Code Listing

Ioana Iulia Lucaci

May 17, 2021

1 auction.txt

```
# Simulation
Number of Rounds = 10
Data Type = 'Efficiency','Speed','Revenue'
Agent Type = 'Auction Types'

# Auction
Number of Bidders = 100
Auction Types = ED,DE,E,D
Reserve Price = 2000
Auctioneer Type = A

# Bidders' Type Percentages
A = 25
B = 25
C = 25
D = 25
```

2 main.py

```
"""The main file."""

import time
from datetime import datetime
from auction import *
import data_analyser as da
import agents_factory as af
import metrics_writer as mw
import parameters_reader as pr


def extract_information():
    global file_name, headers, simulator, parameters, bidders, auctioneer_types, auction_types

    # These are the headers they correspond to in terms of how the auction is coded.
    headers = ('Auction Types', 'Auctioneer Type', 'A', 'B', 'C', 'D', 'Winner Type', 'Starting Bid', 'Revenue',
               'Winner Satisfaction', 'Auctioneer Satisfaction', 'Social Welfare', 'Efficiency', 'Speed')

    # Get the information from the .txt file
    simulator, parameters, bidders = pr.read_parameters(headers)
    auctioneer_types = parameters["Auctioneer Type"]
    auction_types = parameters["Auction Types"].split(',')

    # Set the file name for the metrics
    today = datetime.today()
    file_name = "metrics " + today.strftime('%d-%m-%y') + ".csv"


def run_simulation():
    global agents_factory
    mw.write_metrics(headers, file_name)

    # Start the counter and begin the simulation
    start = time.time()
    list_of_auctions = []
    for counter in range(0, simulator["Number of Rounds"]):
        # For every type of auction, we must use the same bidders
        agents_factory = af.AgentsFactory(parameters, bidders)

        # For every auctioneer type, we will use the same bidders but with updated budget and a new auctioneer
        for auctioneer_type in auctioneer_types:
            parameters["Auctioneer Type"] = auctioneer_type

            agents_factory.create_auctioneer(parameters)

            # Loop dealing with all the different auction types
```

```
            for auction_combination in auction_types:
                parameters["Auction Types"] = auction_combination

                agents_factory.update_prices(parameters["Auction Types"])

                model = Auction(parameters, bidders, agents_factory)
                model.step()
                list_of_auctions.append(model)

        # To avoid caching everything for big number of simulations, we will write every 500 rounds in the .csv
        ↪  file
        if len(list_of_auctions) > 500:
            mw.write_simulators(file_name, list_of_auctions, headers)
            list_of_auctions = []
    # To avoid losing any information, we need to check at the end that we wrote every round
    if len(list_of_auctions) > 0:
        mw.write_simulators(file_name, list_of_auctions, headers)
    end = time.time()
    print("I am done with the models! It took {0} seconds.\n The information is saved in the file
    ↪  '{1}'.".format(end - start, file_name))


def analyse_data():
    # Since the data can be simulated from different aspects, we need to make sure the labels for the agents will be
    ↪  correct
    data_type = simulator["Data Type"]
    agent_type = simulator["Agent Type"]
    types = {"Winner Type": ["A", "B", "C", "D"]}
    if "Auctioneer Type" in agent_type:
        types["Auctioneer Type"] = auctioneer_types
    if "Auction Types" in agent_type:
        types["Auction Types"] = auction_types
    da.analyse_data(file_name, list(data_type.split(',')), list(agent_type.split(',')), types)


if __name__ == "__main__":
    extract_information()
    run_simulation()
    analyse_data()
```

3 agents_factory.py

```
"""
Class in charge of creating agents. Here, the risk, base rate and utility can be updated.
"""
import random


class AgentsFactory:
    """Class in charge of creating the bidders and auctions."""
    def __init__(self, parameters, bidder_types):
        """
        The initialising function.

        :param parameters: The agents' parameters
        :param bidder_types: The bidder types
        """
        self.auctioneer = {}

        # Create bidders
        number_of_bidders = parameters["Number of Bidders"]

        self.bidders = []

        # For every type, we see how many we have to create
        for bidder_type in bidder_types.keys():
            number_of_bidders_type = int(bidder_types[bidder_type] * number_of_bidders / 100)

            # We create the number of bidders for a specific type
            for counter in range(number_of_bidders_type):
                budget = 0

                risk = round(random.uniform(0.1, 0.9), 1)
                base_rate = round(random.uniform(0.01, 0.1), 2)
                utility = round(random.uniform(0.1, 0.99), 1)
                bidder_information = (risk, base_rate, utility)

                self.bidders.append(
                    {
                        "budget": budget,
                        "bidder_type": bidder_type,
                        "bidder_information": bidder_information
                    }
                )
```

```
    def create_auctioneer(self, parameters):
        """
        Creates a new auctioneer.

        :param parameters: parameters for the new auctioneer.
        """
        # Create auctioneer
        reserve_price = parameters["Reserve Price"]
        base_rate = round(random.uniform(0.05, 0.1), 2)

        self.auctioneer = {"starting_bid": 0, "reserve_price": reserve_price,
                           "auctioneer_type": parameters["Auctioneer Type"],
                           "base_rate": base_rate}

    def update_prices(self, current_auction):
        """
        Updates the budget and starting price depending on the auction type.

        :param current_auction: the current auction type
        :return:
        """
        current_auction = list(current_auction.split(','))[0]
        base_rate = self.auctioneer["base_rate"]
        reserve_price = self.auctioneer["reserve_price"]

        # Update auction starting bid
        starting_bid = reserve_price * (1 + base_rate)

        if current_auction == 'D':
            multiplier = round(random.uniform(1.3, 2), 1)
            starting_bid = reserve_price * (multiplier + base_rate)

        if current_auction == 'E':
            multiplier = round(random.uniform(1, 1.1), 1)
            starting_bid = reserve_price * multiplier

        self.auctioneer["starting_bid"] = starting_bid

        # Update bidders' budgets
        updated_bidders = []

        for bidder in self.bidders:
            bidder["budget"] = random.randint(int(reserve_price * 1.1), int(reserve_price * 1.3))
            updated_bidders.append(bidder)

        self.bidders = updated_bidders
```

