Table of Contents

In [1]:

```
1 VERSION = '41'
```

executed in 5ms, finished 17:49:16 2022-04-06

1 Prelude

In [2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import math
import numpy as np
import random

import simpy
```

executed in 554ms, finished 17:49:16 2022-04-06

In [3]:
```
def isNearZero(x, ε = 0.00001):
    return abs(x)<ε
```

executed in 3ms, finished 17:49:16 2022-04-06

In [4]:
```
class SimulationAborted(Exception):
    def __init__(self, cause):
        self.cause = cause
    def getCause(self):
        return self.cause

try:
    raise SimulationAborted("Fatal Error")
except SimulationAborted as e:
    print(e.getCause())
```

executed in 3ms, finished 17:49:16 2022-04-06

```
Fatal Error
```

2 Visualisation [...]

3 Road Network

TOC

A National Road goes straight in North-South direction through a town. It divides the town in two parts, the eastern Old Town and the western New Town.

- `St` for Street going in North-South direction
- `Rd` for Roads going in East-West direction
- `Bd` for Border denoting the limits of the drawing area
- `N`, `S`, `E`, `W`, `M` stands for North, South, East, West and Main
- Streets and Roads have two lanes marked by their direction: `WSt_Nbd` describes the North bound lane of West Street
- Intersections are described by adding `x` and the name of the crossing street or road and an `i` or `o` indicating the point of entering *into* or coming *out of* the crossing, like `WSt_Nbd_xNRd_i`

In [8]:

```
# Global constants indicating main directions

NORTH = "NORTH"
SOUTH = "SOUTH"
EAST = "EAST"
WEST = "WEST"

NORTH_SOUTH = 'NORTH-SOUTH'
EAST_WEST = 'EAST-WEST'

def left(direction):
    if direction is NORTH:
        return WEST
    elif direction is SOUTH:
        return EAST
    elif direction is EAST:
        return NORTH
    elif direction is WEST:
```

```
        return SOUTH
    else:
        raise ValueError("illegal direction: "+direction)

def right(direction):
    return opposite(left(direction))

def cross(direction):
    if direction is NORTH_SOUTH:
        return EAST_WEST
    elif direction is EAST_WEST:
        return NORTH_SOUTH
    else:
        raise ValueError("illegal direction: "+direction)

def opposite(direction):
    if direction is NORTH:
        return SOUTH
    elif direction is SOUTH:
        return NORTH
    elif direction is EAST:
        return WEST
    elif direction is WEST:
        return EAST
    else:
        raise ValueError("illegal direction: "+direction)

def mainDirection(direction):
    if direction is NORTH or direction is SOUTH:
        return NORTH_SOUTH
    elif direction is EAST or direction is WEST:
        return EAST_WEST
    else:
        raise ValueError("illegal direction: "+direction)
```

executed in 4ms, finished 17:49:16 2022-04-06

In [9]:

```
# Global constants indicating relative directions
LEFT = "LEFT"
RIGHT = "RIGHT"

def look(leftOrRight, direction):
    if leftOrRight is LEFT:
        return left(direction)
    elif leftOrRight is RIGHT:
        return right(direction)
    else:
        raise ValueError("unexpected: "+leftOrRight)
```

executed in 2ms, finished 17:49:16 2022-04-06

Testing the above code:

In [10]:

```
x = EAST
while True:
    print(x)
    x = left(x)
    if x is EAST:
        break
```

executed in 2ms, finished 17:49:16 2022-04-06

```
EAST
NORTH
WEST
SOUTH
```

3.1 Roads

In [11]:

```
class Road:
    def __init__(self, network, name, shortName, direction, coord):

        global NORTH_SOUTH, EAST_WEST

```

```
        self.network = network
        self.name = name
        self.shortName = shortName
        self.segments = []
        if direction == NORTH_SOUTH:
            self.direction = NORTH_SOUTH
            self.x = int(round(coord*network.width, 0))
            self.y = None
            self.length = network.height
        elif direction == EAST_WEST:
            self.direction = EAST_WEST
            self.x = None
            self.y = int(round(coord*network.height, 0))
            self.length = network.width
        else:
            raise ValueError('illegal Road direction: '+direction)
        network.roads.append(self)

    def __str__(self):
        return self.name

    def getName(self):
        return self.name

    def getShortName(self):
        return self.shortName

    def getX(self):
        if self.x is not None:
            return self.x
        else:
            raise ValueError('illegal coordinate for: ', self.direction)

    def getY(self):
        if self.y is not None:
            return self.y
        else:
            raise ValueError('illegal coordinate for: ', self.direction)
```

```
    def addSegment(self, segment):
        self.segments.append(segment)

    def getRoadSegment(self, x, y):
        for s in self.segments:
            if s.contains(x, y):
                return s
        return None

    def getLength(self):
        return self.length

    def getIntersection(self, r):
        return self.network.getIntersection(self, r)

    def getCrossRoads(self):
        return self.network.getRoads(cross(self.direction))

    def getIntersections(self):
        return [ self.getIntersection(r)
                    for r in self.getCrossRoads() ]

    def getIntersectionPoints(self):
        intersections = self.getIntersections()
        if self.direction is NORTH_SOUTH:
            return [ i.y for i in intersections]
        else:  # EAST_WEST
            return [ i.x for i in intersections]

    def freeDist(self, direction, r):
        intersection = self.getIntersection(r)
        queueLength = intersection.getQueueLength(direction)
        s = intersection.stop(direction, queueLength)
        if self.direction is NORTH_SOUTH:
            y = s[1] if direction==NORTH else self.length-s[1]
            return y
        else:
            x = s[0] if direction==EAST else self.length-s[0]
            return x

```

```
    def freeDistance(self, direction):
        crossRoads = self.getCrossRoads()
        distances = [ self.freeDist(direction, r)
                        for r in crossRoads ]
        return min(distances)

    def getVehicles(self, direction):
        acc = []
        for v in self.network.rec.vehicles:
            if v.road==self and v.direction==direction:
                acc.append(v)
        return acc
```

executed in 7ms, finished 17:49:16 2022-04-06

3.2 Traffic Lights

In [12]:

```
RED = "RED"
YELLOW = "YELLOW"
GREEN = "GREEN"


class TrafficLight:
    def __init__(self, intersection, NS=[6,2] , EW=[3,2]):
        self.name = intersection.name
        self.NS_green = NS[0]  # timinig of green in NS direction
        self.NS_yellow = NS[1] # timinig of yellow in NS direction
        self.EW_green = EW[0]
        self.EW_yellow = EW[1]
        self.intersection = intersection
        self.intersection.setTrafficLight(self)
        self.rec = intersection.network.rec
        if self.rec is not None:
            self.rec.env.process(self.process())

    def getName(self):
        return self.name
```

```
21
22      def process(self):
23          while True:
24
25              self.NS, self.EW = RED, GREEN
26              self.rec.recordTrafficLight(self, NORTH_SOUTH, RED)
27              self.rec.recordTrafficLight(self, EAST_WEST, GREEN)
28              yield self.rec.env.timeout(self.EW_green)
29
30              self.NS, self.EW = RED, YELLOW
31              self.rec.recordTrafficLight(self, EAST_WEST, YELLOW)
32              yield self.rec.env.timeout(self.EW_yellow)
33
34              self.NS, self.EW = GREEN, RED
35              self.rec.recordTrafficLight(self, NORTH_SOUTH, GREEN)
36              self.rec.recordTrafficLight(self, EAST_WEST, RED)
37              yield self.rec.env.timeout(self.NS_green)
38
39              self.NS, self.EW = YELLOW, RED
40              self.rec.recordTrafficLight(self, NORTH_SOUTH, YELLOW)
41              yield self.rec.env.timeout(self.NS_yellow)
42
43      def getColor(self, direction):
44          if direction is NORTH_SOUTH:
45              return self.NS
46          elif direction is EAST_WEST:
47              return self.EW
48          else:
49              raise ValueError("illegal direction: "+direction)
```

executed in 4ms, finished 17:49:16 2022-04-06

3.3 Intersections

In [13]:

```
1 class Intersection:
2     def __init__(self, network, rNS, rEW):
3         self.name = rNS.name + ' x ' + rEW.name
```

```
        self.rNS = rNS
        self.rEW = rEW
        self.x = rNS.x
        self.y = rEW.y
        self.network = network
        self.NE = None
        self.NW = None
        self.SE = None
        self.SW = None
        self.queues = { NORTH: None, EAST: None, SOUTH: None, WEST: None }
        self.spots = { NORTH: [], EAST: [], SOUTH: [], WEST: [] }
        self.trafficLight = None
        self.env = None
        if self.network.rec is not None:
            self.env = self.network.rec.env
        self.bold = None
        if self.env is not None:
            self.env.process(self.spin())


    def __str__(self):
        return self.name

    def getId(self):
        return self.id

    def setTrafficLight(self, trafficLight):
        self.trafficLight = trafficLight

    def getTrafficLight(self):
        return self.trafficLight

    def centrePoint(self):
        return (self.x, self.y)

    def crossRoad(self, incomingDirection):
        if incomingDirection is NORTH or \
                incomingDirection is SOUTH:
            return self.rEW
        else:
```

```
            return self.rNS

    def crossRoadIsThroughRoad(self, incomingDirection):
        road = self.crossRoad(incomingDirection)
        for s in road.segments:
            if s.begin==self or s.end==self:
                return False
        return True

    def getQueueLength(self, incomingDirection):
        queue = self.queues[incomingDirection]
        if queue is None:
            env = self.network.rec.env
            queue = simpy.Resource(env, capacity=1)
            self.queues[incomingDirection] = queue
        return len(queue.queue)

    def request(self, incomingDirection):
        env = self.network.rec.env
        queue = self.queues[incomingDirection]
        spots = self.spots[incomingDirection]
        if queue is None:
            queue = simpy.Resource(env, capacity=1)
            self.queues[incomingDirection] = queue
        request = queue.request()
        if len(queue.queue)>=len(spots):
            spots += [ simpy.Resource(env, capacity=1) ]
            self.spots[incomingDirection] = spots

        if len(queue.queue)>=self.network.MAX_QUEUE_LENGTH:
            message = f"at t={env.now:7.3f}s Queue Overflow at "+self.name+ \
                      " ("+incomingDirection+")"
            raise SimulationAborted(message)

        return request

    def release(self, incomingDirection, request):
        queue = self.queues[incomingDirection]
        queue.release(request)
```

