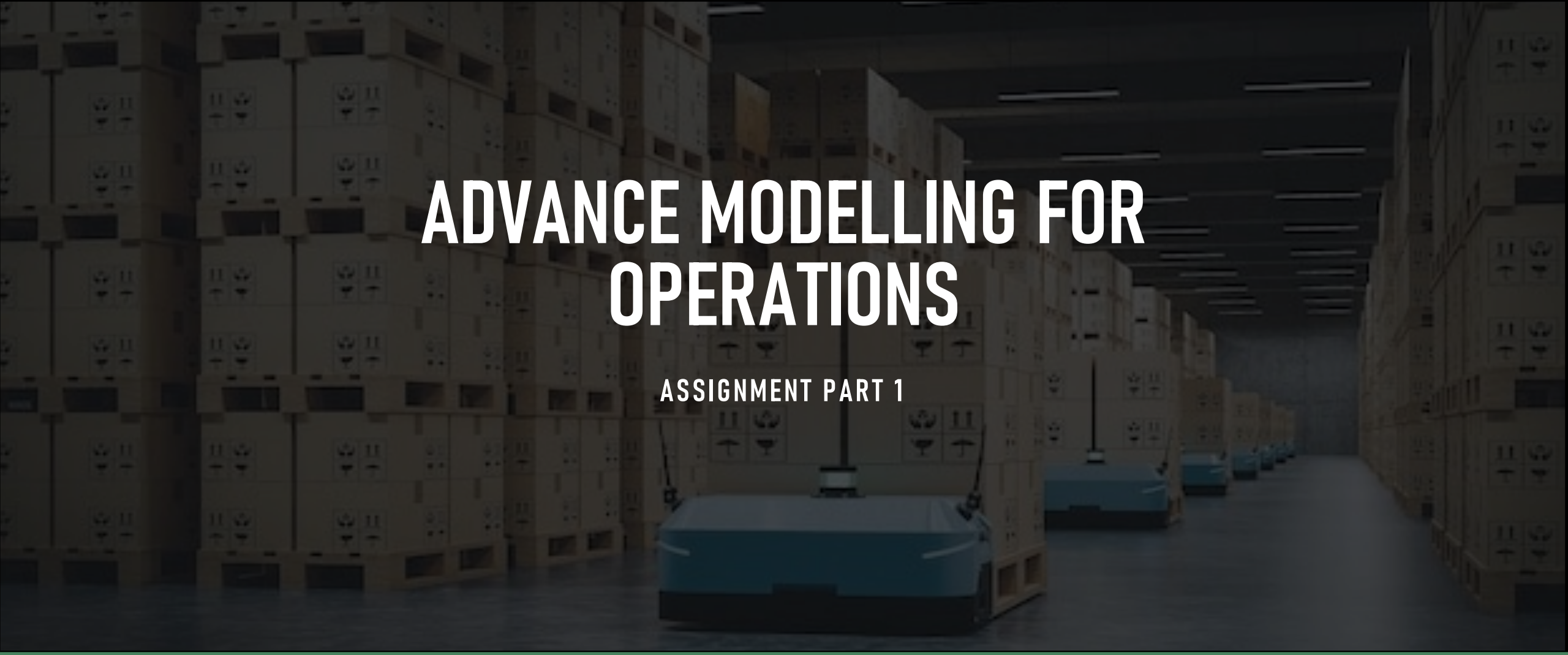


ADVANCE MODELLING FOR OPERATIONS

ASSIGNMENT PART 1

TEAM 1

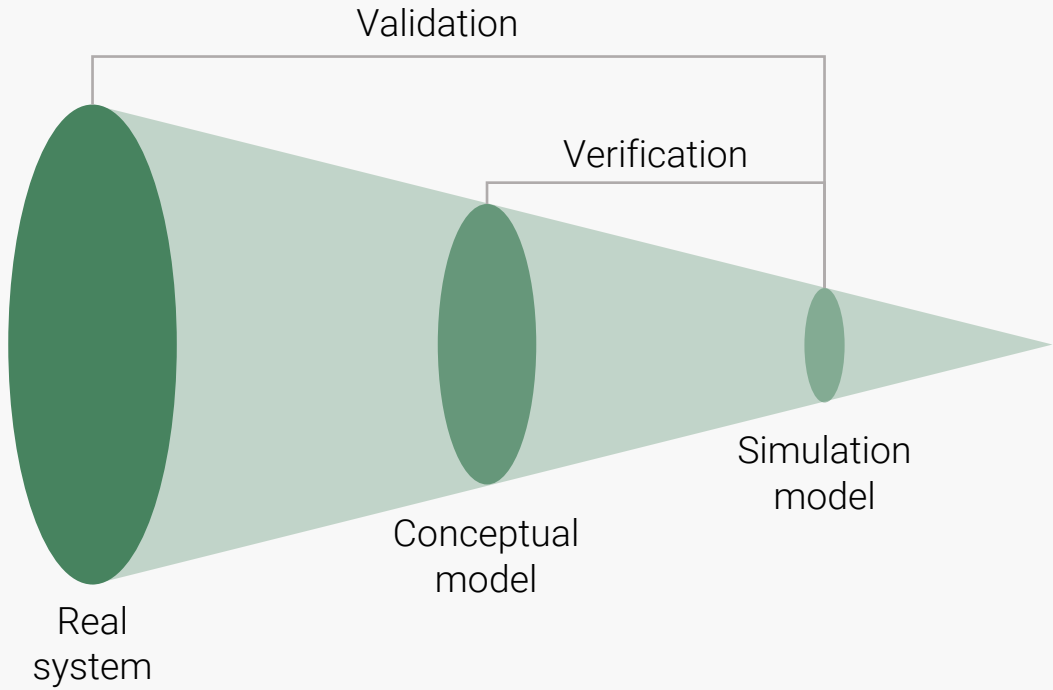
Federico Cantarelli	10596312
Edoardo Gronda	10826399
Silvia Laberinti	10622306
Matteo Mascheroni	10638118
Emanuele Mazzilli	10863187

EXECUTIVE SUMMARY

This presentation aims to explain how the assignment base code, which describes the production and the transportation of finished goods (UL) to the warehouse through one or more autonomous tugger trains, has been enhanced.

More specifically, we adopted a structured approach. Firstly, we performed verification experiments to find out the misalignments between the conceptual and the simulation models. Then, we performed validation experiments to check whether the simulation model possesses sufficient accuracy to represent and address the research problem.

Once the weaknesses of the code have been improved, the focus shifted to creating an updated version of the model to achieve an average idle time lower than 5 minutes.



VERIFICATION

01

Tugger train’s position update

In the base case, the tugger train position is not updated while “on mission”. It moves but the distance does not correspond to the actual one.

02

Distances’ computation

The distance is miscalculated between station 3 and 4.

03

Moving and loading

The tugger verifies the presence of a Unit Load in the next station while it is in the previous line.

04

Step width and initial charging condition

A bias is created due to the discretization of time in minutes despite some functions are performed in continuous time.

05

Weight capacity constraint

The simulation model takes into consideration only the loading capacity constraint of the tugger train.

06

Linear Battery Charging

The function that governs the tugger’s charging time is linear, despite not being so in reality.

07

Anticipated loading correction

The tugger train for the first-round loads units few seconds before the units are created.

08

Initial position of the tugger train

At the beginning of the simulation, the tuggers start from a position as they were stacked.

01

Tugger train's position update

Within the move() function relative to the class Train(), the crossed-out line present in the as-is code was allowing the system to update the tugger's next stop correctly, but we opted to shift it within the added else() condition to give a clearer understanding of the code.
What has been added within the new condition consists of the update of the tuggers' next positions, allowing thus the model to take correctly trace.
Note: In pursuing this slight stylistic modification, we had to keep in mind the condition related to the if() statement. Indeed, we had to add the equal sign, in order to avoid the next line's indexing to go out of range, which is defined by the number of lines, i.e., 5 with index ranging from 0 to 4.