4 auction_information.py

```
"""
The functions used by the bidder and auctioneer types that can be changed at any point.
"""
import math


def update_rate(profile, old_rate, utility, risk):
    """
    Updates the rate of the agent.

    :param profile: the agent's profile.
    :param old_rate: the previous rate
    :param utility: the utility
    :param risk: the risk
    :return: the new rate
    """
    functions = {
        'A': lambda rate, utility, risk: rate,
        'B': lambda rate, utility, risk: rate * risk + utility,
        'C': lambda rate, utility, risk: utility - rate * risk,
        'D': lambda rate, utility, risk: math.sin(rate) + (utility + risk)
    }

    new_rate = functions[profile](old_rate, utility, risk)

    # We ensure rate can only be between 0 and 0.1
    while new_rate > 0.1:
        difference = (new_rate - old_rate) / 100

        new_rate = old_rate + difference

    while new_rate <= 0:
        difference = old_rate / 100
```

```
        new_rate = old_rate - difference

    return new_rate
```

5 auction.py

```
"""
The auction class for the simulation.
"""
import random
from mesa import Model
from mesa.time import SimultaneousActivation
from bidder import Bidder
from auctioneer import Auctioneer


class Auction(Model):
    """Model that simulates an auction situation. Used to dictate the communication between auctioneers and
    ↪  bidders."""

    def __init__(self, parameters, bidder_types, agents_factory):
        """
        Initialisation function for the auction model.

        :param parameters: all the required parameters for the auction.
        :param bidder_types: the types of bidders to join this auction alongside their percentage
        """

        super().__init__()
        self.information = parameters | bidder_types

        self.auction_types = parameters["Auction Types"]
        self.current_auction = self.auction_types[0]

        # Create auctioneer
        auct_info = agents_factory.auctioneer

        self.information["Starting Bid"] = auct_info["starting_bid"]

        self.auctioneer = Auctioneer(-1, auct_info["starting_bid"], auct_info["reserve_price"],
                                     parameters["Auctioneer Type"],
                                     auct_info["base_rate"], self)

        # Create bidders
        self.bid_schedule = SimultaneousActivation(self)

        self.rounds = 0

        bidders_info = agents_factory.bidders
        for counter in range(0, len(bidders_info)):
            bidder_info = bidders_info[counter]
            bidder = Bidder(counter, bidder_info["budget"], bidder_info["bidder_type"],
                            bidder_info["bidder_information"], self)
            self.bid_schedule.add(bidder)

    def step(self):
        """ Simulates an auction or combination of auctions."""
        # First auction type
        self.select_auction_type()

        if len(self.auction_types) == 2:
            # Change the auction type and auctioneer information
            self.current_auction = self.auction_types[1]

            new_base_rate = round(random.uniform(0.01, 0.05), 2)
            self.auctioneer.update_auctioneer(new_base_rate)

            # Second auction type
            self.select_auction_type()

        # Update information
        self.update_metrics()

    def select_auction_type(self):
        """ Determines which auction to be selected. """
        if self.current_auction == 'E' or self.current_auction == 'D':
            self.multi_round_auction()
        elif self.current_auction == 'F' or self.current_auction == 'V':
            self.one_shot_auction(True)

    def multi_round_auction(self):
        """ Simulates an auction with multiple rounds auction."""
        first_round = True
        while True:
```

```
            if self.auctioneer.move_next:
                break
            self.one_shot_auction(first_round)
            first_round = False

    def one_shot_auction(self, first_round):
        """ Simulates an auction with only one round."""
        self.rounds = self.rounds + 1
        self.auctioneer.auction()
        self.bid_schedule.step()
        self.auctioneer.decide(first_round)

    def update_metrics(self):
        """Updates the metrics dictionary."""

        if self.auctioneer.winner == -1 or self.auctioneer.winning_bid < self.information['Reserve Price']:
            # No Winner
            self.information['Winner Type'] = 'auctioneer'
            self.information['Revenue'] = - self.information['Reserve Price']
            self.information['Winner Satisfaction'] = 0
            self.information['Auctioneer Satisfaction'] = -1
            self.information['Efficiency'] = -1 / self.rounds
        else:
            # There is a winner
            winning_bidder = \
                list(filter(lambda bidder: bidder.unique_id == self.auctioneer.winner, self.bid_schedule.agents))
            self.information['Winner Type'] = winning_bidder[0].bidder_type
            self.information['Revenue'] = self.auctioneer.winning_bid - self.information["Reserve Price"]

            self.information['Winner Satisfaction'] = (winning_bidder[
                                                       0].budget - self.auctioneer.winning_bid) /
                                                       ↪  self.auctioneer.winning_bid
            self.information['Auctioneer Satisfaction'] = (self.auctioneer.winning_bid - self.information[
                'Reserve Price']) / self.information['Reserve Price']
            self.information['Efficiency'] = abs(
                (self.information['Winner Satisfaction'] - self.information['Auctioneer Satisfaction'])) /
                ↪  self.rounds

        # Extra Metrics
        self.information['Social Welfare'] = self.information['Revenue'] / len(self.bid_schedule.agents)
        self.information['Speed'] = self.rounds
```

