bmm

Release 1.1

Sam Duffield

Jul 19, 2022

CONTENTS

bmm provides map-matching with uncertainty quantification for both online and offline inference!

Map-matching converts a series of noisy GPS coordinates into a continuous trajectory that is restricted to a graph (i.e. road network) or in the case of bmm a collection of continuous trajectories representing multiple plausible routes!

bmm is built on top of osmnx, an awesome package for retrieving and processing OpenStreetMap data.

The probabilistic model and particle smoothing methodology behind bmm can be found on arXiv and the source code on GitHub.

CHAPTER ONE

DOCS

1.1 Functions

bmm.**offline_map_match**(*graph*, *polyline*, *n_samps*, *timestamps*, *mm_model=<bmm.ExponentialMapMatchingModel object>*, *proposal_func=<function optimal_proposal>*, *d_refine=1*, *initial_d_truncate=None*, *max_rejections=20*, *ess_threshold=1*, *store_norm_quants=False*, *store_filter_particles=False*, *verbose=True*, ***kwargs*)

Runs offline map-matching, i.e. receives a full polyline and returns an equal probability collection of trajectories. Forward-filtering backward-simulation implementation - no fixed-lag approximation needed for offline inference.

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **polyline** (*ndarray*) – series of cartesian cooridnates in UTM
- **n_samps** (*int*) – int number of particles
- **timestamps** (*Union[float, ndarray]*) – seconds either float if all times between observations are the same, or a series of timestamps in seconds/UNIX timestamp
- **mm_model** (*MapMatchingModel*) – MapMatchingModel
- **proposal_func** (*Callable*) – function to propagate and weight single particle defaults to optimal (discretised) proposal
- **d_refine** (*int*) – metres, resolution of distance discretisation
- **initial_d_truncate** (*Optional[float]*) – distance beyond which to assume zero likelihood probability at time zero defaults to 5 * mm_model.gps_sd
- **max_rejections** (*int*) – number of rejections to attempt before doing full fixed-lag stitching 0 will do full fixed-lag stitching and track ess_stitch
- **ess_threshold** (*float*) – in [0,1], particle filter resamples if ess < ess_threshold * n_samps
- **store_norm_quants** (*bool*) – if True normalisation quantities (including gradient evals) returned in out_particles
- **store_filter_particles** (*bool*) – if True filter particles returned in out_particles
- **verbose** (*bool*) – bool whether to print ESS at each iterate
- **kwargs** – optional parameters to pass to proposal i.e. d_max, d_refine or var as well as ess_threshold for backward simulation update

Returns
MMParticles object

Return type
MMParticles

`bmm.initiate_particles`(*graph*, *first_observation*, *n_samps*, *mm_model=<bmm.ExponentialMapMatchingModel object>*, *d_refine=1*, *d_truncate=None*, *ess_all=True*, *filter_store=True*)

Initiate start of a trajectory by sampling points around the first observation. Note that coordinate system of inputs must be the same, typically a UTM projection (not longtitude-latitude!).

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **mm_model** (MapMatchingModel) – MapMatchingModel
- **first_observation** (*ndarray*) – cartesian coordinate in UTM
- **n_samps** (*int*) – number of samples to generate
- **d_refine** (*float*) – metres, resolution of distance discretisation
- **d_truncate** (*Optional[float]*) – metres, distance beyond which to assume zero likelihood probability defaults to 5 * mm_model.gps_sd
- **ess_all** (*bool*) – if true initiate effective sample size for each particle for each observation otherwise initiate effective sample size only for each observation
- **filter_store** (*bool*) – whether to initiate storage of filter particles and weights

Returns
MMParticles object

Return type
MMParticles

`bmm.update_particles`(*graph*, *particles*, *new_observation*, *time_interval*, *mm_model=<bmm.ExponentialMapMatchingModel object>*, *proposal_func=<function optimal_proposal>*, *update='BSi'*, *lag=3*, *max_rejections=20*, ***kwargs*)

