-supported.

1.5 Simulating OSM Street Networks

OpenStreetMap.jl provides some basic street map simulation capabilities. These are hopefully useful for trying things out, like rouing, in a simple grid with known properties. Only highways can be simulated at this time (not features or buildings.

The basic premise is just that you make a list of north/south roads according to their classes, and another of east/west roads. You then give this to the simulator and it gives you back a list of nodes, highways, and the highway classes, all nicely organized in our OpenStreetMap.jl formats. To keep things simple, all roads are separated by 100 meters from one another.

Here is an example:

```
roads_north = [6, 6, 4, 6, 6, 3, 6, 6, 4, 6, 6]
roads_east = [6, 3, 6, 3, 6]
nodes, highways, highway_classes = simCityGrid(roads_north, roads_east)
```

1.6 Working with Data

This page gives details on the functions provided by OpenStreetMap.jl for working with OSM data.

1.6.1 Cropping Maps

OSM XML files do not provide sharp edges on boundaries. Also, it is often the case that one wants to focus on one subregion of a large OSM file. A cropping function is provided for these cases:

```
function cropMap!(nodes::Dict,
                  bounds::Bounds;
                  highways=nothing,
                  buildings=nothing,
                  features=nothing,
                  delete_nodes::Bool=true)
```

1.6.2 Classifying Map Elements

OpenStreetMap.jl can classify map elements according to the following schemes: * Roadways [8 levels] * Cycleways [4 levels] * Walkways [4 levels] * Building Types [5 levels] * Feature Types [7 levels]

Each of these schemes classifies map elements using their OSM tags according to multiple levels. The definitions of these levels is encoded in `classes.jl`.

The following functions take their respective map element lists as the single parameter and output a classification dictionary of type `Dict{Int,Int}`. The `keys` of the dictionary are the highway ID numbers, and the `values` provide the classification of that map element.

- `roadways(highways)`
- `walkways(highways)`
- `cycleways(highways)`
- `classify(buildings)`
- `classify(features)`

These classification dictionaries can be used for both route planning and map plotting.

1.6.3 Converting Map Coordinate Systems

OpenStreetMap.jl is capable of converting map data between LLA, ECEF, and ENU coordinates (see "Data Types") for definitions of these standard coordinates. Because point location data is ONLY stored in the `nodes` dictionary (type `Dict{Int,Point-Type}`), only this object needs to be converted. Note that Bounds objects also need to be converted, although they don't technically store map data. The following functions can be used to convert between coordinate systems:

- `ECEF(nodes::Dict{Int,LLA})`
- `LLA(nodes::Dict{Int,ECEF})`
- `ENU(nodes::Dict{Int,ECEF}, reference::LLA)`
- `ENU(nodes::Dict{Int,LLA}, reference::LLA)`

East-North-Up coordinates require an additional input parameter, `reference`, which gives the origin of the ENU coordinate system. LLA and ECEF coordinates both have their origins fixed at the center of the earth.

Coordinate System Selection

An effort has been made to allow users to work in the coordinate system of their choice. However, often times a specific coordinate system might not make sense for a given task, and thus functionality has not been implemented for it. Below are a few examples:

- Map cropping and plotting do not work in ECEF coordinates (these operations are fundamentally 2D operations, which is convenient only for LLA and ENU coordinates)
- Route planning does not work in LLA coordinates (spherical distances have note been implemented)

1.7 Road Network Analysis

OpenStreetMap.jl provides a user-friendly interface to the Graphs.jl package for route planning on transportation networks. Either shortest or fastest routes may be computed using Dijkstra's algorithm. In addition, driving catchment areas may be computed using Bellman Ford's algorithm.

1.7.1 Transportation Network

In order to plot routes within the map, the streets must first be converted into a transportation network using `createGraph()`:

`createGraph`(*nodes*, *highways*, *classes*, *levels*)

Inputs:

- `nodes` [`Dict{Int,ENU}` or `Dict{Int,ECEF}`]: Dictionary of node locations
- `highways` [`Dict{Int,Highway}`]: Dictionary of highways
- `classes` [`Dict{Int,Int}`]: Dictionary of highway classifications
- `levels` [`Set{Integer}`]: Set of highway classification levels allowed for route planning

Output:

- `Network` type, containing all data necessary for route planning with Graphs.jl

A transportation network graph can alternatively be created using highway "segments" rather than highways. These segments begin and end at intersections, eliminating all intermediate nodes, and can greatly speed up route planning.

`createGraph`(*segments*, *intersections*)

Inputs:

- `segments` [`Vector{Segment}`]: Vector of segments
- `intersections` [`Dict{Int,Intersection}`]: Dictionary of intersections, indexed by node ID

Output:

- `Network` type, containing all data necessary for route planning with Graphs.jl

1.7.2 Route Planning

Shortest Routes

Compute the route with the shortest total distance between two nodes.