6 auctioneer.py

```
"""
The auctioneer class for the simulation.
"""
import random
from mesa import Agent
import auction_information as info


class Auctioneer(Agent):
    """Agents that simulates an auctioneer of a certain type."""

    def __init__(self, unique_id, price, reserved_price, auctioneer_type, base_rate, model):
        """
        Initialisation function for the auctioneer model.

        :param unique_id: the id of the auctioneer
        :param price: the starting price price
        :param reserved_price: the reserved price
        :param auctioneer_type: the type of the auctioneer
        :param base_rate: the starting rate
        :param model: the auction model it belongs to
        """
        super().__init__(unique_id, model)
        # initial information
        self.price = price
        self.reserved_price = reserved_price
        self.auctioneer_type = auctioneer_type
        self.winner = unique_id
        self.rate = base_rate

        # initializations
        self.existing_bids = {}
        self.previous_bids = {}
        self.winning_bid = 0
        self.move_next = False
        self.previous_highest_bid = 0
        self.previous_winner = unique_id

    def auction(self):
        """ Simulates an auctioneer's call."""
```

```
        # We keep track of the previous highest bid
        if self.previous_highest_bid < self.winning_bid:
            self.previous_winner = self.winner
            self.previous_highest_bid = self.winning_bid

        # And we keep track of the current highest bid
        if len(self.existing_bids) > 0:
            self.existing_bids = dict(sorted(self.existing_bids.items(), key=lambda item: item[1], reverse=True))

            # We update the current 'winner' only if it's higher than the current price or we're in a Dutch auction
            if self.price < max(self.existing_bids.values()) or self.model.current_auction == 'D':
                self.winner = list(self.existing_bids.keys())[0]
                self.winning_bid = self.existing_bids[list(self.existing_bids.keys())[0]]

        self.previous_bids = self.existing_bids
        self.existing_bids = {}

    def decide(self, first_round):
        """ Determines whether to change the current bid or determine the winner."""
        if len(self.existing_bids) == 0 and not first_round and self.model.current_auction != 'D':
            self.determine_winner()
        else:
            self.change_current_bid()

    def change_current_bid(self):
        """ Determines how to change the current bid based on the auction type."""

        self.existing_bids = dict(sorted(self.existing_bids.items(), key=lambda item: item[1], reverse=True))
        highest_bid = self.existing_bids[list(self.existing_bids.keys())[0]] if len(self.existing_bids) > 0 else 0

        # If the current winning bid is smaller than the highest bid, we update accordingly
        if self.winning_bid < highest_bid:
            self.previous_winner = self.winner
            self.previous_highest_bid = self.winning_bid
            self.winner = list(self.existing_bids.keys())[0]
            self.winning_bid = highest_bid

        # After that, we choose which function is best for the auction type
        if self.model.current_auction == 'F':
            self.sealedbid_auction()
        if self.model.current_auction == 'V':
            self.vickrey_auction()

        self.rate = info.update_rate(self.auctioneer_type, self.rate, 1, 1)

        # The Dutch has a special case where the price can decrease with no bidders
        if highest_bid < self.reserved_price:
            if self.model.current_auction == 'D':
                self.dutch_auction(highest_bid)
            else:
                self.determine_winner()
        else:
            if self.model.current_auction == 'E':
                self.english_auction(highest_bid)
            elif self.model.current_auction == 'D':
                self.dutch_auction(highest_bid)

    def determine_winner(self):
        """ Determines the winner and how much they have to pay."""
        previous_bids_max = max(self.previous_bids.values()) if len(self.previous_bids) > 0 else 0

        if self.reserved_price < previous_bids_max and self.winning_bid < previous_bids_max:
            self.winner = list(self.previous_bids.keys())[0]
            self.winning_bid = self.previous_bids[self.winner]
        elif self.previous_winner != self.unique_id and self.winning_bid < self.previous_highest_bid:
            self.winner = self.previous_winner
            self.winning_bid = self.previous_highest_bid

        self.move_next = True

    def english_auction(self, highest_bid):
        """ Simulates an English auction. We take the highest current bid and add the rate to it."""
        next_highest_bid = highest_bid * (1 + self.rate)
        self.price = next_highest_bid

    def dutch_auction(self, highest_bid):
        """ Simulates a Dutch auction.We take the highest current bid and add the rate to it."""
        price = max(self.price * (1 - self.rate), highest_bid * (1 + self.rate))

        # If we reached below the reserved price, we exit
        if self.reserved_price > price:
            self.move_next = True
        # If we're in the risk of raising the price or going below what the winning bid currently is, determine
        ↪  winner
        elif self.price < price or price < self.winning_bid:
```

```
            self.determine_winner()
        else:
            self.price = price

    def sealedbid_auction(self):
        """ Simulates a First Price Sealed Bid auction. Since it is one-shot, we only need the winner."""
        self.existing_bids = dict(sorted(self.existing_bids.items(), key=lambda item: item[1], reverse=True))
        self.winner = list(self.existing_bids.keys())[0]
        self.winning_bid = self.existing_bids[self.winner]

        if self.model.current_auction != self.model.auction_types[-1]:
            self.move_next = True

    def vickrey_auction(self):
        """
        Simulates a Vickrey auction. Since it's one-shot, we only need the winner, but the price it'll pay will be
        the second highest bid.
        """
        self.existing_bids = dict(sorted(self.existing_bids.items(), key=lambda item: item[1], reverse=True))
        self.winner = list(self.existing_bids.keys())[0]

        if len(self.existing_bids) >= 2:
            self.winning_bid = self.existing_bids[list(self.existing_bids.keys())[1]]
        else:
            self.winning_bid = self.existing_bids[list(self.existing_bids.keys())[0]]

        if self.model.current_auction != self.model.auction_types[-1]:
            self.move_next = True

    def update_auctioneer(self, new_base_rate):
        """ Updates the auctioneer after the first round of auctions"""
        self.move_next = False
        self.rate = new_base_rate
        self.previous_bids = {}

        # We take the highest bid as the reserved price. This is the most the bidders were willing to pay last
        ↪  auction.
        self.reserved_price = max(self.winning_bid, self.previous_highest_bid, self.price)

        if self.model.auction_types[0] == 'D':
            self.price = self.reserved_price * (1 + self.rate)

        if self.model.auction_types[0] in ['F', 'V']:
            self.price = self.reserved_price

        if self.model.auction_types[-1] == 'D':
            self.reserved_price = max(self.winning_bid, self.previous_highest_bid)
            multiplier = round(random.uniform(1.3, 2), 1)
            self.price = self.reserved_price * multiplier
```