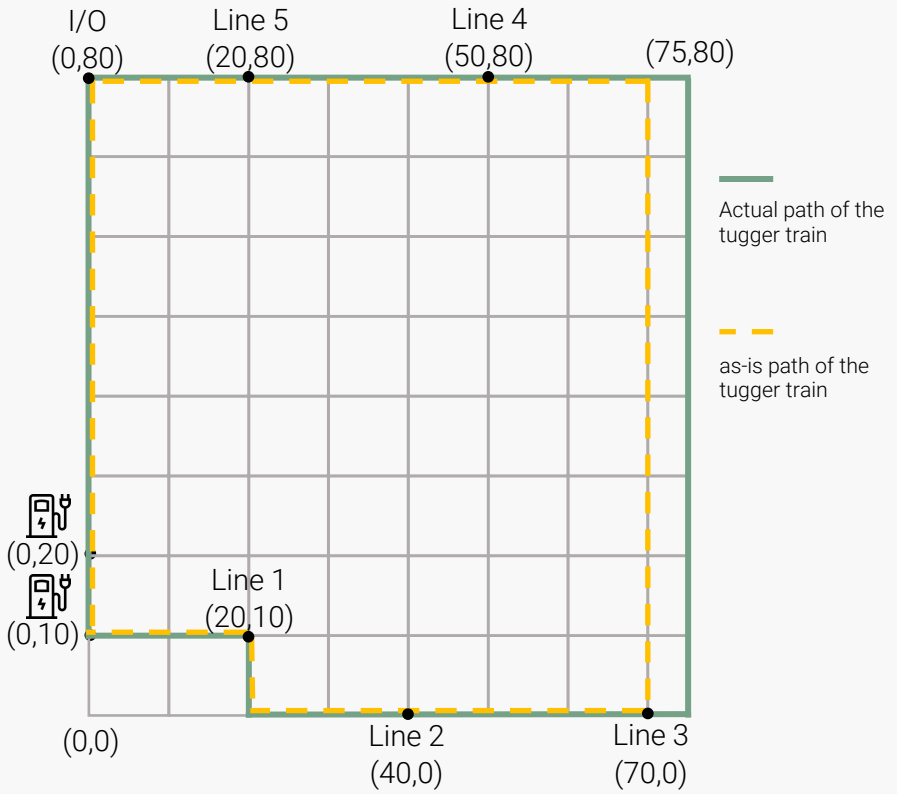
```
if self.next_line >= 4:
            self.next_stop_x = warehouse_coord[0]
            self.next_stop_y = warehouse_coord[1]

else: # Added
            self.next_line += 1
            self.next_stop_x = lines_output_points_x[self.next_line]
            self.next_stop_y = lines_output_points_y[self.next_line]
```

02

Distances' computation

The as-is code computed distances not considering the special case regarding movements between the third and fourth stations losing overall 10 meters. We amended the function compute_distance() taking into account the real path of the tugger train moving along the plant's edges (max_x), therefore returning the correct rectilinear distance.
In order to create a more flexible code, max_x has been added as a parameter.




```
# Parameters
max_x = 75 # Extreme right point of tugger train path

def compute_distance(x1: float, x2: float, y1: float, y2: float, max_x =
max_x):
            if y2 > y1:
                        return abs(max_x - x1) + (max_x - x2) + abs(y1 -
y2)
            else:
                        return abs((max_x - x1) - (max_x - x2)) + abs(y1 -
y2)
```