Updates particle approximation in receipt of new observation

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **particles** (MMParticles) – unweighted particle approximation up to the previous observation time
- **new_observation** (*ndarray*) – cartesian coordinate in UTM
- **time_interval** (*float*) – time between last observation and newly received observation
- **mm_model** (MapMatchingModel) – MapMatchingModel
- **proposal_func** (*Callable*) – function to propagate and weight single particle
- **update** (*str*) –
  - 'PF' for particle filter fixed-lag update
  - 'BSi' for backward simulation fixed-lag update

  must be consistent across updates

- **lag** (*int*) – fixed lag for resampling/stitching
- **max_rejections** (*int*) – number of rejections to attempt before doing full fixed-lag stitching 0 will do full fixed-lag stitching and track ess_stitch
- **kwargs** – optional parameters to pass to proposal i.e. d_max, d_refine or var as well as ess_threshold for backward simulation update

Returns
MMParticles object

Return type
MMParticles

bmm.**_offline_map_match_fl**(*graph*, *polyline*, *n_samps*, *timestamps*, *mm_model=<bmm.ExponentialMapMatchingModel object>*, *proposal_func=<function optimal_proposal>*, *update='BSi'*, *lag=3*, *d_refine=1*, *initial_d_truncate=None*, *max_rejections=20*, *verbose=True*, ***kwargs*)

Runs offline map-matching but uses online fixed-lag techniques. Only recommended for simulation purposes.

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **polyline** (*ndarray*) – series of cartesian coordinates in UTM
- **n_samps** (*int*) – int number of particles
- **timestamps** (*Union[float, ndarray]*) – seconds, either float if all times between observations are the same, or a series of timestamps in seconds/UNIX timestamp
- **mm_model** (MapMatchingModel) – MapMatchingModel
- **proposal_func** (*Callable*) – function to propagate and weight single particle defaults to optimal (discretised) proposal
- **update** (*str*) –
  - 'PF' for particle filter fixed-lag update
  - 'BSi' for backward simulation fixed-lag update

  must be consistent across updates
- **lag** (*int*) – fixed lag for resampling/stitching
- **d_refine** (*int*) – metres, resolution of distance discretisation
- **initial_d_truncate** (*Optional[float]*) – distance beyond which to assume zero likelihood probability at time zero, defaults to 5 * mm_model.gps_sd
- **max_rejections** (*int*) – number of rejections to attempt before doing full fixed-lag stitching, 0 will do full fixed-lag stitching and track ess_stitch
- **verbose** (*bool*) – bool whether to print ESS at each iterate
- **kwargs** – optional parameters to pass to proposal i.e. d_max or var as well as ess_threshold for backward simulation update

Returns
MMParticles object

Return type
MMParticles

bmm.**sample_route**(*graph*, *timestamps*, *num_obs=None*, *mm_model=<bmm.ExponentialMapMatchingModel object>*, *d_refine=1.0*, *start_position=None*, *num_inter_cut_off=None*)

Runs offline map-matching. I.e. receives a full polyline and returns an equal probability collection of trajectories. Forward-filtering backward-simulation implementation - no fixed-lag approximation needed for offline inference.

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **timestamps** (*Union[float, ndarray]*) – seconds either float if all times between observations are the same, or a series of timestamps in seconds/UNIX timestamp
- **num_obs** (*Optional[int]*) – int length of observed polyline to generate
- **mm_model** (MapMatchingModel) – MapMatchingModel
- **d_refine** (*float*) – metres, resolution of distance discretisation
- **start_position** (*Optional[ndarray]*) – optional start position; array (u, v, k, alpha)
- **num_inter_cut_off** (*Optional[int]*) – maximum number of intersections to cross in the time interval

Returns

tuple with sampled route (array with same shape as a single MMParticles) and polyline (array with shape (num_obs, 2))

Return type

Tuple[*ndarray*, *ndarray*]

bmm.**random_positions**(*graph*, *n=1*)

Sample random positions on a graph. :param graph: encodes road network, simplified and projected to UTM :param n: int number of positions to sample, default 1 :return: array of positions (u, v, key, alpha) - shape (n, 4)

Parameters

- **graph** (*MultiDiGraph*) –
- **n** (*int*) –

Return type

ndarray

bmm.**offline_em**(*graph*, *mm_model*, *timestamps*, *polylines*, *save_path*, *n_ffbsi=100*, *n_iter=10*, *gradient_stepsize_scale=0.001*, *gradient_stepsize_neg_exp=0.5*, ***kwargs*)

Run expectation maximisation to optimise parameters of bmm.MapMatchingModel object. Updates the hyperparameters of mm_model in place.

