low carbon mobility Lille

In [121…

```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [122…

```
import site
import sys

# add special development library for network analysis
site.addsitedir('/Users/fabien/Documents/workspace/github/policosm')  # Always appends to end

import itertools
from operator import itemgetter

from scipy.spatial import cKDTree, distance
from scipy.spatial import ConvexHull, convex_hull_plot_2d

from shapely.geometry import Point, LineString, Polygon
from shapely.ops import unary_union

import numpy as np
import pandas as pd
import geopandas as gpd

import matplotlib.pyplot as plt
import seaborn as sns

import graph_tool as gt
import graph_tool.util as gtu
import graph_tool.search as gts
import graph_tool.topology as gtt

from asynchroneBuffer import *

sns.set()
```

load the shp for communes of MEL to get MEL boundaries

To extract the road map from openstreetmap we need to:

1. find the boundaries of MEL
2. get the polygon envelop of MEL
3. use the polygon with `osmium extract` to get MEL.pbf (binary format of osm xml extract
4. from the osm extract, get the road network

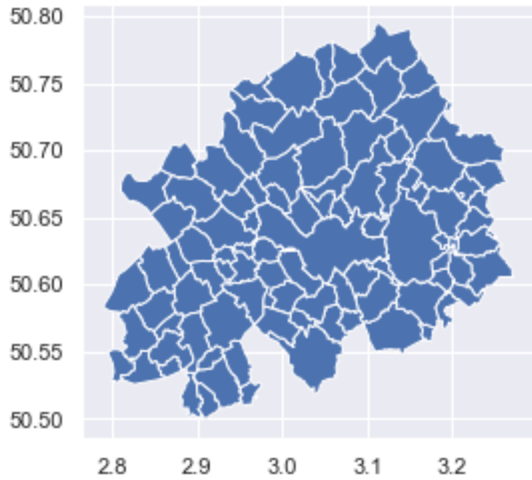
In [93]:

```
mel_cities = gpd.read_file('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/mel_communes.zip')
```

In [94]:

```
mel_cities.plot()
```

Out[94]: <AxesSubplot:>



In [95]:
```
mel_boundary = mel_cities.geometry.unary_union
```

In [96]:
```
mel_boundary
```

Out[96]:

create a geojson from the polygon boundaries by recreating a geo dataframe

In [97]:
```
d = {'name': ['mel_boundary']}
df = pd.DataFrame(d)
gs = gpd.GeoSeries.from_wkt([mel_boundary.to_wkt()])
mel_boundary = gpd.GeoDataFrame(df, geometry=gs, crs="EPSG:4326")
mel_boundary.to_file('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/mel_boundary.geojson', driver='GeoJS
```

Extract MEL

we use a larger region extract found on geofabrik

command is `osmium extract -p mel_boundary.geojson nord-pas-de-calais-latest.osm.pbf -o mel.pbf`

Extract the network from osm using policosm

The network use the **EPSG 3950** aka **CC50** projection in meters from IGN.

The whole network is NOT in 4326 mercator so GPS coordinates are not comparable

In [19]:
```
import policosm
```

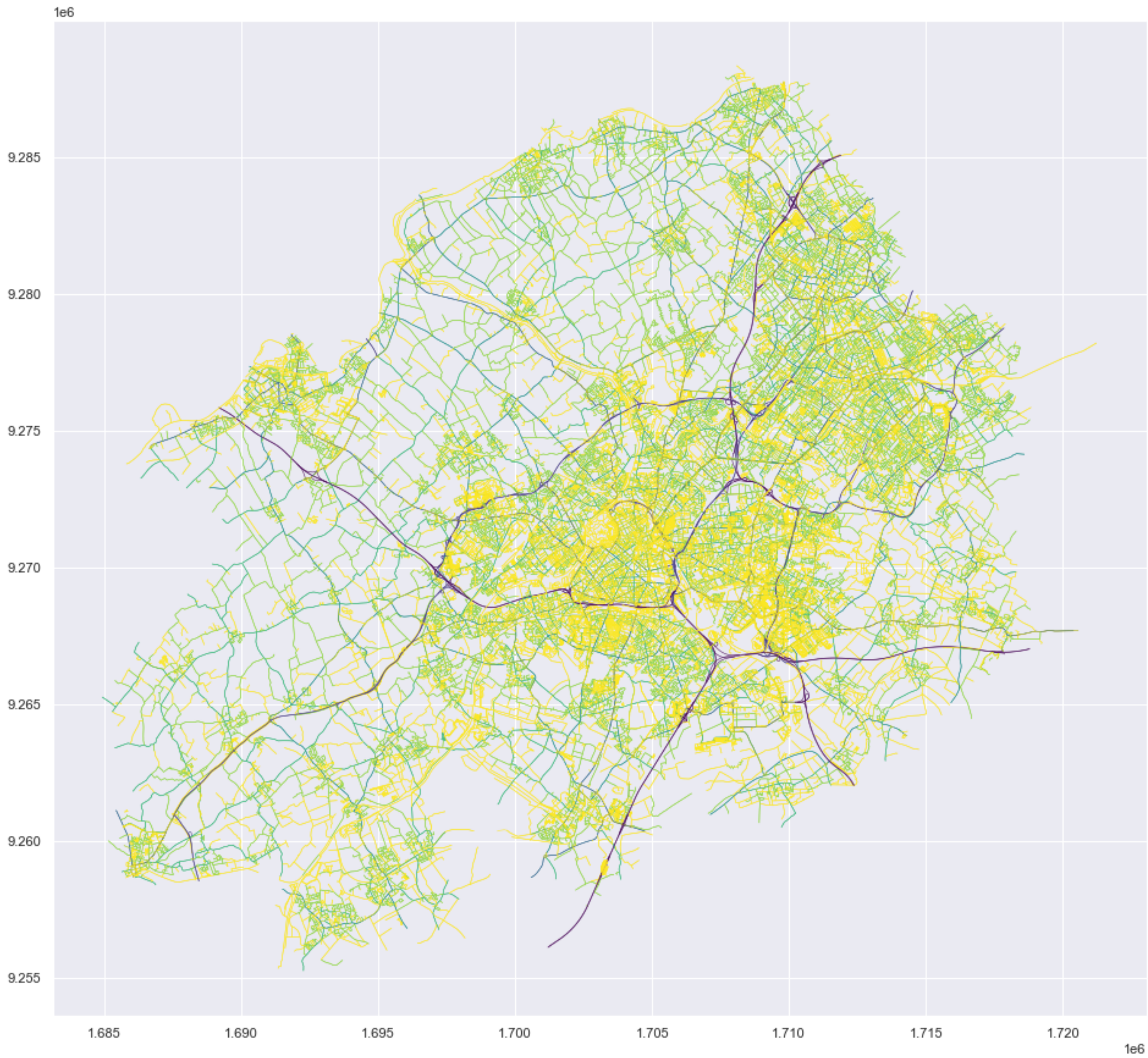
In [20]:
```
roads = policosm.classes.roads.Roads(directed=True, country_iso3="fra")
roads.apply_file('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/mel.pbf')
```

In [22]:
```
roads.osm_to_dataframes(project_overwrite_epsg=3950)
roads.dfe = policosm.geoNetworks.simplify.simplify_directed_as_dataframe(roads.dfe)
```

verification of the map using simple mapping

In [23]:
```
roads.dfe.plot(column='level', cmap='viridis_r', linewidth=0.5, figsize=(20,15))
```

Out[23]: <AxesSubplot:>



saving network data as parquet to save space

In [25]:
```
roads.dfe.to_parquet('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/mel.parquet')
```

```
<ipython-input-25-db0f0053b567>:1: UserWarning: this is an initial implementa
tion of Parquet/Feather file support and associated metadata.  This is tracki
ng version 0.1.0 of the metadata specification at https://github.com/geopanda
s/geo-arrow-spec

This metadata specification does not yet make stability promises.  We do not
yet recommend using this in a production setting unless you are able to rewri
te your Parquet/Feather files.