03

Moving and loading

In the base case, the tugger verifies the presence of a Unit Load in the next station while being in the previous line. As a consequence, if one unit would have been completed while the tugger has been moving toward that station, that particular UL would not have been picked up.
Thus, we created a sequence of tasks by decoupling the moment the tugger begins its journey to the next line from the moment it determines the number of Unit Loads present in the reached production line's buffer.

```
def move(self):
    if (not self.flag_load) or (self.next_stop_x == warehouse_coord[0] and self.next_stop_y == warehouse_coord[1] and
                self.pos_x== 20.0 and self.pos_y == 80.0):
        distance_next_stop = u.compute_distance(self.pos_x, self.next_stop_x, self.pos_y, self.next_stop_y)
        if self.flag_load:
            ...
        else:
            ...
        self.task_endtime += u.compute_time( distance_next_stop,speed=u.compute_speed(self.weight), nextline=self.next_line)

        ...
        self.pos_x = self.next_stop_x
        self.pos_y = self.next_stop_y
        self.flag_load = True
    ...
        else:
            unloading_time = 30 + random.uniform(30, 60)*self.load
            self.task_endtime += unloading_time
            self.remaining_energy -= u.compute_energy_loading(self.weight)
            self.load = 0
            self.weight = 0
            self.next_line = 0
            self.flag_load = False

         if self.next_line >= 4:
              self.next_stop_x = warehouse_coord[0]
              self.next_stop_y = warehouse_coord[1]
         else:
              self.next_line += 1
              self.next_stop_x = lines_output_points_x[self.next_line]
              self.next_stop_y = lines_output_points_y[self.next_line]
              self.flag_load = False
```

In order to do that we have created a *flag* that switches on *True* only when the tugger is in front of the production line and needs to check if there are ULs to be picked up and when it has reached the warehouse and has to unload ULs, while it is set on false when the tugger is travelling or charging.

04

Step width and initial charging condition

The basic time period in the base-case code was expressed in minute and so it was stored as a discrete variable in minutes. Despite this, some operations were made with a “continuous” approach (for example the random uniform distribution in loading and unloading times). Since these kind of operations are stored as floating-point variables, this could lead to a biased solution caused by the approximation of those operations. We decided to convert the basic time period in seconds, so the bias will impact the final output with 60x less magnitude, considering also the fact that our idle target time is expressed in minutes.
To this we have first imported the lines’ cycle times (min) converting them in seconds by simply multiplying them by 60. Moreover, the simulation length, expressed in seconds has been computed by multiplying the number of shifts (in our case 2), the working hours per shift (7.5) and the seconds in 1 hour.

```
def read_line_info(path: str, conversion_factor=60):
    x = []
    y = []
    cycle_times = []
    weights = []

    with open(path, "r") as f:
        f.seek(0)
        for line in f.readlines()[1:]:
            a = line.split(',')
            x.append(float(a[1]))
            y.append(float(a[2]))
            cycle_times.append(float(a[3]) *
conversion_factor)
            weights.append(float(a[4]))
    return x, y, cycle_times, weights
```

```
model = FactoryModel()
for i in range(n_shift*wh*3600): # Seconds
    model.step()
```

We considered that the plant is productive for 2 shifts of 7.5 hours per day instead of 8 hours per day. In the remaining hour (0.5h*2), we considered several activities going on, such as changing the shift, emptying the buffer lines at the end of the day, unloading Unit Loads, and positioning the tuggers at the warehouse, without recharging them. Therefore, we considered that everyday tuggers start from the warehouse with a percentage of charge uniformly distributed between 5% and 98%, considering respectively the worst-case and best-case scenarios to make a ride from the charging stations to the warehouse. The reason for these choices is clearer if we consider part 2 of the assignment, where it was chosen to carry out a finite horizon simulation.

05

Weight capacity constraint

The as-is code did not take into account the maximum weight capacity of the tugger trains in the move() function of the class Train().
Therefore, we firstly added output's weights in "lines_info.cvs", then we imported and saved them within the list output_weight. Afterwards, the attributes weight_capacity = 2000 and weight=0 have been added to class Train(). Lastly, the weight constraint has been added within the move() function of the class Train().
By doing so, the tugger train will check before loading the next UL if there is enough spare weight capacity to actually load it. If there isn't enough spare capacity, tugger train will go on to the next line or, eventually to the warehouse to unload ULs.

```
if (self.weight + output_weight[self.next_line]) <= self.weight_capacity:
    print("\n" + self.unique_id, "going to line", self.next_line, "and picking up a Unit Load")
    loading_time = random.uniform(30, 60)
    self.task_endtime += loading_time
    self.remaining_energy -= u.compute_energy_loading(output_weight[self.next_line])
    self.model.schedule_lines.agents[self.next_line].UL_in_buffer -= 1
    self.load += 1
    self.weight += output_weight[self.next_line]

else:
    print("\n" + self.unique_id, "- Not enough weight capacity left.")
```