7 bidder.py

```
"""
The bidder class for the simulation.
"""
import random
from mesa import Agent
import auction_information as info


class Bidder(Agent):
    """Agent that simulates a bidder."""

    def __init__(self, unique_id, budget, bidder_type, bidder_information, model):
        """
        Initialisation function for the bidder agent.

        :param unique_id: the id of the bidder
        :param budget: the bidder's budget
        :param bidder_type: the type of the bidder
        :param bidder_information: tuple containing risk, base rate and utility
        :param model: the auction model it belongs to
        """
        super().__init__(unique_id, model)
        self.budget = budget
        self.bidder_type = bidder_type
        (self.risk, self.rate, self.utility) = bidder_information

    def step(self):
        """ Takes a step in an auction."""

        current_bid = self.model.auctioneer.price
        if self.decision_to_bid(current_bid):
            self.counter_proposal(current_bid)
```

```
    def advance(self):
        """ Advances the bidder through the model."""
        pass

    def decision_to_bid(self, current_bid):
        """
        Decides if the bidder will bid.

        :param current_bid: the current bid of the auctioneer
        :returns: True if it decides to bid; otherwise False
        """

        if self.model.current_auction in ['F', 'V']:
            return True

        if current_bid > self.budget:
            return False

        chance = random.random()
        if chance < self.risk:
            return True
        else:
            return False

    def counter_proposal(self, current_bid):
        """
        Decides how much to bid.

        :param current_bid: The current bid of the auctioneer.
        """
        personal_bid = 0
        self.rate = info.update_rate(self.bidder_type, self.rate, self.utility, self.risk)

        if self.model.current_auction == 'E':
            personal_bid = self.english_auction(current_bid)
        elif self.model.current_auction == 'D':
            personal_bid = self.dutch_auction(current_bid)
        elif self.model.current_auction == 'F':
            personal_bid = self.sealedbid_auction()
        elif self.model.current_auction == 'V':
            personal_bid = self.vickrey_auction()

        if personal_bid > 0:
            self.model.auctioneer.existing_bids[self.unique_id] = personal_bid

    def english_auction(self, current_bid):
        """
        Simulates an English auction.

        :param current_bid: The current bid of the auctioneer.
        :return: the personal bid to submit.
        """
        personal_bid = current_bid + current_bid * self.rate
        if personal_bid > self.budget:
            return -10
        return personal_bid

    def dutch_auction(self, current_bid):
        """
        Simulates a Dutch auction.

        :param current_bid: The current bid of the auctioneer.
        :return: the personal bid to submit.
        """
        personal_bid = current_bid - current_bid * self.rate

        if personal_bid > self.budget:
            personal_bid = self.budget
        return personal_bid

    def sealedbid_auction(self):
        """
        Simulates a first-price sealed-bid auction.

        :return: the personal bid to submit.
        """
        chance = random.random()
        if chance <= self.risk:
            return self.budget
        else:
            return min(self.budget * (1 - self.rate), self.budget)

    def vickrey_auction(self):
        """
        Simulates a Vickrey auction.
```

```
        :return: the personal bid to submit.
        """
        chance = random.random()
        if chance <= self.risk:
            return self.budget
        else:
            return min(self.budget * (1 - self.rate), self.budget)
```