`shortestRoute`(*network*, *node0*, *node1*)

Inputs:

- `network` [`Network`]: Transportation network
- `node0` [`Int`]: ID of start node
- `node1` [`Int`]: ID of finish node

Outputs:

- `route_nodes` [`Vector{Int}`]: Ordered list of nodes along route
- `distance` [`Float64`]: Total route distance

Fastest Routes

Given estimated typical speeds for each road type, compute the route with the shortest total traversal time between two nodes.

`fastestRoute`(*network*, *node0*, *node1*, *class_speeds=SPEED_ROADS_URBAN*)

Inputs:

- `network` [`Network`]: Transportation network
- `node0` [`Int`]: ID of start node
- `node1` [`Int`]: ID of finish node
- `class_speeds` [`Dict{Int,Real}`]: Traversal speed (km/hr) for each road class

Outputs:

- `route_nodes` [`Vector{Int}`]: Ordered list of nodes along route
- `route_time` [`Float64`]: Estimated total route time

Note 1: A few built-in speed dictionaries are available in `speeds.jl`. Highway classifications are defined in `classes.jl`.

Note 2: Routing does not account for stoplights, traffic patterns, etc. `fastestRoute` merely weights each edge by both distance and typical speed.

Route Distance

It is often of use to compute the total route distance, which is not returned by `fastestRoute()`. An additional function is available for this purpose:

`distance`(*nodes*, *route*)

Inputs:

- `nodes` [`Dict{Int,ENU}` or `Dict{Int,ECEF}`]: Dictionary of node locations
- `route` [`Vector{Int}`]: Ordered list of nodes along route

Outputs:

- `distance` [`Float64`]: Total route distance

For added convenience, `distance()` is additionally overloaded for the following inputs, all of which return a Euclidean distance:

`distance`(*nodes::Dict{Int*, *ECEF}*, *node0::Int*, *node1::Int*)

`distance`(*loc0::ECEF*, *loc1::ECEF*)

`distance`(*nodes::Dict{Int*, *ENU}*, *node0::Int*, *node1::Int*)

`distance`(*loc0::ENU*, *loc1::ENU*)

`distance`(*x0*, *y0*, *z0*, *x1*, *y1*, *z1*)

Edge Extraction

`shortestRoute()` and `fastestRoute()` both return a list of nodes, which comprises the route. `routeEdges()` can then convert this list of nodes into the list of edges, if desired:

`routeEdges`(*network::Network*, *route::Vector{Int}*)

The output is a list of edge indices with type Vector{Int}.

1.7.3 Driving Regions

In addition to providing individual routes, the following functions can also be used for retrieving the set of nodes that are within a driving distance limit from a given starting point.

Drive Distance Regions

nodesWithinDrivingDistance (*network*, *start*, *limit=Inf*)

Inputs:

- `network` [`Network`]: Transportation network
- `start` [`Int` or `Vector{Int}`]: ID(s) of start node(s)
- `limit` [`Float64`]: Driving Distance limit from start node(s)

Outputs:

- `node_indices` [`Vector{Int}`]: Unordered list of indices of nodes within the driving distance limit
- `distances` [`Float64`]: Unordered list of distances corresponding to the nodes in `node_indices`

Note 1: A few built-in speed dictionaries are available in `speeds.jl`. Highway classifications are defined in `classes.jl`.

Note 2: Routing does not account for stoplights, traffic patterns, etc. Each edge is weighted by its distance.

Drive Time Regions

nodesWithinDrivingTime (*network*, *start*, *limit=Inf*, *class_speeds=SPEED_ROADS_URBAN*)

Inputs:

- `network` [`Network`]: Transportation network
- `start` [`Int` or `Vector{Int}` or `ENU`]: ID(s) of start node(s), or any `ENU` location
- `limit` [`Float64`]: Driving time limit from start node(s)
- `class_speeds` [`Dict{Int,Real}`]: Traversal speed (km/hr) for each road class

Outputs:

- `node_indices` [`Vector{Int}`]: Unordered list of indices of nodes within the driving time limit
- `timings` [`Float64`]: Unordered list of driving timings corresponding to the nodes in `node_indices`

Note 1: A few built-in speed dictionaries are available in `speeds.jl`. Highway classifications are defined in `classes.jl`.

Note 2: Routing does not account for stoplights, traffic patterns, etc. Each edge is weighted by both distance and typical speed.

