PyMarket Documentation

Diego Kiedanski

May 06, 2020

Contents:

PyMarket is a python library designed to ease the simulation and comparison of different market mechanisms.

Marketplaces can be proposed to solve a diverse array of problems. They are used to sell ads online, bandwith spectrum, energy, etc. PyMarket provides a simple environment to try, simulate and compare different market mechanisms, a task that is inherent to the process of establishing a new market.

As an example, Local Energy Markets (LEMs) have been proposed to syncronize energy consumption with surplus of renewable generation. Several mechanisms have been proposed for such a market: from double sided auctions to p2p trading.

This library aims to provide a simple interface for such process, making results reproducible.

CHAPTER 1

Installation

1.1 Stable release

To install pymarket, run this command in your terminal:

First check your Python version, PyMarket requires Python 3.5.2 or newer.

```
$ python --version
```

Verify that pip is installed

```
$ python -m pip --version
```

You can proceed to install PyMarket with the following command (the –user flag is optimal but recommended).

```
$ python -m pip install pymarket --user
```

This is the preferred method to install pymarket, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

Warning: Python ‘ >=3.5.2 ‘ is required. PyMarket won't run with Python 2 nor previous versions of Python 3.

1.2 Dependencies

- PyMarket has been tested in Ubuntu 16.04, Ubuntu 18.04, Manjaro 18.1.1 and mac OS 10.14.4 (through travis only).
- PyMarket does not require additional dependencies outside for those specified in the *requeriments.txt* file. Nevertheless, *PulP* might benefit from having access to additional solvers such as CPLEX (not required).

1.3 From sources

The sources for pymarket can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/gus0k/pymarket
```

Or download the tarball:

```
$ curl  -OL https://github.com/gus0k/pymarket/tarball/master
```

Installing from source requires additional dependencies:

```
$ apt-get install --yes pkg-config
$ apt-get install --yes libfreetype6-dev
$ apt-get install --yes libpng12-dev
$ python -m pip install 'setuptools>=27.3' --user
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.4 Running Tests

To run the tests an additional dependency is needed. It can be installed by running:

```
$ python -m pip install pytest --user
```

Test can be run from the main directory of the project by running:

```
$ python -m pytest
```

CHAPTER 2

Getting started

```
[1]: import pprint
```

Standard imports

```
[15]: import numpy as np
      import pandas as pd
      import pymarket as pm

      import pprint
```

We begin by creating an instance of a market, the basic interface for all mechanisms.

```
[16]: mar = pm.Market() # Creates a new market
```

A market accepts buying and selling bids. The standard format of a bid is

$$bid = (quantity, price, userId, isBuying)$$

A buying bid can be interpreted as follows: $userId$ is willing to buy any fraction of $quantity$ at price $price$ or lower.

A selling bid can be interpreted as follows: $userId$ is willing to sell any fraction of $quantity$ at price $price$ or higher.

2.1 Submitting two bids in the market

Each bid gets a unique identifier within the market when it is accepted. That value is returned by the market after accepting the bid.

```
[17]: mar.accept_bid(1, 2, 0, True) # User 0 want to buy (True) 1 unit at price 2
```

```
[17]: 0
```

```
[18]: mar.accept_bid(2, 1, 1, False) # User 1 wants to sell (False) 2 units at price 2
```

```
[18]: 1
```

2.2 The bids dataframe

All bids are stored in a `BidManager` (bm). The bid manager can return a pandas DataFrame describing all available bids.

```
[19]: mar.bm.get_df()
```

```
[19]:    quantity  price  user  buying  time  divisible
      0         1      2     0    True     0       True
      1         2      1     1   False     0       True
```

Bids can have additional attributes (which are optional and do not have to be necesarily supplied while submitting a bid. Those attributes are: `time` (when was the bid added, useful if priority should be given to the first bids) and `divisible` (indicates whether the offer can be fractional or if it is all or nothing).

2.3 Running the market

Each market has a function `run` that executes the market with all available offers. In this case, we are using a peer-to-peer exchange.

```
[20]: transactions, extras = mar.run('p2p') # run the p2p mechanism with the 2 bids
```

Each run of the market returns the list of all the `transactions` between users who traded, as well as `extra` information dependant on each mechanism.

2.4 The transactions dataframe

The `transactions` object returned by `run` is a `TransactionManager` and as well as the `BidManager`, it has a `get_df()` method to get all the transactions in the DataFrame.

```
[21]: transactions.get_df()
```

```
[21]:    bid  quantity  price  source  active
      0    0         1    1.5       1   False
      1    1         1    1.5       0    True
```

The dataframe can be interpreted as follows:

- Bid 0 traded a quantity 1 at price 1.5 with bid 1 and after it, it had traded as much as desired.
- Bid 1 traded a quantity 1 at price 1.5 with bid 0 and after it, it still had some quantity that wished to trade.

Because there were no more players to trade with, bid 1 could not trade all its desired quantity.

2.5 The extra information

For the `P2P` mechanism, the extra information returned concerns how many rounds of trading ocurred and who traded with whom.

[22]:

```
pprint.pprint(extras)
```

```
{'trading_list': [[(0, 1)]]}
```

`trading_list` is a list of rounds. Each round is a list of tuples containing the pairs that traded. We can see that there was only one ronund, and in it, only one trade.

2.6 Statistics

It is possible to get statistics about the market. The available statistics are:

- Percentage of all the tradable quantity traded
- Percentage of the maximum social welfare achieved
- Profits of the market maker
- Profits of the users, asuming that they bided their true valuations
- Profits of the users, given external reservation price information

Assume that user 0 valuated each unit at 3 instead of at 2, and that user 1 bided his true value. We can obtain the statistics from the market as follows:

[23]:

```
reservation_prices = {0: 10} # We do not need to specify the users who bided␣
↪truthfully
statistics = mar.statistics(reservation_prices=reservation_prices)
```

[24]:

```
pprint.pprint(statistics)
```

```
{'percentage_traded': 1.0,
 'percentage_welfare': 1.0,
 'profits': {'market': 0.0,
             'player_bid': array([0.5, 0.5]),
             'player_reservation': array([8.5, 0.5])}}
```

It can be seen that:

- All that could be traded was traded
- The maximum social welfare was achieved
- The market made no profit (reasonable since it is a pure peer to peer exchange)
- Assuming that players bided their valuations, both players obtained a profit of 0.5
- Taking into account player 0 true valuation, player 0 made a profit of 8.5 instead.

This kind of information is useful to see that `P2P` does not incentize users to bid their true valuations (it is not strategy-proof)

2.7 Adding an extra bid

What happens if there was an extra buyer?

Bids are not removed from the market, so we can just add an extra bid and run the market again.

```
[25]: mar.accept_bid(1, 0.5, 5, True, 4)
```

```
[25]: 2
```

For instance, we can model that this bid was added at time 4, but that the market did not trade until time 5, therefore all of the 3 bids get to trade together.

```
[26]: mar.bm.get_df()
```

```
[26]:    quantity  price  user  buying  time  divisible
      0         1    2.0     0    True     0       True
      1         2    1.0     1   False     0       True
      2         1    0.5     5    True     4       True
```

Because the market is not deterministic, we need to pass a random state to it if we want to be able to reproduce its results.

```
[27]: r = np.random.RandomState(1234)
```

```
[28]: transactions, extras = mar.run('p2p', r=r)
```

```
[29]: transactions.get_df()
```

```
[29]:    bid  quantity  price  source  active
      0    2         0    0.0       1    True
      1    1         0    0.0       2    True
      2    0         1    1.5       1   False
      3    1         1    1.5       0    True
```

This time there were 2 rounds with 1 trade each. In the first one, Bid 2 traded with Bid 1, and in the second one, Bid 1 traded with Bid 0. However, because user 5 had a very low buying price, it did not trade exchange any good with user 1.

```
[33]: extras
```

```
[33]: {'trading_list': [[(2, 1)], [(0, 1)]]}
```

```
[34]: reservation_prices = {0: 10} # We do not need to specify the users who bided␣
      ↪truthfully
      statistics = mar.statistics(reservation_prices=reservation_prices)
      pprint.pprint(statistics)
```

```
{'percentage_traded': 1.0,
 'percentage_welfare': 1.0,
 'profits': {'market': 0.0,
             'player_bid': array([0.5, 0.5, 0. ]),
             'player_reservation': array([8.5, 0.5, 0. ])}}
```

We see the same results as in the previous case, with the addition of player 5 who has a 0 profit for not trading

CHAPTER 3

Examples

3.1 P2P

[3]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import networkx as nx
import pymarket as pm
```

3.1.1 Creates new market

[4]:
```
r = np.random.RandomState(1234)
mar = pm.Market()

mar.accept_bid(1, 6.7, 0, True, 0)
mar.accept_bid(1, 6.6, 1, True, 0)
mar.accept_bid(1, 6.5, 2, True, 0)
mar.accept_bid(1, 6.4, 3, True, 0)
mar.accept_bid(1, 6.3, 4, True, 0)
mar.accept_bid(1, 6, 5, True, 0)

mar.accept_bid(1, 1, 6, False, 0)
mar.accept_bid(1, 2, 7, False, 0)
mar.accept_bid(2, 3, 8, False, 0)
mar.accept_bid(2, 4, 9, False, 0)
mar.accept_bid(1, 6.1, 10, False, 0)

bids = mar.bm.get_df()
transactions, extras = mar.run('p2p', r=r)
stats = mar.statistics()
```

```
[5]: bids # bids dataframe
```

```
[5]:     quantity  price  user  buying  time  divisible
     0          1    6.7     0    True     0       True
     1          1    6.6     1    True     0       True
     2          1    6.5     2    True     0       True
     3          1    6.4     3    True     0       True
     4          1    6.3     4    True     0       True
     5          1    6.0     5    True     0       True
     6          1    1.0     6   False     0       True
     7          1    2.0     7   False     0       True
     8          2    3.0     8   False     0       True
     9          2    4.0     9   False     0       True
     10         1    6.1    10   False     0       True
```

```
[6]: transactions.get_df() # transactions dataframe
```

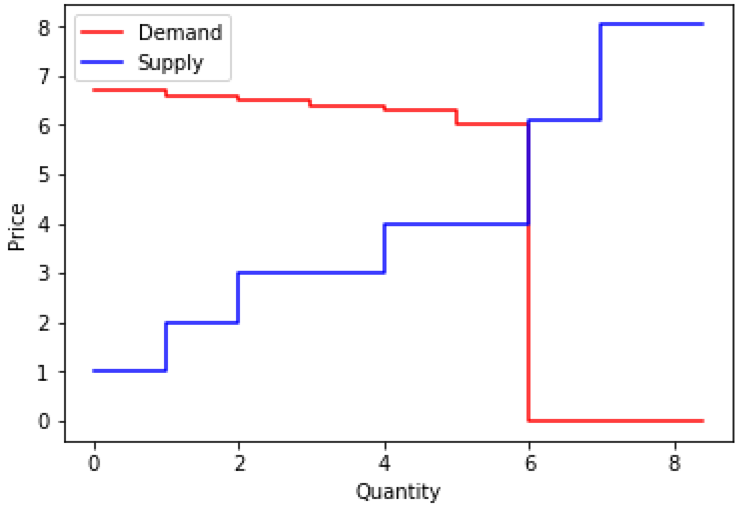
```
[6]:     bid  quantity  price  source  active
     0     3         1   3.70       6   False
     1     6         1   3.70       3   False
     2     5         0   0.00      10    True
     3    10         0   0.00       5    True
     4     2         1   4.25       7   False
     5     7         1   4.25       2   False
     6     4         1   5.15       9   False
     7     9         1   5.15       4    True
     8     0         1   4.85       8   False
     9     8         1   4.85       0    True
     10    5         1   5.00       9   False
     11    9         1   5.00       5   False
     12    1         1   4.80       8   False
     13    8         1   4.80       1   False
```