8 data_analyser.py

```
"""
Analyses the given data in terms of ANOVA test and visualises it.
"""
from scipy import stats
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns


def analyse_data(file_name, data_types, agent_types, types):
    """
    Analyses the data in terms of ANOVA tests and visualisation.

    :param file_name: the file name to be analysed
    :param data_types: the metric to be analysed
    :param agent_types: the agent type to be analysed
    :param types: the types that can be found in the file for the agent type
    :return:
    """
    metrics_data = pd.read_csv(file_name)

    for agent_type in agent_types:
        for data_type in data_types:
            csv_data = {}
            # Separating the data based on the winner type and extracting only what's important
            for element_type in types[agent_type]:
                csv_data[element_type] = list(metrics_data[data_type][metrics_data[agent_type] == element_type])

            visualise_data(csv_data, types[agent_type], data_type, agent_type)

            print("-----------------------------------------------------------")
            print("ANOVA test for '{1}' in terms of '{0}':".format(agent_type, data_type))
            anova_test_data(csv_data, types[agent_type])
            print("-----------------------------------------------------------")


def anova_test_data(csv_data, types):
    """
    Applies ANOVA tests to the data

    :param csv_data: The data to go through the ANOVA test.
    :param types: The different groups to be tested.
    :return:
    """
    f_statistics, p = 0, 0

    if len(types) == 2:
        f_statistics, p = stats.f_oneway(csv_data[types[0]], csv_data[types[1]])
    elif len(types) == 3:
        f_statistics, p = stats.f_oneway(csv_data[types[0]], csv_data[types[1]], csv_data[types[2]])
    elif len(types) == 4:
        f_statistics, p = stats.f_oneway(csv_data[types[0]], csv_data[types[1]], csv_data[types[2]],
        ↪  csv_data[types[3]])

    print('f_statistics={0}; p={1}'.format(f_statistics, p))


def visualise_data(csv_data, types, data_type, agent_type):
    """
    Prints out plots to show the metrics data.

    :param csv_data: The information to be printed.
    :param types: What types we are analysing
    :param data_type: the metric to be analysed
    :param agent_type: the agent type to be analysed
    """
    # Visualisation
    data_for_frame = {}
    data_for_bar = []

    for element in types:
        data_for_frame[element] = csv_data[element]
        data_for_bar.append(sum(csv_data[element]) / len(csv_data[element]))
```

```
    data_to_plot = pd.DataFrame(data_for_frame)
    data_to_bar_plot = pd.DataFrame({agent_type: types, data_type: data_for_bar})

    # Shows a Box with whiskers
    boxplot = sns.boxplot(x='variable', y='value', data=pd.melt(data_to_plot), order=data_for_frame.keys())

    boxplot.set_xlabel("", fontsize=16)
    boxplot.set_ylabel("", fontsize=16)

    for tick in boxplot.get_xticklabels():
        tick.set_fontsize(16)
    for tick in boxplot.get_yticklabels():
        tick.set_fontsize(16)

    # Shows a Bar Chart
    data_to_bar_plot.plot.bar(x=agent_type, y=data_type, rot=0)

    plt.show()
```

9 metrics_writer.py

```
"""Functions in charge of exporting the metrics to csv."""
import csv


def write_metrics(headers, file_name):
    """
    Writes the metrics in a csv file

    :param headers: the headers for the csv file.
    :param file_name: the name of the file.

    """

    with open(file_name, mode='w+', newline='') as metrics_file:
        metrics_writer = csv.writer(metrics_file, delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)

        # First, write the table header.
        metrics_writer.writerow(headers)


def write_simulators(file_name, list_of_auctions, headers):
    """
    Writes the simulators information in a csv file

    :param headers: The headers for the csv file.
    :param file_name: the name of the file.
    :param list_of_auctions: the list of auctions to be exported in the csv file
    """
    with open(file_name, mode='a', newline='') as metrics_file:
        metrics_writer = csv.writer(metrics_file, delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)

        # Then, for each auction, extract the information
        for auction in list_of_auctions:
            to_be_published = []
            for header in headers:
                to_be_published.append(auction.information[header])
            metrics_writer.writerow(to_be_published)
```