06

Linear charging of the battery

The charging function of the tugger's battery is not reflected in reality. In order to improve it, not finding any valuable data on databases, we asked a Politecnico's professor who deals with batteries who suggested that the charging function can be approximated, with considerably low margins, to that of a phone provided with fast charging option. Therefore, we used the data of an iPhone 13 Pro to develop a more realistic function which is shown in the table below. Moreover, we assumed that the time can be linearly interpolated exploiting the percentage levels of charging of the battery.

```python
def compute_charging_time(charge: dict, remaining_energy: float, tot_cap: float):
    percentage = remaining_energy/tot_cap
    for i in charge.keys():
        if percentage <= i:
            max_index = i
            break
        if percentage >= i:
            min_index = i

    upper_time = charge[max_index]
    lower_time = charge[min_index]

    interpolated_time = lower_time*(max_index-percentage) + upper_time*(percentage - min_index)
    interpolated_time = interpolated_time/(max_index - min_index)
    time = charge[1] - upper_time
    time += interpolated_time
    return time
```

iPhone 13 pro

Time (minutes)	% of charge
5	7%
10	17%
15	27%
30	55%
45	76%
60	86%
75	95%
90	98%
98	100%

07

Anticipated loading correction

In the as-is situation the tugger train loads the units few seconds before they have been created by lines for the first round. To solve this issue within the class FactoryModel() we rearranged the step() function updating the model's time before the line's time.

```
class FactoryModel(Model):
    def __init__(self, seed=None):
        super().__init__()
        self.schedule_trains = BaseScheduler(self)
        self.schedule_stations = BaseScheduler(self)
        self.schedule_lines = BaseScheduler(self)
        self.system_time = 0



       ######################################################



def step(self):
    self.system_time += 1
    if verbose and system_time_on:
        print("System time: " + str(self.system_time))
    self.schedule_lines.step()
    self.schedule_trains.step()
    self.schedule_stations.step()
```

08

Initial position of the tugger train

At the beginning of the simulation, all tuggers start from the same position. Actually, it is unrealistic to assume these overlaps, and a simulation model that identifies the lengths of tuggers and creates a queue of them would be more accurate. Nevertheless, considering the simulation is run for just one day, and the final objective is to evaluate the idle time, we considered the impact of this change marginal, so the as-is situation has been kept.

VALIDATION - OPTIMIZATION

01

Constant speed

Average speed meant to account for high-speed travel, low-speed turns, and acceleration/ deceleration times.

02

Compute time

Random.uniform(0,1) is added independently of the distance travelled while it depends on the path the tugger has to do.

03

Compute energy

Energy does not depend solely on time but also on other variables such as loading weight or speed (acceleration, deceleration).

04

Charging conditions

In the as-is code, the tugger train charges the battery when the remaining energy is slightly less than 40% of battery's capacity, despite having sufficient charge to still perform several missions.

05

Loading more than 1 Unit Load

The tugger train can physically load more than one unit belonging to the same line.

06

Non-random charging station picking

The charging station should be chosen given the waiting the time the tugger needs to wait.

07

Stops at every station

Once the tugger is fully load, it stops at every following station until the warehouse.

08

Forecasts

The tugger stops at each station, even if it does not need to load any unit.

01

Constant speed

Assuming a constant speed is not realistic. Indeed, many variables, such as loads' weights, do actually interfere with the tuggers' speed. This is why we had to keep them into consideration. Thus, we have assumed through an educated guess, that the tugger has a maximum and a minimum speed, reached respectively when emptily and fully loaded. Moreover, we evaluated the rate through which the speed decreases in relationship with the weight, as linearly dependent on the ratio between weight carried and the maximum weight capacity times the differences between the maximum reachable speed.

```
speed_min = 1.2
speed_max = 1.6
speed = speed_max - weight / max_weight * (speed_max - speed_min)
return speed
```

02

Compute Time

First of all, we have noticed that in the base case code a random term, intended to take into account unpredictable obstacles on the tugger's route, was added to the traditional hourly law of the rectilinear uniform motion. Since the randomness was independent from any variable included in the system, the probability of finding any delays did not depend on the length of the path travelled by the tugger. An extremely inaccurate and unrealistic assumption. Thus, we designed the function in such a way that the additional time implied by any impediments, such as traffic, was dependent by the distance criss-crossed. Moreover, the random penalty added in our case differs in terms of units of measurement from the base case. Indeed, in the latter it was expressed in minutes, whereas after our modifications, in seconds per meter.