To further ignore this warning, you can do:
```

```
import warnings; warnings.filterwarnings('ignore', message='.*initial impleme
ntation of Parquet.*')
  roads.dfe.to_parquet('/Users/fabien/Dropbox/low-carbon-lille/cartographic-r
esources/mel.parquet')
```

Network Study

Now that the network is available we can use it to study bikes

first we read the parquet back into a geodataframe

In [4]:
```
dfe = gpd.read_parquet('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/mel.parquet')
```

Simple Description

the network holds 278,813 edges which is a reasonably large graph to calculate

- bicycle is a 1 for True and 0 for False column
- bicycle safety use a ranking from 0 (shared with cars) to 3 (dedicated cycleway) to evaluate safety. It is based on osm complicated way of qualifying all the urban situation: osm wiki for cycleway

In [5]:
```
dfe.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Int64Index: 278813 entries, 1108327 to 1108326
Data columns (total 15 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   u               278813 non-null  int64
 1   v               278813 non-null  int64
 2   path            278813 non-null  object
 3   osm_id          278813 non-null  object
 4   highway         278813 non-null  object
 5   level           278813 non-null  int64
 6   lanes           278813 non-null  int64
 7   width           278813 non-null  float64
 8   bicycle         278813 non-null  int64
 9   bicycle_safety  278813 non-null  int64
 10  foot            278813 non-null  int64
 11  foot_safety     278813 non-null  int64
 12  max_speed       278813 non-null  int64
 13  motorcar        278813 non-null  int64
 14  geometry        278813 non-null  geometry
dtypes: float64(1), geometry(1), int64(10), object(3)
memory usage: 34.0+ MB
```

In [6]:
```
dfe.sample(3)
```

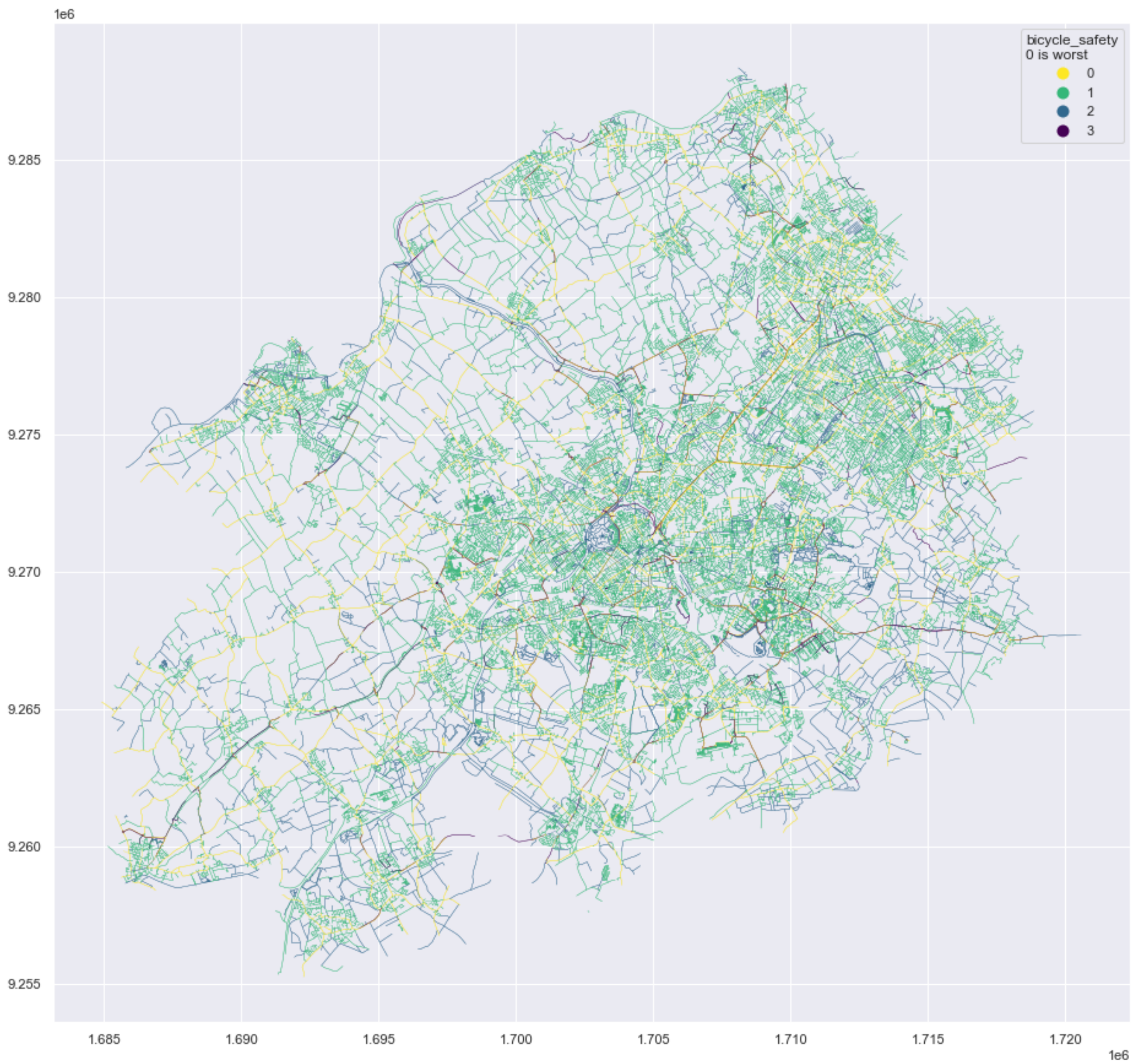
Out[6]:

	u	v	path	osm_id	highway	level	lanes	width
edge_id								
1057552	3106048538	2167223594	[3106048538, 6383598955, 6383598956,	331084356	secondary	5	1	3.0

	u	v	path	osm_id	highway	level	lanes	width
edge_id								
			638359895...					
1038499	6357787824	6357787858	[6357787824, 6357787857, 6357787856, 635778785...	184615789	secondary	5	1	3.0
			[216189993					

map of the roads authorized for bikes

In [7]:
```
ax = dfe[dfe.bicycle == 1].plot(column='bicycle_safety', cmap='viridis_r', categorical=True, linewidth=0.3, figsize=(20,15), l
fig = ax.get_figure()
ax.get_legend().set_title('bicycle_safety\n0 is worst')
#fig.savefig('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/mel-cycleway.pdf')
```



Adding speed to Bike network