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **mm_model** (MapMatchingModel) – MapMatchingModel - of which parameters will be updated
- **timestamps** (*Union[list, float]*) – seconds, either float if all times between observations are the same, or a series of timestamps in seconds/UNIX timestamp, if timestamps given, must be of matching dimensions to polylines
- **polylines** (*list*) – UTM polylines
- **save_path** (*str*) – path to save learned parameters
- **n_ffbsi** (*int*) – number of samples for FFBSi algorithm

- **n_iter** (*int*) – number of EM iterations
- **gradient_stepsize_scale** (*float*) – starting stepsize
- **gradient_stepsize_neg_exp** (*float*) – rate of decay of stepsize, in [0.5, 1]
- **kwargs** – additional arguments for FFBSi

Returns
dict of optimised parameters

bmm.**plot**(*graph*, *particles=None*, *polyline=None*, *label_start_end=True*, *bgcolor='white'*, *node_color='grey'*, *node_size=0*, *edge_color='lightgrey'*, *edge_linewidth=3*, *particles_color='orange'*, *particles_alpha=None*, *polyline_color='red'*, *polyline_s=100*, *polyline_linewidth=3*, ***kwargs*)

Plots particle approximation of trajectory

Parameters

- **graph** – NetworkX MultiDiGraph UTM projection encodes road network e.g. generated using OSMnx
- **particles** – MMParticles object (from inference.particles) particle approximation
- **polyline** – list-like, each element length 2 UTM - metres series of GPS coordinate observations
- **label_start_end** – bool whether to label the start and end points of the route
- **bgcolor** – str background colour
- **node_color** – str node (intersections) colour
- **node_size** – float size of nodes (intersections)
- **edge_color** – str colour of edges (roads)
- **edge_linewidth** – float width of edges (roads
- **particles_color** – str colour of routes
- **particles_alpha** – float in [0, 1] plotting parameter opacity of routes
- **polyline_color** – str colour of polyline crosses
- **polyline_s** – str size of polyline crosses
- **polyline_linewidth** – str linewidth of polyline crosses
- **kwargs** – additional parameters to ox.plot_graph

Returns
fig, ax

bmm.**get_possible_routes**(*graph*, *in_route*, *dist*, *all_routes=False*, *num_inter_cut_off=inf*)

Given a route so far and maximum distance to travel, calculate and return all possible routes on graph.

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **in_route** (*ndarray*) – shape = (_, 9) columns: t, u, v, k, alpha, x, y, n_inter, d t: float, time u: int, edge start node v: int, edge end node k: int, edge key alpha: in [0,1], position along edge x: float, metres, cartesian x coordinate y: float, metres, cartesian y coordinate d: metres, distance travelled
- **dist** (*float*) – metres, maximum possible distance to travel

- **all_routes** (*bool*) – if true return all routes possible <= d otherwise return only routes of length d
- **num_inter_cut_off** (*int*) – maximum number of intersections to cross in the time interval

Returns
list of arrays each array with shape = (_, 9) as in_route each array describes a possible route

Return type
list

bmm.**cartesianise_path**(*graph*, *path*, *t_column=True*, *observation_time_only=False*)

Converts particle or array of edges and alphas into cartesian (x,y) points.

Parameters

- **path** – numpy.ndarray, shape=(_, 5+) columns - (t), u, v, k, alpha, …
- **t_column** – boolean boolean describing if input has a first column for the time variable

Returns
numpy.ndarray, shape = (_, 2) cartesian points

bmm.**get_geometry**(*graph*, *edge*)

Extract geometry of an edge from global graph object. If geometry doesn't exist set to straight line.

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **edge** (*ndarray*) – length = 3 with elements u, v, k * u: int, edge start node * v: int, edge end node * k: int, edge key

Returns
edge geometry

Return type
LineString

bmm.**discretise_edge**(*graph*, *edge*, *d_refine*)

Discretises edge to given edge refinement parameter. Returns array of proportions along edge, xy cartesian coordinates and distances along edge

Parameters

- **graph** (*MultiDiGraph*) – encodes road network, simplified and projected to UTM
- **edge** (*ndarray*) – list-like, length = 3 with elements u, v, k * u: int, edge start node * v: int, edge end node * k: int, edge key
- **d_refine** (*float*) – metres, resolution of distance discretisation

Returns
shape = (_, 4) with columns * alpha: float in (0,1], position along edge * x: float, metres, cartesian x coordinate * y: float, metres, cartesian y coordinate * distance: float, distance from start of edge

Return type
ndarray

bmm.**observation_time_indices**(*times*)

Remove zeros (other than the initial zero) from a series

Parameters
times (*ndarray*) – series of timestamps

Returns
bool array of timestamps that are either non-zero or the first timestamp

Return type
ndarray

`bmm.observation_time_rows(path)`

Returns rows of path only at observation times (not intersections)

Parameters
path (*ndarray*) – numpy.ndarray, shape=(_, 5+) columns - t, u, v, k, alpha, . . .