1.8 Displaying Maps

OpenStreetMap.jl includes a single plotting function. This function has numerous options, allowing a great deal of flexibility when displaying maps:

```
function plotMap(nodes;
                 highways=nothing,
                 buildings=nothing,
                 features=nothing,
                 bounds=nothing,
                 intersections=nothing,
                 roadways=nothing,
                 cycleways=nothing,
```

```
                walkways=nothing,
                feature_classes=nothing,
                building_classes=nothing,
                route=nothing,
                highway_style::Style = Style(0x007CFF, 1.5, "-"),
                building_style::Style = Style(0x000000, 1, "-"),
                feature_style = Style(0xCC0000, 2.5, "."),
                route_style = Style(0xFF0000, 3, "-"),
                intersection_style::Style = Style(0x000000, 3, "."),
                width::Integer=500,
                fontsize::Integer=0,
                km::Bool=false,
                realtime::Bool=false)
```

The function, `plotMap()`, has a single required input: `nodes`. However, providing `plotMap()` with only the list of nodes will result in an empty plot. The user then has the choice between a variety of plotting options. It is important to note that this function is designed for convenience rather than speed. It is highly recommended that a Bounds object is input, as this is used to provided plot scaling.

The following subsecions step through some of the plotting options. Essentially, the user builds up a series of "layers" through providing multiple inputs.

1.8.1 Data Inputs

These parameters provided the actual data to be plotted.

- `nodes` [`Dict{Int,LLA}` or `Dict{Int,ENU}`]: List of all point locations
- `features` [`Dict{Int,Feature}`]: List of features to display
- `buildings` [`Dict{Int,Building}`]: List of buildings to display
- `highways` [`Dict{Int,Highway}`]: List of highways to display
- `intersections` [`Dict{Int,Intersection}`]: List of highway intersections
- `route` [`Vector{Int}` or `Vector{Vector{Int}}`]: List of nodes comprising a highway route OR a list of lists of routes (if multiple routes are to be displayed).

1.8.2 Data Classifiers

These parameters classify the map elements according to a layer specification. When these parameters are passed to `plotMap()`, only the classified map elements are plotted (all map elements not in these dictionaries are ignored).

- `roadways`: Dictionary of highway types suitable for driving
- `cycleways`: Dictionary of highway types suitable for cycling
- `walkways`: Dictionary of highway types suitable for walking
- `building_classes`: Dictionary of building classifications
- `feature_classes`: Dictionary of feature classifications

Note 1: These layers use their own `Layer` dictionaries, containing one `Style` type for each element classification level, to define plotting styles. Therefore, any additional style inputs related to these classifiers will be ignored without any explicit warnings to the user.

Note 2: Using multiple highway classifiers on one plot may cause them to overlap and occlude one another. The ordering, from bottom to top, is `roadways`, `cycleways`, `walkways`.

1.8.3 Plot Display Options

- `bounds` [`Bounds`]: X and Y axes limits of plot, also used to compute appropriate plot aspect ratio
- `width` [`Integer`]: Width of the plot, in pixels
- `fontsize` [`Integer`]: Fontsize of axes labels. If 0, let Winston decide (default). Use this if you need consistency amongst many plots.
- `km` [`Bool`]: If `nodes` is in ENU coordinates, converts plot axes to use kilometers rather than meters
- `realtime` [`Bool`]: When true, elements are added to the map individually (this drastically slows down plotting, but is fun to watch)

1.8.4 Plot Customization

The following optional inputs allow the user to customize the map display.

- `highway_style` [`Style` or `Dict{Int,Style}`]: See note 3 below.
- `building_style` [`Style` or `Dict{Int,Style}`]: See note 3 below.
- `feature_style` [`Style` or `Dict{Int,Style}`]: See note 3 below.
- `route_style` [`Style` or `Vector{Style}`]: Use an vector of `Style` types to plot multiple routes with different appearances.
- `intersection_style` [`Style`]

These inputs all take a `Style` type, which is constructed as follows:

```
style = OpenStreetMap.Style(color, width, spec)
```

For example:

```
highway_style = OpenStreetMap.Style("b", 1.5, "-")
feature_style = OpenStreetMap.Style(0xf57900, 2, ".")
```

Note 1: `color` must be a hex color code.

Note 2: `spec` is a line specification code used by Winston.jl. Common examples are the following:

- `"-"`: Solid line
- `"."`: Filled, square points
- `"o"`: Open, round points

Note 3: For highways, buildings, and features, if an additional classifier is input (e.g., `roadways`), the respective style input must be a dictionary of styles, with type `Dict{Int,Style}`, with a style given for each classification. This dictionary is called a "layer" in OpenStreetMap terminology, and defines how a specific map layer is displayed. The default layers are defined as constants in `layers.jl`.

1.8.5 Saving Map Images

`plotMap()` returns the `Winston.FramedPlot` object. This allows the user to futher modify the plot or save it using the `file` function available from `Winston.jl` with the desired aspect ratio.