To add speed to bike network, we get distance from the geometry column in meters (thanks to the projection in CC50) divided by the speed (m s-1) of the bike set at 15km/h

In [8]:
```
average_speed_kph = 15
average_speed_mps = average_speed_kph * 1000 / 3600
dfe['time'] = dfe.length / average_speed_mps
```

In [9]:
```
from asynchroneBuffer import *
```

exemple of reachable zone for one iris

we start by reading the irises shapefiles and put them in a compatible projection **CC50**

In [10]:
```
irises = gpd.read_file('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/IRIS-GE_2-0_SHP_LAMB93_D059-2(
```

In [11]:
```
irises = irises.to_crs(3950)
```

the irises are for the whole department of 59, nord pas de calais. We will cut it using the MEL boundary. We also reduce the boundaries a little bit (200m) so that the intersect do not capture unnecessary irises

In [12]:
```
mel_boundary = gpd.read_file('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/mel_boundary.geojson')
mel_boundary = mel_boundary.to_crs(3950)
mel_boundary.geometry = mel_boundary.geometry.apply(lambda g: g.buffer(-200))
```

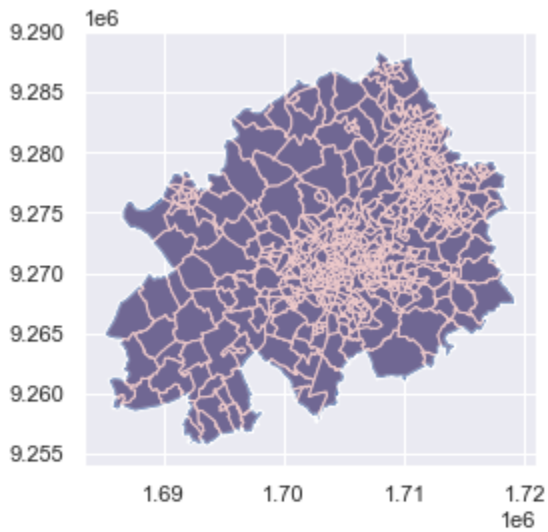
irisMEL will contain our iris data from the MEL

In [13]:
```
irisMEL = gpd.sjoin(irises,mel_boundary,op='intersects')
```

we check on a map superposing the irisMEL and the boundaries (boundaries are transparent red)

In [14]:
```
ax = irisMEL.plot()
mel_boundary.plot(ax=ax,color='r',alpha=0.3)
```

Out[14]: <AxesSubplot:>



30 min from a random IRIS

Select an IRIS at random and calculate 30min on a bike from it

This operation take several steps:

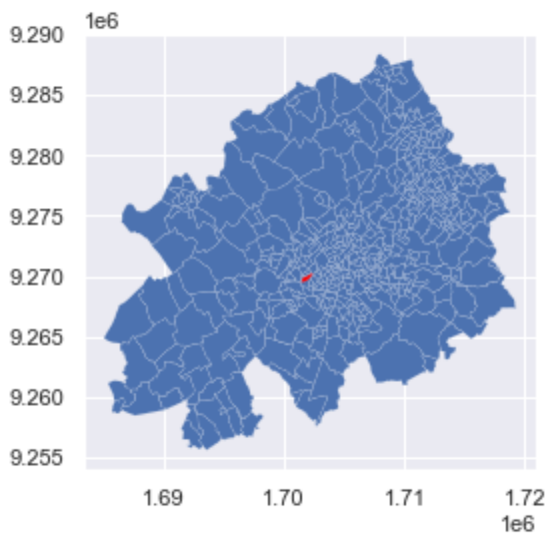
1. select an IRIS to start
2. get the iris boundaries to get starting points
3. simplify iris boundaries to get < 20 points
4. find the index on the network of each coordinates of each point on the boundary (separate object)
5. from the network indexes, cover a distance of 30min using the time column and update all edges
6. from the edges values, create the polygon

0 – select an irisi to start

In [15]:
```
iris = irisMEL[irisMEL.IRIS == '0804'].copy(deep=True)
```

In [16]:
```
ax = irisMEL.plot(linewidth=0.1)
iris.plot(ax=ax, color='red', linewidth=0.1)
```

Out[16]: `<AxesSubplot:>`



In [17]:
```
iris
```

Out[17]:

	INSEE_COM	NOM_COM	IRIS	CODE_IRIS	NOM_IRIS	TYP_IRIS	geometry	index_righ
89	59350	Lille	0804	593500804	Aviateurs	H	POLYGON ((1701386.430 9269601.401, 1701376.845...	

1 – get the iris boundaries to get starting points + 2 – simplify

In [18]:
```
x = iris.iat[0, 6]
print('x original coordinates length', len(x.exterior.coords))
print(x.area)
s = x.simplify(50, preserve_topology=True)
print('s simplified coordinates length', len(s.exterior.coords))
print(s.area)
```