Returns
trimmed path numpy.ndarray, shape like path

Return type
ndarray

`bmm.long_lat_to_utm(points, graph=None)`

Converts a collection of long-lat points to UTM :param points: points to be projected, shape = (N, 2) :param graph: optional graph containing desired crs in graph.graph['crs'] :return: array of projected points

Parameters
points (*Union[list, ndarray]*) –

Return type
ndarray

1.2 Classes

`class bmm.MMParticles(initial_positions)`

Class to store trajectories from a map-matching algorithm.

In particular, contains the `self.particles` object, which is a list of n arrays each with shape = (_, 8)

where _ represents the trajectory length (number of nodes that are either intersection or observation) and columns:

- t: seconds, observation time
- u: int, edge start node
- v: int, edge end node
- k: int, edge key
- alpha: in [0,1], position along edge
- x: float, metres, cartesian x coordinate
- y: float, metres, cartesian y coordinate
- d: float, metres, distance travelled since previous observation time

As well as some useful properties: * `self.n`: number of particles * `self.m`: number of observations * `self.observation_times`: array of observation times * `self.latest_observation_time`: time of most recently received observation * `self.route_nodes`: list of length n, each element contains the series of nodes traversed for that particle

Initiate MMParticles storage of trajectories with some start positions as input.

Parameters
`initial_positions` (*List[ndarray]*) – list, length = n_samps, each element is an array of length 6 with elements

- u: int, edge start node
- v: int, edge end node
- k: int, edge key
- alpha: in [0,1], position along edge
- x: float, metres, cartesian x coordinate
- y: float, metres, cartesian y coordinate

`property latest_observation_time: float`

Extracts most recent observation time. :return: time of most recent observation

`property m: int`

Number of observations received. :return: number of observations received

`property observation_times: ndarray`

Extracts all observation times. :return: array, shape = (m,)

`route_nodes()`

Returns n series of nodes describing the routes :return: length n list of arrays, shape (_,)

where _ represents the trajectory length (number of nodes that are either intersection or observation)

`class bmm.MapMatchingModel`

Class defining the state-space model used for map-matching.

Transition density (assuming constant time interval)

$$p(x_t, e_t | x_{t-1}) \propto \gamma(d_t) \exp(-\beta |d^{\text{gc}}_t - d_t|) \mathbb{1}[d_t < d_{\text{max}}],$$

where d_t is the distance between positions x_{t-1} and x_t along the series of edges e_{t-1}, restricted to the graph/road network. d^{gc}_t is the *great circle distance* between x_{t-1} and x_t, not restricted to the graph/road network.

The $\exp(-\beta |d^{\text{gc}}_t - d_t|)$ term penalises non-direct or windy routes where β is a parameter stored in `self.deviation_beta`, yet to be defined.

d_{max} is defined by `self.d_max` function (metres) and `self.max_speed` parameter (metres per second), defaults to 35.

The $\gamma(d_t)$ term penalises overly lengthy routes and is yet to be defined.

Observation density

$$p(y_t | x_t) = \mathcal{N}(y_t \mid x_t, \sigma^2_{\text{GPS}} \mathbb{I}_2),$$

where σ_{GPS} is the standard deviation (metres) of the GPS noise stored in `self.gps_sd`, yet to be defined. Additional optional `self.likelihood_d_truncate` for truncated Gaussian noise, defaults to inf.

The parameters `self.deviation_beta`, `self.gps_sd` and the distance prior parameters defined in `self.distance_params` and `self.distance_params_bounds` can be tuned using expectation-maximisation with `bmm.offline_em`.

For more details see https://arxiv.org/abs/2012.04602.

d_max(*time_interval*)

Initiates default value of the maximum distance possibly travelled in the time interval. Assumes a maximum possible speed.