Furthermore, we assessed the probability of meeting obstacles as dependent in some way on the factory's area where the tugger is moving. In fact, it has to be larger in those areas in which there is a larger operations' concentration, for instance between two horizontally close lines such as line 1 and 2.

Lastly, we took into consideration the time to reach steady state speed (acceleration) and to stop the train, thus reaching null velocity (deceleration), by the addition of a fixed term (in seconds). The values has been estimated according to a technical data sheets of similar technologies.

Note: Despite having thought that the acceleration and deceleration were dependent on load, we had to restate our believes according to the information that we have gathered .

```
def compute_time(distance: float, speed: float, nextline: int = 0):

    if nextline in (1, 2, 4):
        return distance / speed + random.uniform(0.0, 1.0) * distance + 8
    else:
        return distance / speed + random.uniform(0.0, 0.5) * distance + 8
```

03

Compute Energy

To this regard, we have opted to split the computation of energy consumption according the specific activity carried out by the tugger train, namely rectilinear movements and (un)loading. Indeed, by considering solely a unique function to compute energy consumption as it was in the base case scenario, the energy usage during (un)loading of a UL could have been larger than the one required for horizontal transportation. Something far from reality.
Considering the first, the mathematical formulation has not been modified with respect to the base case, but the dependencies have changed. Indeed, now, time does not depend solely on speed and distance, but on carried weight as well.

```
energy = consumption / 3600 * time
return energy
```

With respect to the second, we exploited the second Newtonian law to compute the force applied to a Unit Load during the activity of (un)loading which has been multiplied by the length of the UL (1.2 meters considering the traditional Euro Pallet) to obtain the work done (Joule). This has been subsequently transformed in KWh by having divided the result by 3600 seconds and by a factor of 10^3. Lastly, the resulting term has been inflated considering eventual friction forces and other energy dispersion, such as the losses incurred during the transformation from electric (from the battery) to kinetic energy (reels rolls).
Note that, in order to keep the consumption calculi as simple as possible, we have assumed that the consumption depends solely on weight (un)loaded and not on the time required to perform the (un)loading activities.

```
def compute_energy_loading(weight: float):
    energy = 1.2 * weight * 9.81 / 3600000 * 1.2
    return energy
```

04

Charging conditions

In the as-is model, the tugger train moves to the charging station when its remaining energy is slightly less than 40% of the battery's capacity although it requires way less to perform a trip.
Thus we have decided to understand, first, which would have been the energy consumption in the worst-case scenario, i.e., where the tugger stops at each line independently from its current remaining capacity, it's fully loaded from the very start (unreal but prudent scenario) until it unloads itself at the warehouse and reaches the farthest charging station.

```
def charge_threshold(self):
    time = 0
    line_output_for_charging_x = [0]+lines_output_points_x+[0]
    line_output_for_charging_y = [80]+lines_output_points_y+[80]
    for i in range(5):
        time += u.compute_time(u.compute_distance(x1=line_output_for_charging_x[i],
                               x2=line_output_for_charging_x[i+1],
                               y1=line_output_for_charging_y[i],
                               y2=line_output_for_charging_y[i+1]),
                               speed=u.compute_speed(weight=self.weight_capacity),
                               random_flag=False)
    time += 190+u.compute_time(u.compute_distance(x1=0, x2=0, y1=80, y2=10), speed=u.compute_speed(weight=0), random_flag=False)
    est_consumption = u.compute_energy(time=time)+u.compute_energy_loading(weight=self.weight_capacity)*2
    return est_consumption
```

```
def check_charge(self):
    if self.remaining_energy < self.charge_threshold():
```

The function charge_threshold has been created to detect a more realistic threshold after which the tugger has to visit one of the two charging stations. This latter has still been designed in such a way that the tugger will opt to charge by comparing the current battery load with the maximum consumption possible considering the aforementioned trip.