10 parameters_reader.py

```
"""Functions in charge of reading the auction.txt file"""
import re


def read_parameters(headers):
    """
    Reads the information about an auction from a file.
    Parameters and bidder % are separated for ease when it comes to creating the bidders.

    :parameter headers: The headers that will be published in the csv file.
    :return: simulator: The information for the simulator.
             parameters: The parameters of the auction
             bidders: The bidder types and their percentages.
    """

    # Declaration of all the information we need
    parameters = {"Number of Bidders": None, "Auction Types": None, "Reserve Price": None, "Auctioneer Type":
    ↪  None}
    bidders = {"A": 0, "B": 0, "C": 0, "D": 0}
```

```
    simulator = {"Number of Rounds": None, "Data Type": None, "Agent Type": None}

    # Populate the dictionaries that need to be returned
    with open("auction.txt", "r") as auction_info:

        for line in auction_info:
            line = line.rstrip()
            # Empty lines and comment lines (starting with '#') are ignored.
            if '#' in line or len(line) == 0:
                continue

            # The regex is looking for the shape 'name = value'
            pattern = r'((\w+ )*\w+) = (([EDVF]{1,2}(,[EDVF]{1,2})*)|([ABCD](,[ABCD])*)|\d+|(\'(\w+
            ↪  )*\w+\')(,\'(\w+ )*\w+\')*)'
            result = re.search(pattern, line)
            try:
                name = result.group(1)
                value = result.group(3)
            except Exception as e:
                raise Exception(
                    "The data could not be parsed. Error raised at the following line: \"{0}\"".format(line), e)
            if result.group(0) is not line:
                raise Exception(
                    "The data could not be parsed. Error raised at the following line: \"{0}\"".format(line))

            format_data(line, name, value, parameters, simulator, bidders)

    check_information(bidders, headers, parameters, simulator)

    return simulator, parameters, bidders


def format_data(line, name, value, parameters, simulator, bidders):
    """
    Formats the data in the correct dictionary.

    :param line: The line currently being checked
    :param name: The name of the 'name = value' pair
    :param value: The value of the 'name = value' pair
    :param parameters: The parameters dictionary
    :param simulator: The simulator dictionary
    :param bidders: The bidders dictionary
    :return:
    """
    # Depending on which dictionary it belongs to, it'll be added based on the 'name' part
    if name in bidders.keys():
        bidders[name] = float(value)
        return

    # There are certain values that must be converted to either int or float
    try:
        if name == 'Auction Types':
            parameters[name] = value
        elif name == 'Auctioneer Type':
            parameters[name] = list(value.split(','))
        elif name == 'Number of Bidders':
            parameters[name] = int(value)
        elif name == "Number of Rounds":
            simulator[name] = int(value)
        elif name in simulator.keys():
            simulator[name] = value.replace('\'', '')
        else:
            parameters[name] = float(value)
    except Exception as e:
        raise Exception("The data could not be parsed. Error raised at the following line: \"{0}\"".format(line),
        ↪  e)


def check_information(bidders, headers, parameters, simulator):
    """
    Checks that all of the information is present.

    :param bidders: The bidder types and their percentages.
    :param headers: The headers that will be published in the csv file.
    :param parameters: The parameters of the auction
    :param simulator: The data for the simulator.
    :return:
    """
    if None in simulator.values() or None in parameters.values():
        list_of_missing = []
        for name, value in simulator.items():
            if value is None:
                list_of_missing.append(name)
        for name, value in parameters.items():
            if value is None:
                list_of_missing.append(name)
```

```
        raise Exception("Incomplete information. You are missing: {0}".format(list_of_missing))

    # Check that bidders add up to 100
    if sum(bidders.values()) != 100:
        raise Exception("Bidders percentages must add up to 100.")

    # Check that we will look for data for the graphs that exists in the file.
    for agent_type in simulator['Agent Type'].split(','):
        if agent_type not in headers:
            raise Exception("Agent type {0} not present in the csv file.".format(agent_type))
    for data_type in simulator['Data Type'].split(','):
        if data_type not in headers:
            raise Exception("Data type {0} not present in the csv file.".format(data_type))
```

11 parameters_writer.py

```
"""
Functions in charge of writing the parameters.
"""


def write_simulation_information(number_of_rounds, data_type, agent_type):
    """Writes the simulation information in the input file.

    :param number_of_rounds: The number of rounds
    :param data_type: The type of data
    :param agent_type: The type of agent
    """

    auction_file = open("auction.txt", "w")

    data_result = f"'{data_type[0]}'"
    for counter in range(1, len(data_type)):
        data_result = data_result + ",'{0}'".format(data_type[counter])

    agent_result = f"'{agent_type[0]}'"
    for counter in range(1, len(agent_type)):
        agent_result = agent_result + ",'{0}'".format(agent_type[counter])

    result = f"# Simulation\nNumber of Rounds = {number_of_rounds}\nData Type = {data_result}\nAgent Type =
    ↪  {agent_result}\n\n"
    auction_file.write(result)


def write_auction_information(number_of_bidders, auction_types, reserve_price, auctioneer_type):
    """ Writes the auction information in the input file.

    :param number_of_bidders: The number of bidders
    :param auction_types: The auction types
    :param reserve_price: The reserve price
    :param auctioneer_type: The auctioneer type
    """
    auction_file = open("auction.txt", "a")
    result = f"# Auction\nNumber of Bidders = {number_of_bidders}\nAuction Types = {auction_types}\nReserve Price
    ↪  = {reserve_price}\nAuctioneer Type = {auctioneer_type}\n\n"
    auction_file.write(result)


def write_bidders_percentages(bidder_a, bidder_b, bidder_c, bidder_d):
    """ Writes the percentages of bidders in the input file.

    :param bidder_a: The percentage of bidders of type A
    :param bidder_b: The percentage of bidders of type B
    :param bidder_c: The percentage of bidders of type C
    :param bidder_d: The percentage of bidders of type D
    """
    auction_file = open("auction.txt", "a")
    result = f"# Bidders' Type Percentages\nA = {bidder_a}\nB = {bidder_b}\nC = {bidder_c}\nD = {bidder_d}\n"
    auction_file.write(result)
```

