mm-case-study-slides

May 24, 2022

[1]:
```
import warnings
warnings.filterwarnings("ignore")
```

1 Reinforcement-learning case-study

Steve Phelps

1.1 Overview

- We will implement a simplified algorithmic trading strategy in Python using reinforcement-learning.
- Our agent will attempt to earn profit from market-making in a high-frequency market.
- We will consider a simplified environment, but the problem can be scaled up to more realistic scenarios.
- It is based on the work of Chan and Shelton 2001; "An Electronic Market-Maker".
- A more structured, and fully commented, version of the code used in these notes is available on github.

1.2 Training an agent in a simulation

- Allowing an agent to learn in the real-world can be risky.
- We can train our agent in a simulation environment.
- The simulation environment is sometimes called a "gym".
- If the simulation is a good model of the real-world then the learned policy will still perform well in real-world.
- The agent can still adapt its policy online in the real-world even if the simulation is not a good model.

1.3 The market model

- We consider three types of agent:
  - informed traders,
  - uninformed traders, and

  - a single market-maker.
- Prices evolve intra-day in discrete time periods $t \in \mathbb{N}$.
- A single asset is traded.
- All trades and orders involve a single share of the asset.
- There is no order crossing between traders.
- The arrival of traders at the market follow a Poisson process.

1.3.1 The fundamental price

- The true price of the asset $p_t^* \in \mathbb{Z}$ follows a Poisson process.
- The parameter $\lambda_p \in [0, 1]$ is the probability of a discrete change in the price.

$$p_t^*(t) = p_0 + \sum_{i=1}^{t} \eta_t \tag{1}$$

where η_t is chosen i.i.d. from $(-1, +1, 0)$ with probabilities $(\lambda_p, \lambda_p, 1 - 2\lambda_p)$ respectively.

1.3.2 The market-maker

- In the simplest form of the model, the market-maker posts a single price p_t^m.
- The market-maker can adjust its price:

$$p_{t+1}^m = p_t + \Delta p_t \tag{2}$$

where the price changes are discrete and finite; $\Delta p_t \in \{-1, 0, +1\}$.

- The reward at time t is the change in the profit:
  - for a sell order: $r_t = p_t^* - p_t^m$.
  - for a buy order: $r_t = p_t^m - p_t^*$.
- More advanced versions of the model consider separate bid and ask prices, and corresponding spread.

1.3.3 Informed traders

- Informed traders have information about the fundamental price p_t^*.
- They can submit market orders for immediate execution at the market-maker's price p_t^m.
- They submit
  - a buy order iff. $p_t^* > p_t^m$.
  - a sell order iff. $p_t^* < p_t^m$.
  - no order iff. $p_t^* = p_t^m$.
- They arrive at the market with probability λ_i.

1.3.4 Uninformed traders

- Uninformed traders arrive at the market with probability $2\lambda_u$.
- They submit a buy order for $+1$ shares with probability λ_u, or a sell order for -1 shares with equal probability λ_u.

1.3.5 The overall process

- All Poisson processes are combined:

$$2\lambda_p + 2\lambda_u + \lambda_i = 1 \tag{3}$$

- There is an event at every discrete time period t.
- Trade occurs a finite period of time $t \in \{1, 2, \ldots, T\}$ where T is the duration of a single trading day.
- The market maker operates over many days.
- The initial conditions for every day are the same; they are independent *episodes*.

1.3.6 The market-maker as an agent

- We can consider the market-maker as an adaptive agent.
- The environment consists of the observable variables in the market.
- Initially the observable state is the total order-imbalance IMB_t.
- The variables are discrete, therefore there is a discrete state space.
- The market-maker chooses actions in discrete time periods t.
- The set of actions $\mathbb{A}$ available to the agent is $\Delta p \in \mathbb{A} = \{-1, 0, +1\}$.
- It can choose actions conditional on observations in order to maximise expected return $E[G]$ where $G = \sum_t \gamma^t r_t$.

1.3.7 Importing the required modules

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

1.3.8 Parameterising the model

- In the following examples we use the parameterisation of the model:

$$
\begin{aligned}
\lambda_p &= 0.2 \\
\lambda_u &= 0.1 \\
\lambda_i &= 0.4 \\
T &= 150 \\
p_0^* &= 200
\end{aligned}
$$

1.3.9 Setting up the parameters in Python

[3]:
```
INITIAL_PRICE = 200
MAX_T = 150

PROB_PRICE = 0.2

PROB_PRICE_UP = PROB_PRICE
PROB_PRICE_DOWN = PROB_PRICE

PROB_UNINFORMED = 0.1

PROB_UNINFORMED_BUY = PROB_UNINFORMED
PROB_UNINFORMED_SELL = PROB_UNINFORMED

PROB_INFORMED = 0.4

ALL_PROB = [PROB_PRICE_DOWN, PROB_PRICE_UP, PROB_UNINFORMED_BUY,␣
 ↪PROB_UNINFORMED_SELL, PROB_INFORMED]
```

[4]:
```
np.sum(ALL_PROB)
```

[4]:
```
1.0
```

1.3.10 Representing events

[5]:
```
EVENT_PRICE_CHANGE_UP    = 0
EVENT_PRICE_CHANGE_DOWN  = 1
EVENT_UNINFORMED_BUY     = 2
EVENT_UNINFORMED_SELL    = 3
EVENT_INFORMED_ARRIVAL   = 4
```

[6]:
```
ALL_EVENT = \
    [EVENT_PRICE_CHANGE_DOWN, EVENT_PRICE_CHANGE_UP, EVENT_UNINFORMED_SELL,
         EVENT_UNINFORMED_BUY, EVENT_INFORMED_ARRIVAL]
```