```
x original coordinates length 122
280859.8301837968
s simplified coordinates length 9
256032.98973009738
```

In [19]:
```
iris_df = pd.DataFrame.from_records(s.exterior.coords, columns=['longitude','latitude'])
iris_df = gpd.GeoDataFrame(iris_df, geometry=gpd.points_from_xy(iris_df.longitude, iris_df.latitude), crs=3950)
```

In [20]:
```
iris_df.head()
```

Out[20]:

	longitude	latitude	geometry
0	1.701386e+06	9.269601e+06	POINT (1701386.430 9269601.401)
1	1.701492e+06	9.269970e+06	POINT (1701492.443 9269970.072)
2	1.701689e+06	9.270079e+06	POINT (1701688.605 9270079.487)
3	1.701747e+06	9.269989e+06	POINT (1701746.707 9269989.143)
4	1.702175e+06	9.270281e+06	POINT (1702174.967 9270281.251)

3 - find the index on the network of each coordinates

we start with cleaning the network and removign empty coordinates if any

In [21]:
```
dfe_bike = dfe[dfe['bicycle']>0].copy(deep=True)
dfe_bike = dfe_bike.dropna()
dfe_bike = dfe_bike[~(dfe_bike.is_empty | dfe_bike.geometry.isna())]
```

- prepare B an array with the list of coordinates from dfe_bike
- prepare B_ix an array of the index where to find each coordinate in the list of edges

Note len(B) = len(B_ix)

In [22]:
```
B = [np.array(geom.coords) for geom in dfe_bike.geometry.to_list()]
B_ix = tuple(itertools.chain.from_iterable([itertools.repeat(i, x) for i, x in enumerate(list(map(len, B)))]))
B = np.concatenate(B)
ckd_tree = cKDTree(B)
```

In [23]:
```
def ckdnearest(gdfA, gdfB, gdfB_cols=['g']):
    A = np.concatenate([np.array(geom.coords) for geom in gdfA.geometry.to_list()])
    dist, idx = ckd_tree.query(A, k=1)
    idx = itemgetter(*idx)(B_ix)
    return dfe_bike.iloc[idx].name
```

for each position of iris_df we find the nearest edge, get its edge_id value (to get the value of an index one use name instead of value)

In [88]:
```
edge_ids_steps = []
for row in iris_df.itertuples():
    name = ckdnearest(gpd.GeoDataFrame([list(row)], columns=row._fields), dfe_bike)
    edge_ids_steps.append(name)