12 parameters_writer.py

```
"""
Functions in charge of writing the parameters.
"""


def write_simulation_information(number_of_rounds, data_type, agent_type):
    """Writes the simulation information in the input file.

    :param number_of_rounds: The number of rounds
```

```
    :param data_type: The type of data
    :param agent_type: The type of agent
    """

    auction_file = open("auction.txt", "w")

    data_result = f"'{data_type[0]}'"
    for counter in range(1, len(data_type)):
        data_result = data_result + ",'{0}'".format(data_type[counter])

    agent_result = f"'{agent_type[0]}'"
    for counter in range(1, len(agent_type)):
        agent_result = agent_result + ",'{0}'".format(agent_type[counter])

    result = f"# Simulation\nNumber of Rounds = {number_of_rounds}\nData Type = {data_result}\nAgent Type =
    ↪  {agent_result}\n\n"
    auction_file.write(result)


def write_auction_information(number_of_bidders, auction_types, reserve_price, auctioneer_type):
    """ Writes the auction information in the input file.

    :param number_of_bidders: The number of bidders
    :param auction_types: The auction types
    :param reserve_price: The reserve price
    :param auctioneer_type: The auctioneer type
    """
    auction_file = open("auction.txt", "a")
    result = f"# Auction\nNumber of Bidders = {number_of_bidders}\nAuction Types = {auction_types}\nReserve Price
    ↪  = {reserve_price}\nAuctioneer Type = {auctioneer_type}\n\n"
    auction_file.write(result)


def write_bidders_percentages(bidder_a, bidder_b, bidder_c, bidder_d):
    """ Writes the percentages of bidders in the input file.

    :param bidder_a: The percentage of bidders of type A
    :param bidder_b: The percentage of bidders of type B
    :param bidder_c: The percentage of bidders of type C
    :param bidder_d: The percentage of bidders of type D
    """
    auction_file = open("auction.txt", "a")
    result = f"# Bidders' Type Percentages\nA = {bidder_a}\nB = {bidder_b}\nC = {bidder_c}\nD = {bidder_d}\n"
    auction_file.write(result)
```

13 interactive.ipynb

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "source": [
    "# Auction Simulator\n",
    "\n",
    "### Configuring the simulator\n",
    "To run the simulator, we first need to gather some information.\n",
    "\n",
    "First things first, we need to know what we will investigate.\n",
    "In data_type we will store what metrics we want to analyse, while in agent_type we store what agents we want
    ↪  to analyse the metrics for. The number of rounds determines how many simulations should be run for each
    ↪  combination of auction type and auctioneer."
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "import parameters_writer as pw\n",
    "\n",
    "number_of_rounds = 100\n",
    "data_type = ['Efficiency', 'Speed', 'Revenue']\n",
    "agent_type = ['Auction Types']\n",
    "\n",
    "pw.write_simulation_information(number_of_rounds, data_type, agent_type)"
   ],
   "metadata": {
    "collapsed": false,
    "pycharm": {
     "name": "#%%\n"
    }
```

```
 }
},
{
 "cell_type": "markdown",
 "source": [
  "There are 4 types of auctions that can be combined:\n",
  "- E - English\n",
  "- D - Dutch\n",
  "- F - First-price sealed-bid\n",
  "- V - Vickrey\n",
  "\n",
  "There are also four types of auctioneers, each with a different behaviour in terms of how the rate evolves
  ↪  over time:\n",
  "- *A* : constant function\n",
  "- *B* : increasing function\n",
  "- *C* : decreasing function\n",
  "- *D* : non-monotonous function\n",
  "\n",
  "We can choose multiple auction types and auctioneer types, but they must be separated by a comma and no
  ↪  space.\n",
  "The number of bidders represents how many bidders each auction will have. The reserve_price refers to the
  ↪  minimum price the auctioneer is willing to sell the item for."
 ],
 "metadata": {
  "collapsed": false
 }
},
{
 "cell_type": "code",
 "execution_count": null,
 "outputs": [],
 "source": [
  "number_of_bidders = 100\n",
  "auction_types = \"ED,DE,E,D\"\n",
  "reserve_price = 2000\n",
  "auctioneer_type = \"A,B,C,D\"\n",
  "\n",
  "pw.write_auction_information(number_of_bidders, auction_types, reserve_price, auctioneer_type)"
 ],
 "metadata": {
  "collapsed": false,
  "pycharm": {
   "name": "#%%\n"
  }
 }
},
{
 "cell_type": "markdown",
 "source": [
  "Finally, we need to establish what types of bidders we will have. All types of bidders **must** have a value,
  ↪  but it can be set to 0.\n",
  "These values will represent the percentages of bidders of a certain type. For example, if we put our number
  ↪  of bidders previously to be 200 and bidder_a is set to 25, that will mean there will be 50 bidders of type
  ↪  A (25% of 200)."
 ],
 "metadata": {
  "collapsed": false
 }
},
{
 "cell_type": "code",
 "execution_count": null,
 "outputs": [],
 "source": [
  "bidder_a = 25\n",
  "bidder_b = 25\n",
  "bidder_c = 25\n",
  "bidder_d = 25\n",
  "\n",
  "pw.write_bidders_percentages(bidder_a, bidder_b, bidder_c, bidder_d)"
 ],
 "metadata": {
  "collapsed": false,
  "pycharm": {
   "name": "#%%\n"
  }
 }
},
{
 "cell_type": "markdown",
 "source": [
  "### Running the simulator\n",
  "With all the information completed, we just need to run the simulator. The metrics will be automatically
  ↪  saved in a .csv file."
 ],
 "metadata": {
```