1.3.11 Simulating the Poisson process

[7]:
```
def simulate_events(probabilities=ALL_PROB):
     return np.random.choice(ALL_EVENT, p=probabilities, size=MAX_T)
```

[8]:
```
events = simulate_events()
```

The first ten events:

[9]:
```
events[:10]
```

[9]:
```
array([2, 1, 0, 0, 1, 3, 0, 4, 0, 2])
```

1.3.12 Simulating the price process

[10]:
```
fundamental_price_changes = np.zeros(MAX_T)
```

[11]:
```
fundamental_price_changes[events == EVENT_PRICE_CHANGE_DOWN] = -1
fundamental_price_changes[events == EVENT_PRICE_CHANGE_UP] = +1
```

[12]:
```
fundamental_price_changes[:10]
```

[12]:
```
array([ 0., -1.,  1.,  1., -1.,  0.,  1.,  0.,  1.,  0.])
```

[13]:
```
fundamental_price = \
    INITIAL_PRICE + np.cumsum(fundamental_price_changes)
```

[14]:
```
fundamental_price[:10]
```

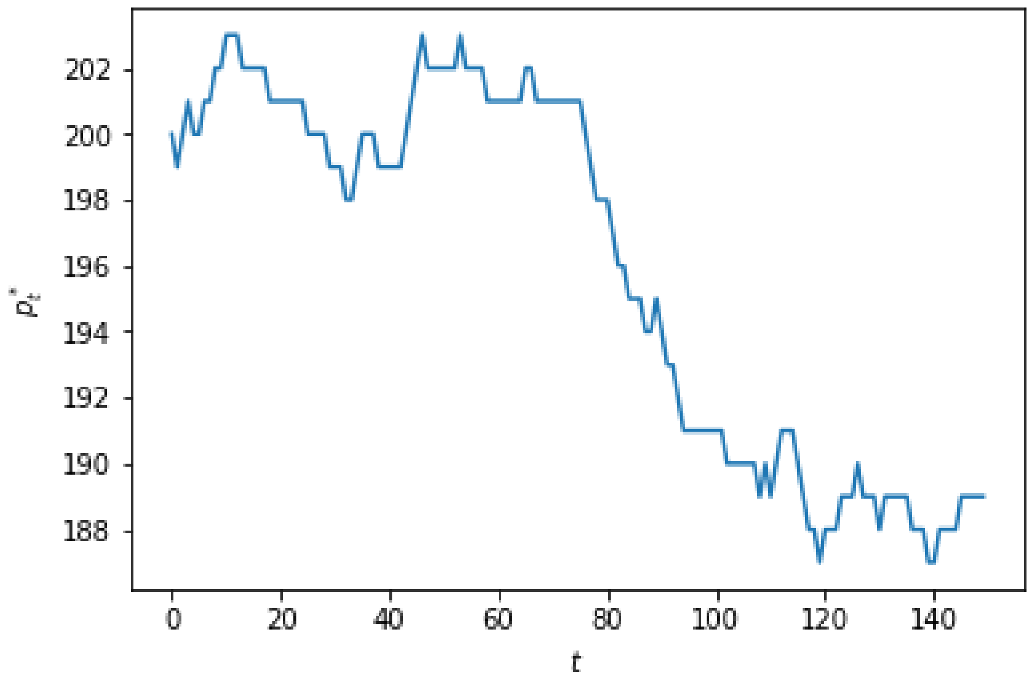
[14]:
```
array([200., 199., 200., 201., 200., 200., 201., 201., 202., 202.])
```

As a function

[15]:
```
def simulate_fundamental_price(events):
    price_changes = np.zeros(MAX_T)
    price_changes[events == EVENT_PRICE_CHANGE_DOWN] = -1
    price_changes[events == EVENT_PRICE_CHANGE_UP] = +1
    return INITIAL_PRICE + np.cumsum(price_changes)

fundamental_price = simulate_fundamental_price(events)
```

A single realisation of the price process.

[16]:
```
plt.plot(fundamental_price)
plt.xlabel('$t$'); plt.ylabel('$p_t^*$')
plt.show()
```