iris_df['nearest'] = edge_ids_steps
iris_df.head()
```

Out[88]:

	longitude	latitude	geometry	nearest
0	1.701386e+06	9.269601e+06	POINT (1701386.430 9269601.401)	973677
1	1.701492e+06	9.269970e+06	POINT (1701492.443 9269970.072)	767195
2	1.701689e+06	9.270079e+06	POINT (1701688.605 9270079.487)	1036215
3	1.701747e+06	9.269989e+06	POINT (1701746.707 9269989.143)	979283
4	1.702175e+06	9.270281e+06	POINT (1702174.967 9270281.251)	316004

now we have the coordinate on the graph of the nearest node close to the boundary points

4 – 30 min distance

first we prepare the graph on graph tool (network dataframe -> networlk graph-tool)

Graph tool will handle all the network search

In [89]:
```
dfe_bike.head(2)
```

Out[89]:

	u	v	path	osm_id	highway	level	lanes	width	b
edge_id									
970003	4028550600	4028550592	[4028550600, 6532006256, 2091208107, 209120814...	27799236	primary	6	1	3.0	
122	133263733	2562615356	[133263733, 2562615356]	14037709	residential	3	1	3.0	

- g is a graph representation independant of the dataframe, we need to build it
- time is an edge property we will fill
- edges_id is also an edge property to link the edge_id inside oour dataframe with the edge id in the graph (different)
- edgelist get the list of edges with the chosen properties from the dataframe (it is an array of len(df) rows and len(properties) columns)
- nodes_id is the list of nodes corresponding to each edge added to the graph

In [90]:
```
g = gt.Graph(directed=True)
time = g.new_edge_property('float')
edges_id = g.new_edge_property('int')

edgelist = dfe_bike.reset_index()[['u','v', 'time', 'edge_id']].values
nodes_id = g.add_edge_list(edgelist, hashed=True, eprops=[time, edges_id])
```

In [102…
```
def get_reachable_egdes(source, threshold):
    """
    :params source is edge of the graph
    :params threshold is in second

    :return a polygon
    """

    # this visitor class will stop the search once the time threshold will be reached
    class VisitorIsochrone(gt.search.DijkstraVisitor):
        def __init__(self, dist, thresh):
            self.dist = dist
            self.thresh = thresh

        def examine_vertex(self, u):
            if self.dist[u] > self.thresh:
                raise gt.search.StopSearch()

    # dist is a graph property to "save" the time property we use to represent distance
    dist = g.new_vertex_property("double")
    visitor = VisitorIsochrone(dist, threshold)

    # we have the vertex directly
    # dist is a vertex property map with the computed distances from the source.
    # pred is a vertex property map with the predecessor tree.
    dist, pred =  gt.search.dijkstra_search(g, time, source=source, visitor=visitor, dist_map=dist, infinity=np.inf)