```
83
84      def getSpotRequest(self, incomingDirection, pos):
85          env = self.network.rec.env
86          spots = self.spots[incomingDirection]
87          while pos>=len(spots):
88              spots += [ simpy.Resource(env, capacity=1) ]
89              self.spots[incomingDirection] = spots
90          request = spots[pos].request()
91          return request
92
93      def releaseSpot(self, incomingDirection, pos, request):
94          spots = self.spots[incomingDirection]
95          spots[pos].release(request)
96
97      def stop(self, incomingDirection, queueLength):
98          LW = self.network.LW
99          dist = LW+4
100         if incomingDirection is NORTH:
101             return (self.x+LW/2, self.y-(queueLength+1)*dist)
102         elif incomingDirection is SOUTH:
103             return (self.x-LW/2, self.y+(queueLength+1)*dist)
104         elif incomingDirection is EAST:
105             return (self.x-(queueLength+1)*dist, self.y+LW/2)
106         elif incomingDirection is WEST:
107             return (self.x+(queueLength+1)*dist, self.y-LW/2)
108
109     def isDeadLocked(self):
110         all = [NORTH, EAST, SOUTH, WEST]
111         return min([ self.getQueueLength(dir) for dir in all ])>0
112
113     def spin(self):
114         while self.trafficLight is None:
115             self.bold = None
116             yield self.env.timeout(2)
117             self.bold = random.sample([NORTH, EAST, SOUTH, WEST],1)[0]
118             yield self.env.timeout(5)
119
120     def isBold(self, direction):
121         return self.bold is direction
```

executed in 9ms, finished 17:49:16 2022-04-06

3.4 Road Segments

In [14]:

```
class RoadSegment:
    def __init__(self, road, begin, end):
        if road.direction==NORTH_SOUTH:
            self.direction = SOUTH
            self.giveWayNORTH = begin is not None
            self.giveWaySOUTH = end is not None
        elif road.direction==EAST_WEST:
            self.direction = WEST
            self.giveWayEAST = begin is not None
            self.giveWayWEST = end is not None
        self.road = road
        self.begin = road.network.getIntersection(road, begin)
        self.end = road.network.getIntersection(road, end)
        self.name = road.name + "(" + self.direction + ") ["+str(begin)+","+str(end)+"]"
        road.addSegment(self)

    def __str__(self):
        return self.name

    def contains(self, x, y):
        if self.road.direction is NORTH_SOUTH:
            return min(begin.y, end.y) <= y <= max(begin.y, end.y) and \
                    self.road.x-LW <= x <= self.road.x+LW
        elif self.road.direction is EAST_WEST:
            return min(begin.x, end.x) <= x <= max(begin.x, end.x) and \
                    self.road.y-LW <= y <= self.road.y+LW
        else:
            raise ValueError('Illegal direction:', self.road.name)
```

executed in 4ms, finished 17:49:16 2022-04-06

3.5 Road Network

In [15]:

```
class RoadNetwork:
    def __init__(self, name, width, height, rec=None):
        self.rec = rec
        if rec is not None:
            rec.network = self
        self.name = name
        self.width = width
        self.height = height
        self.roads = []
        self.intersections = dict()
        self.background = None

        self.LW = 8  # [m] lane width
        self.SL = 40 # [m] length of keep clear line near crossing
        self.VL = 10 # [m] enlarged length of a vehicle
        self.VW = 4  # [m] enlarged width of a vehicle

        self.MAX_QUEUE_LENGTH = (max(width, height)-50)/12 # for testing only

    def addRoad(self, name, shortName, direction, coord):
        for r in self.roads:
            if name in [r.getName(), r.getShortName()]:
                raise ValueError("Road name re-used: ", name)
            if shortName in [r.getName(), r.getShortName()]:
                raise ValueError("Road name re-used: ", shortName)
        r = Road(self, name, shortName, direction, coord)
        return r

    def getRoads(self, direction):
        return [ r for r in self.roads if r.direction is direction ]

    def getRoad(self, name):
        for r in self.roads:
            if r.name == name  or r.shortName == name:
                return r
```

```
36          return None
37
38      def getIntersection(self, r1, r2):
39          if r1 is None or r2 is None:
40              return None
41          if type(r1) is str:
42              r1 = self.getRoad(r1)
43          if type(r2) is str:
44              r2 = self.getRoad(r2)
45          if r1.network is not self or r2.network is not self:
46              raise ValueError('intersection roads from different networks')
47          rNS, rEW = (r1, r2) if r1.direction is NORTH_SOUTH else (r2, r1)
48          if rNS.direction is not NORTH_SOUTH or \
49                  rEW.direction is not EAST_WEST:
50              raise ValueError('problem with intersection:', rNS.name, rEW.name)
51          name = rNS.name + ' x ' + rEW.name
52          if name not in self.intersections:
53              intersection = Intersection(self, rNS, rEW)
54              self.intersections[name] = intersection
55          return self.intersections[name]
56
57      def getIntersectionByName(self, name):
58          if name in self.intersections:
59              return self.intersections[name]
60          else:
61              return None
```

executed in 6ms, finished 17:49:16 2022-04-06

4 Vehicles

TOC

In [16]:

```
# emergency brake decelleration
#   Tesla:   -8.0 m/s²
#   normal:  -4.0 m/s²
A_BRAKE = -4.0 # [m/s²]

# average decelleration when using engine braking
#   Tesla:   -1.8 m/s²   (regenerative braking)
#   default: -0.6 m/s²
A_COAST = -0.6 # [m/s²]

# max accelleration depending on car class
#   Tesla:   4.6 m/s²
#   default: 2.5 m/s² corresponds to 0-100km/h om 11s
A_MAX =  2.5 # [m/s²]
```

executed in 2ms, finished 17:49:16 2022-04-06

In [17]:

```
class Vehicle:

    def __init__(self, rec, road, direction,
                 t0=0, v=0, a=0, vmax=None,
                 color='red', plan=None):

        # the simulation wide vehicle registry is
        # anchored in the recorder
        self.id = rec.register(self)

        self.a_brake = A_BRAKE
        self.a_coast = A_COAST
        self.a_max = A_MAX

        self.length = rec.network.VL # [m]  Length of the vehicle

        self.nomore_tolerance = 2 # [s]
        self.time_tolerance = 5 # [s]

        ## if not None the preferred max free velocity
        self.vmax = vmax
        self.color = color
```

```
        self.plan = plan

        self.env = rec.env
        self.rec = rec

        self.t0 = t0
        self.road = road

        # self.x0 and self.y0 [m] specify the position of the
        # reference point in the front center of the vehicle
        if direction==SOUTH:
            self.direction = SOUTH
            self.cosφ, self.sinφ = 0, -1
            self.x0 = road.x+rec.network.LW//2
            self.y0 = rec.network.height-1

        elif direction==NORTH:
            self.direction = NORTH
            self.cosφ, self.sinφ = 0, +1
            self.x0 = road.x-rec.network.LW//2
            self.y0 = 0

        elif direction==EAST:
            self.direction = EAST
            self.cosφ, self.sinφ = +1, 0
            self.x0 = 0
            self.y0 = road.y+rec.network.LW//2

        elif direction==WEST:
            self.direction = WEST
            self.cosφ, self.sinφ = -1, 0
            self.x0 = rec.network.width-1
            self.y0 = road.y-rec.network.LW//2

        else:
            raise ValueError("illegal direction", direction)

        self.startPoint = (self.x0, self.y0)
        self.dx0, self.dy0 = v*self.cosφ, v*self.sinφ
```

```
        self.ddx0, self.ddy0 = a*self.cosφ, a*self.sinφ

        self.stopQueueReq = None  # request object for queueing at stop
        self.positionInQueue = None
        self.spotQueueReq = None # request for position in queue

        # trace flags
        self.traceEvents = False
        self.traceCrossing = False
        self.traceAdjustVelocity = False
        self.traceCruising = False
        self.traceInterrupt = False
        self.traceBraking = False

        self.t_target = []
        self.v_target = []

        # start process
        self.aborted = False
        self.running = False

        # Flags used for temporarily exclusive behaviour
        # This flag is used to prevent interrupting
        # braking for short distance moving

        self.braking = False
        self.moving = False
        # exclusive for stopping/crossing at intersection
        self.stopping = False
        self.patience = None

        self.processRef = None
        self.mainProcessRef = None
        self.env.process(self.encapsulatedProcess())

    def __str__(self):
        return f"v{self.id:d}"

    def abort(self, cause=None):
        if not self.aborted:
```

```
        if not self.aborted:
            if cause is not None:
                print(cause)
            self.aborted = True
            self.running = False
            if cause is None:
                if self.mainProcessRef is not None and \
                        self.mainProcessRef.is_alive:
                    self.mainProcessRef.interrupt('Killing')


    def encapsulatedProcess(self):
        self.mainProcessRef = self.env.process(self.process())
        try:
            yield self.mainProcessRef
            self.mainProcessRef = None
        except SimulationAborted as exp:
            self.mainProcessRef = None
            self.abort(exp.getCause())
            self.rec.abort()
        except simpy.Interrupt:
            pass

    def trace(self, message):
        print(f"t={self.t0:5,.1f}s "
              f"x={self.x0:5,.1f}m y={self.y0:5,.1f}m "
              f"v={self.v():4.1f}m/s v{self.id:02d} "
              f"on {self.road.shortName:s}[{self.direction[0]:s}]",
              message)

    def Δs(self, P=None):
        if P is None:
            P = self.startPoint
        return math.sqrt((self.x0-P[0])**2+(self.y0-P[1])**2)