Example of saving a plot as an image in png, eps, and pdf formats:

```
p = plotMap(nodes, bounds=bounds, highways=highways)
width = 500
aspect_ratio = OpenStreetMap.aspectRatio(bounds)
height = int(width / aspect_ratio)
Winston.file(p, "filename.png", "width", width, "height", height)
Winston.file(p, "filename.eps", "width", width, "height", height)
Winston.file(p, "filename.pdf", "width", width, "height", height)
```

1.9 Examples

The following example walks through a sample workflow using OpenStreetMap.jl. This page does not cover all functionality available in OpenStreetMap.jl, but hopefully helps new users get started quickly. See also "test/examples.jl" for all of these examples together in a single Julia file.

Read data from an OSM XML file:

```
nodesLLA, highways, buildings, features = getOSMData(MAP_FILENAME)

println("Number of nodes: $(length(nodesLLA))")
println("Number of highways: $(length(highways))")
println("Number of buildings: $(length(buildings))")
println("Number of features: $(length(features))")
```

Define map boundary:

```
boundsLLA = Bounds(42.365, 42.3675, -71.1, -71.094)
```

Define reference point and convert to ENU coordinates:

```
lla_reference = center(boundsLLA)
nodes = ENU(nodesLLA, lla_reference)
bounds = ENU(boundsLLA, lla_reference)
```

Crop map to boundary:

```
cropMap!(nodes, bounds, highways=highways, buildings=buildings, features=features, delete_nodes=false
```

Find highway intersections:

```
inters = findIntersections(hwys)

println("Found $(length(inters)) intersections.")
```

Extract map components and classes:

```
roads = roadways(hwys)
peds = walkways(hwys)
cycles = cycleways(hwys)
bldg_classes = classify(builds)
feat_classes = classify(feats)
```

Convert map nodes to East-North-Up (ENU) coordinates:

```
reference = center(bounds)
nodesENU = ENU(nodes, reference)
boundsENU = ENU(bounds, reference)
```

Extract highway classes (note that OpenStreetMap calls paths of any form "highways"):

```
roads = roadways(highways)
peds = walkways(highways)
cycles = cycleways(highways)
bldg_classes = classify(buildings)
feat_classes = classify(features)
```

Find all highway intersections:

```
intersections = findIntersections(highways)
```

Segment only specific levels of roadways (e.g., freeways (class 1) through residential streets (class 6)):

```
segments = segmentHighways(nodes, highways, intersections, roads, Set(1:6))
```

Create transportation network from highway segments:

```
network = createGraph(segments, intersections)
```

Compute the shortest and fastest routes from point A to B:

```
loc_start = ENU(-5000, 5500, 0)
loc_end = ENU(5500, -4000, 0)

node0 = nearestNode(nodes, loc_start, network)
node1 = nearestNode(nodes, loc_end, network)
shortest_route, shortest_distance = shortestRoute(network, node0, node1)

fastest_route, fastest_time = fastestRoute(network, node0, node1)
fastest_distance = distance(nodes, fastest_route)

println("Shortest route: $(shortest_distance) m  (Nodes: $(length(shortest_route)))")
println("Fastest route: $(fastest_distance) m  Time: $(fastest_time/60) min  (Nodes: $(length(fastest
```

Display the shortest and fastest routes:

```
fignum_shortest = plotMap(nodesENU, highways=hwys, bounds=boundsENU, roadways=roads, route=shortest_r

fignum_fastest = plotMap(nodesENU, highways=hwys, bounds=boundsENU, roadways=roads, route=fastest_rou
```

Extract Nodes near to (within range) our route's starting location:

```
loc0 = nodes[node0]
filteredENU = filter((k,v)->haskey(network.v,k), nodes)
local_indices = nodesWithinRange(filteredENU, loc0, 100.0)
```

Identify Driving Catchment Areas (within limit):

```
start_index = nearestNode(filteredENU, loc0)
node_indices, distances = nodesWithinDrivingDistance(network, local_indices, 300.0)
```

Alternatively, switch to catchment areas based on driving time, rather than distance:

```
node_indices, distances = nodesWithinDrivingTime(network, local_indices, 50.0)
```

Display classified roadways, buildings, and features:

```
fignum = plotMap(nodes,
                 highways=highways,
                 buildings=buildings,
                 features=features,
                 bounds=bounds,
```

```
                  width=500,
                  feature_classes=feat_classes,
                  building_classes=bldg_classes,
                  roadways=roads)

Winston.savefig("osm_map.png")
```

Note: Winston currently distorts figures slightly when it saves them. Therefore, whenever equal axes scaling is required, export figures as EPS and rescale them as necessary.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

C

D

F

G

H

N

R

S