Parameters
time_interval (*float*) – float seconds time between observations

Returns
float defaulted d_max

Return type
float

deviation_prior_evaluate(*previous_cart_coord*, *route_cart_coords*, *distances*)

Evaluate deviation prior/transition density Vectorised to handle multiple evaluations at once :param previous_cart_coord: shape = (2,) or (_, 2) cartesian coordinate(s) at previous observation time :param route_cart_coords: shape = (_, 2), cartesian coordinates - positions along road network :param distances: shape = (_,) route distances between previous_cart_coord(s) and route_cart_coords :return: deviation prior density evaluation(s)

Parameters

- **previous_cart_coord** (*ndarray*) –
- **route_cart_coords** (*ndarray*) –
- **distances** (*ndarray*) –

Return type
ndarray

distance_prior_bound(*time_interval*)

Extracts bound on the distance component of the prior/transition density :param time_interval: seconds, time between observations :return: bound on distance prior density

Parameters
time_interval (*float*) –

Return type
float

distance_prior_evaluate(*distance*, *time_interval*)

Evaluate distance prior/transition density Vectorised to handle multiple evaluations at once

Parameters

- **distance** (*Union[float, ndarray]*) – metres array if multiple evaluations at once
- **time_interval** (*Union[float, ndarray]*) – seconds, time between observations

Returns
distance prior density evaluation(s)

Return type
Union[float, *ndarray*]

distance_prior_gradient(*distance*, *time_interval*)

Evaluate gradient of distance prior/transition density in distance_params Vectorised to handle multiple evaluations at once

Parameters

- **distance** (*Union[float, ndarray]*) – metres array if multiple evaluations at once

- **time_interval** (*Union[float, ndarray]*) – seconds, time between observations

Returns
distance prior density evaluation(s)

Return type
Union[float, *ndarray*]

`likelihood_evaluate`(*route_cart_coords*, *observation*)

Evaluate probability of generating observation from cartesian coords Vectorised to evaluate over many cart_coords for a single observation Isotropic Gaussian with standard dev self.gps_sd :param route_cart_coords: shape = (_, 2), cartesian coordinates - positions along road network :param observation: shape = (2,) observed GPS cartesian coordinate :return: shape = (_,) likelihood evaluations

Parameters

- **route_cart_coords** (*ndarray*) –
- **observation** (*ndarray*) –

Return type
Union[float, *ndarray*]

`pos_distance_prior_bound`(*time_interval*)

Extracts bound on the distance component of the prior/transition density given the distance is > 0 :param time_interval: seconds, time between observations :return: bound on distance prior density

Parameters
time_interval (*float*) –

Return type
float

class `bmm.ExponentialMapMatchingModel`(*zero_dist_prob_neg_exponent=0.133*, *lambda_speed=0.068*, *deviation_beta=0.052*, *gps_sd=5.23*)

Class defining the state-space model used for map-matching with exponential prior on distance travelled.

Transition density (assuming constant time interval)

$$p(x_t, e_t | x_{t-1}) \propto \gamma(d_t) \exp(-\beta |d_t^{\text{gc}} - d_t|) \mathbb{1}[d_t < d_{\max}],$$

where d_t is the distance between positions x_{t-1} and x_t along the series of edges e_{t-1}, restricted to the graph/road network. d_t^{gc} is the *great circle distance* between x_{t-1} and x_t, not restricted to the graph/road network.

The $\exp(-\beta |d_t^{\text{gc}} - d_t|)$ term penalises non-direct or windy routes where β is a parameter stored in `self.deviation_beta` defaults to 0.052.

$d_{\max}$ is defined by `self.d_max` function (metres) and `self.max_speed` parameter (metres per second), defaults to 35.

The $\gamma(d_t)$ term

$$\gamma(d_t) = p^0 \mathbb{1}[d_t = 0] + (1 - p^0) \mathbb{1}[d_t > 0] \lambda \exp(-\lambda d_t / \Delta t),$$

penalises overly lengthy routes, defined as an exponential distribution with probability mass at $d_t = 0$ to account for traffic, junctions etc.

where $p^0 = \exp(-r^0 \Delta t)$ with Δt being the time interval between observations. The r^0 parameter is stored in `self.zero_dist_prob_neg_exponent` and defaults to 0.133. Exponential distribution parameter λ is stored in `self.lambda_speed` and defaults to 0.068.