    def setV(self, v):
        self.dx0, self.dy0 = v*self.cosφ, v*self.sinφ

    def v(self):
        return math.sqrt(self.dx0**2+self.dy0**2)
```

```
141
142     def Δv(self, other):
143         return math.sqrt((self.dx0-other.dx0)**2+(self.dy0-other.dy0)**2)
144
145     def setA(self, a):
146         self.ddx0, self.ddy0 = a*self.cosφ, a*self.sinφ
147
148     def a(self):
149         return self.ddx0/self.cosφ if self.cosφ!=0 else self.ddy0/self.sinφ
150
151     # compute distance to the car in front
152     # i.e. distance between the front bumpers of both
153     # vehicles minus the car length of the car in the front
154     def dist(self, v):
155         if v is None:
156             return math.inf
157         else:
158             return self.Δs((v.x0, v.y0)) - v.length
159
160     # returns the vehicle that is on the same road in the same
161     # direction directly in front
162     def vehicleInFront(self):
163         other = None
164         for v in self.rec.vehicles:
165             if v is not self and v.road == self.road and \
166                     v.direction == self.direction:
167                 if self.direction is NORTH and v.y0>self.y0:
168                     if other is None or v.y0<other.y0: other = v
169                 elif self.direction is SOUTH and v.y0<self.y0:
170                     if other is None or v.y0>other.y0: other = v
171                 elif self.direction is EAST and v.x0>self.x0:
172                     if other is None or v.x0<other.x0: other = v
173                 elif self.direction is WEST and v.x0<self.x0:
174                     if other is None or v.x0>other.x0: other = v
175         return other
176
177     # updates (vectorised) position and speed
178     def update(self):
179         t = self.env.now
180         if t < self.t0 or not self.running:
```

```
180         if t < self.t0 or not self.running:
181             return False
182 
183         if t > self.t0:
184             Δt = t - self.t0
185             Δdx = self.ddx0*Δt
186             Δdy = self.ddy0*Δt
187             Δx = self.dx0*Δt + self.ddx0*Δt*Δt/2
188             Δy = self.dy0*Δt + self.ddy0*Δt*Δt/2
189             self.t0 = t
190             self.x0, self.y0 = self.x0+Δx, self.y0+Δy
191             self.dx0, self.dy0 = self.dx0+Δdx, self.dy0+Δdy
192 
193             # stop when leaving the area of the current network
194             if self.x0<0 or self.x0>self.rec.network.width or \
195                 self.y0<0 or self.y0>self.rec.network.height:
196                 self.running = False
197 
198         return True
199 
200     # frequent periodic status check controlled by Recorder
201     # triggers emergency action in the vehicle process
202     def checkStatus(self):
203         inFront = self.vehicleInFront()
204         # if there is a vehicle in front which drives slower and
205         # the distance to that vehicle in front is at current speed
206         # less than the critical time tolerance of this driver
207         if inFront is not None and \
208                 not self.braking and \
209                 not self.moving and \
210                 not self.stopping and \
211                 inFront.v() < self.v() and \
212                 self.dist(inFront)-self.length < self.time_tolerance*self.Δv(inFront):
213             # action is required
214             # note that Δv is positive and self.a_coast is negative
215             Δv = self.Δv(inFront)
216             Δs = self.dist(inFront)
217             if -Δv**2/self.a_coast < Δs/2:
218                 Δt = -self.Δv(inFront)/self.a_coast
219             else:
```

```
220                 Δt = -self.Δv(inFront)/self.a_brake
221             self.setTarget(Δt, inFront.v())
222 
223     # allows setting of control parameters
224     # from an independent process
225     def setTarget(self, t, v):
226         self.t_target = [ t ] + self.t_target
227         self.v_target = [ v ] + self.v_target
228         self.interruptProcess()
229 
230     # defines the life cycle of a vehicle
231     def process(self):
232 
233         # delay start to the given time t
234         if self.t0>self.env.now:
235             yield self.env.timeout(self.t0-self.env.now)
236 
237         while self.road.freeDistance(self.direction)<50:
238             yield self.env.timeout(5)
239 
240         self.t0 = self.env.now
241         if self.aborted:
242             return
243         self.running = True
244         self.rec.startRecording(self)
245 
246         while self.update():
247 
248             inFront = self.vehicleInFront()
249 
250             # if the car in front is slower and we are a bit too near on its heals...
251             if inFront is not None and \
252                     not self.braking and not self.moving and \
253                     inFront.v() < self.v() and \
254                     self.dist(inFront)-self.length < \
255                         self.nomore_tolerance*self.Δv(inFront):
256                 # inFront.trace(f"being followed v={inFront.v():4.1f}m/s a={inFront.a():1.2f}m/s² by
257                 yield from self.emergencyBraking(inFront.v())
258                 if not isNearZero(self.v()-inFront.v()):
```

```
                    # after emergency breaking adjust to the speed of the car in front...
                    Δt = 1
                    self.setTarget(Δt, inFront.v())
                continue

            elif len(self.t_target)>0:
                # normally len(self.t_target)<=1
                Δt = self.t_target[0]
                Δv = self.v_target[0]-self.v()
                self.t_target = self.t_target[1:]
                self.v_target = self.v_target[1:]
                if isNearZero(Δv):
                    yield from self.continueAtSameSpeed(Δt)
                else:
                    yield from self.adjustVelocity(Δv, Δt)


            if self.plan is not None and len(self.plan)>0:

                command = self.plan[0]

                # split action and position from command
                split = command.find('@')
                if 0 < split < len(command)-1:
                    action = command[:split]
                    position = command[split+1:]
                else:
                    action = command
                    position = ""

                # split numeric parameters from action
                split1 = action.find('(')
                split2 = action.find(')')
                if 0 < split1 < split2:
                    params = action[split1+1:split2].split(',')
                    pars = [ float(p) for p in params ]
                    action = action[:split1]
                else:
                    pars = []
```

```
299
300                 if action == "Acc":
301                     ## test action Acc(a,t)
302                     self.setA(pars[0])
303                     self.update()
304                     yield self.env.timeout(pars[1])
305                     # action completed
306                     self.plan = self.plan[1:]
307                     continue
308
309                 if action == "Move":
310                     ## test action Move(s)
311                     yield from self.move(pars[0])
312                     # action completed
313                     self.plan = self.plan[1:]
314                     continue
315
316                 if action == "Wait":
317                     ## test action Acc(a,t)
318                     Δt = pars[0]
319                     self.update()
320                     # the car should be stationary
321                     self.setA(0)
322                     self.setV(0)
323                     yield self.env.timeout(Δt)
324                     self.update()
325                     # action completed
326                     self.plan = self.plan[1:]
327                     continue
328
329                 if action == "Exit":
330                     ## take vehicle out
331                     self.running = False
332                     # action completed
333                     self.plan = self.plan[1:]
334                     continue
335
336                 if action == "Stop":
337                     if len(pars)>0:
```

```
338                     ## test action Stop(s)
339                     Δs = pars[0]
340                     yield from self.stop(Δs)
341                     # action completed
342                     self.plan = self.plan[1:]
343                     continue
344 
345                 if self.traceCrossing:
346                     self.trace("action Stop")
347                 intersection = self.road.getIntersection(position)
348                 if intersection is None:
349                     raise ValueError(f"unknown position {position:s}")
350 
351                 self.stopping = True
352                 self.stopQueueReq = intersection.request(self.direction)
353                 inFront = self.vehicleInFront()
354                 if inFront is None or inFront.positionInQueue is None or \
355                     self.dist(inFront)>self.Δs(intersection.centrePoint()):
356                         self.positionInQueue = intersection.getQueueLength(self.direction)
357                 else:
358                     self.positionInQueue = inFront.positionInQueue+1
359                 self.rec.record(self, "queue")
360                 self.spotQueueReq = intersection.getSpotRequest(self.direction, self.positionIn(
361                 stopPoint = intersection.stop(self.direction, self.positionInQueue)
362                 distance = self.Δs(stopPoint)
363                 if self.traceCrossing:
364                     self.trace(f"stopping at position {self.positionInQueue:d}")
365                 yield from self.stop(distance)
366                 if self.traceCrossing:
367                     self.trace(f"stopped at position {self.positionInQueue:d}")
368 
369                 # stopped at the end of the queue...
370                 yield self.spotQueueReq
371 
372                 while self.positionInQueue>0:
373 
374                     if self.traceCrossing:
375                         self.trace(f"waiting for spot at position {self.positionInQueue-1:d}")
376                     nextReq = intersection.getSpotRequest(self.direction, self.positionInQueue-1
377                     yield nextReq
```

```
                    yield nextReq
                    if self.traceCrossing:
                        self.trace(f"moving up to position {self.positionInQueue-1:d}")
                    stopPoint = intersection.stop(self.direction, self.positionInQueue-1)
                    yield from self.move(self.Δs(stopPoint))
                    if self.traceCrossing:
                        self.trace(f"moved up to position {self.positionInQueue-1:d}")
                    intersection.releaseSpot(self.direction, self.positionInQueue, self.spotQueu
                    self.spotQueueReq = nextReq
                    self.positionInQueue -= 1

                # vehicle is now at top of the queue
                if self.traceCrossing:
                    self.trace("action Stop finished")
                self.stopping = False
                # action completed
                self.plan = self.plan[1:]
                continue

            if action == 'X':  # cross the intersection
                if self.traceCrossing:
                    self.trace("action X")
                intersection = self.road.getIntersection(position)
                if intersection is None:
                    raise ValueError(f"unknown position {position:s}")
                trafficLight = intersection.getTrafficLight()
                crossRoad = intersection.crossRoad(self.direction)
                crossingDist = 2*self.rec.network.LW+self.length
                crossingTime = self.timeRequired(crossingDist)

                if intersection.crossRoadIsThroughRoad(self.direction):
                    if self.traceCrossing:
                        self.trace("checking to cross")
                    Δt = min(self.nextCrossTraffic(intersection, self.direction, RIGHT),
                             self.nextCrossTraffic(intersection, self.direction, LEFT))
                    if Δt>crossingTime+2:
                        # Enough time for crossing
                        if self.traceCrossing:
                            self.trace("crossing")
                        #### This is a deliberate coding error, should be. yield from...
```