```
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "import main as m\n",
    "\n",
    "m.extract_information()\n",
    "m.run_simulation()"
   ],
   "metadata": {
    "collapsed": false,
    "pycharm": {
     "name": "#%%\n"
    }
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "Now that we have the data, all we need to do is analyse it. This function will show not only a bar chart and
    ↪  a box and whiskers chart, but also the ANOVA results for the specified metric."
   ],
   "metadata": {
    "collapsed": false,
    "pycharm": {
     "name": "#%% md\n"
    }
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "m.analyse_data()"
   ],
   "metadata": {
    "collapsed": false,
    "pycharm": {
     "name": "#%%\n"
    }
   }
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 2
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython2",
   "version": "2.7.6"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 0
}
```


14 Possible Output

Auction Types	Auctioneer Type	A	B	C	D	Winner Type	Starting Bid	Revenue	Winner Satisfaction	Auctioneer Satisfaction
ED	A	25	25	25	25	C	2120	544.3105	0.016385393	0.272155237
DE	A	25	25	25	25	D	2120	598.473	0.000202798	0.299236518
E	A	25	25	25	25	D	2000	546.1398	0.006621882	0.273069882
D	A	25	25	25	25	B	2920	468.1824	0.023425192	0.234091178
ED	A	25	25	25	25	A	2120	504.5876	0.024919231	0.252293801
DE	A	25	25	25	25	D	2120	448.6	0.02058319	0.2243
E	A	25	25	25	25	A	2200	420	0.06322314	0.21
D	A	25	25	25	25	A	2720	480.096	0.038669471	0.240048
ED	A	25	25	25	25	A	2160	480.2135	0.041442609	0.240106741
DE	A	25	25	25	25	B	2160	447.28	0.000702821	0.22364
E	A	25	25	25	25	A	2000	542.32	0.00498757	0.27116
D	A	25	25	25	25	B	3760	421.3474	0.066348427	0.210673704
ED	A	25	25	25	25	A	2100	507.0259	0.026315684	0.253512949
DE	A	25	25	25	25	A	2100	545.3165	0.009697635	0.272658225
E	A	25	25	25	25	D	2200	411.1981	0.028119579	0.205599062
D	A	25	25	25	25	D	2900	444.1189	0.0617323	0.222059458
ED	A	25	25	25	25	D	2120	478.066	0.049205302	0.239033007
DE	A	25	25	25	25	C	2120	529.4083	0.011303712	0.264704148
E	A	25	25	25	25	C	2000	534.6042	0.000156174	0.26730208
D	A	25	25	25	25	B	3320	532.7278	0.024981854	0.266363883
ED	A	25	25	25	25	A	2120	509.3272	0.014614589	0.254663607
DE	A	25	25	25	25	B	2120	532.578	0.008458587	0.266288985
E	A	25	25	25	25	D	2200	420	0.035123967	0.21
D	A	25	25	25	25	C	3120	344.5404	0.040289172	0.172270204
ED	A	25	25	25	25	C	2180	446.5664	0.049634308	0.223283186
DE	A	25	25	25	25	D	2180	469.94	0.051847413	0.23497
E	A	25	25	25	25	A	2000	565.86	0.010967083	0.28293
D	A	25	25	25	25	A	3180	372.3922	0.086245342	0.186196111
ED	A	25	25	25	25	A	2160	360.1383	0.036803646	0.180069152
DE	A	25	25	25	25	D	2160	460.143	0.030427901	0.230071506
E	A	25	25	25	25	D	2000	542.3186	0.011674913	0.271159325
D	A	25	25	25	25	B	3960	367.6695	0.074474269	0.183834771
ED	A	25	25	25	25	C	2120	578.2169	0.002630932	0.289108443
DE	A	25	25	25	25	C	2120	509.4359	0.008991718	0.254717931
E	A	25	25	25	25	D	2200	420	0.034297521	0.21
D	A	25	25	25	25	C	2920	483.5384	0.045282808	0.241769204
ED	A	25	25	25	25	A	2180	381.1824	0.013782049	0.190591214
DE	A	25	25	25	25	A	2180	519.828	0.006417902	0.259913996
E	A	25	25	25	25	C	2200	420	0.040909091	0.21
D	A	25	25	25	25	A	3780	335.2531	0.112085027	0.167626547

Social Welfare	Efficiency	Speed
5.4431047	0.0170513	15
5.9847304	0.0747584	4
5.4613976	0.088816	3
4.6818236	0.0526665	4
5.045876	0.0113687	20
4.486	0.0679056	3
4.2	0.0733884	2
4.80096	0.1006893	2
4.8021348	0.0090302	22
4.4728	0.0743124	3
5.4232	0.0887241	3
4.2134741	0.0240542	6
5.070259	0.0051636	44
5.4531645	0.0657401	4
4.1119812	0.0887397	2
4.4411892	0.0400818	4
4.7806601	0.0051305	37
5.294083	0.0633501	4
5.3460416	0.0890486	3
5.3272777	0.0482764	5
5.0932721	0.0042114	57
5.3257797	0.0644576	4
4.2	0.087438	2
3.4454041	0.0263962	5
4.4656637	0.0173649	10
4.6994	0.0610409	3
5.6586	0.0906543	3
3.7239222	0.0249877	4
3.6013831	0.0026048	55
4.6014301	0.0665479	3
5.4231865	0.0864948	3
3.6766954	0.0156229	7
5.7821689	0.0106103	27
5.0943586	0.0614316	4
4.2	0.0878512	2
4.8353841	0.0654955	3
3.8118243	0.0025625	69
5.1982799	0.063374	4
4.2	0.0845455	2
3.3525309	0.0092569	6