Observation density

$$p(y_t|x_t) = \mathcal{N}(y_t \mid x_t, \sigma_{\text{GPS}}^2 \mathbb{I}_2),$$

where σ_{GPS} is the standard deviation (metres) of the GPS noise stored in `self.gps_sd`, defaults to 5.23. Additional optional `self.likelihood_d_truncate` for truncated Gaussian noise, defaults to inf.

The parameters `self.deviation_beta`, `self.gps_sd` as well as the distance prior parameters `self.zero_dist_prob_neg_exponent` and `self.lambda_speed` can be tuned using expectation-maximisation with `bmm.offline_em`.

For more details see https://arxiv.org/abs/2012.04602.

Parameters

- **zero_dist_prob_neg_exponent** (*float*) – Positive parameter such that stationary probability is $p^0 = \exp(-r^0 \Delta t)$, defaults to 0.133.
- **lambda_speed** (*float*) – Positive parameter of exponential distribution over average speed between observations.
- **deviation_beta** (*float*) – Positive parameter of exponential distribution over route deviation.
- **gps_sd** (*float*) – Positive parameter defining standard deviation of GPS noise in metres.

distance_prior_bound(*time_interval*)

Extracts bound on the prior/transition density :param time_interval: seconds, time between observations :return: bound on distance prior density

Parameters
time_interval (*float*) –

Return type
float

distance_prior_evaluate(*distance*, *time_interval*)

Evaluate distance prior/transition density Vectorised to handle multiple evaluations at once

Parameters

- **distance** (*Union[float, ndarray]*) – metres array if multiple evaluations at once
- **time_interval** (*Union[float, ndarray]*) – seconds, time between observations

Returns
distance prior density evaluation(s)

Return type
Union[float, *ndarray*]

distance_prior_gradient(*distance*, *time_interval*)

Evaluate gradient of distance prior/transition density in distance_params Vectorised to handle multiple evaluations at once

Parameters

- **distance** (*Union[float, ndarray]*) – metres array if multiple evaluations at once
- **time_interval** (*Union[float, ndarray]*) – seconds, time between observations

Returns
distance prior gradient evaluation(s)

Return type
Union[float, *ndarray*]

`pos_distance_prior_bound`(*time_interval*)

Extracts bound on the distance component of the prior/transition density given the distance is > 0 :param time_interval: seconds, time between observations :return: bound on distance prior density

Parameters
time_interval (*float*) –

Return type
float

`zero_dist_prob`(*time_interval*)

Probability of travelling a distance of exactly zero :param time_interval: time between last observation and newly received observation :return: probability of travelling zero metres in time_interval

Parameters
time_interval (*Union[float, ndarray]*) –

Return type
Union[float, *ndarray*]

1.3 Index

CHAPTER

TWO

INSTALL

```
pip install bmm
```

CHAPTER

THREE

QUICKSTART

Load graph and convert to UTM (Universal Transverse Mercator), a commonly used projection of spherical longtitude-latitude coordinates into square x-y coordinates:

```
import numpy as np
import pandas as pd
import osmnx as ox
import json
import bmm

graph = ox.graph_from_place('Porto, Portugal')
graph = ox.project_graph(graph)
```

Beware that downloading graphs using `osmnx` can take a few minutes, especially for large cities.

Load polyline and convert to UTM:

```
data_path = 'simulations/porto/test_route.csv'
polyline_longlat = json.loads(pd.read_csv(data_path)['POLYLINE'][0])
polyline_utm = bmm.long_lat_to_utm(polyline_longlat, graph)
```

3.1 Offline map-matching

```
matched_particles = bmm.offline_map_match(graph, polyline=polyline_utm, n_samps=100,␣
↪timestamps=15)
```

3.2 Online map-matching

Initiate with first observation:

```
matched_particles = bmm.initiate_particles(graph, first_observation=polyline_utm[0], n_
↪samps=100)
```

Update when new observation comes in

```
matched_particles = bmm.update_particles(graph, matched_particles, new_
↪observation=polyline_utm[1], time_interval=15)
```

CHAPTER FOUR

SANITY CHECK

You can manually test that bmm is working sensibly for a given graph by generating synthetic data:

```
graph = ox.graph_from_place('London, UK')
graph = ox.project_graph(graph)
generated_route, generated_polyline = bmm.sample_route(graph, timestamps=15, num_obs=20)
```