```
[7]: extras # additional information characteristic of P2P trading
```

```
[7]: {'trading_list': [[(3, 6), (5, 10), (2, 7), (4, 9), (0, 8)], [(5, 9), (1, 8)]]}
```

3.1.2 Orignal supply and demand curves

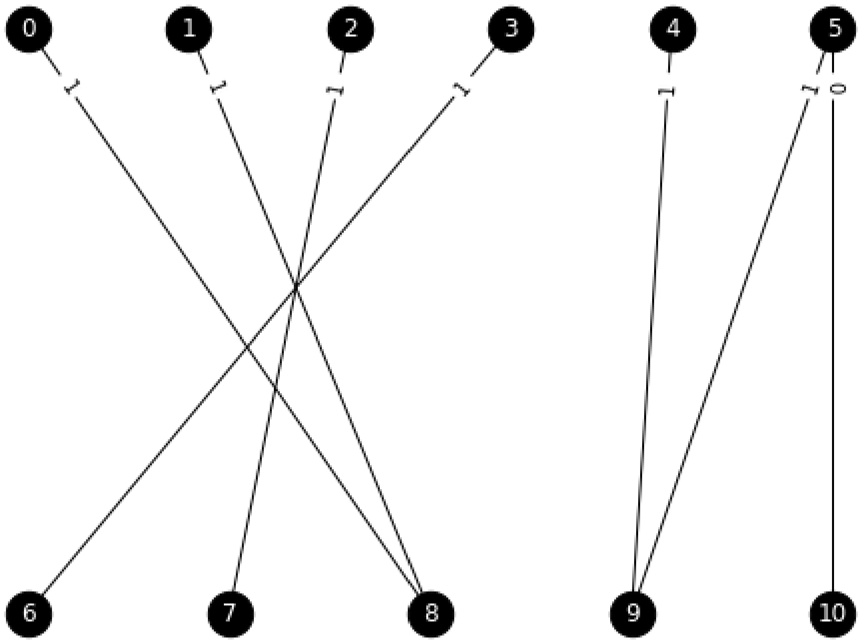
```
[8]: mar.plot()
```



3.1.3 Trades among participants

```
[9]: ax = mar.plot_method('p2p')
```

```
/home/guso/anaconda3/lib/python3.6/site-packages/networkx/drawing/nx_pylab.py:611:␣
↪MatplotlibDeprecationWarning: isinstance(..., numbers.Number)
  if cb.is_numlike(alpha):
```



3.1.4 Analysis of the results

Round 1

- 3 trades with 6, they both trade all their quantity and are not considered for next round
- 5 trades with 10, the asked price by 10 is to high and no trade happens, they continue in next round
- 2 trades with 7, they both trade all their quantity and are not considered for next round
- 4 trades with 9, they trade one unit and 9 goes to next one with one remaining unit
- 0 trades with 8, they trade one unit and 8 goes to next one with one remaining unit
- 1 is not paired with anyone and continues to round 2

Round 2

- 5 trades with 9, they both trade all their remaining quantity and are not considered for the next round
- 1 trades with 8, they both trade all their remaining quantity and are not considered for the next round
- 10 is not paired and continues to the round 3

Round 3

- Only 10 remains, so no trade can ocurr, the algorithm ends.

3.1.5 Statistics

[10]:
```python
print('Percentage of the maximum possible traded quantity')
stats['percentage_traded']
```

```
Percentage of the maximum possible traded quantity
```

[10]:
```
0.9999999999989999
```

[11]:
```python
print('Percentage of the maximum possible total welfare')
stats['percentage_welfare']
```

```
Percentage of the maximum possible total welfare
```

[11]:
```
1.0
```

[12]:
```python
print('Profits per user')
for u in bids.user.unique():
    print(f'User {u:2} obtained a profit of {stats["profits"]["player_bid"][u]:0.2f}')
```

```
Profits per user
User  0 obtained a profit of 1.85
User  1 obtained a profit of 1.80
User  2 obtained a profit of 2.25
User  3 obtained a profit of 2.70
User  4 obtained a profit of 1.15
User  5 obtained a profit of 1.00
User  6 obtained a profit of 2.70
User  7 obtained a profit of 2.25
User  8 obtained a profit of 3.65
User  9 obtained a profit of 2.15
User 10 obtained a profit of 0.00
```

```
[13]: print(f'Profit to Market Maker was {stats["profits"]["market"]:0.2f}')
```

```
Profit to Market Maker was 0.00
```

3.2 MUDA

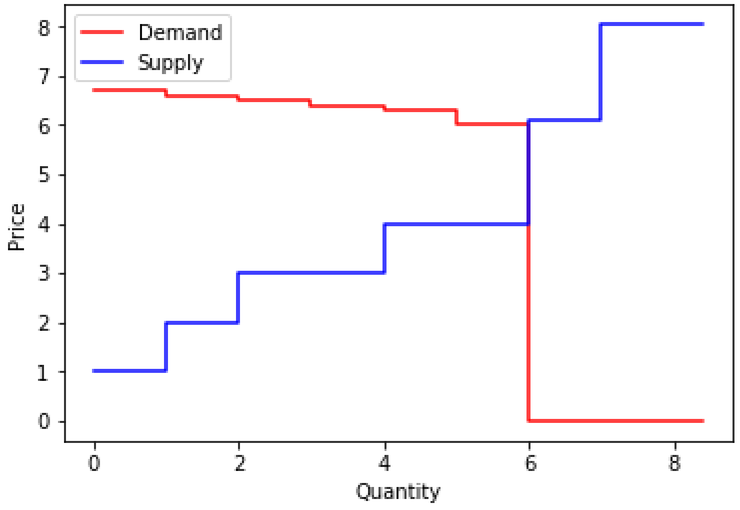
```
[1]: %matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import networkx as nx
import pymarket as pm
```

3.2.1 Creates new market

```
[2]: r = np.random.RandomState(1234)
mar = pm.Market()

mar.accept_bid(1, 6.7, 0, True, 0)
mar.accept_bid(1, 6.6, 1, True, 0)
mar.accept_bid(1, 6.5, 2, True, 0)
mar.accept_bid(1, 6.4, 3, True, 0)
mar.accept_bid(1, 6.3, 4, True, 0)
mar.accept_bid(1, 6, 5, True, 0)

mar.accept_bid(1, 1, 6, False, 0)
mar.accept_bid(1, 2, 7, False, 0)
mar.accept_bid(2, 3, 8, False, 0)
mar.accept_bid(2, 4, 9, False, 0)
mar.accept_bid(1, 6.1, 10, False, 0)

bids = mar.bm.get_df()
transactions, extras = mar.run('muda', r=r)
stats = mar.statistics()
```

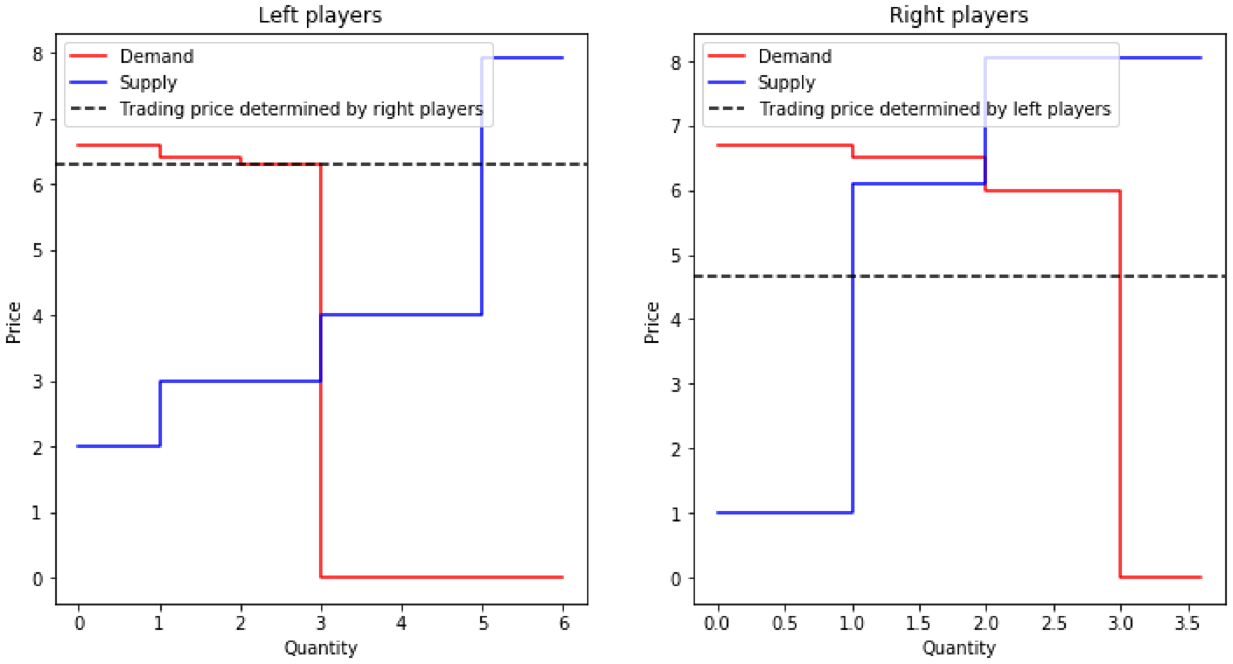
3.2.2 Orignal supply and demand curves

```
[3]: mar.plot()
```



3.2.3 Supply and demand curves after market is splitted

```
[4]: ax = mar.plot_method('muda')
```



3.2.4 Analysis of the Left Side

Participants

- Buying: 1, 3, 4
- Selling: 7, 8, 9

Trading price

- 6.3

Results

- The long side is the supply, all demand side buys as much as they want
- The demand side pays no fees, they are the short side
- Bid 7, results in bid 9 not trading a unit, so the fee is $ 1 :nbsphinx-math:‘times ‘(6.3 - 4) = 2.3$
- Bid 8, results in bid 9 not trading a 2 units so the fee is $2 \times (6.3 - 4) = 4.6$

3.2.5 Analysis of the Right Side

Participants

- Buying: 0, 2, 5
- Selling: 6, 10 (10 does not trade because bid price is greater than trading price)

Trading price

- 4.65

Results

- The long side is the demand, all supply side buys as much as they want
- The supply side pays no fees, they are the short side
- Bid 0, results in bid 2 not trading a unit, so the fee is $ 1 :nbsphinx-math:‘times ‘(6.5 - 4.65) = 1.85$

3.2.6 Statistics

```
[5]: print('Percentage of the maximum possible traded quantity')
     stats['percentage_traded']
```

```
Percentage of the maximum possible traded quantity
```

```
[5]: 0.6666666666659999
```

```
[6]: print('Percentage of the maximum possible total welfare')
     stats['percentage_welfare']
```

```
Percentage of the maximum possible total welfare
```

```
[6]: 0.7906976744186046
```

```
[7]: print('Profits per user')
     for u in bids.user.unique():
         print(f'User {u:2} obtained a profit of {stats["profits"]["player_bid"][u]:0.2f}')
```

```
Profits per user
User  0 obtained a profit of 2.05
User  1 obtained a profit of 0.30
User  2 obtained a profit of 0.00
User  3 obtained a profit of 0.10
User  4 obtained a profit of 0.00
User  5 obtained a profit of 0.00
User  6 obtained a profit of 3.65
User  7 obtained a profit of 4.30
User  8 obtained a profit of 6.60
User  9 obtained a profit of 0.00
User 10 obtained a profit of 0.00
```

[8]:
```
print(f'Profit to Market Maker was {stats["profits"]["market"]:0.2f}')
```