```
                    yield from self.accelerateAndCruise(crossingTime)
                    self.rec.record(self, "dequeue")
                    intersection.releaseSpot(self.direction, 0, self.spotQueueReq)
                    intersection.release(self.direction, self.stopQueueReq)
                    if self.traceCrossing:
                        self.trace("action X finished")
                    self.stopQueueReq = None

                    # action completed
                    self.plan = self.plan[1:]
                    continue

                else:
                    # wait for cross traffic to pass and try again...
                    yield self.env.timeout(Δt+0.5)
                    continue

            elif trafficLight is None:
                # equal crossing
                bold = intersection.isDeadLocked() and intersection.isBold(self.direction)
                Δt = self.nextCrossTraffic(intersection, self.direction, RIGHT)
                if bold or math.isinf(Δt) or Δt>2*crossingTime+2:
                    # plenty of time for crossing
                    if bold:
                        self.trace("breaking deadlock")
                    yield from self.accelerateAndCruise(crossingTime)
                    self.rec.record(self, "dequeue")
                    intersection.releaseSpot(self.direction, 0, self.spotQueueReq)
                    intersection.release(self.direction, self.stopQueueReq)
                    self.stopQueueReq = None
                    self.plan = self.plan[1:]
                    continue
                else:
                    yield self.env.timeout(min(5, Δt+0.5))
                    continue

            else: # TrafficLight
                while trafficLight.getColor(mainDirection(self.direction))!=GREEN:
                    yield self.env.timeout(self.rec.timeStep)
                yield from self.accelerateAndCruise(crossingTime)
```

```
456                         yield from self.accelerateAndCruise(crossingTime)
457                         self.rec.record(self, "dequeue")
458                         intersection.releaseSpot(self.direction, 0, self.spotQueueReq)
459                         intersection.release(self.direction, self.stopQueueReq)
460                         self.stopQueueReq = None
461                         self.plan = self.plan[1:]
462                         continue
463 
464             # cruise along with potentially slightly modified speed
465             elif self.vmax is not None:
466                 # as long there is no vehicle in front or the vehicle in front
467                 # is far enough ahead adjust to random speed around vmax
468                 if inFront is None or \
469                     self.time_tolerance*self.Δv(inFront) < self.dist(inFront):
470                     yield from self.adjustVelocity(self.vmax-self.v(), 5)
471                 else:
472                     self.setA(0)
473                     yield self.env.timeout(self.rec.timeStep)
474             else:
475                 self.setA(0)
476                 yield self.env.timeout(self.rec.timeStep)
477 
478 
479         self.rec.stopRecording(self)
480 
481     # check if the vehicle is approaching the intersection
482     # from the given direction or if it hasn't yet crossed
483     # the intersection completely
484     def approaching(self, intersection, direction):
485         LW = self.rec.network.LW
486         if direction is SOUTH:
487             return self.y0+self.length>intersection.y-LW
488         elif direction is NORTH:
489             return self.y0-self.length<intersection.y+LW
490         elif direction is EAST:
491             return self.x0+self.length<intersection.x+LW
492         elif direction is WEST:
493             return self.x0-self.length>intersection.x-LW
494 
495     # time required to cross a distance under max acceleration
```

```
    # while not exceeding vmax
    def timeRequired(self, dist):
        v0 = self.v()
        accTime = (self.vmax-v0) / self.a_max
        accDist = v0*accTime + accTime**2*self.a_max/2
        if accDist>dist:
            # we never reach vmax over the distance
            return math.sqrt(2*dist/self.a_max)
        else:
            # we accelerate to vmax and then continue
            # cruising along with vmax.
            return accTime+(dist-accDist)/self.vmax

    # estimates the time that approaching vehicle takes to
    # cross the intersection completely
    def crossingTime(self, intersection, direction):
        LW = self.rec.network.LW
        if direction is SOUTH:
            dist = self.y0+self.length-(intersection.y-LW)
        elif direction is NORTH:
            dist = (intersection.y+LW)-(self.y0-self.length)
        elif direction is EAST:
            dist = (intersection.x+LW)-(self.x0+self.length)
        elif direction is WEST:
            dist = self.x0-self.length-(intersection.x-LW)
        return self.timeRequired(dist)

    # a vehicle from the incoming direction standing at
    # an intersection and looking towards left or right,
    # estimating the time until the next vehicle coming
    # might cross its way
    def nextCrossTraffic(self, intersection,
                         incomingDirection, leftOrRight):
        # self.trace("Waiting for Traffic from "+leftOrRight)
        crossroad = intersection.crossRoad(incomingDirection)
        crossDir = opposite(look(leftOrRight, incomingDirection))
        critTime = math.inf
        for v in self.rec.vehicles:
            if v.road==crossroad and \
```

```
                        v.direction==crossDir and \
                        v.approaching(intersection, crossDir):
                    time = v.crossingTime(intersection, crossDir)
                    if time<critTime:
                        critTime = time
        return critTime

    # decelerate as fast as possible to v
    def emergencyBraking(self, v):
        if self.traceBraking:
            self.trace(f"Braking from v={self.v():4.1f}m/s to {v:4.1f}m/s")
        self.rec.record(self, 'brake')
        self.setA(self.a_brake)
        v = max(0, min(v, self.v()-2))
        Δv = v-self.v()
        Δt = max(0.5, Δv/self.a())
        self.setA(Δv/Δt)
        yield self.env.timeout(Δt)

        self.update()
        self.setA(0)
        self.rec.record(self, 'brake end')
        if self.traceBraking:
            self.trace(f"Braking end  v={self.v():4.1f}m/s")

    def stop(self, Δs):
        self.update()
        D = -2
        A = 4
        v0 = self.v()
        sd, td = v0**2/(-2*D), -v0/D
        # solve quadratic equation
        a, b, c = A/2-A**2/(2*D), v0*(1-A/D), Δs-sd
        if c<0:
            A = -0.5*v0**2/Δs
            Δt = 2*Δs/v0
            self.setA(A)
            yield self.env.timeout(Δt)
            self.update()
        else:
```

```
            else:
                r = math.sqrt(b**2+4*a*c)
                x1 = (-b+r)/(2*a)
                t1, t2 = x1, -A/D*x1
                # phase 1: accelerate
                self.setA(A)
                yield self.env.timeout(t1)
                self.update()
                # phase 2: decelerate
                self.setA(D)
                yield self.env.timeout(t2+td)
                self.update()
            self.setV(0)
            self.setA(0)

    # move a short distance in a queue using only marginal
    # acceleration and deceleration
    def move(self, Δs):
        a = 2 # [m/s²]
        self.setA(a)
        Δt = math.sqrt(Δs/a)
        yield self.env.timeout(Δt)
        self.update()
        self.setA(-a)
        yield self.env.timeout(Δt)
        self.update()
        self.setV(0)
        self.setA(0)

    # change velocity by Δv over the period Δt
    def adjustVelocity(self, Δv, Δt):
        self.update()
        if self.traceAdjustVelocity:
            self.trace(f"Adjusting Velocity by Δv={Δv:4,.1f}m/s over {Δt:4,.1f}s")
        self.setA(Δv/Δt)
        yield self.env.timeout(Δt)
        self.update()
        self.setA(0)
        if self.traceAdjustVelocity:
            self.trace(f"Adjusted Velocity")
```

```
614
615     def continueAtSameSpeed(self, Δt):
616         self.update()
617         # don't change the current velocity
618         self.setA(0)
619         if self.traceCruising:
620             self.trace(f"Cruising for {Δt:4,.1f}s")
621         yield self.env.timeout(Δt)
622         self.update()
623         if self.traceCruising:
624             self.trace(f"End Cruising")
625
626     def accelerateAndCruise(self, crossingTime):
627         crossV = crossingTime*self.a_max
628         if  crossV > self.vmax:
629             accT = (self.v_max-self.v())/self.a_max
630             yield from self.adjustVelocity(self.vmax,  accT)
631             yield from self.continueAtSameSpeed(crossingTime-accT)
632         else:
633             yield from self.adjustVelocity(crossV, crossingTime)
634
635     # interrupting a sub process
636     def interruptProcess(self):
637         #print("interrupting...")
638         #traceback.print_stack(limit=5)
639         if self.processRef is not None and self.processRef.is_alive:
640             self.processRef.interrupt('There are more important things to do...')
641
642
```

executed in 40ms, finished 17:49:17 2022-04-06

5 Recorder

TOC

In [18]:

```
1 class Recorder:
```

In [18]:

```
class Recorder:

    def __init__(self, startTime=0, stopTime=0, timeStep=1):

        self.env = simpy.Environment()
        self.network = None

        self.startTime = startTime
        self.stopTime = stopTime
        self.timeStep = timeStep

        # list of all currently running vehicles
        self.vehicles = []

        # list of all known vehicles (including those
        # that haven't yet started and thosse that have already stopped
        self.allVehicles = []

        self.running = True

        cols=['t', 'x', 'y', 's', 'v', 'a',
              'rd', 'dir', 'id', 'col', 'event', 'ql']
        self.data = pd.DataFrame(columns=cols)

    def register(self, vehicle):
        self.allVehicles.append(vehicle)
        return len(self.allVehicles)

    # runs the simulation
    def run(self):
        self.env.process(self.process())
        self.env.run(self.stopTime+self.timeStep)

    def abort(self):
        if self.running:
            print("Aborting Simulation")
            self.running = False
            for v in self.allVehicles:
                v.abort()

```