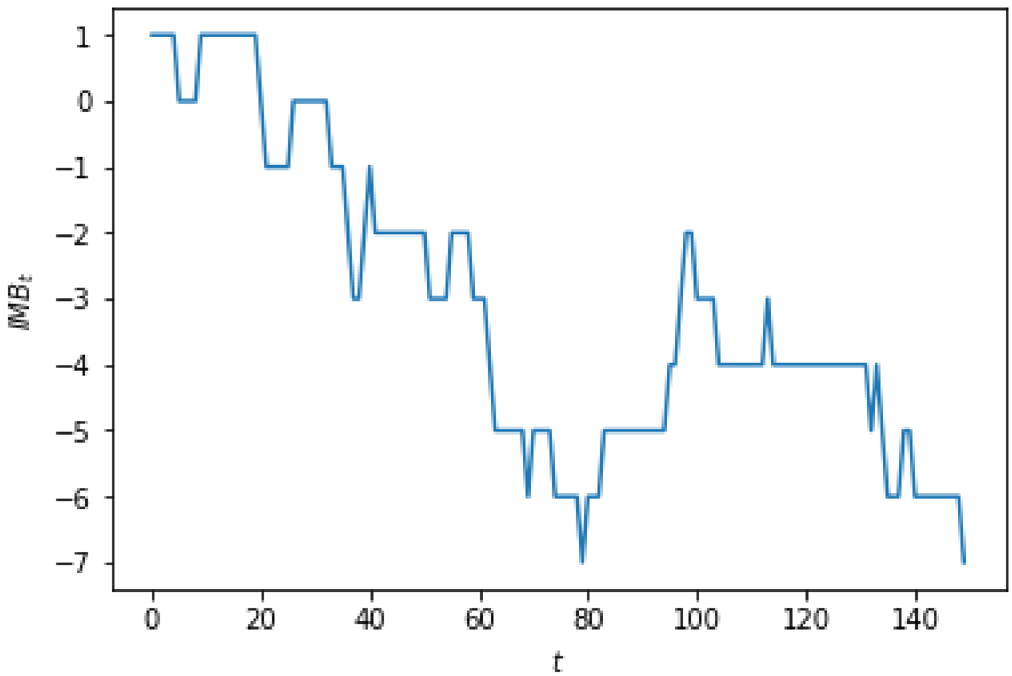
1.3.13 Uninformed traders

[17]:
```python
def simulate_uninformed_orders(events):
    orders = np.zeros(MAX_T)
    orders[events == EVENT_UNINFORMED_BUY] = +1
    orders[events == EVENT_UNINFORMED_SELL] = -1
    return orders

uninformed_orders = simulate_uninformed_orders(events)
uninformed_orders[:10]
```

[17]:
```
array([ 1.,  0.,  0.,  0.,  0., -1.,  0.,  0.,  0.,  1.])
```

1.3.14 Uninformed order-imbalance

[18]:
```python
plt.plot(np.cumsum(uninformed_orders))
plt.xlabel('$t$'); plt.ylabel('$IMB_t$')
plt.show()
```



1.3.15 Informed traders

[19]:
```
def informed_strategy(current_price, mm_price):
    if current_price > mm_price:
        return 1
    elif current_price < mm_price:
        return -1
    else:
        return 0
```

1.3.16 A simple market-making strategy

- Initially we consider a very simple policy for our market-making agent.
- The policy π_h is parameterised by a single threshold parameter h.
  - Increase the price by a single tick if the order-imbalance is $+h$.
  - Decrease the price by a single tick if the order-imbalance is $-h$.

[20]:
```
def mm_threshold_strategy(order_imbalance, threshold=2):
    if order_imbalance == -threshold:
        return -1
    elif order_imbalance == +threshold:
        return +1
    else:
```

```
        return 0
```

1.3.17 The reward function

[21]:
```
def mm_reward(current_fundamental_price, mm_current_price, order_sign):
    if order_sign < 0:
        return current_fundamental_price - mm_current_price
    elif order_sign > 0:
        return mm_current_price - current_fundamental_price
    else:
        return 0
```

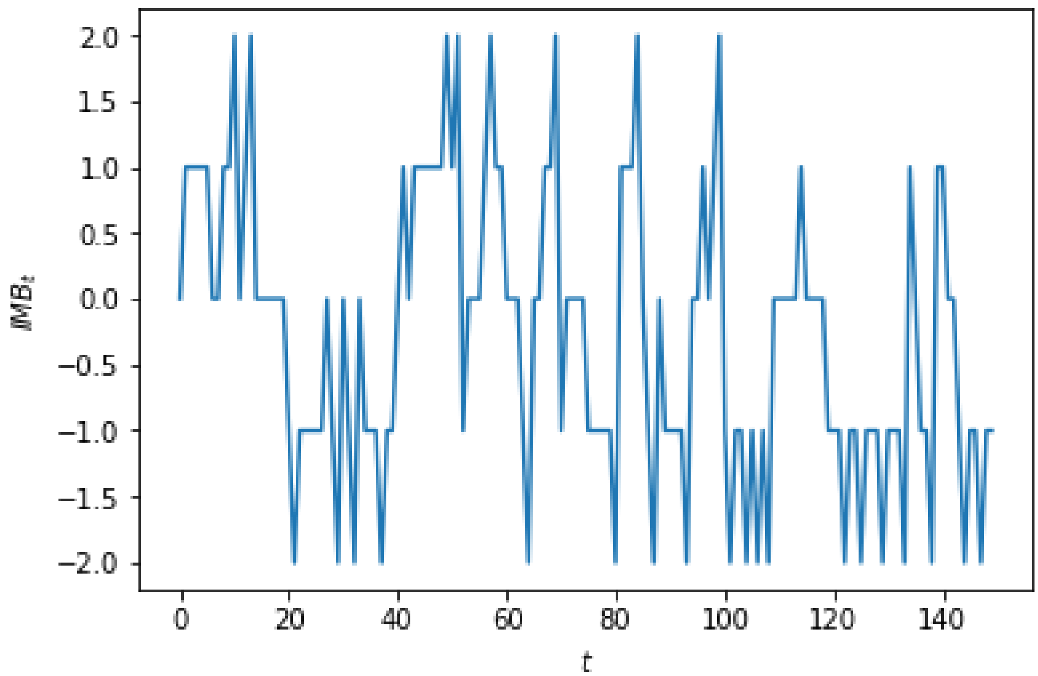
1.3.18 The market simulation

[22]:
```
def simulate_market(events, uninformed_orders, fundamental_price,
                        mm_strategy=mm_threshold_strategy, threshold=1):

    mm_prices = np.zeros(MAX_T); order_imbalances = np.zeros(MAX_T)
    informed_orders = np.zeros(MAX_T); rewards = np.zeros(MAX_T);
    actions = np.zeros(MAX_T)
    t_mm = 0; mm_current_price = INITIAL_PRICE

    for t in range(MAX_T):

        if events[t] == EVENT_INFORMED_ARRIVAL:
            order = informed_strategy(fundamental_price[t], mm_current_price)
            informed_orders[t] = order
        else:
            order = uninformed_orders[t]

        imbalance = np.sum(informed_orders[t_mm:t] + uninformed_orders[t_mm:t])

        mm_price_delta = mm_strategy(imbalance, threshold)
        if mm_price_delta != 0:
            t_mm = t
            mm_current_price += mm_price_delta

        order_imbalances[t] = imbalance; mm_prices[t] = mm_current_price
        actions[t] = mm_price_delta;
        rewards[t] = mm_reward(fundamental_price[t], mm_current_price, order)

    return mm_prices, order_imbalances, rewards, actions
```