05

Loading more than 1 Unit Load

In the as-is model, the tugger is not allowed to pick up more than a Unit Load per line, despite not being an impossible scenario. Thus, we have opted, for efficiency purposes to change this aspect through a simple «while» loop, inside the tugger train's move() function, in which 3 conditions need to be respected:

1. If there are any units to be loaded
2. If the tugger can load an extra UL
3. If the extra UL weight is such that the maximum capacity in terms of weight is not exceeded

```
while self.model.schedule_lines.agents[self.next_line].UL_in_buffer >= 1:
    if self.load < self.capacity:
        if (self.weight + output_weight[self.next_line]) <= self.weight_capacity:
            print("\n"+ self.unique_id, "going to line", self.next_line, "and picking up a Unit Load")
            loading_time = random.uniform(30, 60)
            self.task_endtime += loading_time
            self.remaining_energy -= u.compute_energy_loading(output_weight[self.next_line])
            self.model.schedule_lines.agents[self.next_line].UL_in_buffer -= 1
            self.load += 1
            self.weight += output_weight[self.next_line]
        else:
            print("\n"+ self.unique_id, "- Not enough weight capacity left.")
            break
    else:
        print("\n"+ self.unique_id, "going to line", self.next_line, "- Not enough loading capacity left."))
        break
```

06

Non-random charging station picking

Moreover, we have thought that the charging station should not be chosen randomly but rather according to the time the tugger should wait before being charged. In simple words, if there were only 2 tuggers, we believe that an intelligent tugger would opt for the charging station that is not occupied. This is something required whenever we have a model that is populated by more than one tugger.

```
self.selected_charging_station = self.model.schedule_stations.agents.index(min(self.model.schedule_stations.agents,
                                                                                key=lambda x: x.waiting_time))
```

07

Stops at every station

In the as-is model, the tugger train, once fully loaded in terms of space capacity or weight capacity, stops at every following station checking whether it is able to load a further unit or not.
Having thought thoroughly about how to solve the problem, we have recognized that after having done so, another problem would have arose: calculating the distances properly. Indeed, by simply changing the stopping aspect, the tugger would have not been able to take into consideration the distances related to all the intermediate lines. For example, if the tugger finds itself at station 1 and is already fully loaded, thus headed to the warehouse, it would be required to take into consideration the cumulative distances from station 1 to station 2, from station 2 to station 3, and so on until its final destination. This could have been done through a *for loop* of *distance_next_stop* but, despite appearing seemingly easy, would have required a complete refactoring of the code, starting from the *move function*. Thus, we have gone through a trade-off analysis which resulted in the withdrawal of the proposal. In fact, the benefits that the model would have obtained (e.g. reduction in travelling time due to avoiding acceleration and deceleration times) did not outmatched the costs of performing such task in terms of coding's effort.

08

Forecasts

The tugger train stops at each station because in the as-is situation is not able to forecast its path, and therefore optimize it by avoiding useless stops. We took the decision of not solving this problem due to its complexity. Anyhow, we have identified the main steps to solve it. Firstly, each train must be able to forecast all buffers' situation to have visibility on the overall system. This problem includes many other sub-problems.

1.1 Variability of time of the tugger: When computing time, the tugger train needs to sum the standard time with a random factor, that depends on the distance in the to-be model and considers extra time due to the possibility of meeting obstacles (see Validation 02). So, it is necessary to define a unique way to consider this variability when the tugger forecasts how many units will be present in the buffer of lines when it arrives.

1.2 Priority rules: It is necessary to define some priority rules for the picking of items. For example, to minimize the idle time would be better to give a higher priority to the station that has a longer cycle time, but we have to consider some secondary rules such as the maximum weight that the tugger can carry in order to avoid bottlenecks as much as possible.