```
Profit to Market Maker was 8.75
```

[ ]:

3.3 Huang

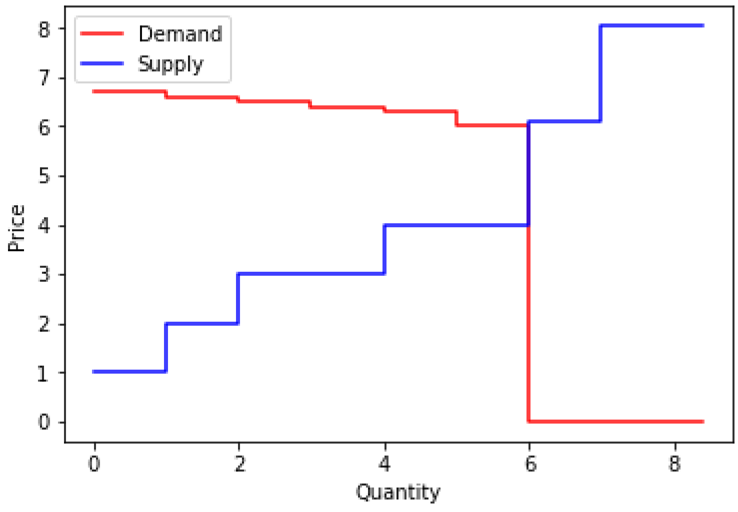
[2]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import networkx as nx
import pymarket as pm
```

3.3.1 Creates new market

[4]:
```
mar = pm.Market()

mar.accept_bid(1, 6.7, 0, True, 0)
mar.accept_bid(1, 6.6, 1, True, 0)
mar.accept_bid(1, 6.5, 2, True, 0)
mar.accept_bid(1, 6.4, 3, True, 0)
mar.accept_bid(1, 6.3, 4, True, 0)
mar.accept_bid(1, 6, 5, True, 0)

mar.accept_bid(1, 1, 6, False, 0)
mar.accept_bid(1, 2, 7, False, 0)
mar.accept_bid(2, 3, 8, False, 0)
mar.accept_bid(2, 4, 9, False, 0)
mar.accept_bid(1, 6.1, 10, False, 0)

bids = mar.bm.get_df()
transactions, extras = mar.run('huang')
stats = mar.statistics()
```

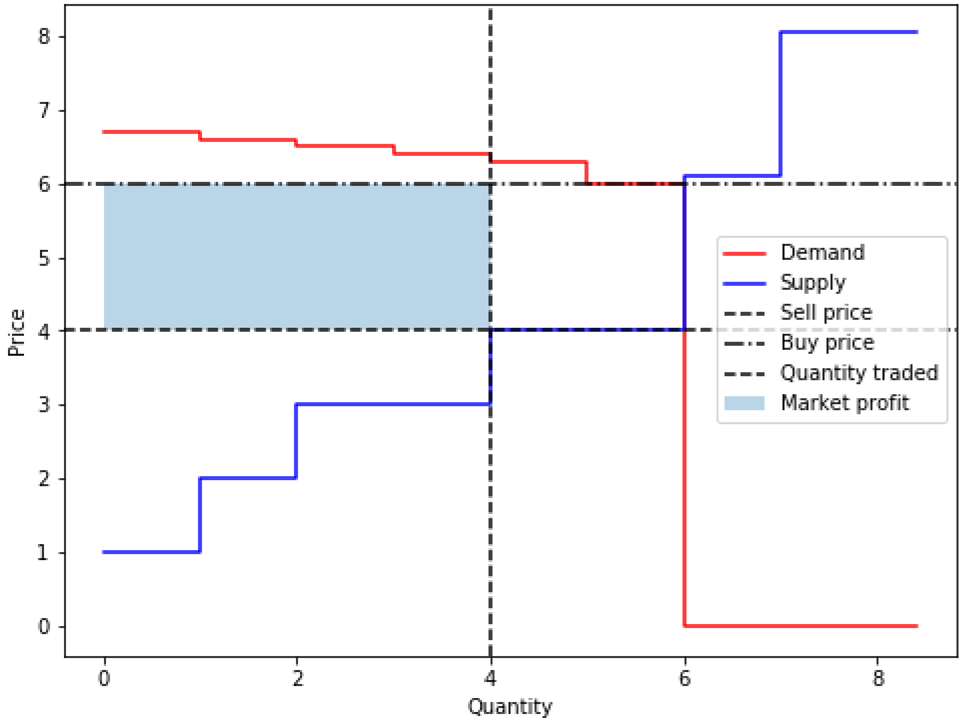
3.3.2 Orignal supply and demand curves

```
[5]: mar.plot()
```



3.3.3 Supply and demand curves after market is splitted

```
[11]: fig, ax = plt.subplots(figsize=(8, 6))
      ax = mar.plot_method('huang', ax=ax)
```



3.3.4 Analysis of the trade

Trading price

- Selling Price: 4, defined by bid 9, consequently, 9 does not trade
- Buying Price: 6, defined by bid 5, consequently, 5 does not trade

Actually trading

- Buying: 0, 1, 2, 3
- Selling: 6, 7, 8

Results

- Supply and demand have the same size.
- The profit of the market maker coincides with the blue shaded area

3.3.5 Statistics

[7]:

```
print('Percentage of the maximum possible traded quantity')
stats['percentage_traded']
```

```
Percentage of the maximum possible traded quantity
```

[7]:

```
0.6666666666659999
```

[8]:

```
print('Percentage of the maximum possible total welfare')
stats['percentage_welfare']
```

```
Percentage of the maximum possible total welfare
```

[8]:

```
0.4186046511627907
```

[9]:

```
print('Profits per user')
for u in bids.user.unique():
    print(f'User {u:2} obtained a profit of {stats["profits"]["player_bid"][u]:0.2f}')
```

```
Profits per user
User  0 obtained a profit of 0.56
User  1 obtained a profit of 0.48
User  2 obtained a profit of 0.40
User  3 obtained a profit of 0.32
User  4 obtained a profit of 0.24
User  5 obtained a profit of 0.00
User  6 obtained a profit of 3.00
User  7 obtained a profit of 2.00
User  8 obtained a profit of 2.00
User  9 obtained a profit of 0.00
User 10 obtained a profit of 0.00
```

[10]:

```
print(f'Profit to Market Maker was {stats["profits"]["market"]:0.2f}')
```

```
Profit to Market Maker was 8.00
```

```
[ ]:
```

3.4 Efficiency and Performance

```
[1]: %matplotlib inline
     import time
     import pymarket as pm
     import numpy as np
     import matplotlib.pyplot as plt
```

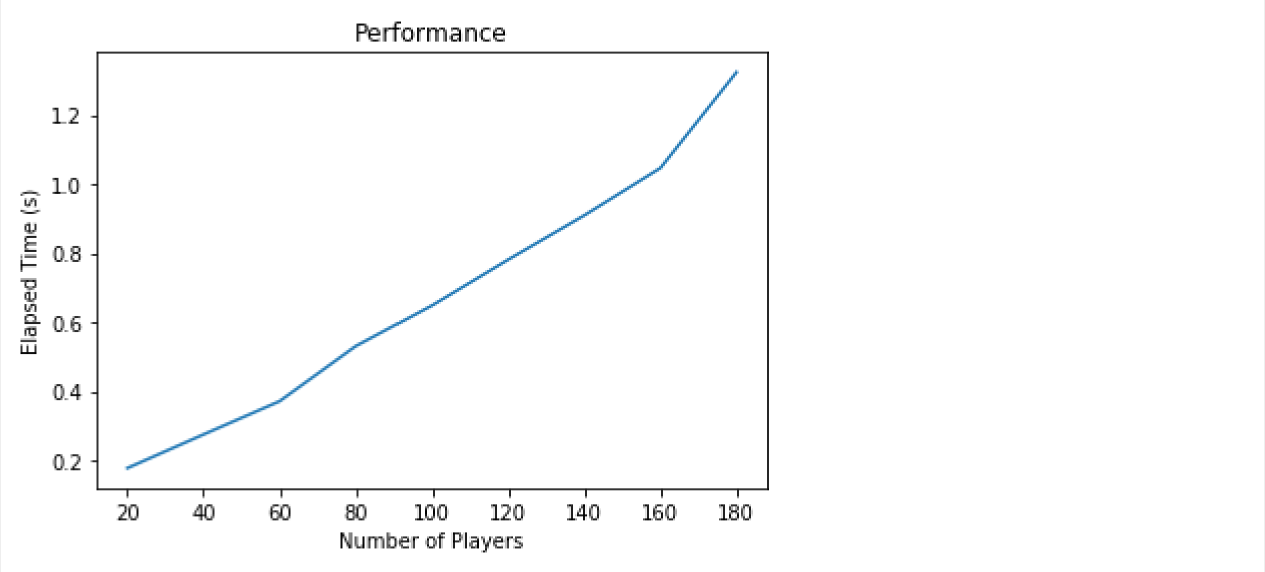
3.4.1 Create a set of markets with varying number of participants

```
[41]: markets = []
      range_players = np.arange(20, 200, 20)
      M = len(range_players)

      for i in range_players:
          bids = pm.datasets.generate(i, i, 2, 1)
          mar = pm.Market()
          for b in bids:
              mar.accept_bid(*b)
          markets.append(mar)
```

3.4.2 Run the diferent markets

```
[42]: elapsed = np.zeros(M)
      for i in range(M):
          mar = markets[i]
          start = time.time()
          mar.run('huang')
          stop = time.time()
          elapsed[i] = stop - start
```

```
[43]: fig, ax = plt.subplots()
      ax.plot(range_players, elapsed)
      _ = ax.set_xlabel('Number of Players')
      _ = ax.set_ylabel('Elapsed Time (s)')
      _ = ax.set_title('Performance')
```

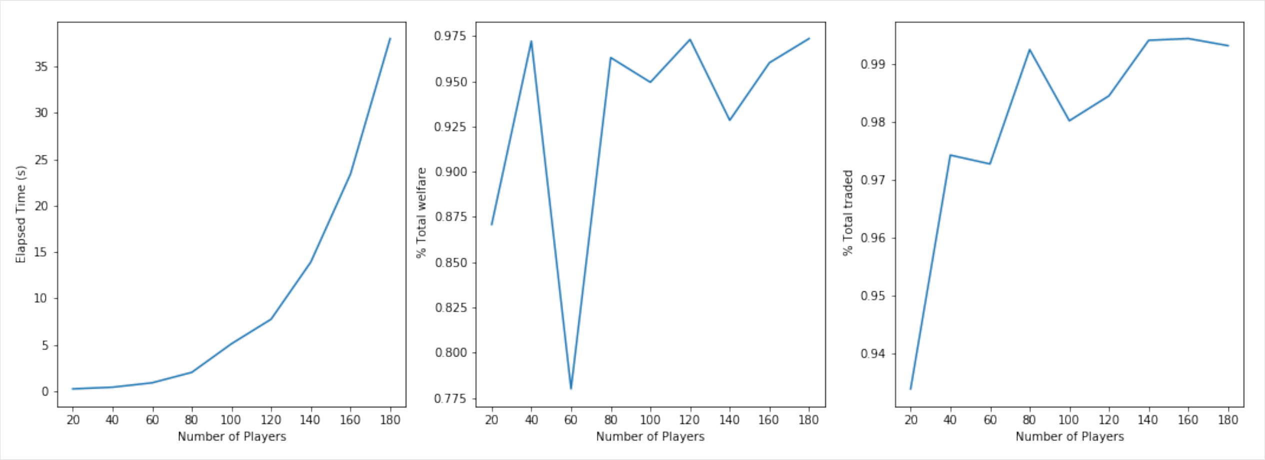


3.4.3 Obtains the statistics (optimization problems have to be solved)

[49]:
```
traded = np.zeros(M)
welfare = np.zeros(M)
stats_time = np.zeros(M)

limit = M
for i in range(limit):
    mar = markets[i]
    start = time.time()
    stats = mar.statistics()
    stop = time.time()
    stats_time[i] = stop - start
    welfare[i] = stats['percentage_welfare']
    traded[i] = stats['percentage_traded']
```

3.4.4 Plots the results

[51]:
```
fig, ax = plt.subplots(1, 3, figsize=(18, 6))

ax[0].plot(range_players[:limit], stats_time[:limit])
ax[0].set_ylabel('Elapsed Time (s)')

ax[1].plot(range_players[:limit], welfare[:limit])
ax[1].set_ylabel(' % Total welfare')

ax[2].plot(range_players[:limit], traded[:limit])
ax[2].set_ylabel(' % Total traded')

for ax_ in ax:
    ax_.set_xlabel('Number of Players')
```



[ ]:

3.5 Creating a new mechanism

[1]:

```
import numpy as np
import pandas as pd
import pymarket as pm
import matplotlib.pyplot as plt
from pprint import pprint
```

One of the advantages of `PyMarket` is the ability to easily implement and test a new idea for a mechanism. Here we will show how to implement a new mechanism and use it.

3.5.1 The uniform price mechanism

We are going to implement a uniform price mechanism that charges every trading player the clearing price.

As a reference we are going to be implement the example Here

We can begin by adding the corresponding bids to a new market

[2]:

```
mar = pm.Market()

buyers_names = ['CleanRetail', 'El4You', 'EVcharge', 'QualiWatt', 'IntelliWatt']

mar.accept_bid(250, 200, 0, True) # CleanRetail  0
mar.accept_bid(300, 110, 1, True) # El4You       1
mar.accept_bid(120, 100, 2, True) # EVcharge     2
mar.accept_bid( 80,  90, 3, True) # QualiWatt    3
mar.accept_bid( 40,  85, 4, True) # IntelliWatt  4
mar.accept_bid( 70,  75, 1, True) # El4You       5
mar.accept_bid( 60,  65, 0, True) # CleanRetail  6
mar.accept_bid( 45,  40, 4, True) # IntelliWatt  7
mar.accept_bid( 30,  38, 3, True) # QualiWatt    8
mar.accept_bid( 35,  31, 4, True) # IntelliWatt  9
mar.accept_bid( 25,  24, 0, True) # CleanRetail  10
mar.accept_bid( 10,  21, 1, True) # El4You       11
```

(continues on next page)

(continued from previous page)

```
sellers_names = ['RT', 'WeTrustInWind', 'BlueHydro', 'KøbenhavnCHP', 'DirtyPower',
↪'SafePeak']

mar.accept_bid(120,   0, 5, False) # RT            12
mar.accept_bid(50,    0, 6, False) # WeTrustInWind 13
mar.accept_bid(200,  15, 7, False) # BlueHydro     14
mar.accept_bid(400,  30, 5, False) # RT            15
mar.accept_bid(60, 32.5, 8, False) # KøbenhavnCHP  16
mar.accept_bid(50,   34, 8, False) # KøbenhavnCHP  17
mar.accept_bid(60,   36, 8, False) # KøbenhavnCHP  18
mar.accept_bid(100,37.5, 9, False) # DirtyPower    19
mar.accept_bid(70,   39, 9, False) # DirtyPower    20
mar.accept_bid(50,   40, 9, False) # DirtyPower    21
mar.accept_bid(70,   60, 5, False) # RT            22
mar.accept_bid(45,   70, 5, False) # RT            23
mar.accept_bid(50,  100, 10, False) # SafePeak     24
mar.accept_bid(60,  150, 10, False) # SafePeak     25
mar.accept_bid(50,  200, 10, False) # SafePeak     26
```

[2]:

```
26
```

[3]:

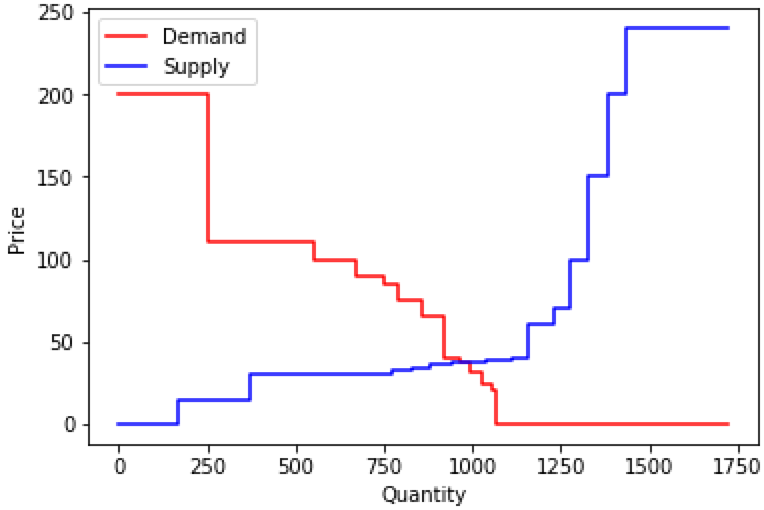
```
12, 15, 22, 23
```

[3]:

```
(12, 15, 22, 23)
```

[4]:

```
mar.plot()
```



3.5.2 Implementing the mechanism

All market mechanisms take as arguements a bids dataframe (as well as possibly extra parameters) and returns a `TransactionManager` and an `extras` dictionary.

[13]:

```
def uniform_price_mechanism(bids: pd.DataFrame) -> (pm.TransactionManager, dict):

    trans = pm.TransactionManager()
```

(continues on next page)

(continued from previous page)

```
    buy, _ = pm.bids.demand_curve_from_bids(bids) # Creates demand curve from bids
    sell, _ = pm.bids.supply_curve_from_bids(bids) # Creates supply curve from bids

    # q_ is the quantity at which supply and demand meet
    # price is the price at which that happens
    # b_ is the index of the buyer in that position
    # s_ is the index of the seller in that position
    q_, b_, s_, price = pm.bids.intersect_stepwise(buy, sell)

    buying_bids  = bids.loc[bids['buying']].sort_values('price', ascending=False)
    selling_bids = bids.loc[~bids['buying']].sort_values('price', ascending=True)

    ## Filter only the trading bids.
    buying_bids = buying_bids.iloc[: b_ + 1, :]
    selling_bids = selling_bids.iloc[: s_ + 1, :]

    # Find the long side of the market
    buying_quantity = buying_bids.quantity.sum()
    selling_quantity = selling_bids.quantity.sum()


    if buying_quantity > selling_quantity:
        long_side = buying_bids
        short_side = selling_bids
    else:
        long_side = selling_bids
        short_side = buying_bids

    traded_quantity = short_side.quantity.sum()

    ## All the short side will trade at `price`
    ## The -1 is there because there is no clear 1 to 1 trade.
    for i, x in short_side.iterrows():
        t = (i, x.quantity, price, -1, False)
        trans.add_transaction(*t)

    ## The long side has to trade only up to the short side
    quantity_added = 0
    for i, x in long_side.iterrows():

        if x.quantity + quantity_added <= traded_quantity:
            x_quantity = x.quantity
        else:
            x_quantity = traded_quantity - quantity_added
        t = (i, x_quantity, price, -1, False)
        trans.add_transaction(*t)
        quantity_added += x.quantity

    extra = {
        'clearing quantity': q_,
        'clearing price': price
    }



    return trans, extra
```

3.5.3 Wrapping the algorithm as a mechanism

[14]:
```
# Observe that we add as the second argument of init the algorithm just coded
class UniformPrice(pm.Mechanism):
    """
    Interface for our new uniform price mechanism.

    Parameters
    -----------
    bids
        Collection of bids to run the mechanism
        with.
    """

    def __init__(self, bids, *args, **kwargs):
        """TODO: to be defined1. """
        pm.Mechanism.__init__(self, uniform_price_mechanism, bids, *args, **kwargs)
```

3.5.4 Adding the new mechanism to the list of available mechanism of the market

[15]:
```
pm.market.MECHANISM['uniform'] = UniformPrice
```

3.5.5 Running the new mechanism and comparing it with Huang's and P2P

[24]:
```
stats = {}
for mec in ['uniform', 'huang', 'p2p']:
    t, e = mar.run(mec)
    stat = mar.statistics()
    stats[mec] = stat
```

Profits for the players in the different mechanism

[33]:
```
profits = pd.DataFrame([v['profits']['player_bid'] for k, v in stats.items()]).T
profits.columns = stats.keys()
profits
```

[33]:
```
    uniform      huang      p2p
0   42275.0  41529.375  22890.0
1   24375.0  23849.375  12980.0
2    7500.0   7246.250   3150.0
3    4215.0   3997.500   2630.0
4    2012.5   1816.875   1162.5
5    7500.0   7500.000  14647.5
6    1875.0   1875.000    810.0
7    4500.0   4500.000  18500.0
8     565.0    565.000   3910.0
9       0.0      0.000   4570.0
10      0.0      0.000    375.0
```

Percentage traded by mechanism

```
[36]: traded = pd.DataFrame([v['percentage_traded'].round(3) for k, v in stats.items()]).T
traded.columns = stats.keys()
traded
```

```
[36]:    uniform  huang    p2p
0     0.934  0.883  0.995
```

Percentage of the maximum social welfare achieved by mechanism

```
[38]: welfare = pd.DataFrame([v['percentage_welfare'].round(3) for k, v in stats.items()]).T
welfare.columns = stats.keys()
welfare
```

```
[38]:    uniform  huang    p2p
0       1.0   0.98  0.903
```

```
[ ]:
```

CHAPTER 4

pymarket

4.1 pymarket package

4.1.1 Subpackages

pymarket.bids package

Top-level package for pymarket.

Submodules

pymarket.bids.bids module

class `pymarket.bids.bids.`**`BidManager`**

Bases: `object`

A class used to store and manipulate a collection of all the bids in the market.

`col_names`

Column names for the different attributes in the dataframe to be created. Currently and in order: *quantity*, *price*, *user*, *buying*, *time*, *divisible*.

Type `list` of `str`

`n_bids`

Number of bids currently stored. Used as a unique identifier for each bid within a BidManager.

Type int

`bids`

A list where all the recieved bids are stored.

Type `list` of `tuple`

add_bid(*quantity*, *price*, *user*, *buying=True*, *time=0*, *divisible=True*)

Appends a bid to the bid list

Parameters

- **quantity** (*float*) – Quantity of good desired. If *divisible=True* then any fraction of the good is an acceptable outcome of the market.
- **price** (*float*) – Uniform price offered in the market for each unit of the the good.
- **user** (*int*) – Identifier of the user submitting the bid.
- **buying** (*bool*) – *True* if the bid is for buying the good and *False'otherwise. Default is 'True*.
- **time** (*float*) – Instant at which the offer was made. This is relevant only if the market mechanism has perferences for earlier bids. Default is *0*
- **divisible** (*bool*) – *True* is the user accepts a fraction of the asked quantity as a result and *False* otherwise.

Returns Unique identifier of the added bid.

Return type int

Examples

```
>>> bm = pm.BidManager()
>>> bm.add_bid(2, 1, 0)
0
```

col_names = ['quantity', 'price', 'user', 'buying', 'time', 'divisible']

get_df()

Creates a dataframe with the bids

Returns Dataframe with each row a different bid and each column each of the different attributes.

Return type pd.DataFrame

Examples

```
>>> bm = pm.BidManager()
>>> bm.add_bid(2, 1, 0)
0
>>> bm.add_bid(1, 3, 1, buying=False)
1
>>> print(bm.get_df())
   quantity  price  user  buying  time  divisible
0         2      1     0    True     0       True
1         1      3     1   False     0       True
```

pymarket.bids.demand_curves module

pymarket.bids.demand_curves.**demand_curve_from_bids**(*bids*)

Creates a demand curve from a set of buying bids. It is the inverse cumulative distribution of quantity as a function of price.

Parameters **`bids`** – Collection of all the bids in the market. The algorithm filters only the buying bids.

Returns

- **demand_curve** (*np.ndarray*) – Stepwise constant demand curve represented as a collection of the N rightmost points of each interval (N-1 bids). It is stored as a (N, 2) matrix where the first column is the x-coordinate and the second column is the y-coordinate. An extra point is a))dded with x coordinate at infinity and price at 0 to represent the end of the curve.
- **index** (*np.ndarray*) – The order of the identifier of each bid in the demand curve.

Examples

A minimal example, selling bid is ignored:

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 1, 0, buying=True)
0
>>> bm.add_bid(1, 1, 1, buying=False)
1
>>> dc, index = pm.demand_curve_from_bids(bm.get_df())
>>> dc
array([[ 1.,   1.],
       [inf,   0.]])
>>> index
array([0])
```

A larger example with reordering of bids:

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 1, 0, buying=True)
0
>>> bm.add_bid(1, 1, 1, buying=False)
1
>>> bm.add_bid(3, 0.5, 2, buying=True)
2
>>> bm.add_bid(2.3, 0.1, 3, buying=True)
3
>>> dc, index = pm.demand_curve_from_bids(bm.get_df())
>>> dc
array([[1. , 1. ],
       [4. , 0.5],
       [6.3, 0.1],
       [inf, 0. ]])
>>> index
array([0, 2, 3])
```

`pymarket.bids.demand_curves.`**`get_value_stepwise`**(*x*, *f*)

Returns the value of a stepwise constant function defined by the right extrems of its interval Functions are assumed to be defined in (0, inf).

Parameters

- **`x`** (*`float`*) – Value in which the function is to be evaluated
- **`f`** (*`np.ndarray`*) – Stepwise function represented as a 2 column matrix. Each row is the rightmost extreme point of each constant interval. The first column contains the x coordinate and is sorted increasingly. f is assumed to be defined only in the interval :math: (0, infty)

Returns The image of x under f: *f(x)*. If *x* is negative, then None is returned instead. If x is outside the range of the function (greater than *f[-1, 0]*), then the method returns None.

Return type float or None

Examples

```
>>> f = np.array([
...     [1, 1],
...     [3, 4]])
>>> [pm.get_value_stepwise(x, f)
...     for x in [-1, 0, 0.5, 1, 2, 3, 4]]
[None, 1, 1, 1, 4, 4, None]
```

`pymarket.bids.demand_curves.`**`intersect_stepwise`**(*f*, *g*, *k=0.5*)

Finds the intersection of two stepwise constants functions where f is assumed to be bigger at 0 than g. If no intersection is found, None is returned.

Parameters

- **f** (*np.ndarray*) – Stepwise constant function represented as a 2 column matrix where each row is the rightmost point of the constat interval. The first column is sorted increasingly. Preconditions: f is non-increasing.
- **g** (*np.ndarray*) – Stepwise constant function represented as a 2 column matrix where each row is the rightmost point of the constat interval. The first column is sorted increasingly. Preconditions: g is non-decreasing and *f[0, 0] > g[0, 0]*
- **k** (*float*) – If the intersection is empty or an interval, a convex combination of the y-values of f and g will be returned and k will be used to determine hte final value. *k=1* will be the value of g while *k=0* will be the value of f.

Returns

- **x_ast** (*float or None*) – Axis coordinate of the intersection of both functions. If the intersection is empty, then it returns None.
- **f_ast** (*int or None*) – Index of the rightmost extreme of the interval of *f* involved in the intersection. If the intersection is empty, returns None
- **g_ast** (*int or None*) – Index of the rightmost extreme of the interval of *g* involved in the intersection. If the intersection is empty, returns None.
- **v** (*float or None*) – Ordinate of the intersection if it is uniquely identified, otherwise the k-convex combination of the y values of *f* and *g* in the last point when they were both defined.

Examples

Simple intersection with diferent domains

```
>>> f = np.array([[1, 3], [3, 1]])
>>> g = np.array([[2,2]])
>>> pm.intersect_stepwise(f, g)
(1, 0, 0, 2)
```

Empty intersection, returning the middle value

```
>>> f = np.array([[1,3], [2, 2.5]])
>>> g = np.array([[1,1], [2, 2]])
>>> pm.intersect_stepwise(f, g)
(None, None, None, 2.25)
```

`pymarket.bids.demand_curves.`**`supply_curve_from_bids`**(*bids*)

Creates a supply curve from a set of selling bids. It is the cumulative distribution of quantity as a function of price.

Parameters **`bids`** (*`pd.DataFrame`*) – Collection of all the bids in the market. The algorithm filters only the selling bids.

Returns

- **supply_curve** (*np.ndarray*) – Stepwise constant demand curve represented as a collection of the N rightmost points of each interval (N-1 bids). It is stored as a (N, 2) matrix where the first column is the x-coordinate and the second column is the y-coordinate. An extra point is added with x coordinate at infinity and price at infinity to represent the end of the curve.
- **index** (*np.ndarray*) – The order of the identifier of each bid in the supply curve.

Examples

A minimal example, selling bid is ignored:

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0, False)
0
>>> bm.add_bid(2.1, 3, 3, True)
1
>>> sc, index = pm.supply_curve_from_bids(bm.get_df())
>>> sc
array([[ 1.,  3.],
       [inf, inf]])
>>> index
array([0])
```

A larger example with reordering:

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0, False)
0
>>> bm.add_bid(2.1, 3, 3, True)
1
>>> bm.add_bid(0.2, 1, 3, False)
2
>>> bm.add_bid(1.7, 6, 4, False)
3
>>> sc, index = pm.supply_curve_from_bids(bm.get_df())
>>> sc
array([[0.2, 1. ],
       [1.2, 3. ],
       [2.9, 6. ],
       [inf, inf]])
>>> index
array([2, 0, 3])
```

pymarket.bids.processing module

Implements processing techniques applied to bids before mechanisms can use them

`pymarket.bids.processing.`**`merge_same_price`**(*df*, *prec=5*)

Process a collection of bids by merging in each side (buying or selling) all players with the same price into a new user with their aggregated quantity

Parameters

- **`df`** (*`pd.DataFrame`*) – Collection of bids to process
- **`prec`** (*`float`*) – Number of digits to use after the comma while comparing floating point prices as equal.

Returns

- **dataframe_new** (*pd.DataFrame*) – The new collection of bids where players with the same price have been merged into one.
- **final_maping** (*dict*) – Maping from new bids index to the old bids index.

Examples

```
>>> bm = BidManager()
>>> bm.add_bid(0.3, 1, 0)
0
>>> bm.add_bid(0.7, 1, 1)
1
>>> bm.add_bid(2, 1, 2, False)
2
>>> bm.add_bid(1, 2.444446, 3, False)
3
>>> bm.add_bid(3, 2.444447, 4, False)
4
>>> bm.get_df()
   quantity     price  user  buying  time  divisible
0       0.3  1.000000     0    True     0       True
1       0.7  1.000000     1    True     0       True
2       2.0  1.000000     2   False     0       True
3       1.0  2.444446     3   False     0       True
4       3.0  2.444447     4   False     0       True
>>> bids, index = pm.merge_same_price(bm.get_df(), 5)
>>> bids
   quantity    price  user  buying  time  divisible
0       1.0  1.00000     5    True     0       True
1       2.0  1.00000     2   False     0       True
2       4.0  2.44445     6   False     0       True
>>> index
{0: [0, 1], 1: [2], 2: [3, 4]}
```

```
>>> mar = pm.Market()
>>> mar.accept_bid(250, 200, 0, True) # CleanRetail
0
>>> mar.accept_bid(300, 110, 1, True) # El4You
1
>>> mar.accept_bid(120, 100, 2, True) # EVcharge
2
```

(continues on next page)

(continued from previous page)

```
>>> mar.accept_bid( 80,  90, 3, True) # QualiWatt
3
>>> mar.accept_bid( 40,  85, 4, True) # IntelliWatt
4
>>> mar.accept_bid( 70,  75, 1, True) # El4You
5
>>> mar.accept_bid( 60,  65, 0, True) # CleanRetail
6
>>> mar.accept_bid( 45,  40, 4, True) # IntelliWatt
7
>>> mar.accept_bid( 30,  38, 3, True) # QualiWatt
8
>>> mar.accept_bid( 35,  31, 4, True) # IntelliWatt
9
>>> mar.accept_bid( 25,  24, 0, True) # CleanRetail
10
>>> mar.accept_bid( 10,  21, 1, True) # El4You
11
```

```
>>> mar.accept_bid(120,   0, 5, False) # RT
12
>>> mar.accept_bid(50,    0, 6, False) # WeTrustInWind
13
>>> mar.accept_bid(200,  15, 7, False) # BlueHydro
14
>>> mar.accept_bid(400,  30, 5, False) # RT
15
>>> mar.accept_bid(60, 32.5, 8, False) # KøbenhavnCHP
16
>>> mar.accept_bid(50,   34, 8, False) # KøbenhavnCHP
17
>>> mar.accept_bid(60,   36, 8, False) # KøbenhavnCHP
18
>>> mar.accept_bid(100,37.5, 9, False) # DirtyPower
19
>>> mar.accept_bid(70,   39, 9, False) # DirtyPower
20
>>> mar.accept_bid(50,   40, 9, False) # DirtyPower
21
>>> mar.accept_bid(70,   60, 5, False) # RT
22
>>> mar.accept_bid(45,   70, 5, False) # RT
23
>>> mar.accept_bid(50,  100, 10, False) # SafePeak
24
>>> mar.accept_bid(60,  150, 10, False) # SafePeak
25
>>> mar.accept_bid(50,  200, 10, False) # SafePeak
26
>>> bids, index = pm.merge_same_price(mar.bm.get_df())
>>> mar.bm.get_df()
    quantity  price  user  buying  time  divisible
0        250  200.0     0    True     0       True
1        300  110.0     1    True     0       True
2        120  100.0     2    True     0       True
3         80   90.0     3    True     0       True
```

(continues on next page)

(continued from previous page)

```
4          40   85.0     4    True      0        True
5          70   75.0     1    True      0        True
6          60   65.0     0    True      0        True
7          45   40.0     4    True      0        True
8          30   38.0     3    True      0        True
9          35   31.0     4    True      0        True
10         25   24.0     0    True      0        True
11         10   21.0     1    True      0        True
12        120    0.0     5   False      0        True
13         50    0.0     6   False      0        True
14        200   15.0     7   False      0        True
15        400   30.0     5   False      0        True
16         60   32.5     8   False      0        True
17         50   34.0     8   False      0        True
18         60   36.0     8   False      0        True
19        100   37.5     9   False      0        True
20         70   39.0     9   False      0        True
21         50   40.0     9   False      0        True
22         70   60.0     5   False      0        True
23         45   70.0     5   False      0        True
24         50  100.0    10   False      0        True
25         60  150.0    10   False      0        True
26         50  200.0    10   False      0        True
```

`pymarket.bids.processing.`**`new_player_id`**`(`*index*`)`

Helper function for merge_same_price. Creates a function that returns consecutive integers.

Parameters **`index`** (*`int`*) – First identifier to use for the new fake players

Returns **Callable** – Function that maps a list of user ids into a new user id.

Return type function

Examples

```
>>> id_gen = new_player_id(6)
>>> id_gen([3])
3
>>> id_gen([5])
5
>>> id_gen([0, 1])
6
>>> id_gen([2, 4])
7
```

pymarket.datasets package

Top-level package for pymarket.

Submodules

pymarket.datasets.uniform_bidders module

`pymarket.datasets.uniform_bidders.`**`generate`**(*cant_buyers*, *cant_sellers*, *offset_sellers=0*, *offset_buyers=0*, *r=None*, *eps=0.0001*)

Generates random bids. All the volumes and reservation prices are sampled independently from a uniform distribution. For sellers, the reservation price is shifted *offset_seller* while for the buyers is shifter *offset_buyers*. If there are two sellers or two buyers with the same price, the reservation price of one of them is resampled until in both side of the market, all players have different values.

The maximum number of players is limited by 1/eps, although the parameter currently updates itself to allow the requested quantity of buyers and sellers.

Parameters

- **`cant_buyers`** (*`int`*) – Number of buyers to generate. Has to be positiv
- **`cant_sellers`** (*`int`*) – Number of sellers to generate. Has to be positive.
- **`offset_sellers`** (*`float`*) – Quantity to shift the reservation price of sellers
- **`offset_buyers`** (*`float`*) – Quantity to shift the reservation price of buyers
- **`r`** (*`optional, RandomState`*) – RandomState used to generate the data
- **`eps`** (*`optional, float`*) – Minimum precision of the prices.

Returns List of tuples of all the bids generated

Return type bids

Examples

```
>>> r = np.random.RandomState(420)
>>> generate(2, 3, 1, 2, r, 0.1)
[(0.5, 2.8, 0, True, 0, True), (0.7000000000000001, 2.5, 1, True, 0, True), (0.
↪6000000000000001, 1.2, 2, False, 0, True), (0.1, 1.7000000000000002, 3, False,␣
↪0, True), (0.2, 1.3, 4, False, 0, True)]
```

pymarket.mechanisms package

Top-level package for pymarket.

Submodules

pymarket.mechanisms.huang_auction module

class `pymarket.mechanisms.huang_auction.`**`HuangAuction`**(*bids*, **args*, ***kwargs*)

Bases: *`pymarket.mechanisms.mechanism.Mechanism`*

Iinterface for the HuangAuction

Parameters

- **`bids`** (*`pd.DataFrame`*) – Collection of bids to use in the market
- **`merge`** (*`bool`*) – Wheather to merge players with the same price. Always *True*

`pymarket.mechanisms.huang_auction.`**`huang_auction`**`(`*bids*`)`

Implements the auction described in [1]

Parameters **`bids`** (*`pd.DataFrame`*) – Collection of all the bids to take into account by the mechanism

Returns

- **trans** (*TransactionManager*) – Collection of all the trasactions cleared by the mechanism
- **extra** (*dict*) – Extra information provided by the mecanism. Keys: * price_sell: price at which sellers traded * price_buy: price at which the buyers traded * quantity_traded: the total quantity traded

Notes

[1] Huang, Pu, Alan Scheller–Wolf, and Katia Sycara. "Design of a multi–unit double auction e–market." Computational Intelligence 18.4 (2002): 596-617.

Examples

No trade because price setters don't trade:

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(2, 1, 1)
1
>>> bm.add_bid(2, 2, 2, False)
2
>>> trans, extra = huang_auction(bm.get_df())
>>> trans.get_df()
Empty DataFrame
Columns: [bid, quantity, price, source, active]
Index: []
>>> extra
OrderedDict([('price_sell', 2.0), ('price_buy', 3.0), ('quantity_traded', 0)])
```

Adding small bids at the beginning, those can trade because they don't define de market price:

```
>>> bm.add_bid(0.3, 1, 3, False)
3
>>> bm.add_bid(0.2, 3.3, 4)
4
>>> trans, extra = huang_auction(bm.get_df())
>>> trans.get_df()
   bid  quantity  price  source  active
0    3       0.2    2.0      -1   False
1    4       0.2    3.0      -1   False
>>> extra
OrderedDict([('price_sell', 2.0), ('price_buy', 3.0), ('quantity_traded', 0.2)])
```

`pymarket.mechanisms.huang_auction.`**`update_quantity`**`(`*quantity*, *gap*`)`

Implements the footnote in page 8 of [1], where the long side updates their trading quantities to match the short side.

Parameters

- **`quantity`** (*`np.ndarray`*) – List of the quantities to be traded by each player.
- **`gap`** (*`float`*) – Difference between the short and long side

Returns **quantity** – Updated list of quantities to be traded by each player

Return type np.ndarray

Notes

[1] Huang, Pu, Alan Scheller–Wolf, and Katia Sycara. "Design of a multi–unit double auction e–market." Computational Intelligence 18.4 (2002): 596-617.

Examples

All keep trading, with less quantity

```
>>> l, g = np.array([1, 2, 3]), 0.6
>>> update_quantity(l, g)
array([0.8, 1.8, 2.8])
```

The gap is to big for small trader:

```
>>> l,g = np.array([1, 0.5, 2]), 1.8
>>> update_quantity(l, g)
array([0.35, 0.  , 1.35])
```

pymarket.mechanisms.mechanism module

class `pymarket.mechanisms.mechanism.`**`Mechanism`**(*algo*, *bids*, **args*, *merge=False*, ***kwargs*)

Bases: `object`

Implements a standard interface for mechanisms

`algo`
Algorithm to execute to solve the market.

Type Callable

`bids`
Collection of bids to use, with processing.

Type pd.DataFrame

`old_bids`
Collection of bids previous to proecssing.

Type pd.DataFrame

`maping`
Map from the new bids to the old bids

Type dict

`merge`
Wheather to merge different players with the same price into one player. Useful for algorithms that require players to have different prices.

Type bool

Examples

Run p2p mechanism channging parameters with default parameters.

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 0.5, 1)
1
>>> bm.add_bid(1, 1, 2, False)
2
>>> bm.add_bid(1, 2, 3, False)
3
>>> r = np.random.RandomState(420)
>>> p2p = pm.mechanisms.p2p_random
>>> mec = Mechanism(p2p, bm.get_df(), r=r)
>>> trans, extra = mec.run()
>>> extra
{'trading_list': [[(0, 3), (1, 2)]]}
>>> trans.get_df()
   bid  quantity  price  source  active
0    0         1    2.5       3   False
1    3         1    2.5       0   False
2    1         0    0.0       2    True
3    2         0    0.0       1    True
```

```
>>> r = np.random.RandomState(420)
>>> mec = Mechanism(p2p, bm.get_df(), r=r, p_coef=1)
>>> trans, extra = mec.run()
>>> extra
{'trading_list': [[(0, 3), (1, 2)]]}
>>> trans.get_df()
   bid  quantity  price  source  active
0    0         1    3.0       3   False
1    3         1    3.0       0   False
2    1         0    0.0       2    True
3    2         0    0.0       1    True
```

`run()`
Runs the mechanisms

pymarket.mechanisms.muda_auction module

class `pymarket.mechanisms.muda_auction.`**`MudaAuction`**(*bids*, **args*, ***kwargs*)
Bases: *pymarket.mechanisms.mechanism.Mechanism*

Interface for MudaAuction.

Parameters **bids** – Collection of bids to run the mechanism with.

`pymarket.mechanisms.muda_auction.`**`compute_fee`**(*df*, *index*, *user*, *quantity*, *price*)
Computes the fee that a user has to pay by not letting others trade

Parameters

- **df** (*pd.DataFrame*) – Dataframe for one side of the market resulting from reseting the index of a bid dataframe, getting the bid as the first column in addition to all the standard ones. Precondition: all bids should be willing to trade at the trading price.
- **index** (*int*) – Index of the last trading bid
- **user** (*int*) – User identifier for which the fee should be computed
- **quantity** (*float*) – Total quantity that the side of the market trades
- **price** (*float*) – Price at which the market clears.

Returns **fee** – Fee that user *ùser* will have to pay for not letting others trade as well.

Return type float

Examples

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 1, 1)
0
>>> bm.add_bid(1, 2, 3)
1
>>> compute_fee(bm.get_df(), 0, 1, 1, 2.5)
0.5
```

`pymarket.mechanisms.muda_auction.`**`find_competitive_price`**(*bids*)

Finds the unique trading price of the intersection of supply and demand.

Parameters **bids** (*pd.DataFrame*) – Collection of bids to process the mechanism with.

Returns **price** – The price at which the market clears.

Return type float

Notes

See also: intersect_stepwise.

`pymarket.mechanisms.muda_auction.`**`get_trading_bids`**(*bids*, *quantity_traded*)

Finds the index of the rightmost trading bid in a side of the market. If the bid has to be split, it does so, and returns the a new bid dataframe with two bids in stade of the original one.

Parameters

- **bids** (*pd.DataFrame*) – Collection of bids in one side of the market Precondition: the dataframe is sorted by price. Reverse order for buying and selling side.
- **quantity_traded** (*float*) – Total quantity that the side of the market can trade.

Returns

- **bids_trading** (*pd.DataFrame*) – Same as *bids*, but the index (which represent the bid identifier) is added as the first column. If a bid had to be splitted, that bid is replaced by two, with the quantity in both summing up to the original quantity. The index is reseted but both splitted bids retain the oringal bid number in the column.
- **bid_index** (*int*) – Index of the *worst* bid that gets to trade.

Examples

No splitting needed

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 2, 1)
1
>>> bm.get_df()
   quantity  price  user  buying  time  divisible
0         1      3     0    True     0       True
1         1      2     1    True     0       True
>>> bids, index = get_trading_bids(bm.get_df(), 1)
>>> bids
   bid  quantity  price  user  buying  time  divisible
0    0         1      3     0    True     0       True
1    1         1      2     1    True     0       True
>>> index
0
```

Splitting needed:

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 2, 1)
1
>>> bm.get_df()
   quantity  price  user  buying  time  divisible
0         1      3     0    True     0       True
1         1      2     1    True     0       True
>>> bids, index = get_trading_bids(bm.get_df(), 0.3)
>>> bids
   bid quantity price user buying time divisible
0    0      0.3     3    0    True     0      True
1    0      0.7     3    0    True     0      True
2    1        1     2    1    True     0      True
>>> index
0
```

`pymarket.mechanisms.muda_auction.`**`muda`**(*bids*, *r=None*)

Implements the Vickrey MUDA as described in [1].

The mechanism does not support two players in the same side of the market with the same price.

Parameters

- **`bids`** (*`pd.DataFrame`*) – Collection of bids to be used in the market
- **`r`** (*`np.random.RandomState`*) – A numpy random state generator. If not given, a new one will be created and the output will be random.

Returns

- **trans** (*TransactionManager*) – A collection of all the transactions performed.
- **extra** (*dict*) – Dictionary with extra information provided by the mechanism. Keys: * left: players in the left market * right: players in the right market * price_left: clearing price of the left market * price_right: clearing price of the right_market * fees: Fees that players have to pay to participate

Notes

[1] Segal-Halevi, Erel, Avinatan Hassidim, and Yonatan Aumann. “MUDA: a truthful multi-unit double-auction mechanism.” Thirty-Second AAAI Conference on Artificial Intelligence. 2018.

Examples

A case in which the market puts all the players in the same side and no one trades.

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 4, 0)
0
>>> bm.add_bid(1, 2, 1)
1
>>> bm.add_bid(1, 3, 2, False)
2
>>> bm.add_bid(1, 1, 3, False)
3
>>> r = np.random.RandomState(420)
>>> trans, extra = muda(bm.get_df(), r)
>>> extra
OrderedDict([('left', []), ('right', [0, 1, 2, 3]), ('price_left', inf), ('price_
↪right', 2.5), ('fees', array([0., 0., 0., 0.]))])
>>> trans.get_df()
Empty DataFrame
Columns: [bid, quantity, price, source, active]
Index: []
```

A case in which there are 2 players in each side but the cleared prices makes it impossible to trade:

```
>>> r = np.random.RandomState(69)
>>> trans, extra = muda(bm.get_df(), r)
>>> extra
OrderedDict([('left', [1, 3]), ('right', [0, 2]), ('price_left', 1.5), ('price_
↪right', 3.5), ('fees', array([0., 0., 0., 0.]))])
>>> trans.get_df()
Empty DataFrame
Columns: [bid, quantity, price, source, active]
Index: []
```

A case with trade:

```
>>> bm.add_bid(1, 5, 4)
4
>>> r = np.random.RandomState(69)
>>> trans, extra = muda(bm.get_df(), r)
>>> trans.get_df()
   bid  quantity  price  source  active
0    3         1    3.5      -1   False
1    4         1    3.5      -1   False
2    2         1    3.0      -1   False
3    0         1    3.0      -1   False
>>> extra
OrderedDict([('left', [1, 3, 4]), ('right', [0, 2]), ('price_left', 3.0), ('price_
↪right', 3.5), ('fees', array([0., 0., 0., 0., 0.]))])
```

`pymarket.mechanisms.muda_auction.`**`solve_market_side_with_exogenous_price`**(*bids*, *price*, *fees*)

Clears the market based on an external price. First it removes all biders that are not willing to trade at the given price, and then it fits the best allocation. Fees are calculated based on users that were willing but could not trade.

Parameters

- **`bids`** (*`pd.DataFrame`*) – Collection of bids to clear the market with
- **`price`** (*`float`*) – Price at which all the trades will ocurr
- **`fees`** (*`list of floats`*) – List of all the fees that players will have to pay. It gets updated.

Returns

- **trans** (*TransactionManager*) – Collection of the transactions that clear the market
- **fees** (*list of floats*) – Fees to be paid by each player. Is a list where the fee of player with id *u* is located at *fees[u]*.

Examples

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 0.5, 1)
1
>>> bm.add_bid(1, 1, 2, False)
2
>>> bm.add_bid(1, 2, 3, False)
3
>>> fees = [0, 0, 0, 0]
>>> trans, fees = solve_market_side_with_exogenous_price(bm.get_df(),2.5, fees)
>>> trans.get_df()
   bid  quantity  price  source  active
0    0         1    2.5      -1   False
1    2         1    2.5      -1   False
>>> fees
[0, 0, 0.5, 0]
```

pymarket.mechanisms.p2p_random module

`class` `pymarket.mechanisms.p2p_random.`**`P2PTrading`**(*bids*, **args*, ***kwargs*)

Bases: *`pymarket.mechanisms.mechanism.Mechanism`*

Interface for P2PTrading.

Parameters **`bids`** (*`pd.DataFrame`*) – Collections of bids to use

`pymarket.mechanisms.p2p_random.`**`p2p_random`**(*bids*, *p_coef=0.5*, *r=None*)

Computes all the trades using a P2P random trading process inspired in [1].

Parameters

- **`bids`** (*`pd.DataFrame`*) – Collection of bids that will trade. Precondition: a user participates only in one side of the market, i.e, it cannot sell and buy in the same run.

- **`p_coef`** (*`float`*) – coefficient to calculate the trading price as a convex combination of the price of the seller and the price of the buyer. If 1, the seller gets all the profit and if 0, the buyer gets all the profit.
- **`r`** (*`np.random.RandomState`*) – Random state to generate stochastic values. If None, then the outcome of the market will be different on each run.

Returns

- **trans** (*TransactionManger*) – Collection of all the transactions that ocurred in the market
- **extra** (*dict*) – Extra information provided by the mechanisms. Keys:
  - trading_list: list of list of tuples of all the pairs that traded in each round.

Notes

[1] Blouin, Max R., and Roberto Serrano. “A decentralized market with common values uncertainty: Non-steady states.” The Review of Economic Studies 68.2 (2001): 323-346.

Examples

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 0.5, 1)
1
>>> bm.add_bid(1, 1, 2, False)
2
>>> bm.add_bid(1, 2, 3, False)
3
>>> r = np.random.RandomState(420)
>>> trans, extra = p2p_random(bm.get_df(), r=r)
>>> extra
{'trading_list': [[(0, 3), (1, 2)]]}
>>> trans.get_df()
   bid  quantity  price  source  active
0    0         1    2.5       3   False
1    3         1    2.5       0   False
2    1         0    0.0       2    True
3    2         0    0.0       1    True
```

pymarket.plot package

Submodules

pymarket.plot.demand_curves module

`pymarket.plot.demand_curves.`**`plot_demand_curves`**(*bids*, *ax=None*, *margin_X=1.2*, *margin_Y=1.2*)

Plots the demand curves. If ax is none, creates a new figure

Parameters

- **`bids`** – Collection of bids to be used

- **ax** (*TODO, optional*) – (Default value = None)
- **margin_X** – (Default value = 1.2)
- **margin_Y** – (Default value = 1.2)

pymarket.plot.huang module

`pymarket.plot.huang.`**`plot_huang_auction`**(*bids*, *price_sell*, *price_buy*, *quantity_traded*, *ax=None*)

Plots the results of the huang auction with some of the characteristics of such auction

Parameters

- **(pandas dataframe)** (*bids*) – Table with all the bids submitted
- **(list)** (*price_buy*) – The price at which all sellers sell
- **(list)** – The price at which all players buy
- **traded (float)** (*quantity*) – The total quantity traded

Returns **axe** – The axe in which the figure was plotted.

Return type matplotlib.axes._subplots.AxesSubplot

pymarket.plot.muda module

`pymarket.plot.muda.`**`plot_both_side_muda`**(*bids*, *left_players*, *right_players*, *left_price*, *right_price*, *FIGSIZE=(12, 6)*, ***kwargs*)

Plots the two sides in which MUDA divides the trades with the corresponding prices

Parameters

- **(pandas dataframe)** (*bids*) – Table with all the bids submitted
- **(list)** (*right*) – List of players in the left side
- **(list)** – List of players in the right side
- **(float)** (*right_price*) – Price obtained from the left side to be used in the right side
- **(float)** – Price obtained from the right side to be used in the left side
- **(tuple)** (*FIGSIZE*) – Tuple (width, height) of the figure to be created

Returns **axe** – The axe in which the figure was plotted.

Return type matplotlib.axes._subplots.AxesSubplot

pymarket.plot.trades module

`pymarket.plot.trades.`**`plot_trades_as_graph`**(*bids*, *transactions*, *ax=None*)

Plots all the bids as a bipartit graph with buyers and trades and an edge between each pair that traded

Parameters

- **bids** (*pd.DataFrame*) – Collection of bids to be used
- **transactions** (*pd.DataFrame*) – Collection of transactions to be used
- **ax** (*pyplot.axe*) – The axe in which the figure should be ploted

Returns **axe** – The axe in which the figure was plotted.

Return type matplotlib.axes._subplots.AxesSubplot

pymarket.statistics package

Submodules

pymarket.statistics.maximum_aggregated_utility module

`pymarket.statistics.maximum_aggregated_utility.`**`maximum_aggregated_utility`**(*bids*, **args*, *reservation_prices=None*)

Maximizes the total welfare

Parameters

- **`bids`** (*`pd.DataFrame`*) – Collection of bids
- **`reservation_prices`** (*`dict of floats or None, (Default value = None)`*) – A maping from user ids to reservation prices. If no reservation price for a user is given, his bid will be assumed to be his true value.

Returns

- **status** (*str*) – Status of the optimization problem. Desired output is 'Optimal'
- **objective** (*float*) – Maximum aggregated utility that can be obtained
- **variables** (*dict*) – A set of values achieving the objective. Maps a pair of bids to the quantity traded by them.

Examples

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 2, 1)
1
>>> bm.add_bid(1.5, 1, 2, False)
2
>>> s, o, v = maximum_aggregated_utility(bm.get_df())
>>> s
'Optimal'
>>> o
2.5
>>> v
OrderedDict([((0, 2), 1.0), ((1, 2), 0.5)])
```

If in reality the seller had 0 value for his commodity, the social welfare will be 1.5 units larger

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 2, 1)
```

(continues on next page)

(continued from previous page)

```
1
>>> bm.add_bid(1.5, 1, 2, False)
2
>>> rp = {2: 0}
>>> s, o, v = maximum_aggregated_utility(bm.get_df(),
...           reservation_prices=rp)
>>> s
'Optimal'
>>> o
4.0
>>> v
OrderedDict([((0, 2), 1.0), ((1, 2), 0.5)])
```

`pymarket.statistics.maximum_aggregated_utility.`**`percentage_welfare`**(*bids*, *transactions*, *reservation_prices=None*, ***kwargs*)

Percentage of the total welfare that could be achieved calculated based on the transaction lists

Parameters

- **`(pandas dataframe)`** (*`transactions`*) – Table with all the submited bids
- **`(pandas dataframe)`** – Table with all the transactions that ocurred in the market
- **`(dict, optional)`** (*`reservation_prices`*) – Reservation prices of the different participants. If None, the bids will be assumed to be the truthfull values.

Returns **ratio** – The ratio of the maximum social welfare achieved by the collection of transactions.

Return type float

Examples

Only bid 0 and 2 trade. That represents a net utility of 2 which is 80% of the total max utility 2.5

```
>>> tm = pm.TransactionManager()
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 2, 1)
1
>>> bm.add_bid(1.5, 1, 2, False)
2
>>> tm.add_transaction(0, 1, 2, 2, False)
0
>>> tm.add_transaction(2, 1, 2, 0, False)
1
>>> percentage_welfare(bm.get_df(), tm.get_df())
0.8
```

pymarket.statistics.maximum_traded_volume module

`pymarket.statistics.maximum_traded_volume.`**`maximum_traded_volume`**(*bids*, **args*, *reservation_prices={}*)

Parameters

- **`bids`** (*`pd.DataFrame`*) – Collections of bids
- **`reservation_prices`** (*`dict of floats or None, (Default value = None)`*) – A maping from user ids to reservation prices. If no reservation price for a user is given, his bid will be assumed to be his true value.

Returns

- **status** (*str*) – Status of the optimization problem. Desired output is 'Optimal'
- **objective** (*float*) – Maximum tradable volume that can be obtained
- **variables** (*dict*) – A set of values achieving the objective. Maps a pair of bids to the quantity traded by them.

Examples

```
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 2, 1)
1
>>> bm.add_bid(1.5, 1, 2, False)
2
>>> s, o, v = maximum_traded_volume(bm.get_df())
>>> s
'Optimal'
>>> o
1.5
>>> v
OrderedDict([((0, 2), 0.5), ((1, 2), 1.0)])
```

`pymarket.statistics.maximum_traded_volume.`**`percentage_traded`**(*bids*, *transactions*, *reservation_prices={}*, ***kwargs*)

Calculates from the transaction dataframe the percentage of the total maximum possible traded quantity.

Parameters

- **`(pandas dataframe)`** (*`transactions`*) – Table with all the submited bids
- **`(pandas dataframe)`** – Table with all the transactions that ocurred in the market
- **`(dict, optional)`** (*`reservation_prices`*) – Reservation prices of the different participants. If None, the bids will be assumed to be the truthfull values.

Returns **ratio** – The ratio of the maximum social welfare achieved by the collection of transactions.

Return type float

Examples

Only bid 0 and 2 trade 1 unit. That represents the 66% of all that could have been traded.

```
>>> tm = pm.TransactionManager()
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 2, 1)
1
>>> bm.add_bid(1.5, 1, 2, False)
2
>>> tm.add_transaction(0, 1, 2, 2, False)
0
>>> tm.add_transaction(2, 1, 2, 0, False)
1
>>> percentage_traded(bm.get_df(), tm.get_df())
0.6666666666666666
```

pymarket.statistics.profits module

`pymarket.statistics.profits.`**`calculate_profits`**(*bids*, *transactions*, *reservation_prices=None*, *fees=None*, ***kwargs*)

Extras from the transactions and the bids the profit of each player and the market maker

Parameters

- **bids** (*pd.DataFrame*) – Collections of bids to be used
- **transactions** (*pd.DataFrame*) – Collection of transactions to be taken into account
- **reservation_prices** (*dict, (Default value = None)*) – Maping between users and their reservation prices. If None, it is assumed that each user bided truthfully and the information is extracted from the bid.
- **fees** (*np.ndarray, (Default value = None)*) – List of fees that each user has to pay to trade in the market.

Returns **profit** – A dictionary with three values: * player_bid: A list with the profits of each user using their bids as reservation prices * player_reservation: Same as above but using their reservation prices, if none are provided, it is the same as *player_bid* * market: profit of the market maker

Return type dict

Examples

```
>>> tm = pm.TransactionManager()
>>> bm = pm.BidManager()
>>> bm.add_bid(1, 3, 0)
0
>>> bm.add_bid(1, 2, 1)
1
>>> bm.add_bid(1.5, 1, 2, False)
2
```

(continues on next page)

(continued from previous page)

```
>>> tm.add_transaction(0, 1, 2, 2, False)
0
>>> tm.add_transaction(2, 1, 2, 0, False)
1
>>> rp = {2: 0}
>>> profits = calculate_profits(bm.get_df(), tm.get_df(),
...         reservation_prices=rp)
>>> profits['player_bid']
array([1., 0., 1.])
>>> profits['player_reservation']
array([1., 0., 2.])
>>> profits['market']
0.0
```

`pymarket.statistics.profits.`**`get_gain`**`(`*row*`)`

Finds the gain of the row

Parameters **`row`** (`pandas row`) – Row obtained by merging a transaction with a bid dataframe

Returns The gain obtained by the row

Return type gain

pymarket.statistics.statistics module

pymarket.transactions package

Submodules

pymarket.transactions.processing module

Some processing functions to deal with transactions

`pymarket.transactions.processing.`**`split_transactions_merged_players`**`(`*transactions*, *bids*, *maping*, *fees=None*`)`

Splits the transactions of a market that used merged bids into the original bids Uses a proportional split, based on the offered (or asked) quantity by each player.

Parameters

- **`transactions`** (`TransactionManager`) – the transactions manager returned by the mechanism.
- **`bids`** (`pandas dataframe`) – the original bid dataframe where some players might be repeated
- **`maping`** (`pandas dataframe`) – A maping between the bids in the transaction dataframe and the original bids.

Returns

- **transactions_splited** (*pandas dataframe*) – the result of splitting each merged bid in the transactions dataframe
- **fees** (*dict or None*) – dictionary obtained by splitting the fees equal to the transactions

Examples

```
>>> bm = pm.BidManager()
>>> tm = pm.TransactionManager()
>>> bm.add_bid(1, 1, 0)
0
>>> bm.add_bid(2, 1, 1)
1
>>> tm.add_transaction(0, 1, 1, -1, False)
0
>>> tm_2 = split_transactions_merged_players(tm, bm.get_df(), {0:[0,1]})
>>> tm_2.get_df()
   bid  quantity  price  source  active
0    0  0.333333      1      -1   False
1    1  0.666667      1      -1   False
```

pymarket.transactions.transactions module

class `pymarket.transactions.transactions.`**`TransactionManager`**

Bases: `object`

An interaface to store and manage all transactions. Transactions are the minimal unit to represent the outcome of a market.

`name_col`

Name of the columns to use in the dataframe returned.

Type list of str

`n_trans`

Number of transactions currently in the Manager

Type int

`trans`

List of the actual transactions available

Type list of tuples

`add_transaction`(*bid*, *quantity*, *price*, *source*, *active*)

Add a transaction to the transactions list

Parameters

- **`bid`** (*`int`*) – Unique identifier of the bid
- **`quantity`** (*`float`*) – transacted quantity
- **`price`** (*`float`*) – transacted price
- **`source`** (*`int`*) – Identifier of the second party in the trasaction, -1 if there is no clear second party, such as in a double auction.
- **`active`** – True‘ if the bid is still active after the transaction.

Returns **trans_id** – id of the added transaction, -1 if fails

Return type int

Examples

```
>>> tm = pm.TransactionManager()
>>> tm.add_transaction(1, 0.5, 2.1, -1, False)
0
>>> tm.trans
[(1, 0.5, 2.1, -1, False)]
>>> tm.n_trans
1
```

`get_df`()

Returns the transaction dataframe

Returns **df** – A pandas dataframe representing all the transactions stored.

Return type pd.DataFrame

Examples

```
>>> tm = pm.TransactionManager()
>>> tm.add_transaction(1, 0.5, 2.1, -1, False)
0
>>> tm.add_transaction(5, 0, 0, 3, True)
1
>>> tm.get_df()
   bid  quantity  price  source  active
0    1       0.5    2.1      -1   False
1    5       0.0    0.0       3    True
```

`merge`(*other*)

Merges two transaction managers with each other There are no checks on whether the new TransactionManger is consisten after the merge.

Parameters **`other`** (`TransactionManager`) – A different transaction manager to merge with

Returns **trans** – A new transaction Manager with the transactions of the two.

Return type *TransactionManager*

Examples

```
>>> tm_1 = pm.TransactionManager()
>>> tm_1.add_transaction(1, 0.5, 2.1, -1, False)
0
>>> tm_2 = pm.TransactionManager()
>>> tm_2.add_transaction(5, 0, 0, 3, True)
0
>>> tm_3 = tm_1.merge(tm_2)
>>> tm_3.get_df()
   bid  quantity  price  source  active
0    1       0.5    2.1      -1   False
1    5       0.0    0.0       3    True
```

`name_col = ['bid', 'quantity', 'price', 'source', 'active']`

pymarket.utils package

Top-level package for pymarket.

Submodules

pymarket.utils.decorators module

`pymarket.utils.decorators.`**`check_equal_price`**`(`*f*`)`

CHeck wheather there are two bids with the same price in the same side and in that case rises an error

Parameters **`f`** (*`(function, mechanisms)`*) – Mechanisms to be tested

4.1.2 Submodules

pymarket.conftest module

`pymarket.conftest.`**`add_namespace`**`(`*doctest_namespace*`)`

pymarket.market module

`class` `pymarket.market.`**`Market`**

Bases: `object`

General interface for calling the different market mechanisms

Parameters

- **`bm`** (`BidManager`) – All bids are stored in the bid manager
- **`transactions`** (`TransactionManager`) – The set of all tranasactions in the Market. This argument get updated after the market ran.
- **`extra`** (*`dict`*) – Extra information provided by the mechanisms. Gets updated after an execution of the run.

Examples

If everyone is buying, the transaction dataframe is returned empty as well as the extra dictionary.

```
>>> mar = pm.Market()
>>> mar.accept_bid(1, 2, 0, True)
0
>>> mar.accept_bid(2, 3, 1, True)
1
>>> trans, extra = mar.run('huang')
>>> extra
OrderedDict()
>>> trans.get_df()
Empty DataFrame
Columns: [bid, quantity, price, source, active]
Index: []
```

If everyone is buying, the transaction dataframe is returned empty as well as the extra dictionary.

```
>>> mar = pm.Market()
>>> mar.accept_bid(1, 2, 0, False)
0
>>> mar.accept_bid(2, 3, 1, False)
1
>>> trans, extra = mar.run('huang')
>>> extra
OrderedDict()
>>> trans.get_df()
Empty DataFrame
Columns: [bid, quantity, price, source, active]
Index: []
```

A very simple auction where nobody trades

```
>>> mar = pm.Market()
>>> mar.accept_bid(1, 3, 0, True)
0
>>> mar.accept_bid(1, 2, 1, False)
1
>>> trans, extra = mar.run('huang')
>>> extra
OrderedDict([('price_sell', 2.0), ('price_buy', 3.0), ('quantity_traded', 0)])
>>> trans.get_df()
Empty DataFrame
Columns: [bid, quantity, price, source, active]
Index: []
```

`accept_bid`(**args*)

Adds a bid to the bid manager

Parameters **`*args`** – List of parameters requried to create a bid. See *BidManager* documentation.

Returns **bid_id** – The id of the new created bid in the BidManger

Return type int

`plot`()

Plots both demand curves

`plot_method`(*method*, *ax=None*)

Plots a figure specific for a given method, reflecting the main characteristics of its solution. It requires that the algorithm has run before.

Parameters

- **`method`** (*`str`*) – One of *p2p*, *muda*, *huang*
- **`ax`** – (Default value = None)

`run`(*algo*, **args*, ***kwargs*)

Runs a given mechanism with the current bids

Parameters

- **`algo`** (*`str`*) –

  One of:

  - ’p2p’
  - ’huang’

    - ’muda’

- **`*args`** – Extra arguments to pass to the algorithm.
- **`**kwargs`** – Extra keyworded arguments to pass to the algorithm

Returns

- **transactions** (*TransactionManager*) – The transaction manager holding all the transactions returned by the mechanism.
- **extra** (*dict*) – Dictionary with extra information returned by the executed method.

`statistics`(*reservation_prices=None*, *exclude=[]*)

Computes the standard statistics of the market

Parameters

- **`(dict, optional)`** (*`reservation_prices`*) – the reservation prices of the users. If there is none, the bid will be assumed truthfull
- **`reservation_prices`** – (Default value = None)
- **`exclude`** – List of mechanisms to ignore will comuting statistics

Returns

stats –

Dictionary with the differnt statistics. Currently:

- percentage_welfare
- percentage_traded
- profits

Return type dict

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/gus0k/pymarket/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

5.1.4 Write Documentation

pymarket could always use more documentation, whether as part of the official pymarket docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/gus0k/pymarket/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *pymarket* for local development.

1. Fork the *pymarket* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pymarket.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pymarket
$ cd pymarket/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 pymarket tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/gus0k/pymarket/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Credits

6.1 Team

- Diego Kiedanski
- Daniel Kofman
- José Horta

6.2 Development Lead

- Diego Kiedanki <gusok@protonmail.com>

6.3 Contributors

None yet. Why not be the first?

CHAPTER 7

References

7.1 Algorithms Used

- Segal-Halevi, Erel, Avinatan Hassidim, and Yonatan Aumann. “MUDA: a truthful multi-unit double-auction mechanism.” Thirty-Second AAAI Conference on Artificial Intelligence. 2018.
- Huang, Pu, Alan Scheller–Wolf, and Katia Sycara. “Design of a multi–unit double auction e–market.” Computational Intelligence 18.4 (2002): 596-617.
- Blouin, Max R., and Roberto Serrano. “A decentralized market with common values uncertainty: Non-steady states.” The Review of Economic Studies 68.2 (2001): 323-346.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index