1.3.19 Order-imbalance time series

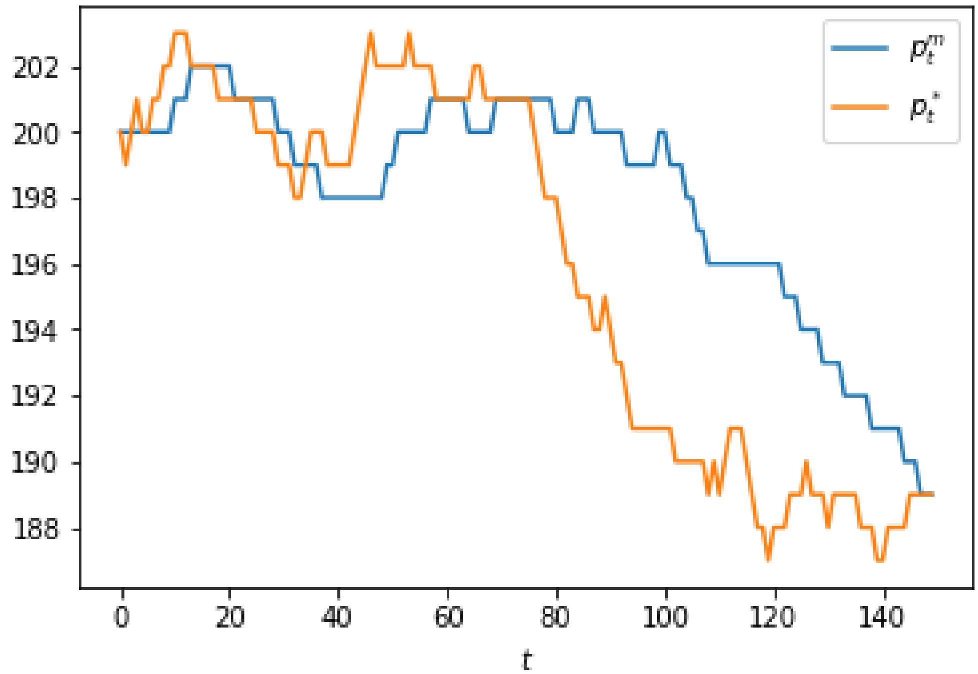
[23]:
```
mm_prices, order_imbalances, rewards, actions = \
    simulate_market(events, uninformed_orders, fundamental_price, threshold=2)
```

[24]:
```
plt.plot(order_imbalances); plt.xlabel('$t$'); plt.ylabel('$IMB_t$')
plt.show()
```



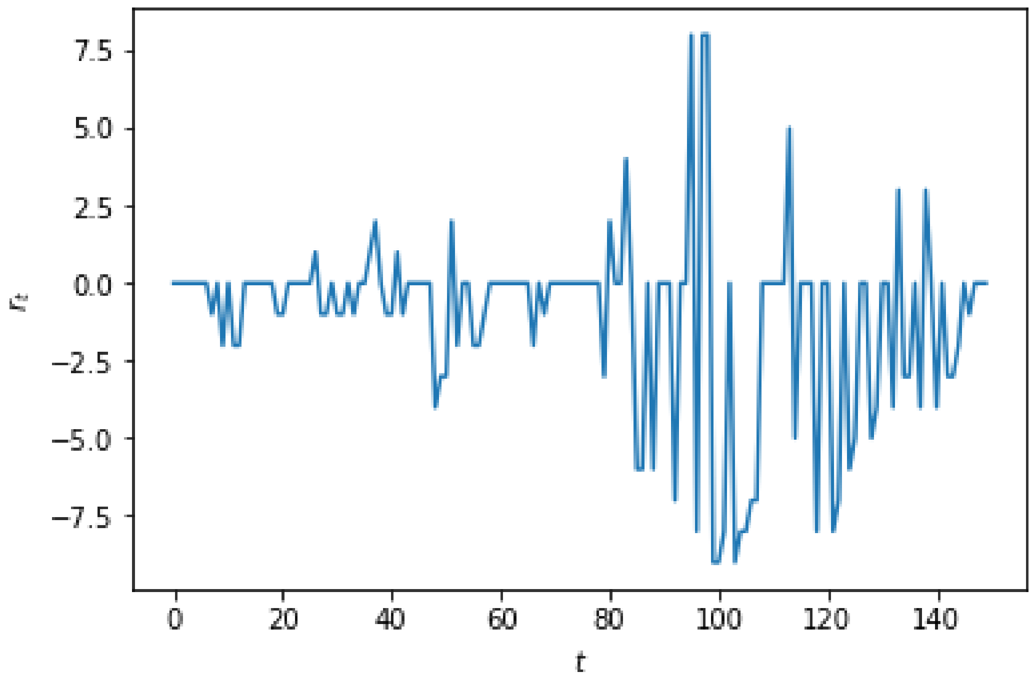
1.3.20 Price time series

[25]:
```
plt.plot(mm_prices); plt.plot(fundamental_price)
plt.xlabel('$t$'); plt.legend(['$p^m_t$','$p^*_t$'])
plt.show()
```



1.3.21 Reward time series r_t

[26]:
```
plt.plot(rewards)
plt.xlabel('$t$'); plt.ylabel('$r_t$')
plt.show()
```



1.4 Policy evaluation

- Each value of the threshold parameter h defines a policy π_h.
- In the experiments above we set $h = 2$.
- We can estimate the expected reward, and hence the expected return, across the episode.
- With no time-discounting, i.e. $\gamma = 1$, the return G is approximately:

[27]:
```
np.mean(rewards)*MAX_T
```

[27]: -165.0

- We could use this as an estimate of the expected return to the policy $E(G)$.
- However, we have only used a single sample.