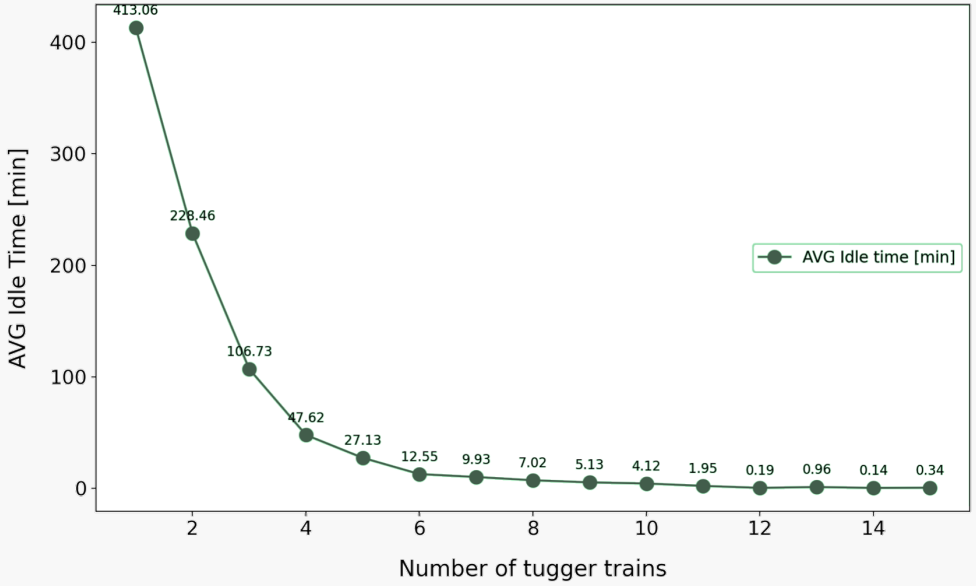
1.3 When forecasting: We identified two specific moments when doing forecasts. The first one concerns the moment after the unloading phase in case the tugger has still enough charge to another full lap. The second one concerns the moment right after it has completed the charging phase. A prerogative for making predictions is that the train knows its position.

1.4 Update of the system: Once the tugger has decided which items to load and therefore its path, the system will be automatically updated to avoid conflicts with other tuggers. For instance, if tugger_1 at time 0 plans to pick at time 3 a unit at line 1, at time 0 the line 1's buffer will be updated as the unit has already been picked. By doing so, the other tuggers would not plan to pick the same unit.

The second macro problem consists of the *function move*: it needs to be redefined to allow the train to go directly to the next station in which it will load a unit, without stopping at every line.

OUTCOME

After having corrected the model and improved its ability to represent the reality we performed the research for the number of tugger train to satisfy the conditions to have an average idle time of the bottling lines under 5 minutes, on average. The major constraint here consists of balancing the precision of the simulations and the computing resources needed. We opted for analysing the performance of the system from 1 tugger train to 15.
50 runs with every configuration were run (750 runs in total), to address the high variability of the system, and then the average performance of the 50 runs has been evaluated to draw conclusions about the optimal number of tuggers trains needed.
The graph shows the observed behavior for the increasing number of tuggers trains. We can see that there is a reduction of the average idle time along with an increase of the number of tugger trains. The performance converges to a value close to 0.5 minutes after 10 tugger trains. The graph also shows that, despite the high number of simulation per level, the results are still variable, and the convergence is obtained with a very high number of tugger trains.



We think that acquiring 10 tugger trains could not be justified from an economic point of view, since after 5 tugger trains it appears that the marginal increase of the performances adding an extra tugger flattens considerably. Given that, due to the variability of the system, taking 5 tuggers trains could be risky, therefore we think that a good balance between risks and costs is given by 6 tugger trains. For a deeper analysis of the system, we believe it might be more reasonable to explore if we can reach similar results varying other factors and with an overall cheaper solution.
Moreover, being the idle time with 10 tuggers already low, we could not measure the impact of the change of other parameters.
For these reasons, in the second part of the analysis we will consider a system with 6 tuggers and not with 10 tuggers.

ASSIGNMENT PART 1
TEAM 1

For a more comprehensive view of the work done, the script of the code is available. In addition, to facilitate understanding of the script, the file "*HOW-TO*“ better explains which simulations can be performed through the script and how.