```
    def process(self):
        yield self.env.timeout(self.startTime-self.env.now)
        while self.env.now <= self.stopTime:
            if self.running:
                self.record()
            yield self.env.timeout(self.timeStep)

    def startRecording(self, v):
        self.vehicles.append(v)
        self.record(vehicle=v, event='start')

    def stopRecording(self, v):
        self.record(vehicle=v, event='end')
        self.vehicles.remove(v)

    def record(self, vehicle=None, event='timer'):
        if vehicle is not None:
            v = vehicle
            if v.update() or event!='timer':
                ix = len(self.data)
                ql = v.positionInQueue if event=='queue' else 0
                self.data.loc[ix]=[self.env.now, \
                                   round(v.x0,2), round(v.y0,2), \
                                   round(v.Δs(v.startPoint),2), \
                                   round(v.v(),2), round(v.a(),2), \
                                   v.road.name, v.direction, \
                                   v.id, v.color, event, ql]
                if event=='timer':
                    v.checkStatus()
        else:
            for vehicle in self.vehicles:
                self.record(vehicle, event)

    def recordTrafficLight(self, trafficLight, direction, color):
        event = 'trafficlight'
        intersection = trafficLight.intersection
        ix = len(self.data)
        self.data.loc[ix]=[self.env.now, \
                           intersection.x, intersection.y, \
```

```
                            0, 0, 0, \
                            trafficLight.name, direction, \
                            0, color, event, 0]

    def getTrafficLightData(self, name, direction):
        tf = self.data[self.data.event=='trafficlight']
        tf = tf[tf.rd==name]
        tf = tf[tf.dir==direction]
        dropcols = ['s','v','a','rd','dir','id','event','ql']
        return tf.copy(deep=True).drop(columns=dropcols)

    def getData(self):
        return self.data.copy(deep=True)

    def getTimerEvents(self):
        return self.data[self.data.event!='timer'].copy(deep=True)

    def selectData(self, roads, directions):
        data = self.data
        if roads is None:
            roads = list(data.rd.unique())
        else:
            if type(roads) is str:
                roads = [ roads ]
            rds = list(data.rd.unique())
            roads = [ r for r in roads if r in rds ]
        data = data[data.rd.isin(roads)]

        if directions is None:
            directions = list(data.dir.unique())
        else:
            if type(directions) is str:
                directions = [ directions ]
            dirs = list(data.dir.unique())
            directions = [ d for d in directions if d in dirs ]
        data = data[data.dir.isin(directions)]
        return roads, directions, data

    def maxQueueLength(self, roads=None, directions=None):
        _, _, data = self.selectData(roads, directions)
```

```
        _, _, data = self.selectData(roads, directions)
        data = data[data.event=='queue']
        if len(data)>0:
            return data.ql.max()
        else:
            return 0

    def maxWaitTime(self, roads=None, directions=None):
        roads, directions, data = self.selectData(roads, directions)
        d0 = data[data.event=='queue']
        d1 = data[data.event=='dequeue']
        times = []
        id0 = d0.id.unique()
        id1 = d1.id.unique()
        for id in id0:
            if id in id1:
                t0 = d0.t[d0.id==id].min()
                t1 = d1.t[d1.id==id].max()
                times += [ t1-t0 ]
        if len(times)>0:
            return round(max(times),2)
        else:
            return 0

    def avgWaitTime(self, roads=None, directions=None):
        roads, directions, data = self.selectData(roads, directions)
        d0 = data[data.event=='queue']
        d1 = data[data.event=='dequeue']
        times = []
        id0 = d0.id.unique()
        id1 = d1.id.unique()
        for id in id0:
            if id in id1:
                t0 = d0.t[d0.id==id].min()
                t1 = d1.t[d1.id==id].max()
                times += [ t1-t0 ]
        if len(times)>0:
            return round(sum(times)/len(times), 2)
        else:
            return 0
```

```
159
160     ## new code: computes the average travelling time in seconds
161     ## on a road in a given direction
162     def avgTravelTime(self, roads=None, directions=None):
163         roads, directions, data = self.selectData(roads, directions)
164         if len(roads)>1:
165             print("avg travelling time across different roads not defined")
166             return 0
167         d0 = data[data.event=='start']
168         d1 = data[data.event=='end']
169         times = []
170         id0 = d0.id.unique()
171         id1 = d1.id.unique()
172         for id in id0:
173             if id in id1:
174                 t0 = d0.t[d0.id==id].min()
175                 t1 = d1.t[d1.id==id].max()
176                 times += [ t1-t0 ]
177         if len(times)==0:
178             raise ValueError("No times measured")
179         return round(sum(times)/len(times), 2)
180
181     ## computes the average speed in km/h of cars travelling
182     ## on the given road in the given direction
183     def avgSpeed(self, roads=None, directions=None):
184         roads, _, _ = self.selectData(roads, directions)
185         if len(roads)>1:
186             raise ValueError("avgSpeed undefined for multiple roads")
187         road = self.network.getRoad(roads[0])
188         t = self.avgTravelTime(roads=roads, directions=directions)
189         return round(3.6*road.getLength()/t, 2)
190
191     ## computes traffic flow in vehicles/h for a given road
192     ## and direction based on vehicles reaching the end of the road
193     def flow(self, roads=None, directions=None):
194         roads, directions, data = self.selectData(roads, directions)
195         df = data[data.event=='end']
196         if len(df)<=1:
197             raise ValueError('not enough data')
198         f = (len(df)-1)/(df.t.max()-df.t.min())*3600
```

```
198         f = (len(df)-1)/(df.t.max()-df.t.min())*3600
199         return round(f, 2)
200
201     ## computes traffic density in vehicles/km for a given
202     ## road and direction at a given moment in time.
203     ## When no time is specified it returns a
204     ## list of traffic densities over time.
205     def density(self, roads=None, directions=None,
206                 time=None, plot=False):
207         roads, directions, data = self.selectData(roads, directions)
208         if len(roads)>1 or len(directions)>1:
209             print("not yet implemented")
210             return None
211         road = self.network.getRoad(roads[0])
212         timerEvents = data[data.event=='timer']
213         times = timerEvents.t.unique()
214         if len(times)<1:
215             raise ValueError('not enough data')
216         roadLength = road.getLength()
217
218         if time is None:
219             x, y = [], []
220             for t in times:
221                 events = data[data.t==t]
222                 # ignore the initial period before the first
223                 # vehicle has nearly finished the course
224                 if len(x)>0 or \
225                         events.s.max()>0.9*roadLength:
226                     # cut-off overshooting events
227                     events = events[events.s<=roadLength]
228                     d = len(events)*1000/roadLength
229                     x.append(t)
230                     y.append(round(d,2))
231             μ = round(sum(y)/len(y),2)
232             if plot:
233                 plt.figure(figsize=(5, 3), dpi=120)
234                 plt.plot(x, y)
235                 plt.xlabel('Time [s]')
236                 plt.ylabel('Density [veh/km]')
237                 plt.title("Traffic Density "+roads[0]+
```

```
                        " "+directions[0][0]+"-bound")
                plt.xlim((self.startTime, self.stopTime))
                ylim = plt.ylim()
                plt.ylim((0, ylim[1]))
                plt.axhline(y=μ, ls='--', c='red')
                plt.grid(True)
                plt.show()
            return μ

        # find the timestamp nearest to the requested time
        if time in times:
            t = time
        else:
            # take the nearest point in time
            diff = list((times-time)**2)
            t = times[diff.index(min(diff))]
        events = timerEvents[timerEvents.t==t]
        d = len(events)*1000/roadLength
        return round(d,2)

    def plot(self, x, y,
             vehicles=None, roads=None, directions=None,
             style='', lw=1, decoration=True,
             x0=None, x1=None, y0=None, y1=None, fillColor=None,
             xmin=None, xmax=None, ymin=None, ymax=None):

        columns = ['t', 's', 'v', 'a']
        labels =  ['Time [s]',
                   'Distance [m]',
                   'Velocity [m/s]',
                   'Acceleration [m/s²]']
        try:
            xindex = columns.index(x)
            yindex = columns.index(y)
        except ValueError:
            print(f"Supports only plots of 't', 's', 'v', 'a'")
            return

        xcolix = list(self.data.columns).index(x)
```

```
277         ycolix = list(self.data.columns).index(y)
278 
279         plt.figure(figsize=(5, 3), dpi=120)
280         if xmin is not None and xmax is not None:
281             plt.xlim((xmin, xmax))
282         if ymin is not None and ymax is not None:
283             plt.ylim((ymin, ymax))
284 
285         roads, directions, data = self.selectData(roads, directions)
286 
287         if x=='t':
288             if xmin is None:
289                 xmin = self.startTime
290             if xmax is None:
291                 xmax = self.stopTime
292             plt.xlim((xmin, xmax))
293 
294         if len(roads)==1 and len(directions)==1:
295             plt.title(roads[0]+" "+directions[0][0]+"-bound")
296             road = self.network.getRoad(roads[0])
297             if x=='t' and y=='s':
298                 if ymin is None:
299                     ymin = 0
300                 if ymax is None:
301                     ymax = road.getLength()
302                 plt.ylim((ymin, ymax))
303                 # draw cross roads and traffic light status
304                 intersections = road.getIntersections()
305                 crossRoads = road.getIntersectionPoints()
306                 for i in range(len(intersections)):
307                     name = intersections[i].name
308                     crossRoad = crossRoads[i]
309                     if intersections[i].getTrafficLight() is None:
310                         plt.axhline(y=crossRoad, ls='--', c='black')
311                     else:
312                         direction = mainDirection(directions[0])
313                         tf = self.getTrafficLightData(name, direction)
314                         t = list(tf.t)
315                         col = list(tf.col)
316                         for i in range(len(t)):
```

```
                    t0 = t[i]/xmax
                    t1 = t[i+1]/xmax if i<len(t)-1 else 1
                    plt.axhline(y=crossRoad,
                                xmin=t0, xmax=t1,
                                c=col[i], lw=5)

        if vehicles is None:
            vehicles = list(data.id.unique())