1.5 Monte-carlo policy evaluation

- We will attempt to obtain more accurate value estimates using Monte-Carlo estimation.
- First we restructure our code so that we can easily rerun an entire simulation with given parameters.

[28]:
```
def simulate_all(mm_strategy=mm_threshold_strategy, threshold=1,
                 probabilities=ALL_PROB):
```

```python
    events = simulate_events(probabilities)
    fundamental_price = simulate_fundamental_price(events)
    uninformed_orders = simulate_uninformed_orders(events)

    return simulate_market(events, uninformed_orders, fundamental_price,␣
↪mm_strategy, threshold)
```

[29]:
```python
mm_prices, order_imbalances, rewards, actions = simulate_all(threshold=1)
np.mean(rewards)
```

[29]: -0.36

1.5.1 Using the sample-mean to estimate expected value

- We realise the model many times.
- Each realisation is a single episode or trajectory.
- We can consider each episode as a sample.
- We use the sample mean as the best estimator for the expectation.

[30]:
```python
def evaluate(policy, probabilities=ALL_PROB, samples=1000):
    return np.mean([np.mean(simulate_all(threshold=policy)[2])
                        for i in range(samples)])
```

1.5.2 Comparing policies

$v(\pi_1) \approx$

[31]:
```python
evaluate(policy=1)
```

[31]: -0.4362533333333333

$v(\pi_2) \approx$

[32]:
```python
evaluate(policy=2)
```

[32]: -0.5420799999999999

$v(\pi_3) \approx$

[33]:
```python
evaluate(policy=3)
```

[33]: -0.6780799999999999

1.5.3 The value of states

- We can also estimate the value of a given state s assumming a fixed policy π
- For a threshold $h = 2$, i.e. $\pi = \pi_2$ the states are $s \in \{-2, -1, 0, +1, +2\}$.
- We first simulate a single episode.

[34]:
```
mm_prices, order_imbalances, rewards, actions = simulate_all(threshold=2)
```

[35]:
```
np.mean(rewards)
```

[35]:
```
-0.54
```

1.5.4 The value of states

- Now we estimate $v_{\pi_2}(s) \approx \bar{r}$ for those rewards obtained in the given state:

[36]:
```
value_fn = {(state, np.mean(rewards[order_imbalances == state])) \
                    for state in [-2, -1, 0, +1, +2]}
value_fn
```

[36]:
```
{(-2, -0.5833333333333334),
 (-1, -0.631578947368421),
 (0, -0.34782608695652173),
 (1, -0.4878048780487805),
 (2, -1.0769230769230769)}
```

- It's more elegant to represent this as a Python dictionary:

[37]:
```
value_dict = dict(value_fn)
```

To compute $v_{\pi_2}(1)$ we can use the following

[38]:
```
value_dict[1]
```

[38]:
```
-0.4878048780487805
```

1.5.5 The value of states

- Note that in the previous slide our estimate was based on a single episode.
- Below we extend the code on the previous slide to average over many episodes (samples).

[39]:
```
def expected_reward_by_state(mm_strategy, threshold,
                                probabilities=ALL_PROB, samples=1000):

    states = range(-threshold, threshold+1)
    result = np.zeros((samples, len(states)))
```

```
    for i in range(samples):
        rewards = simulate_all(mm_strategy, threshold)[2]
        result[i, :] = [np.mean(rewards[order_imbalances == state]) \
                                for state in states]

    return dict(zip(states, np.nanmean(result, axis=0)))
```

1.5.6 Policy comparision

$v_{\pi_1}(s) \approx$

[40]:
```
expected_reward_by_state(mm_threshold_strategy, threshold=1)
```

[40]:
```
{-1: -0.4366842105263162, 0: -0.4183260869565217, 1: -0.4551463414634147}
```

$v_{\pi_2}(s) \approx$

[41]:
```
expected_reward_by_state(mm_threshold_strategy, threshold=2)
```

[41]:
```
{-2: -0.5339999999999994,
 -1: -0.5252894736842114,
 0: -0.5089347826086954,
 1: -0.5504634146341455,
 2: -0.5315384615384616}
```

1.5.7 The value of state action *pairs*

- Ideally we would like to compute $v_\pi(s, a)$.

[42]:
```
def q_table(result, all_actions, all_states):
    return pd.DataFrame(result,
                        columns=["$\Delta p=%s$" % a for a in all_actions],
                        index=all_states)
```

[43]:
```
def expected_reward_by_state_action(mm_strategy, threshold,
                                    probabilities=ALL_PROB, samples=1000):

    all_states = range(-threshold, threshold+1)
    all_actions = [-1, 0, +1]
    result = np.zeros((samples, len(all_states), len(all_actions)))

    for i in range(samples):

        _, states, rewards, actions = simulate_all(mm_strategy, threshold)

        result[i, :, :] = \
            np.reshape(
```

```
                 [np.nanmean(rewards[(states == state) &
                                     (actions == action)]) \
                     for state in all_states for action in all_actions],
                 (len(all_states), len(all_actions)))

    return np.nanmean(result, axis=0)
```

1.5.8 The value function obtained from π_2

```
[44]: Q = expected_reward_by_state_action(mm_threshold_strategy, threshold=2,␣
       ↪samples=10000)
      q_table(Q, [-1, 0, +1], range(-2, 3))
```

```
[44]:     $\Delta p=-1$  $\Delta p=0$  $\Delta p=1$
      -2       -0.38593           NaN           NaN
      -1            NaN     -0.567512           NaN
       0            NaN     -0.564312           NaN
       1            NaN     -0.567192           NaN
       2            NaN           NaN     -0.386951
```