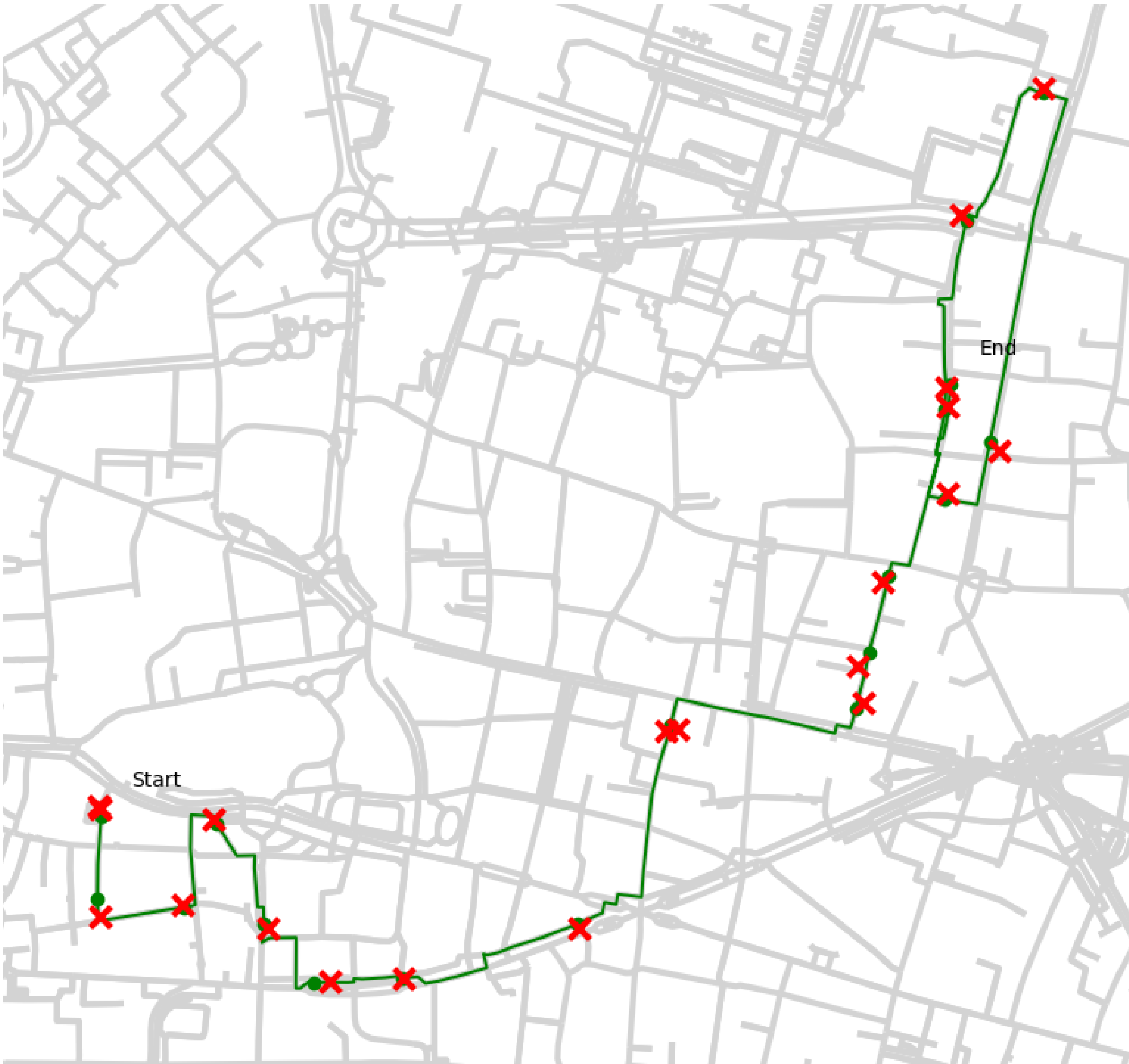
Note that the London graph takes some time (~10mins) to download and for testing on synthetic data it may be worth considering a smaller region (although not so small that the sample_route function consistently terminates early due to reaching the edge of the graph).

Run map-matching on the generated polyline:

```
matched_particles = bmm.offline_map_match(graph, generated_polyline, n_samps=100,␣
↪timestamps=15)
```

Plot true generated route:

```
bmm.plot(graph, generated_route, generated_polyline, particles_color='green')
```



Plot map-matched particles:

```
bmm.plot(graph, matched_particles, generated_polyline)
```


Start
End

INDEX