    # get the reachable edges from source with the established threshold
    # if the vertex can be reached, we get the predecessor vertex, get the edge
    # and add the edge id with both distances to reachable array (edge_id(u,v), dist_u, dist_v)
    reachable = []
    for i,value in enumerate(dist.a):
        if not np.isinf(value):
            e = g.edge(s=pred[i], t=i)
            if e is not None:
                reachable.append((edges_id[e],dist[pred[i]], dist[i]))

    # reachable contains all the reachable edges indexed with the edge id in the dataframe
    # it also contains the time to get to   u (t1) ----> v (t2)
    #                                          edge_id


    return reachable
```

- threshold is the cutoff value in seconds
- from all the nearest edges we found before, we get the id of the source vertex in the graph (different from the dataframe one, yes it is complicated) and we call the function above to list all reachable nodes

In [97]:
```
threshold = 30 * 60
reachables = []
for nearest_edge in iris_df.nearest.to_list():
    vertex = gt.util.find_edge(g, edges_id, nearest_edge)[0].source()
    reachables += get_reachable_egdes(vertex,threshold)
```

we create a clean copy of the bike dataframe because we want to cut this one

In [98]:
```
bike_30 = dfe_bike.copy(deep=True)
```

we use the reachable nodes of the dataframe to set the time needed to reach them

In [103…
```
bike_30['isochrone_u']=np.inf
bike_30['isochrone_v']=np.inf
for i,t1,t2 in reachables:
    if bike_30.at[i,'isochrone_u'] > t1 and bike_30.at[i,'isochrone_v'] > t2:
        bike_30.at[i,'isochrone_u'] = t1
        bike_30.at[i,'isochrone_v'] = t2
```

we remove all we did not reached

In [104…
```
bike_30_cut  = bike_30[bike_30.isochrone_u != np.inf]
```

plot the isochrone graph, the legend is in seconds

- 600 -> 10min
- 1200 -> 20 min
- 1800 -> 30 min

In [109…
```
ax = bike_30_cut.plot(column='isochrone_u',
           categorical=False,
           cmap='viridis_r',#sns.cubehelix_palette(start=.4, rot=-.5, as_cmap=True),
           linewidth=0.5, figsize=(20,15), legend=True)
iris.plot(ax=ax, color = 'red')
```

Out[109…
```
<AxesSubplot:>
```

create the polygon

to create the polygon, for each reachable edge, we load the corresponding linestring (the actual geographic feature). the listring is cut into segment. for each segment make a circle around the source to represent the area reachable by a human walking (0.8 mps), same with the destination and then make a polygon to envelop them both. Then all the segment polygons are merge together into a kind of asymmetric sausage and finally all the linestring are merged into a final polygon.

tomake different isochrone, you only need to select in bike_30_cut the node with a isochrone value lesser than th ethreshold you decide

In [124…

```
isochrone_polygon = []
speed = 0.8

for i,r in bike_30_cut.iterrows():
    time_left_u = threshold - r['isochrone_u']
    time_left_v = threshold - r['isochrone_v']
    line_length = r['geometry'].length

    if time_left_v < 0:
        stop = time_left_u * line_length / (time_left_u - time_left_v)
        line = cut_linestring(r['geometry'],stop)[0]
        time_left_v = 0
        line_length = line.length
    else:
        line = r['geometry']

    isochrone_polygon.append(asymmetric_line_buffer(line, time_left_u * speed, time_left_v * speed))