1.5.9 Exploration of the state-space

```
[45]: def mm_exploration_strategy(order_imbalance, threshold=2, epsilon=0.025):
          if np.random.random() <= epsilon:
              return np.random.choice([-1, 0, +1])
          else:
              if order_imbalance == +threshold:
                  return -1
              elif order_imbalance == -threshold:
                  return +1
              else:
                  return 0
```

1.5.10 Results from Monte-Carlo policy evaluation

```
[46]: Q = expected_reward_by_state_action(mm_exploration_strategy, threshold=2,␣
       ↪samples=50000)
```

```
[47]: q_table(Q, [-1, 0, +1], range(-2, 3))
```

```
[47]:     $\Delta p=-1$  $\Delta p=0$  $\Delta p=1$
      -2      -5.393120     -6.398448     -5.971927
      -1      -5.820319     -6.092052     -6.168338
       0      -5.823990     -5.799513     -5.802399
       1      -6.240433     -6.094727     -5.838191
```

```
2    -5.945665    -5.921463    -5.777265
```

The greedy policy:

```
[48]: dict({(s, np.where(Q[s+2, :] == np.max(Q[s+2, :]))[0][0] - 1) for s in range(-2,␣
      ↪3)})
```

```
[48]: {-2: -1, -1: -1, 0: 0, 1: 1, 2: 1}
```

1.6 Policy improvement

- We have estimated $Q_\pi(s,a) = \hat{v}_\pi(s,a)\ \forall s \in \mathbb{S}\ \forall a \in \mathbb{A}$ for a given policy π.
- If our estimates are accurate, we can use the function Q to find a better policy.
- To improve the policy we can simply take the greedy action for a given state:

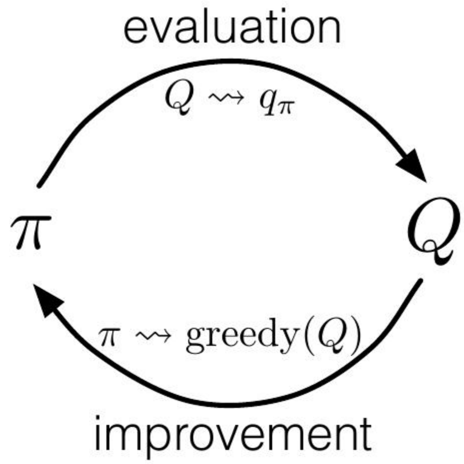
  $\pi'(s) = \operatorname{argmax}_a Q_\pi(s,a)$
- For the corresponding proof, see the *policy improvement theorem*.

1.7 Policy iteration

- Note, however, that if we change the policy from π to π', that our value estimates are outdated.
- Value estimates $Q_\pi(s,a)$ are obtained from following a given policy π.
- Value estimates and policies are not independent of each other.
- Therefore we should re-estimate $Q'_{\pi'}(s,a)$ for the new policy as $\hat{v}_{\pi'}(s,a)$
- We iterate until the policy and value estimates converge:

$$\pi(s,a) \to Q_\pi(s,a) \to \pi'(s,a) \to Q'_{\pi'}(s,a) \to \pi''(s,a) \dots \tag{4}$$

1.7.1 Policy iteration figure



1.8 Generalised Policy Iteration

- Many reinforcement-learning interleaving policy improvement and value estimation;
  - improve policy based on incomplete samples, while
  - improving value estimates by following a (sub-optimal) policy.
- These techniques are called Generalised Policy Iteration (GPI) methods.

1.9 Temporal-difference (TD) learning

Recall that:

$$
\begin{aligned}
v_\pi(s) &= E_\pi[G_t | s_t = s] && (5)\\
&= E_\pi[r_{t+1} + \gamma G_{t+1} | s_t = s] && (6)\\
&= E_\pi[r_{t+1} + \gamma v_\pi(s_{t+1}) | s_t = s] && (7)\\
& && (8)
\end{aligned}
$$

- When we use Monte-Carlo methods, G_t is unknown, so we estimate G_t from sampled returns.
- When we use dynamic-programming $v_\pi(s_{t+1})$ is unknown, so we use our existing estimate $\hat{v}_\pi(s_{t+1}) = V(s_{t+1})$ instead.
- Temporal difference learning combines both estimates:

$$V(s) \leftarrow V(s) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \tag{9}$$

- We can generalise this to Q:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{10}$$

1.10 Initialising the Q table

- We first define our action space $\Delta p_t \in \mathbb{A}$ and state space $\text{IMB}_t \in \mathbb{S}$

[49]:
```
all_actions = [-1, 0, +1]
all_states = range(-2, +3)
```

- We will use a matrix to represent our current estimates Q

[50]:
```
def initialise_learner():
    return np.zeros((len(all_states), len(all_actions)))
```

[51]:
```
Q = initialise_learner()
q_table(Q, all_actions, all_states)
```

[51]:

	$\Delta p=-1$	$\Delta p=0$	$\Delta p=1$
-2	0.0	0.0	0.0
-1	0.0	0.0	0.0
0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0

1.11 Functions to manipulate Q

- We define the following functions to map from the state and action space into indices of the matrix.

[52]:
```
def state(imbalance, all_states=range(-2, +3)):
    s = int(imbalance) - all_states[0]
    ms = len(all_states)-1
    if s > ms:
        return ms
    elif s < 0:
        return 0
    else:
        return s

def action(price_delta):
    return int(price_delta) + 1
```

- We then define functions to obtain Q values from specified actions and states

[53]:
```
def q_values(Q, imbalance):
    return Q[state(imbalance), :]
```

[54]:
```
def q_value(Q, imbalance, price_delta):
    return Q[state(imbalance), action(price_delta)]
```

1.12 Temporal-difference learning in Python

We can translate the following equation

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{11}$$

into Python:

[55]:
```python
def update_learner(s, a, r, s_, a_, Q, alpha=0.01, gamma=0.0):
        Q[state(s), action(a)] += alpha * (r + gamma * q_value(Q, s_, a_) -␣
 ↪q_value(Q, s, a))
```

- Now we must modify our simulation code to provide feedback to the agent.

1.13 The market-maker as a reinforcement-learning agent

[56]:
```python
def simulate_learning_mm(Q, events, uninformed_orders, fundamental_price,
                             mm_policy):
    mm_prices = np.zeros(MAX_T, dtype=int); order_imbalances = np.zeros(MAX_T,␣
 ↪dtype=int)
    informed_orders = np.zeros(MAX_T, dtype=int); rewards = np.zeros(MAX_T);
    actions = np.zeros(MAX_T, dtype=int);  mm_t_last_change = 0
    mm_current_price = INITIAL_PRICE
    for t in range(MAX_T):
        if events[t] == EVENT_INFORMED_ARRIVAL:
            order = informed_strategy(fundamental_price[t], mm_current_price)
            informed_orders[t] = order
        else:
            order = uninformed_orders[t]
        imbalance = np.sum(informed_orders[mm_t_last_change:t] +
                            uninformed_orders[mm_t_last_change:t])
        mm_price_delta = mm_policy(imbalance)
        if mm_price_delta != 0:
            mm_t_last_change = t
            mm_current_price += mm_price_delta
        order_imbalances[t] = imbalance; mm_prices[t] = mm_current_price
        actions[t] = mm_price_delta;
        rewards[t] = mm_reward(fundamental_price[t], mm_current_price, order)
        if t>0:
            update_learner(order_imbalances[t-1], actions[t-1], rewards[t-1],
                             imbalance, mm_price_delta, Q)
    return fundamental_price, mm_prices, order_imbalances, rewards, actions, Q
```

1.14 On-policy control

- Now we can combine policy improvement and policy estimation in a single step.

- This algorithm is called SARSA, which is named after the arguments to the function `update_learner`.
- We use TD learning to bootstrap Q values, and then form an ϵ-greedy policy using our value estimates.

[57]:
```
def mm_learning_strategy(Q, s, epsilon=0.1):
    if np.random.random() <= epsilon:
        action = np.random.choice([-1, 0, +1])
    else:
        values = q_values(Q, s)
        max_value = np.max(values)
        action = np.random.choice(np.where(values == max_value)[0]) - 1
    return action
```

[58]:
```
def simulate_learning(Q, probabilities=ALL_PROB):

    events = simulate_events(probabilities)
    fundamental_price = simulate_fundamental_price(events)
    uninformed_orders = simulate_uninformed_orders(events)

    def sarsa(s):
        return mm_learning_strategy(Q, s)

    return simulate_learning_mm(Q, events, uninformed_orders, fundamental_price,
 ↪mm_policy=sarsa)
```

1.15 Learning over a single episode

[59]:
```
Q = initialise_learner()
```

[60]:
```
fundamental_price, mm_prices, order_imbalances, rewards, actions, Q =
 ↪simulate_learning(Q)
```

[61]:
```
q_table(Q, all_actions, all_states)
```

[61]:

	$\Delta p=-1$	$\Delta p=0$	$\Delta p=1$
-2	0.000000	0.000000	0.000000
-1	-0.401995	-0.266438	-0.260303
0	-0.629396	-0.570168	-0.329424
1	-0.764471	-0.882239	-0.889121
2	-0.377703	-0.503057	-0.215830

1.16 Learning over many episodes

- We simply iterate over many trading days (i.e. episodes) in order to gradually learn the optimal policy.

- Each episode is independent, but notice that we re-use the Q-values from the previous episode.
- This ensures that we learn across episodes.

[62]:
```
EPISODES = 5000

for i in range(EPISODES):
    fundamental_price, mm_prices, order_imbalances, rewards, actions, Q =␣
 ↪simulate_learning(Q)
```

1.17 The results

The learned Q values:

[63]:
```
q_table(Q, all_actions, all_states)
```

[63]:

	$\Delta p=-1$	$\Delta p=0$	$\Delta p=1$
-2	-0.908272	-1.945495	-1.884303
-1	-0.404848	-0.941013	-1.036575
0	-0.945292	-0.496658	-0.874284
1	-1.331283	-1.248187	-0.343388
2	-2.161748	-2.306827	-1.399997

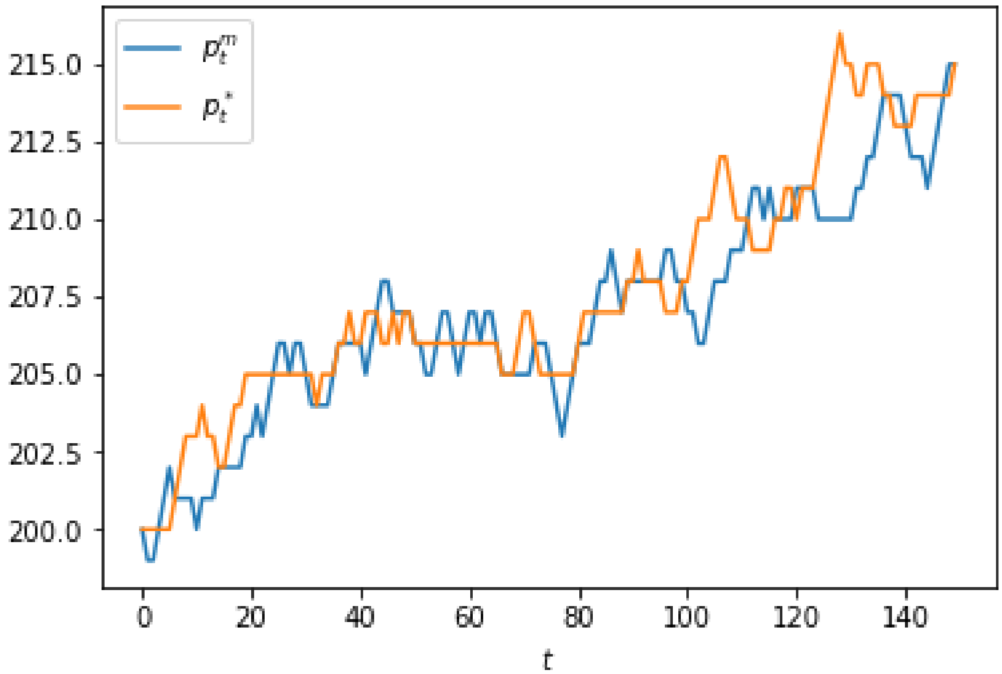
The greedy policy:

[64]:
```
{(s, np.where(Q[state(s), :] == np.max(Q[state(s), :]))[0][0] - 1) for s in␣
 ↪all_states}
```

[64]: {(-2, -1), (-1, -1), (0, 0), (1, 1), (2, 1)}

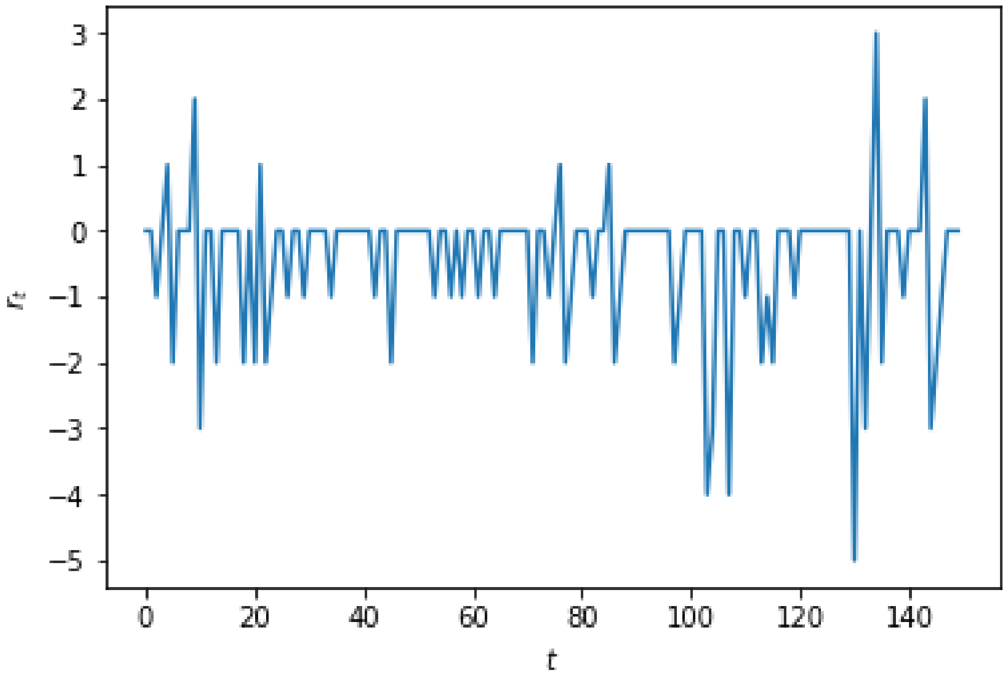
1.18 The prices from the final trading day

[65]:
```
plt.plot(mm_prices); plt.plot(fundamental_price)
plt.xlabel('$t$'); plt.legend(['$p^m_t$','$p^*_t$'])
plt.show()
```



1.19 The rewards in the final day

[66]:
```
plt.plot(rewards)
plt.xlabel('$t$'); plt.ylabel('$r_t$')
plt.show()
```



1.20 Conclusion

- We have implemented a very simple market-making strategy in a simplified model of a financial market.
- The framework is very flexible, and can be extended to more realistic applications, e.g. separate bid and ask qutes.
- We can use a simulation model to initially train our agent.
- However, the agent can learn directly from the environment in the absence of a model.
- Reinforcement-learning can be very useful in reducing model-risk.

2 Bibliography

Chan, N. T., & Shelton, C. (2001). An electronic market-maker.

Ganchev, K., Nevmyvaka, Y., Kearns, M., & Vaughan, J. W. (2010). Censored exploration and the dark pool problem. Communications of the ACM, 53(5), 99-107.

Mani, M., Phelps, S., & Parsons, S. (2019). Applications of Reinforcement Learning in Automated Market-Making.

Sutton, R. S., & Barto, A. G. (2011). Reinforcement learning: An introduction.

2.1 Further reading on agent-based modeling

Farmer, J. D., & Foley, D. (2009). The economy needs agent-based modelling. Nature, 460(7256), 685-686.

Lo, A. W. (2004). The adaptive markets hypothesis. The Journal of Portfolio Management, 30(5), 15-29.

Phelps, S. (2012). Applying dependency injection to agent-based modeling: the JABM toolkit. WP056-12, Centre for Computational Finance and Economic Agents (CCFEA), Tech. Rep.

Tesfatsion, L., & Judd, K. L. (Eds.). (2006). Handbook of computational economics: agent-based computational economics. Elsevier.

2.2 Links to software toolkits

- JABM
- JASA