        # if there are many lines to be drawn, use thin lines
        if len(vehicles)>50:
            lw = 0.5*lw

        for id in vehicles:
            df = data[data.id==id]
            colors = list(df.col.unique())
            if len(colors)==1:
                plt.plot(x, y, style, lw=lw, data=df, c=colors[0])
            else:
                plt.plot(x, y, style, lw=lw, data=df)
            plt.xlabel(labels[xindex])
            plt.ylabel(labels[yindex])

            # use small red circle to indicate emergency braking
            dc = df[df.event=='brake']
            for i in range(len(dc)):
                X = dc.iloc[i, xcolix]
                Y = dc.iloc[i, ycolix]
                plt.plot([X], [Y], 'ro')
            db = df[df.event=='brake end']
            for i in range(len(db)):
                X = db.iloc[i, xcolix]
                Y = db.iloc[i, ycolix]
                plt.plot([X], [Y], marker='o', mec='r', fillstyle='none')

        # fill area with background color
        if fillColor is not None:
            if x0 is None:
                x0=self.data[x].min()
```

```
356             if x1 is None:
357                 x1=self.data[x].max()
358             if y0 is None:
359                 y0=self.data[y].min()
360             if y1 is None:
361                 y1=self.data[y].max()
362             plt.fill_between( [x0, x1], [y0, y0], [y1, y1], color=fillColor)
363 
364         plt.grid(True)
365         plt.show()
```

executed in 62ms, finished 17:49:17 2022-04-06

5.1 Backup and Restore

In [19]:

```
1  def saveData(rec, filename):
2      rec.getData().to_csv(filename, index=False)
3  
4  def loadData(filename):
5      data = pd.read_csv(filename)
6      r = Recorder()
7      if list(data.columns) == list(r.data.columns):
8          r.data = data
9          return r
10     else:
11         return None
12 
13 def sameData(r1, r2):
14     try:
15         return all(r1.getData() == r2.getData())
16     except ValueError:
17         return False
```

executed in 3ms, finished 17:49:17 2022-04-06

6 Simulation

TOC

6.1 Traffic Lights

TOC

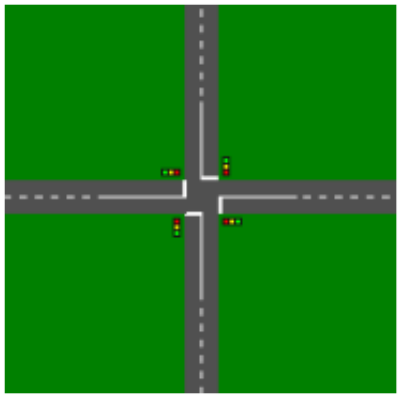
6.1.1 Test Traffic Lights

In [20]:

```
def TestTrafficLight(Tmax=100, NS=[6,2], EW=[4,2] ):

    rec = Recorder(0, Tmax, 0.5)

    network = RoadNetwork("Traffic Light Crossing", 200, 200, rec)
    MSt = network.addRoad("Main St",  "MSt", NORTH_SOUTH, 0.5)
    CRd = network.addRoad("Cross Rd", "CRd", EAST_WEST, 0.5)
    RoadSegment(CRd, None, MSt)
    RoadSegment(CRd, MSt, None)
    RoadSegment(MSt, None, CRd)
    RoadSegment(MSt, CRd, None)
    intersection = network.getIntersection(MSt, CRd)
    TrafficLight(intersection, NS=NS, EW=EW)

    displayMap(network)

    rec.run()

    return rec
```

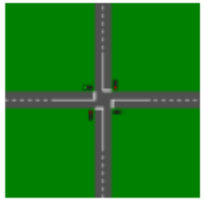
executed in 3ms, finished 17:49:17 2022-04-06

In [21]:
```
1 rec3T = TestTrafficLight(30)
```
executed in 98ms, finished 17:49:17 2022-04-06



In [22]:
```
1 animate(rec3T, 'animation[3T]')
```
executed in 8.30s, finished 17:49:25 2022-04-06

```
MovieWriter imagemagick unavailable; using Pillow instead.

.............................................................done
```



6.1.2 Code Traffic Light Crossing

In [23]:
```
1 def TrafficLightCrossing(Tmax,
2                           VMAXmain=20, VMAXcross=15,
3                           loc=1, IATmain=4, IATcross=10,
4                           NS=[40,10], EW=[20,10]):
```

```
5
6       rec = Recorder(0, Tmax, 0.5)
7
8       network = RoadNetwork("Traffic Light Crossing", 300, 300, rec)
9       MSt = network.addRoad("Main St",  "MSt", NORTH_SOUTH, 0.5)
10      CRd = network.addRoad("Cross Rd", "CRd", EAST_WEST, 0.5)
11      RoadSegment(CRd, None, MSt)
12      RoadSegment(CRd, MSt, None)
13      RoadSegment(MSt, None, CRd)
14      RoadSegment(MSt, CRd, None)
15
16      intersection = network.getIntersection(MSt, CRd)
17      TrafficLight(intersection, NS=NS, EW=EW)
18
19      # displayMap(network)
20
21      seeds = random.sample(range(1000), k=5)
22
23      ## Generate Main Traffic
24      random.seed(seeds[0])
25      t = 0
26      while t<Tmax:
27          Δt = loc + random.expovariate(1/(IATmain-loc))
28          vmax = VMAXmain
29          t = round(t+Δt,2)
30          v = Vehicle(rec, MSt, SOUTH, t0=t, v=vmax, vmax=vmax,
31                      color='black', plan=['Stop@CRd', 'X@CRd'])
32          # v.traceAdjustVelocity = True
33
34      random.seed(seeds[1])
35      t = 0
36      while t<Tmax:
37          Δt = loc + random.expovariate(1/(IATmain-loc))
38          vmax = VMAXmain
39          t = round(t+Δt,2)
40          v = Vehicle(rec, MSt, NORTH, t0=t, v=vmax, vmax=vmax,
41                      color='orange', plan=['Stop@CRd', 'X@CRd'])# v.traceEvents = True
42
43      ## Generate Cross Traffic
44      random.seed(seeds[2])
```

```
    random.seed(seeds[2])
    t = 0
    while t<Tmax:
        Δt = loc + random.expovariate(1/(IATcross-loc))
        vmax = VMAXcross
        t = round(t+Δt,2)
        v = Vehicle(rec, CRd, EAST, t0=t, v=vmax, vmax=vmax,
                    color='red', plan=['Stop@MSt', 'X@MSt'])
        # v.traceEvents = True
        # v.traceCrossing = True
        # v.traceInterrupt = True
        # v.traceAdjustVelocity = True
        # v.traceCruising = True

    random.seed(seeds[3])
    t = 0
    while t<Tmax:
        Δt = loc + random.expovariate(1/(IATcross-loc))
        vmax = VMAXcross
        t = round(t+Δt,2)
        v = Vehicle(rec, CRd, WEST, t0=t, v=vmax, vmax=vmax,
                    color='blue', plan=['Stop@MSt', 'X@MSt'])
        # v.traceAdjustVelocity = True
        # v.traceBraking = True

    random.seed(seeds[4])
    rec.run()

    return rec
```

executed in 6ms, finished 17:49:25 2022-04-06

6.1.3 Simulation Traffic Light Crossing

TOC

In [24]:

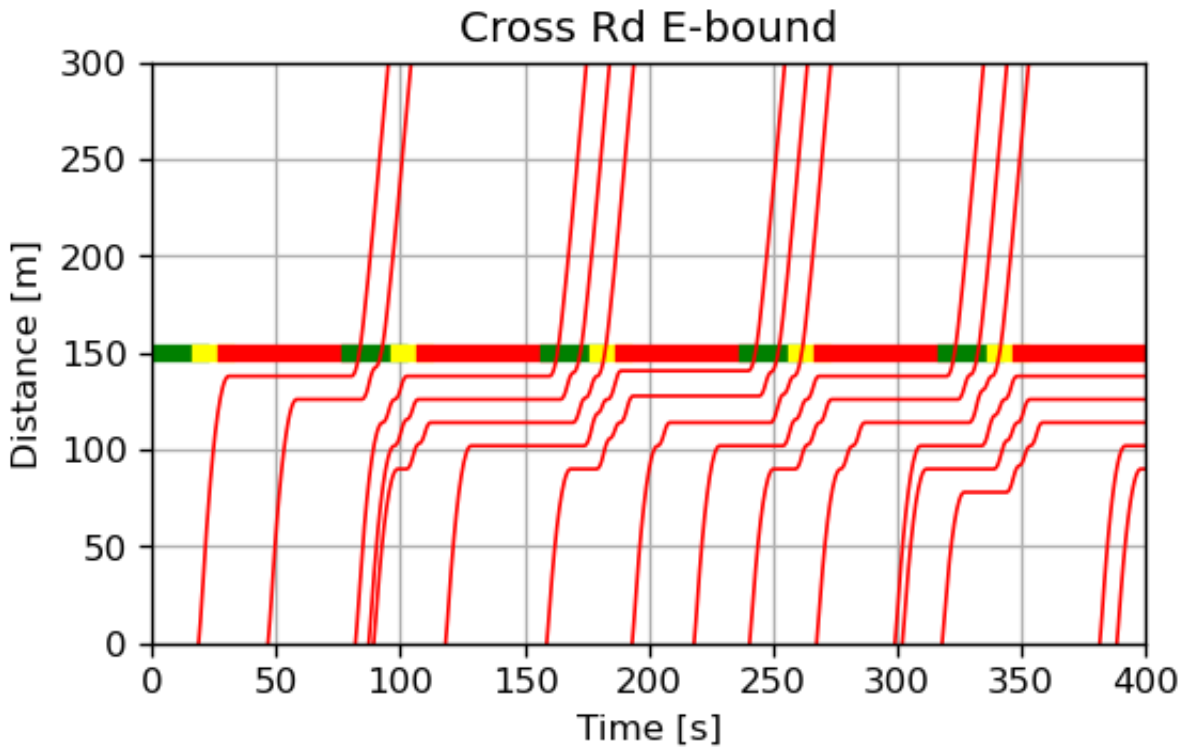
```
IATmain = 30
IATcross = 30
VMAXmain = 50/3.6
VMAXcross = 50/3.6
random.seed(0)
rec3 = TrafficLightCrossing(400, loc=2,
                            VMAXmain=VMAXmain,
                            VMAXcross=VMAXcross,
                            IATmain=IATmain,
                            IATcross=IATcross)

```

executed in 5.93s, finished 17:49:31 2022-04-06

6.1.4 Visualisation Traffic Light Crossing

In [25]:
```
1 rec3.plot('t', 's', roads='Cross Rd', directions='EAST')
```

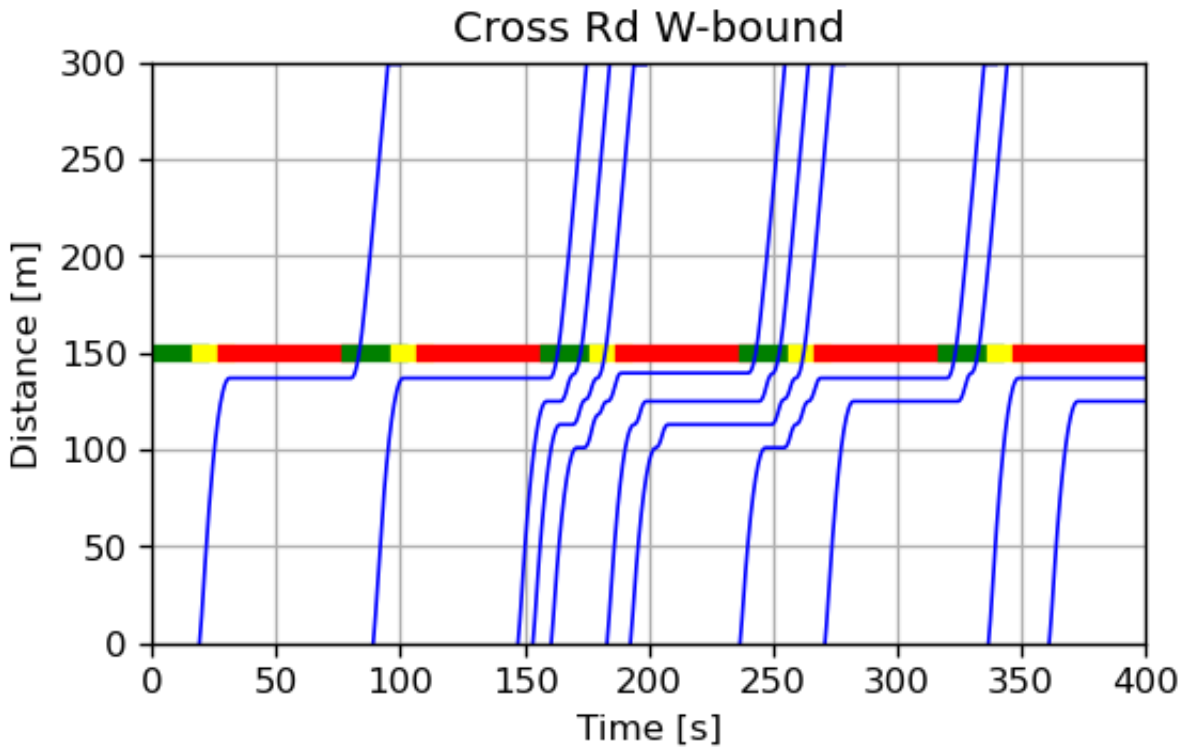
executed in 480ms, finished 17:49:31 2022-04-06



In [26]:

```
1 rec3.plot('t', 's', roads='Cross Rd', directions='WEST')
```

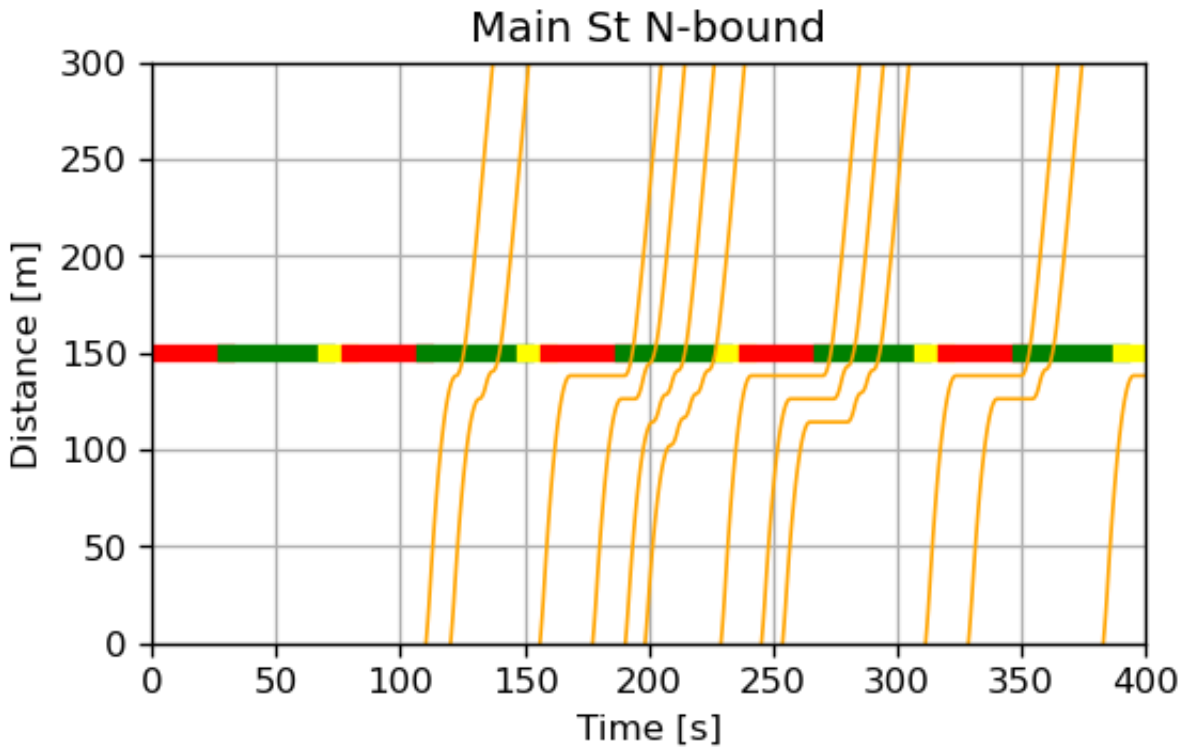
executed in 273ms, finished 17:49:32 2022-04-06



In [27]:

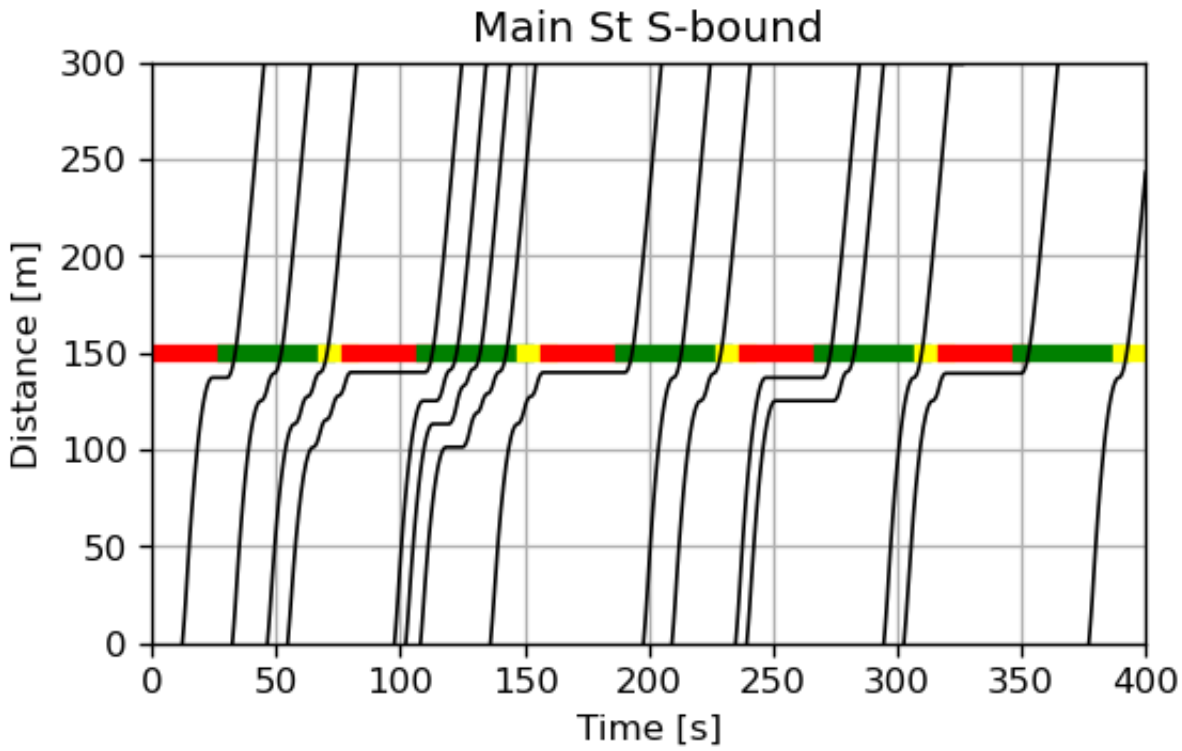
```
1 rec3.plot('t', 's', roads='Main St', directions='NORTH')
```

executed in 291ms, finished 17:49:32 2022-04-06



In [28]:
```
1 rec3.plot('t', 's', roads='Main St', directions='SOUTH')
```

executed in 344ms, finished 17:49:32 2022-04-06



Beware: Generating the animation for 300s simulation takes about 50 min, or 10s per second of animation time.

In [29]:
```
1 # animate(rec3, 'TrafficLight Animation', start_time=0, end_time=300)
```

executed in 2ms, finished 17:49:32 2022-04-06

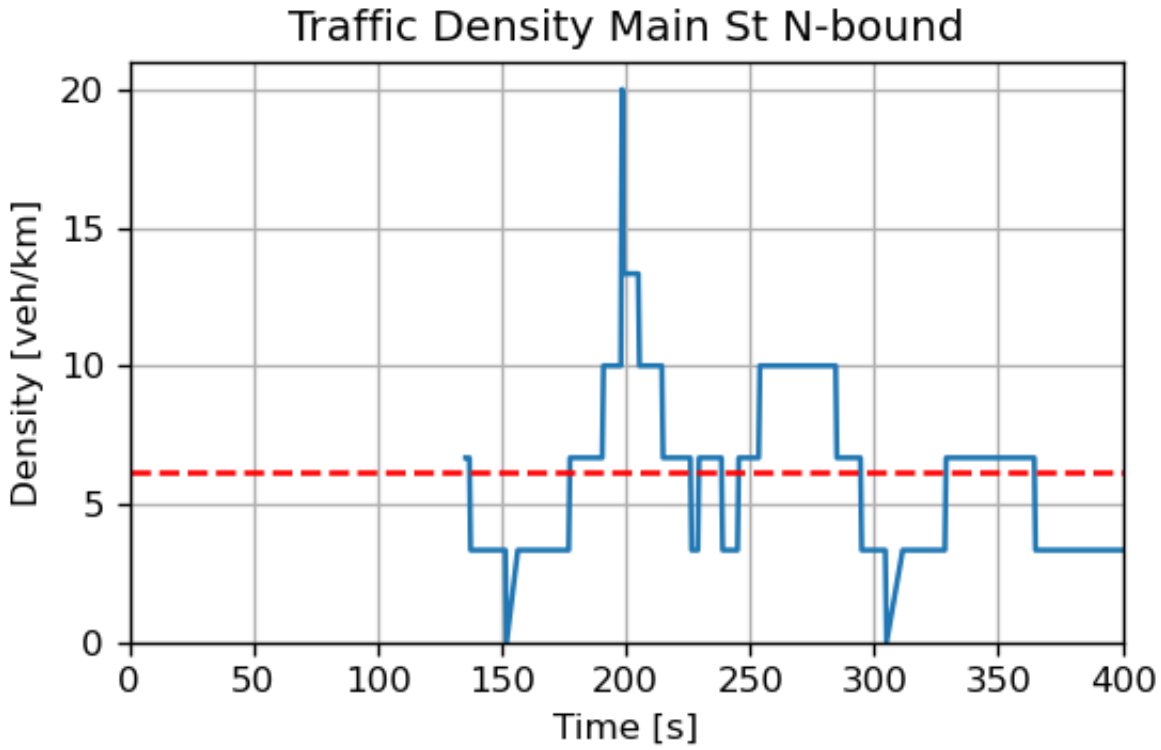
6.1.5 Statistics Traffic Light Crossing

TOC

6.1.5.1 Traffic Density in veh/km

In [30]:
```
1 rec3.density(roads='Main St', directions='NORTH', plot=True)
```

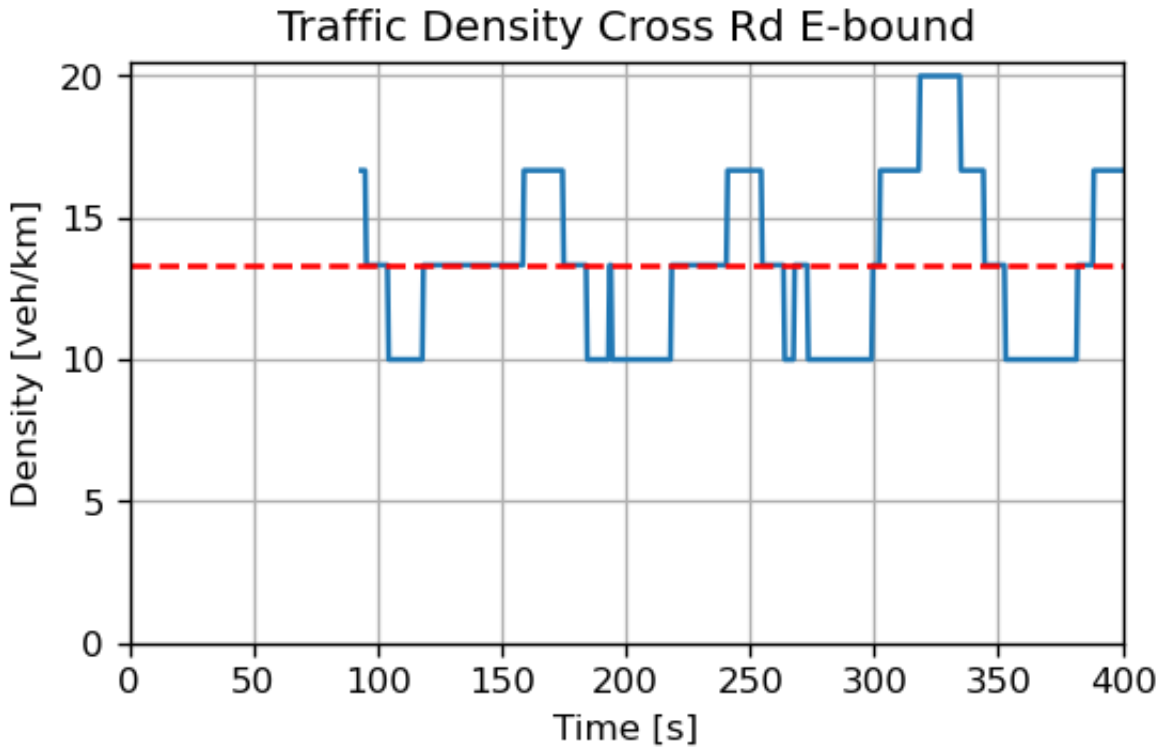
executed in 173ms, finished 17:49:33 2022-04-06



Out[30]: 6.15

In [31]:
```
1 rec3.density(roads='Cross Rd', directions='EAST', plot=True)
```

executed in 217ms, finished 17:49:33 2022-04-06



Out[31]: 13.29

6.1.5.2 Traffic Flow in veh/h

In [32]:

```
VL = rec3.network.VL
flowN = rec3.flow(roads='Main St', directions='NORTH')
expectedFlowN = 3600/IATmain
print(f"N-bound Flow: {flowN:6.1f}veh/h  "
      f"expected flow: {expectedFlowN:6.1f}veh/h ")
flowS = rec3.flow(roads='Main St', directions='SOUTH')
expectedFlowS = 3600/IATmain
print(f"S-bound Flow: {flowS:6.1f}veh/h  "
      f"expected flow: {expectedFlowS:6.1f}veh/h ")
flowE = rec3.flow(roads='Cross Rd', directions='EAST')
expectedFlowE = 3600/IATcross
print(f"E-bound Flow: {flowE:6.1f}veh/h  "
      f"expected flow: {expectedFlowE:6.1f}veh/h ")
flowW = rec3.flow(roads='Cross Rd', directions='WEST')
expectedFlowW = 3600/IATcross
print(f"W-bound Flow: {flowW:6.1f}veh/h  "
      f"expected flow: {expectedFlowW:6.1f}veh/h ")
```

executed in 13ms, finished 17:49:33 2022-04-06

```
N-bound Flow:  154.8veh/h  expected flow:  120.0veh/h
S-bound Flow:  146.4veh/h  expected flow:  120.0veh/h
E-bound Flow:  139.7veh/h  expected flow:  120.0veh/h
W-bound Flow:  115.5veh/h  expected flow:  120.0veh/h
```

When a direction is not specified, the flows in both directions are actually added up.

In [33]:

```
rec3.flow(roads='Cross Rd')
```

executed in 6ms, finished 17:49:33 2022-04-06

Out[33]: 265.15

6.1.5.3 Average Travelling Time in s

In [34]:
```
1 rec3.avgTravelTime(roads='Main St')
```
executed in 12ms, finished 17:49:33 2022-04-06

Out[34]: 47.41

In [35]:
```
1 rec3.avgTravelTime(roads='Cross Rd', directions='EAST')
```
executed in 9ms, finished 17:49:33 2022-04-06

Out[35]: 100.38

In [36]:
```
1 rec3.avgTravelTime(roads='Cross Rd')
```
executed in 9ms, finished 17:49:33 2022-04-06

Out[36]: 90.76

6.1.5.4 Average Speed in km/h

In [37]:
```
1 rec3.avgSpeed(roads='Main St')
```
executed in 10ms, finished 17:49:33 2022-04-06

Out[37]: 22.78

In [38]:
```
1 rec3.avgSpeed(roads='Cross Rd', directions='EAST')
```
executed in 8ms, finished 17:49:33 2022-04-06

Out[38]: 10.76

In [39]:
```
1 rec3.avgSpeed(roads='Cross Rd')
```
executed in 10ms, finished 17:49:33 2022-04-06

Out[39]: 11.9

6.1.5.5 Average and Maximum Wait Time in s

There is no wait time to be expected on the main road:

In [40]:
```
1 rec3.avgWaitTime(roads='Main St')
```
executed in 9ms, finished 17:49:33 2022-04-06

Out[40]: 32.95

In [41]:
```
1 rec3.maxWaitTime(roads='Main St')
```
executed in 9ms, finished 17:49:33 2022-04-06

Out[41]: 60.03

In [42]:
```
1 rec3.avgWaitTime(roads='Cross Rd', directions='EAST')
```
executed in 7ms, finished 17:49:33 2022-04-06

Out[42]: 85.84

6.1.5.6 Maximum Queue Length

In [43]:
```
1 rec3.maxQueueLength(roads='Main St')
```
executed in 6ms, finished 17:49:33 2022-04-06

Out[43]: 3

In [44]:
```
1 rec3.maxQueueLength(roads='Cross Rd', directions='EAST')
```
executed in 4ms, finished 17:49:33 2022-04-06

Out[44]: 5

In [ ]:

```
1
```