isochrone_polygon = unary_union(isochrone_polygon)
```

In [128…

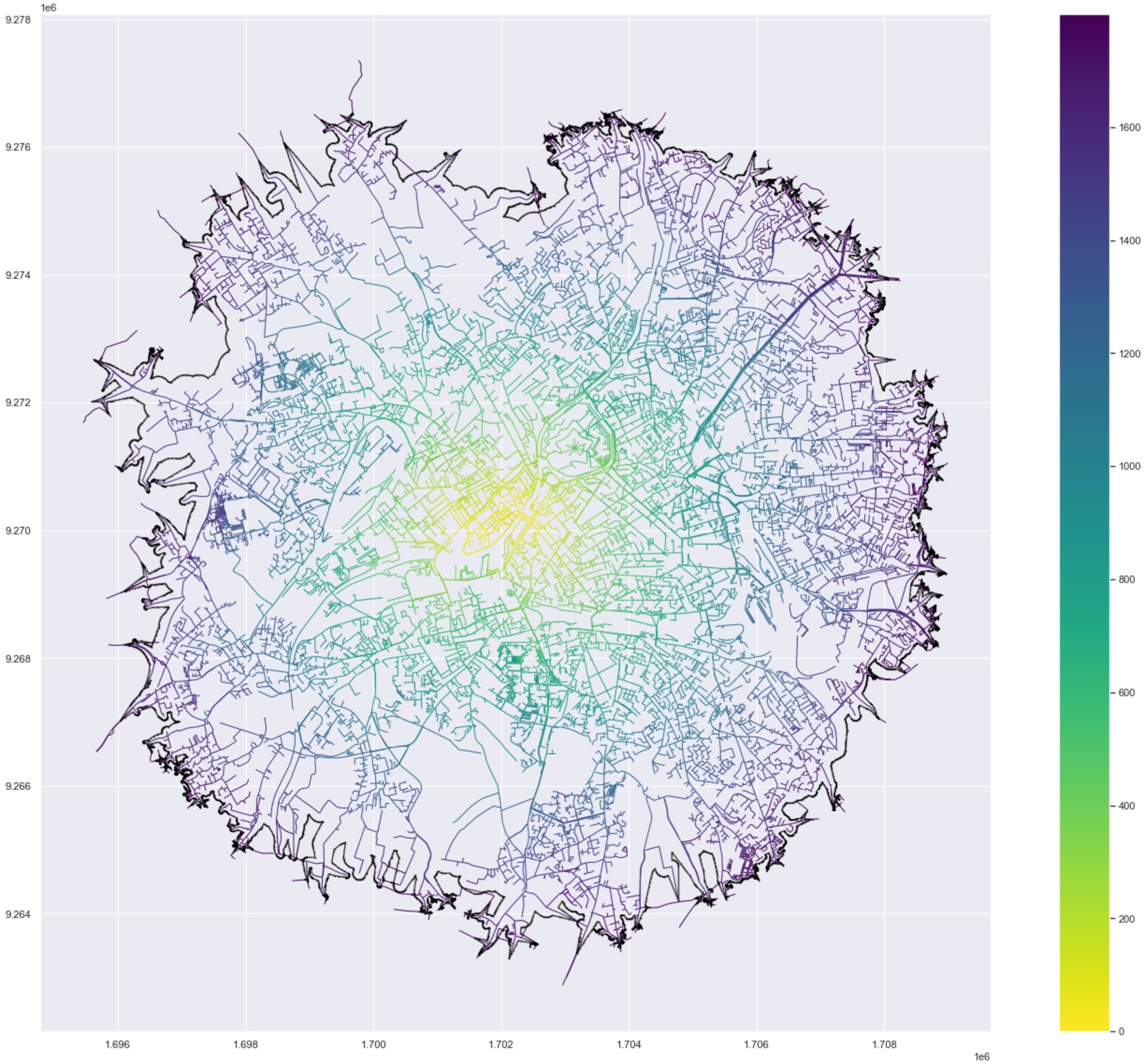
```
d = {'name': ['isochrone']}
df = pd.DataFrame(d)
gs = gpd.GeoSeries.from_wkt([isochrone_polygon.to_wkt()])
isochrone_polygon_gdf = gpd.GeoDataFrame(df, geometry=gs, crs="EPSG:3950")
isochrone_polygon_gdf.to_file('/Users/fabien/Dropbox/low-carbon-lille/cartographic-resources/iris-example-isochrone.shp')
```

In [129…

```

fig, ax = plt.subplots(figsize=(26, 20))
bike_30_cut.plot(ax=ax,linewidth=1, column='isochrone_u', cmap='viridis_r', figsize=(15,15), legend=True)
ax.plot(*isochrone_polygon.exterior.xy, marker='o',color='black',alpha=0.7,markersize=0.5)
```

Out[129… [<matplotlib.lines.Line2D at 0x16bfb2b50>]



In [ ]: