Libraries

[ ] ↳ *2 cells hidden*

Entities

[ ] ↳ *1 cell hidden*

Lanes

[ ] ↳ *2 cells hidden*

Vehicles

```
def isRunning(p):
    return p is not None and p.running

def isCrashed(p):
    return p is not None and p.crashed
```

Saved successfully!

```
class Vehicle:
    def __init__(self, env, rec,
                 startingLane=None, startingPos=0,
                 t0=0, x0=0, dx0=0, ddx0=0, dddx0=0,
                 t=[], v=[]):

        global VEHICLE_ID
        self.id = VEHICLE_ID
        VEHICLE_ID += 1

        self.a_min = -4 # [m/s²]
        self.a_max = 2.5 # [m/s²] corresponds to 0-100km/h om 12s

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

```
        self.startingLane = startingLane
        self.startingPos = startingPos
        self.lane = None
        self.pos = 0

        ## second lane reference during changing of lanes
        self.oldLane = None

        self.t0 = t0
        self.x0 = x0
        self.dx0 = dx0
        self.ddx0 = ddx0
        self.dddx0 = dddx0

        self.t = t
        self.v = v
        self.t_target = []
        self.v_target = []

        self.running = False
        self.crashed = False
        self.braking = False
        self.changingLane = False

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

        ## this allows to trigger trace messages for
        ## the new feature Surround
        self.traceSurround = False
        self.traceOvertake = False
        self.traceBrake = False
```

Saved successfully!

```
    def isNotFasterThan(self, other):
        return True if other is None else self.dx0 <= other.dx0

    def isNotSlowerThan(self, other):
        return True if other is None else other.dx0 <= self.dx0

    #REQUIRES TO MERGE ON THE RIGHT SIDE LANE

    def MergeToRight(self):
        if self.lane.length-self.pos <= 500 and \
            self.surround.rightLane is not None and \
            self.surround.leftLane  is None and \
            self.surround.rightLane.length >=500:
                return True
        else:
            return False
```

```
            return False

    def updateOnly(self):
        if self.crashed:
            return False
        t = self.env.now
        if t < self.t0:
            return False
        if self.running and t > self.t0:
            dt = t - self.t0
            ddx = self.ddx0 + self.dddx0*dt
            dx = round(self.dx0 + self.ddx0*dt + self.dddx0*dt*dt/2,4)
            Δx = self.dx0*dt + self.ddx0*dt*dt/2 + self.dddx0*dt*dt*dt/6
            x = round(self.x0 + Δx, 2)
            self.t0, self.x0, self.dx0, self.ddx0 = t, x, dx, ddx

            self.pos = round(self.pos+Δx, 2)
            # update lane information if necessary
            if self.pos >= self.lane.length:
                nextPos = self.pos - self.lane.length
                nextLane = self.lane.next
                self.lane.leave(self)
                if nextLane is None:
                    self.rec.record(self, event='end')
                    self.running = False
                    return False
                else:
                    nextLane.enter(self, pos=nextPos)
        return True

    def update(self):
        active = self.updateOnly()
        if not active:
```

Saved successfully!

```
        self.surround = Surround(self)

        ## instead of direct link, call method
        inFront = self.surround.front
        if (isRunning(inFront) or isCrashed(inFront)) \
               and inFront.x0 < self.x0 + CAR_LENGTH:
            self.crash(inFront)
            return True

        if inFront is not None and not self.braking and \
                self.dx0 > inFront.dx0 and \
                self.x0 + CRITICAL_TIME_TOLERANCE*self.dx0 > inFront.x0:
            Δt = max(MIN_TIME_DIFF, (inFront.x0-self.x0)/self.dx0)
            self.setTarget(Δt, inFront.dx0)
            self.interruptProcess()
            return True
```

```
        ## new code: start overtaking maneuver by changing into fast lane
        if inFront is not None and \
                not self.braking and not self.changingLane and \
                self.dx0 > inFront.dx0 + MIN_SPEED_DIFF and \
                self.x0 + (LANE_CHANGE_TIME+CRITICAL_TIME_TOLERANCE)*self.dx0 > inFront.x0 an
                self.surround.rightLane is not None and \
                self.surround.right is None and \
                self.isNotFasterThan(self.surround.rightFront) and \
                self.isNotSlowerThan(self.surround.rightBack):
            if self.traceOvertake:
                print(f"t={self.t0:7,.1f}s Overtaking v{self.id:d} overtakes v{inFront.id:d}
            self.setTarget(LANE_CHANGE_TIME, 'fast')
            self.interruptProcess()
            return True

        ## new code: end overtaking by returning to slow lane
        if self.surround.leftLane is not None and \
                not self.braking and not self.changingLane and \
                self.surround.left is None and \
                self.isNotFasterThan(self.surround.leftFront) and \
                self.surround.leftBack is None:
            if self.traceOvertake:
                print(f"t={self.t0:7,.1f}s Overtaking v{self.id:d} returns to slow lane at x=
            self.setTarget(LANE_CHANGE_TIME, 'slow')
            self.interruptProcess()
            return True

    def setTarget(self, Δt, v):
        self.t_target = [ Δt ] + self.t_target
        self.v_target = [ v ] + self.v_target

    def process(self):

Saved successfully!                                                    ime t-

            yield self.env.timeout(self.t0-self.env.now)
        self.t0 = self.env.now
        self.running = True
        self.rec.startRecording(self)
        self.startingLane.enter(self, pos=self.startingPos)

        while self.running:
            self.updateOnly()

            self.surround = Surround(self)

            inFront = self.surround.front
            if inFront is not None:

                # if the car in front is slower and we are a bit too near on its heals...
                if inFront.dx0 < self.dx0 and \
                        inFront.x0 < self.x0 + CRITICAL_TIME_TOLERANCE*self.dx0:
```

```
                    if self.traceBrake:
                        print(f"t={self.t0:7,.1f}s Braking v{self.id:d} v={self.dx0:4.4f}m/s


                    yield from self.emergencyBraking(inFront.dx0)
                    if not isZero(self.dx0-inFront.dx0):
                        # after emergency breaking adjust to the speed of the car in front...
                        Δt = 2
                        self.setTarget(Δt, inFront.dx0)
                    continue

            if len(self.t_target)==0:
                self.t_target = self.t.copy()
                self.v_target = self.v.copy()

            if len(self.t_target)>0 or self.MergeToRight()==True:

                ## add code for explicit change of lane
                if type(self.v_target[0]) is str:
                    direction = normaliseDirection(self.v_target[0])
                    t = self.t_target[0]
                    self.t_target = self.t_target[1:]
                    self.v_target = self.v_target[1:]
                    if self.lane.getLane(direction) is not None:
                        yield from self.changeLane(direction, t)

                ## the rest is what was there before
                else:
                    v0 = self.dx0
                    v1 = self.v_target[0]
                    t = self.t_target[0]
                    self.t_target = self.t_target[1:]
                                    self.v_target[1:]
                                    :
                        yield from self.wait(t)
                    else:
                        yield from self.adjustVelocity(v1-v0, t)
            else:
                yield from self.wait(10)

        self.rec.stopRecording(self)

    def emergencyBraking(self, v):

        def emergencyBrakingProcess(v):
            self.rec.record(self, 'brake')
            minΔt = 0.2
            self.dddx0 = (self.a_min-self.ddx0)/minΔt
            yield self.env.timeout(minΔt)

            self.updateOnly()
```

Saved successfully!

```
            self.dddx0=0
            self.ddx0=self.a_min
            v = min(v, self.dx0-2)
                # the brake time estimate is for perfect timing for
                # autonomous cars. For manual driving leave out the
                # -minΔt/2 or use a random element.
            Δt = max(0.5, (v-self.dx0)/self.ddx0 - minΔt/2)
            yield self.env.timeout(Δt)

            self.updateOnly()
            self.dddx0 = -self.ddx0/minΔt
            yield self.env.timeout(minΔt)

            self.updateOnly()
            self.ddx0 = 0
            self.dddx0 = 0


        ## The 'braking' bit prevents the interruption of an emergency breaking process
        self.braking = True
        self.processRef = self.env.process(emergencyBrakingProcess(v))
        try:
            yield self.processRef
        except simpy.Interrupt:
            pass
        self.processRef = None
        self.braking = False

    ## make changeLane robust against interrupt:
    def changeLane(self, direction, Δt):

        # smoothly adjust velocity by Δv over the time Δt
        def changeLaneProcess(oldLane, newlane, Δt):
```

Saved successfully!

```
            hange '+direction)
            self.oldLane = oldLane
            newLane.enter(self, pos=self.pos)
            self.ddx0 = 1
            self.dddx0 = 0
            yield self.env.timeout(Δt)
            self.updateOnly()
            self.oldLane.leave(self)
            self.lane = newLane
            self.oldLane = None
            self.rec.record(self, 'done change '+direction)
            self.updateOnly()
            self.ddx0 = 0
            self.dddx0 = 0

        ## keep record of current lane, as in case of aborting
        ## the lane change
        ## when interrupted go back into original lane
```

```
            oldLane = self.lane
            newLane = self.lane.getLane(direction)
            self.changingLane = True
            try:
                self.processRef = self.env.process(changeLaneProcess(oldLane, newLane, Δt))
                yield self.processRef
                self.processRef = None
            except simpy.Interrupt:
                # if interrupted go quickly back into old lane
                # but this is not interruptible
                self.processRef = None
                self.env.process(changeLaneProcess(newLane, oldLane, Δt/4))
            self.changingLane = False

        def adjustVelocity(self, Δv, Δt):

            # smoothly adjust velocity by Δv over the time Δt
            def adjustVelocityProcess():
                self.updateOnly()
                minΔt = 0.1*Δt
                a = Δv/(Δt-minΔt)
                tt = Δt-2*minΔt

                self.dddx0 = (a-self.ddx0)/minΔt
                yield self.env.timeout(minΔt)

                self.updateOnly()
                self.dddx0 = 0
                self.ddx0 = a
                yield self.env.timeout(tt)

                self.updateOnly()
```

Saved successfully!

```
inΔt)
                self.updateOnly()
                self.dddx0 = 0
                self.ddx0 = 0

            self.processRef = self.env.process(adjustVelocityProcess())
            try:
                yield self.processRef
            except simpy.Interrupt:
                self.dddx0 = 0
                pass
            self.processRef = None

        def wait(self, Δt):

            def waitProcess():
                yield self.env.timeout(Δt)
```

```
        self.processRef = self.env.process(waitProcess())
        try:
            yield self.processRef
        except simpy.Interrupt:
            pass
        self.processRef = None

    def interruptProcess(self):
        if self.processRef is not None and self.processRef.is_alive:
            self.processRef.interrupt('change')

    def crash(self, other):

        def recordCrash(self):
            self.rec.record(self, 'crash')
            self.running = False
            self.crashed = True
            self.dx0 = 0
            self.ddx0 = 0
            self.dddx0 = 0

        if self.running:
            print(f"Crash p{self.id:d} into p{other.id:d} at t={self.t0:7.3f} x={self.x0:7.1f
            recordCrash(self)
            if other.running:
                recordCrash(other)
```

Surroundings of car

[ ] ↳ *1 cell hidden*

Saved successfully!

Recorder

[ ] ↳ *1 cell hidden*

Multiple vehicles with fixed speed Lane change

```
VMAX = 120/3.6
N = 50
DT = 6 # time difference between start
env = simpy.Environment()
rec = SimpleRecorder(env, 0, 800, 1)

c = Lane(2000, VMAX)
```

```
l = c.widenLeft()
c.extend(Lane(1000, VMAX))
r = c.widenRight()
print("Left Lane:  ", l)
print("Centre Lane:", c)
print("Right Lane: ", r)
for i in range(N):
    v = Vehicle(env, rec, startingLane=l, t0=i*DT, dx0=0, t=[10], v=[VMAX+3*i])
    v = Vehicle(env, rec, startingLane=c, t0=i*DT, dx0=0, t=[10], v=[VMAX+3*i])
    v.traceOvertake = True
rec.run()
```

```
 Left Lane:   [1 2000m R:0]
 Centre Lane: [0 2000m L:1 R:3]-[2 1000m R:4]
 Right Lane:  [3 2000m L:0]-[4 1000m L:2]
 t=   23.0s Overtaking v5 overtakes v3 at x=  236.0m
 t=   55.0s Overtaking v9 returns to slow lane at x=1,163.8m
 t=   55.0s Overtaking v11 overtakes v9 at x=  966.6m
 t=   58.0s Overtaking v11 returns to slow lane at x=1,115.6m
 t=   61.0s Overtaking v7 returns to slow lane at x=1,574.7m
 t=   67.0s Overtaking v11 returns to slow lane at x=1,514.8m
 Crash p11 into p8 at t= 68.000 x= 1548.2
 t=   79.0s Overtaking v13 returns to slow lane at x=1,661.4m
 t=   80.0s Overtaking v15 returns to slow lane at x=1,657.5m
 t=   81.0s Overtaking v17 overtakes v15 at x=1,514.4m
 t=   84.0s Overtaking v17 returns to slow lane at x=1,629.7m
 t=   87.0s Overtaking v17 returns to slow lane at x=1,740.6m
 Crash p18 into p12 at t= 91.000 x= 1529.7
 t=   95.0s Overtaking v21 returns to slow lane at x=1,776.0m
 Crash p20 into p22 at t=107.000 x= 1940.3
 Crash p38 into p18 at t=136.000 x= 1528.4
 t=  138.0s Overtaking v39 returns to slow lane at x=1,716.3m
 Crash p46 into p38 at t=158.000 x= 1527.7
 t=  159.0s Overtaking v47 returns to slow lane at x=1,637.3m
```



```
df1.to_csv(r'/content/drive/MyDrive/Colab Notebooks/1.csv', index = False)
```

```
df1.head(40)
```

```
df1[df1["lane"]==1].head(40)
```

Average Time

```
start, end = {},{}
time_taken = []
for i in range(len(df1)):
    if df1["id"][i] not in start:
```

```
        if df1["id"][i] not in start:
            start[df1["id"][i]] = df1["t"][i]
            end[df1["id"][i]] = df1["t"][i]
        end[df1["id"][i]] = df1["t"][i]

    for i in start:
        time_taken.append(end[i]-start[i])
    average = sum(time_taken) / len(time_taken)
    print(f"The average time is {average} seconds ")
```

```
The average time is 35.61375394344311 seconds
```

Throughput

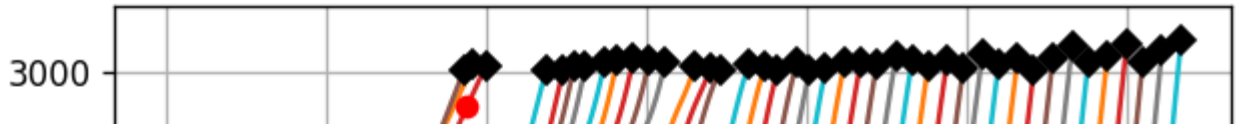
```
event = "end"
for index, row in df1.iloc[::-1].iterrows():
    if df1["event"][index]=="end":
        number = df1["id"][index]
        time = df1["t"][index]
        break

factor = 3600/time
throughput = number*factor
print(f"The Throughput is {throughput} cars per hour")
```

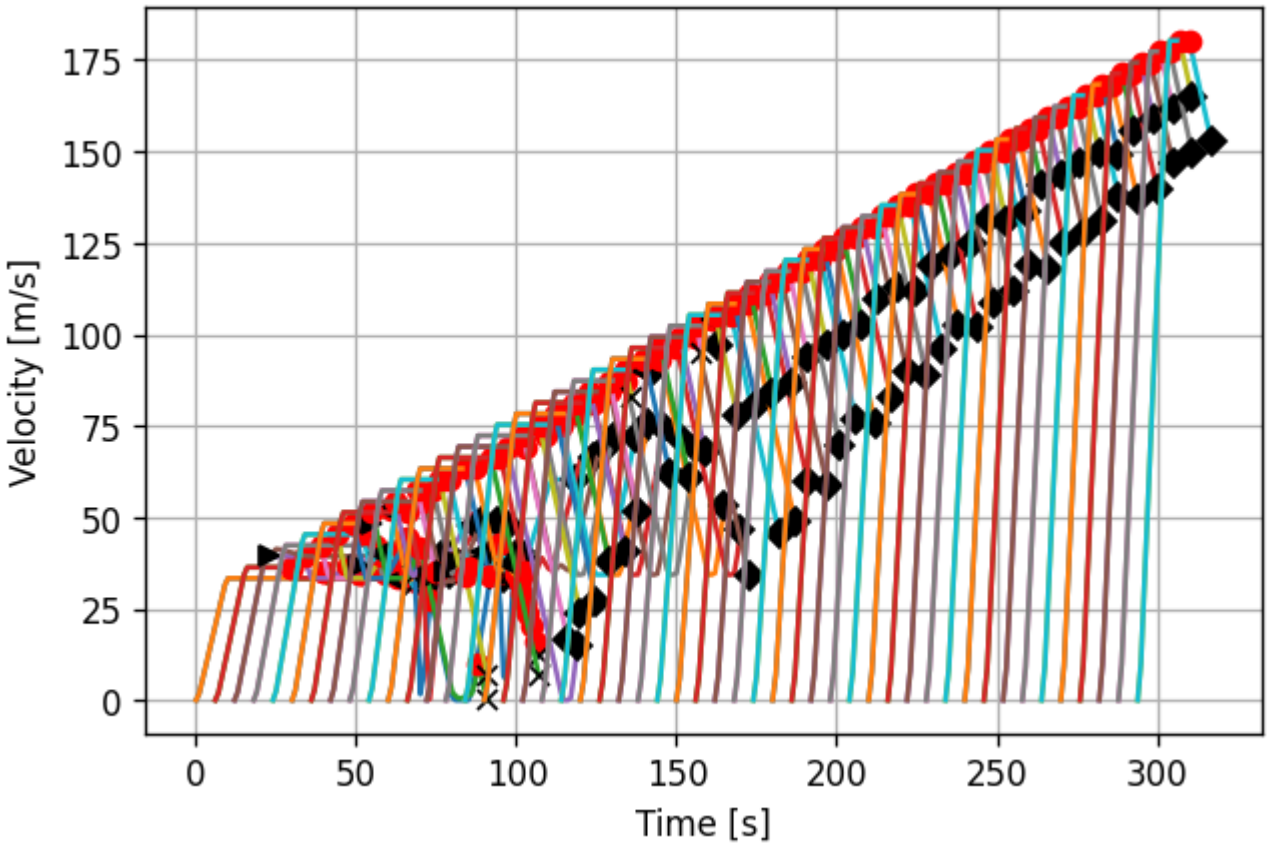
```
The Throughput is 1124.2902208201892 cars per hour
```

```
rec.plot('t', 'x')
```

Saved successfully!

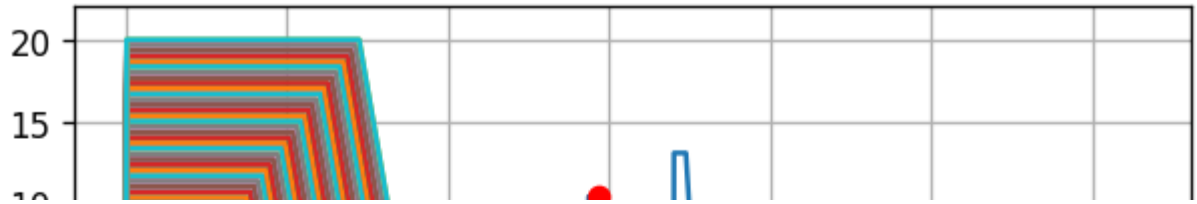



```
rec.plot('t', 'v')
```

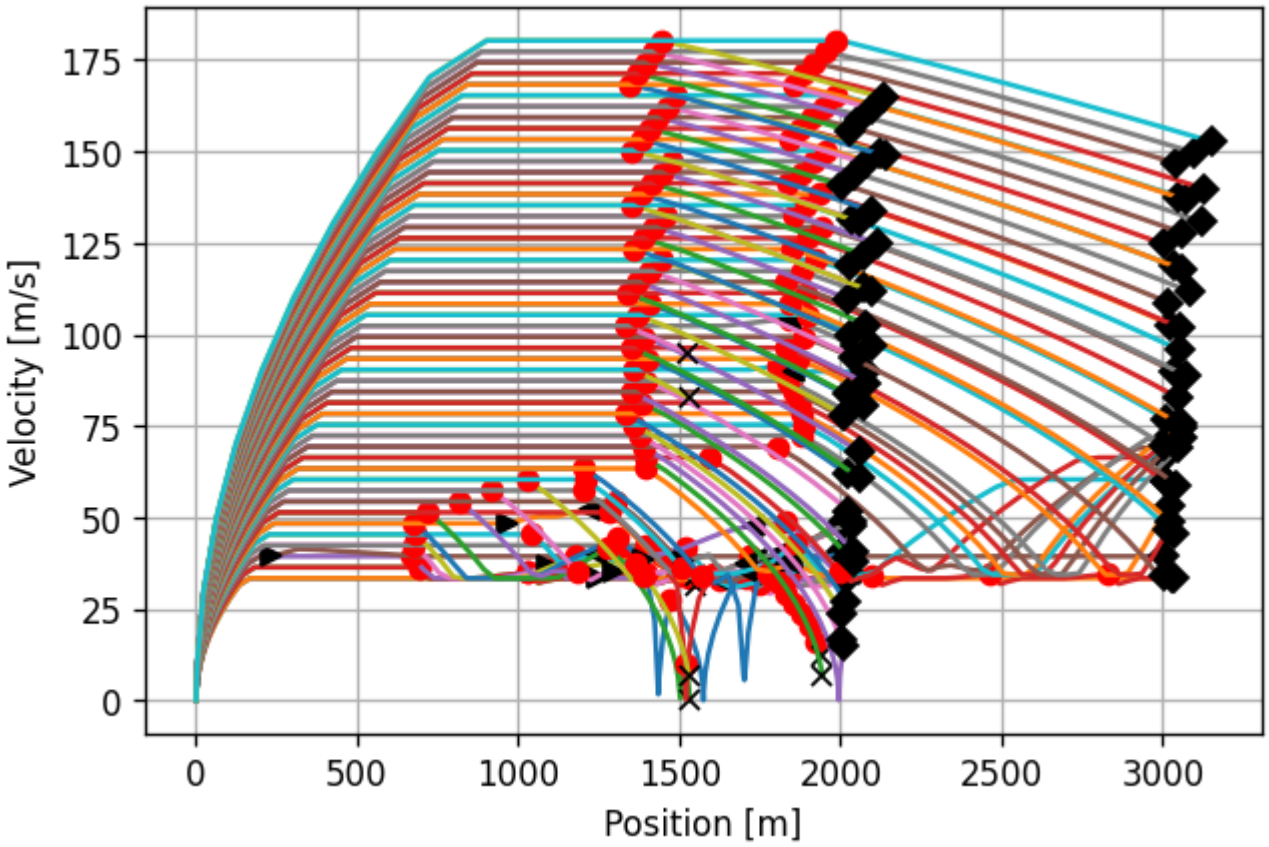



```
rec.plot('x', 'a')
```

Saved successfully!




```
rec.plot('x', 'v')
```



Saved successfully!

✓ 1s completed at 11:37 PM

Saved successfully!