Price Volatility and Shortages: Modelling Panic Buying

April 23, 2021

Chenyang Li

Eton College Keynes Prize 2021

Word count: 1000 (excluding figures and references)

Price Volatility and Shortages: Modelling Panic Buying

Chenyang Li

April 23, 2021

1 Introduction

For those who have lived through a pandemic, price volatility is no stranger to the ears. In the early days of lockdown restrictions, dramatic images of empty shelves surfaced across media outlets, prompting fears of widespread shortages and significant price increases. Economic theory provides a standard explanation: "a change in consumer preferences results in a rightwards shift in demand. This leads to shortages, which producers respond to by increasing quantity supplied and raising prices." Such conditions commonly occur in *commodity markets*, defined as markets for *fungible* economic goods (whose individual units are interchangeable and indistinguishable).

So far, so simple.

The size of price increase depends on the price elasticity of supply $\epsilon_{\langle s \rangle}$ and demand $\epsilon_{\langle d \rangle}$ ($\epsilon := \frac{p}{q}\frac{dq}{dp}$)[1], respectively a proxy measure of the ability of producers to respond to changes in demand by adjusting production, as well as how necessary consumers view that product. For price inelastic goods with $0 \leq \epsilon_{\langle s \rangle} < 1$ and $0 \leq |\epsilon_{\langle d \rangle}| < 1$, any change in the market will result in substantial price fluctuations, harmful to both producers and consumers alike.[2] This type of market failure is special as it preserves *Pareto efficiency*[3], while still leading to undesirable market outcomes.

Uncertainty over the market price leads to risk, the mitigation of which acts as an additional cost to participants; systemic risk makes planning investment in new productive capacity or future patterns of consumption almost impossible. Historical evidence suggests this uncertainty also facilitates speculation, amplifying price volatility through the accumulation of speculative bubbles (Dash, 2011; Vogel, 2009).

Any initial price instability may lead to long-term price volatility, as there is a tendency for participants in the market to overreact to the initial price shock. For instance, take a staple such as pasta. Pasta may be treated as a quasi-commodity since it is associated with inelastic price elasticities, estimated by Tiffin to be $\epsilon_{\langle s \rangle} = 0.375$ and $\epsilon_{\langle d \rangle} = -0.789$ in the short-term. If the theory is correct, one would expect price increases from March to April 2021, followed by overcorrection in the markets and a subsequent price decrease.

[1] by definition.

[2] In times of shortages and high prices, poorer consumers spend much of what resources they have on essentials, reducing consumption at great personal cost. Wealthier consumers reduce consumption very little even when prices soar (Bulíř, 2001). Conversely, in times of plenty, producer incomes will shrink to unprofitable margins, potentially leading to a missing market or oligopolistic market conditions as the majority of producers divest into substitute industries.

[3] Pareto efficient: an allocation of resources such that no changes to its distribution can be made without hurting someone else.

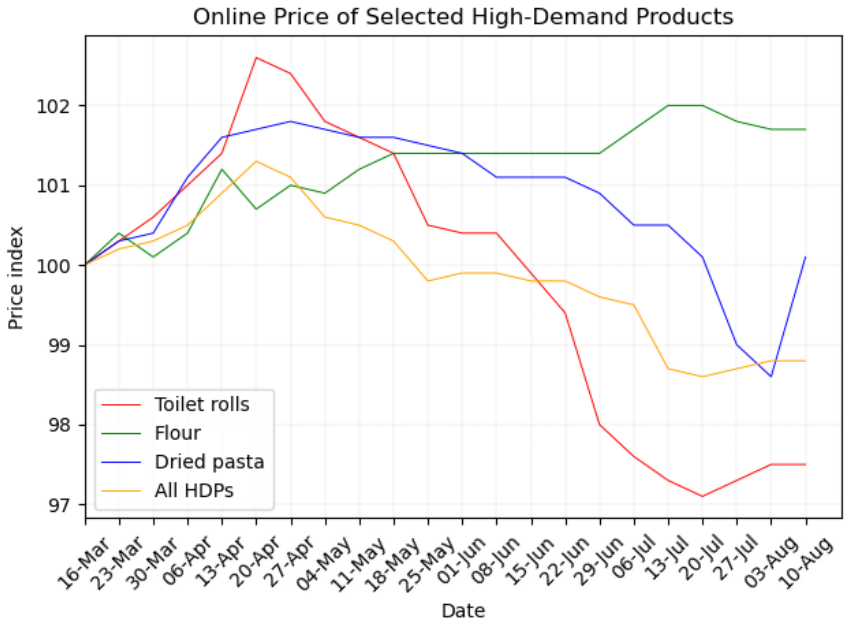

Figure 1: A selection of three goods from an experimental index of online prices for products in a high-demand basket. The introduction of lockdown across the UK coincides with a general price increase for *high-demand products* (HDPs). *Source*: Office for National Statistics, 2021. *Graphics*: Chenyang Li, via `matplotlib`.

This is indeed what we see. Time series *All HDPs*, *Toilet rolls*, and *Dried pasta* follow the trend as described, with *All HDPs* spiking on April 20$^{\text{th}}$ by over 2%, before slumping to almost 3% below the base value by July 20$^{\text{th}}$. We observe, however, that *Dried pasta* has re-corrected to pre-pandemic levels; interestingly, *Flour* remains an anomaly in the time series, showing a sustained increase of 0.08% every week, with respect to the base date.

---

2 The Model

Pivoting away from textbook economics, we now reformulate such modelling towards a more fundamental, participant-based model of the market.

Much of economics centres on two simple axioms (Varian, 2014):

- *The optimisation principle*: people try to choose the best patterns of consumption that they can afford.
- *The equilibrium principle*: prices tend to adjust until the quantity demanded equals the quantity supplied.

Precisely how equilibrium arises receives less attention, and any explanatory adjustment process often assumes perfect information. "Alternative assumptions will necessitate explicit modelling of individual agents and their behaviour, which greatly increases the level of complication of the model" (Coad and Van De Panne, 1996).

Our model is a numerical simulation of a closed market for an arbitrary commodity. In it, we instantiate producers and consumers, each buying/selling one unit of the good for an arbitrary timeframe. Consumers will seek to buy at as low a price as possible, whereas producers will sell at as high a price as possible; both adjust prices each time they succeed or consecutively fail, and have a maximum and minimum price limit, respectively. [4]

During a transaction, consumers will:

1. not buy if the offered price is too high.
2. lower prices if they succeed.
3. raise prices if they fail consecutively for `consumer_tolerance` days, until they are at their maximum price.
4. leave the market if they fail to purchase for `threshold` days.

Producers will:

1. sell to the best price, given a pool of consumers.
2. not sell if the best price is below their current acceptance price.
3. raise prices if a transaction occurred.
4. lower prices if they fail consecutively for `producer_tolerance` days, until they are at their minimum price.
5. leave the market if they fail to sell for `threshold` days.

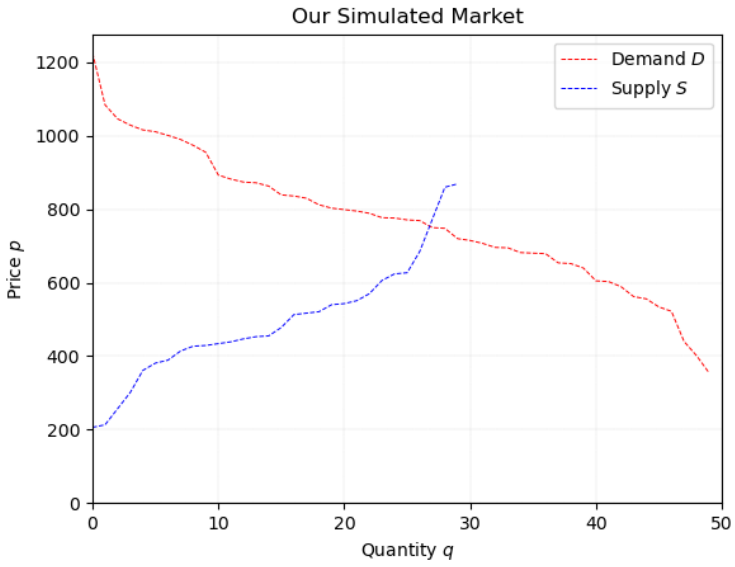

Figure 2: Initial supply and demand diagram for the simulated market. *Graphics*: Chenyang Li, via `matplotlib`.

[4] For computational simplicity, we designed the transaction mechanism in a similar way to a *first-price sealed-bid* (or *blind*) auction, "auctions in which the highest bid wins and the highest bidder pays a price equal to his bid" (Milgrom, 2004). Producers are assigned a pool of consumers and will make a transaction if the price is suitable to both parties. The order of search is random.

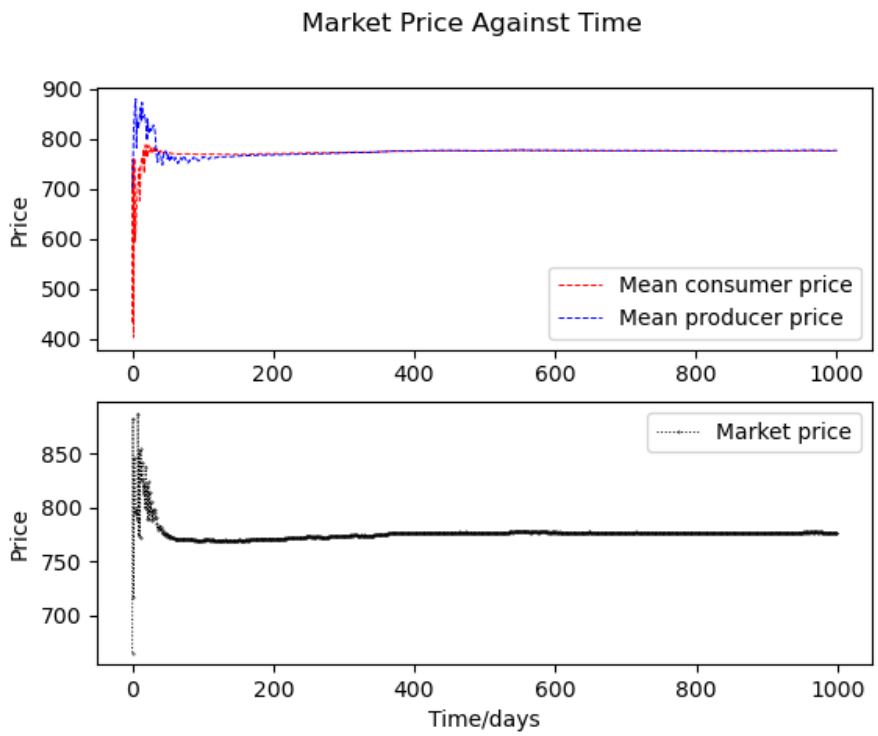

Figure 3: Variation of the market price over time with no changes to the system. Both graphs clearly show average prices settling to an equilibrium price after approximately 50 days. *Graphics*: Chenyang Li, via `matplotlib`.

We see that the model obeys both axioms without assuming perfect information. It also matches the predicted equilibrium price of the supply-demand graph.[5]

---

3 Simulation

We now run the model under the following conditions, for 1000 days (note that price and quantity are unitless): [6]

- 50 consumers with $p \sim \mathcal{N}(750, 200^2)$.
- 30 producers with $p \sim \mathcal{N}(500, 150^2)$.

Then:

- after day 250, a panic is simulated by raising the maximum price of all consumers by 500.

[5] Since quantity is discrete in our simulation, the supply and demand graph predicts an equilibrium quantity of either 26 or 27, and an equilibrium price between 685-770, depending on the aggressiveness of participants (controlled in our simulation by `aggression_factor`, `producer_tolerance`, and `consumer_tolerance`). The equilibrium point of the model is (27, 764), within the predicted boundary.

[6] $p \sim \mathcal{N}(\mu, \sigma^2)$ is read as "the price is distributed normally such that the mean is μ and the standard deviation is σ." For simplicity, prices are discretised to the nearest integer.

- after day 500, additional producers, where $p \sim \mathcal{N}(400, 100^2)$, flood into the market because of high prices.
- after day 750, additional consumers, where $p \sim \mathcal{N}(1000, 300^2)$, flood into the market because of low prices.

Simulations ran under two assumptions:

1. each transaction made per day remained *closed* for the remainder of that iteration (successful participants retire until the next day).
2. there is no *incumbency* (the possibility of extending transactions to future periods).

---

4 Results

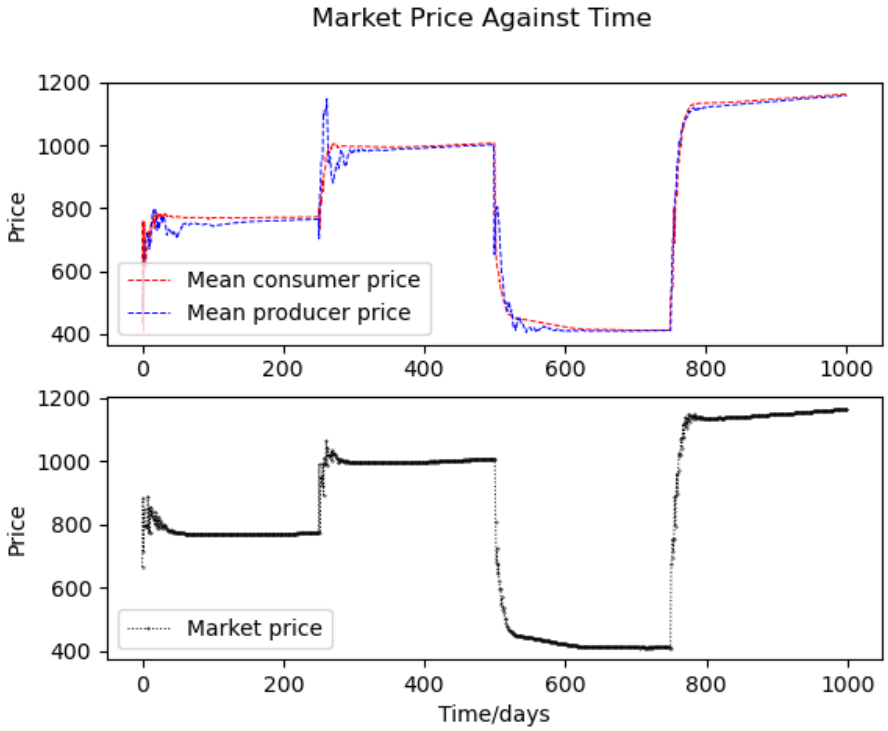

Figure 4: Our model of price volatility. After each disturbance, the market mechanism will cause prices to adjust to a new equilibrium, causing significant price fluctuations. *Graphics*: Chenyang Li, via `matplotlib`.

From Figure 4, we see our results are exactly what the theory and empirical data predict: sudden price changes may lead to large overcorrections by transacting participants, turning short-term panics into long-term price volatility. With a range of 800, prices in our simulated market are truly wild.

However, there are caveats in our simulation, as we have made simplifying modelling assumptions that may prove to be unrealistic. Most notably, the transaction mechanism is based on a blind auction for a good sold in indivisible units, of which participants can only ever sell or buy one.

This simplification means that our model will not capture the "hoarding" behaviour witnessed in demand-side panics, although it remains unclear what implications this omission may have for the accuracy of our model.

---

5 Further Development

A natural extension of the project would be to allow transactions of multiple quantities, examining what changes this would have on the simulated outcome. Furthermore, we would conduct a rigorous evaluation of our results against empirical data, refining the model towards accurately predicting real-world situations. Due to the chaotic nature of human interactions, however, we would expect significant deviation between our results and reality no matter how accurate the model is, especially given that the initial conditions for real consumers and producers can only be roughly measured, and constantly change over time.

References

Bulíř, Aleš (2001). "Income Inequality: Does Inflation Matter?" In: *IMF Staff Papers* 48.1, pp. 139–159. ISSN: 10207635, 15645150. URL: http://www.jstor.org/stable/4621662.

Coad, L.A. and C. Van De Panne (1996). "Computer Simulation for Supply-Demand Interaction". In: *The Canadian Journal of Economics* 29.1 (Special Issue), S308–S312. URL: https://www.jstor.org/stable/136006. Retrieved 11 April, 2021.

Dash, Mike (2011). *Tulipomania: The story of the world's most coveted flower and the extraordinary passions it aroused*. Hachette UK.

Milgrom, P. R. (2004). *Putting auction theory to work*. Cambridge University Press, p. 112.

Office for National Statistics (2021). "Dataset: Online price changes for high-demand products (March to August 2021)". In: URL: https://www.ons.gov.uk/economy/inflationandpriceindices/datasets/onlinepricechangesforhighdemandproducts. Retrieved 10 April, 2021.

Tiffin, R. et al. (2011). "Estimating Food and Drink Elasticities". In: *Working Paper for the Department for Environment, Food and Rural Affairs*, p. 115. URL: https://www.gov.uk/government/publications/food-and-drink-elasticities. Retrieved 20 April, 2021.

Varian, H. R. (2014). *Intermediate Microeconomics*. W. W. Norton & Company, p. 3. ISBN: 9780393919677.

Vogel, H. L. (2009). *Financial Market Bubbles and Crashes*. Cambridge University Press. DOI: 10.1017/CBO9780511806650.

Appendix: Full Code

```
# 12/04/2021
# Author: Chenyang Li

# This is a simple numerical model of a market for an arbitrary economic good.
# A Consumer and Producer class are instantiated and put through a market␣
 ↪simulation.
# Producers are randomly assigned a pool of consumers and will make a transaction
# if the price is suitable to both parties.
# Order of search is random.
# Results are shown in a matplotlib plot.

import matplotlib.pyplot as plt
import numpy as np

# hyperparameters
days = 1000
aggression_factor = 0.8  # ratio of rate of change of new price, 0<x<1
threshold = 100  # number of days after which producers/consumers withdraw after␣
 ↪successive failure

panic_day = 250  # day after which consumers panic and want more
overcompensate_day = 500  # day after which producers overcompensate
more_consumers_day = 750  # day after which more consumers flood in to the market

producer_tolerance = 5  # days after which producers will decrease prices
consumer_tolerance = 2  # days after which consumers will increase prices

np.random.seed(10)  # fixed random seed for reproducibility


class Consumer:
    """
    Consumer object that models a simplistic rational consumer.

    Consumers will:
    1. not buy if the price offered is too high
    2. lower prices if a transaction occurred
    3. raise prices if no transaction in two previous days, until price ==␣
 ↪max_price
    """

    def __init__(self, max_price, price, success=False, attempts=0):
```

```
        """
        :param max_price: maximum price a consumer will tolerate
        :param price: current offered price
        :param success: whether the previous purchase was a success
        :param attempts: # of attempts without success. Resets to 0 after␣
↪transaction completed
        """
        if max_price < price:
            raise ValueError("max_price must be greater than price")

        self.max_price = max_price
        self.price = price
        self.success = success
        self.attempts = attempts
        self.price_inc = max(max_price - price, 100)  # price_inc: increment of␣
↪price change.
        self.price_inc_list = [self.price_inc]
        self.price_list = [price]
        self.attempts_list = [attempts]

    def __str__(self):
        return f"Consumer. Current offered price: {self.price}. Maximum price:␣
↪{self.max_price}."

    def set_price(self):
        """
        Method that determines how producers change their prices
        :global aggression_factor: hyperparameter determining how price_change␣
↪changes (second derivative)
        """
        self.attempts += 1  # increase attempts by 1
        if self.success:
            new_price = self.price - self.price_inc
            if new_price < 0:
                self.price = 0
            else:
                self.price = new_price
            # increment of price change is reduced every time a successful␣
↪transaction is made
            self.price_inc = int(self.price_inc * aggression_factor)  # by a␣
↪ratio of aggression_factor
            if self.price_inc < 1:
                # |price_change| >= 1
                self.price_inc = 1
            self.attempts = 0  # reset attempts
        else:
```

```
            if self.attempts > consumer_tolerance:
                new_price = self.price + self.price_inc
                if new_price <= self.max_price:
                    # prices must not exceed the maximum price limit
                    self.price = new_price
                else:
                    self.price = self.max_price
            else:
                # price increases only after x consecutive days of failure
                pass

        self.price_list.append(self.price)
        self.price_inc_list.append(self.price_inc)
        self.attempts_list.append(self.attempts)


class Producer:
    """
    Producer object that models a simplistic profit-maximising producer.

    Producers will:
    1. sell to the best price, given a pool of Consumers.
    2. not sell if the best price is below the current acceptance price
    3. raise prices if a transaction occurred
    4. lower prices if no transaction
    """

    def __init__(self, min_price, price, success=False, attempts=0):
        """
        :param min_price: minimum price a consumer will tolerate
        :param price: current acceptance price
        :param success: whether the previous purchase was a success
        :param attempts: # of attempts without success. Resets to 0 after␣
↪transaction completed
        """
        if min_price > price:
            raise ValueError("min_price must be less than price")

        self.min_price = min_price
        self.price = price
        self.success = success
        self.attempts = attempts
        self.price_inc = max(price - min_price, 100)  # price_inc: increment of␣
↪price change.
        self.price_inc_list = [self.price_inc]
        self.price_list = [price]
```

```
    def __str__(self):
        return f"Producer. Current acceptance price: {self.price}. Minimum price:
↪ {self.min_price}."

    def set_price(self):
        """
        Method that determines how producers change their prices
        """
        self.attempts += 1  # increase attempts by 1
        if self.success:
            self.price += self.price_inc
            # increment of price change is reduced every time a successful␣
↪transaction is made
            self.price_inc = int(self.price_inc * aggression_factor)  # by a␣
↪ratio of aggression_factor
            if self.price_inc < 1:
                # |price_change| >= 1
                self.price_inc = 1
            self.attempts = 0  # reset attempts
        else:
            if self.attempts > producer_tolerance:
                # producers reduce prices if they cannot sell
                new_price = self.price - self.price_inc
                if new_price >= self.min_price:
                    # prices must not exceed the minimum price limit
                    self.price = new_price
                else:
                    self.price = self.min_price
            else:
                # price increases only after x consecutive days of failure
                pass

        self.price_list.append(self.price)
        self.price_inc_list.append(self.price_inc)


def populate_consumers(mu, sigma, n, time=0):
    """
    Populate a group of consumers with normally distributed max_prices and␣
↪random prices.
    The normal distribution is chosen as it is a good approximation for unknown
    distributions of continuous variables in real life.
    See: Central Limit Theorem

    :param mu: population mean
    :param sigma: standard deviation
    :param n: number of consumers
```

```
    :param time: if additional consumers join, their average prices before must␣
 ↪be filled in
    :return consumers: list of Consumer objects
    """
    consumers = []
    max_prices = list(np.random.normal(mu, sigma, n))
    max_prices = [int(price) for price in max_prices]  # round to int
    initial_prices_seed = list(np.random.random(n))
    for index, max_price in enumerate(max_prices):
        price = int(max_price * initial_prices_seed[index])
        instance = Consumer(max_price, price)
        consumers.append(instance)

    if time != 0:
        filler = np.empty(time)
        filler.fill(np.nan)
        filler = list(filler)
        for consumer in consumers:
            consumer.price_list = filler + consumer.price_list

    return consumers


def populate_producers(mu, sigma, n, time=0):
    """
    Populate a group of producers  with normally distributed min_prices and␣
 ↪random prices.
    The normal distribution is chosen as it is a good approximation for unknown
    distributions of continuous variables in real life.
    See: Central Limit Theorem

    :param mu: population mean
    :param sigma: standard deviation
    :param n: number of producers
    :param time: if additional producers join, their average prices before must␣
 ↪be filled in
    :return producers : list of Producer objects
    """
    producers = []
    min_prices = list(np.random.normal(mu, sigma, n))
    min_prices = [int(price) for price in min_prices]  # round to int
    initial_prices_seed = list(np.random.random(n))
    for index, min_price in enumerate(min_prices):
        # prices will not be greater 2 times min_price
        price = int(min_price * (1 + initial_prices_seed[index]))
        instance = Producer(min_price, price)
        producers.append(instance)
```

```
    if time != 0:
        filler = np.empty(time)
        filler.fill(np.nan)
        filler = list(filler)
        for producer in producers:
            producer.price_list = filler + producer.price_list

    return producers


def plot_supply_demand(producers, consumers):
    """
    :param producers: list of Producer objects
    :param consumers: list of Consumer objects
    """
    consumers.sort(key=lambda consumer: consumer.max_price, reverse=True)  #␣
 ↪sort consumers by price descending
    producers.sort(key=lambda producer: producer.min_price)  # sort producers by␣
 ↪price ascending

    c_prices, c_quantity = [], []
    for num, consumer in enumerate(consumers):
        if consumer.max_price not in c_prices:
            c_prices.append(consumer.max_price)
            c_quantity.append(num)

    p_prices, p_quantity = [], []
    for num, producer in enumerate(producers):
        if producer.min_price not in c_prices:
            p_prices.append(producer.min_price)
            p_quantity.append(num)

    plt.plot(c_quantity, c_prices, color="r", linestyle="--", linewidth=0.7,␣
 ↪label="Demand $D$")
    plt.plot(p_quantity, p_prices, color="b", linestyle="--", linewidth=0.7,␣
 ↪label="Supply $S$")
    plt.title("Our Simulated Market")
    plt.xlabel("Quantity $Q$")
    plt.ylabel("Price $P$")
    plt.legend()
    plt.grid(color='grey', linestyle='--', linewidth=0.1)

    plt.xlim(0, 50)
    plt.ylim(0)

    plt.show()
```

```
def simulate_market(producers, consumers, transaction_prices):
    """
    IMPORTANT FUNCTION
    Simulates the market
    :param consumers: list of Consumer objects
    :param producers: list of Producer objects
    :param transaction_prices: history of successful transaction prices
    :return: producers, consumers, transaction_prices (all updated)
    """
    consumers.sort(key=lambda consumer: consumer.price, reverse=True)  # sort␣
↪consumers by price descending
    np.random.shuffle(producers)  # shuffle producers
    prices_today = []  # initialise

    index = 0  # search index for consumers (consumers is sorted, so index␣
↪denotes best price)
    for num, producer in enumerate(producers):
        if index == len(consumers):
            # all consumers have purchased
            break

        if consumers[index].price >= producer.price:
            producer.success = True
            consumers[index].success = True
            prices_today.append(consumers[index].price)
            index += 1
        else:
            producer.success = False
            consumers[index].success = False

    new_consumers = []
    # once transactions are all made consumers set prices
    for num, consumer in enumerate(consumers):
        if num >= index:
            consumer.success = False
        consumer.set_price()

        if consumer.attempts < threshold:
            new_consumers.append(consumer)
        else:
            if (consumer.max_price - consumer.price) <= consumer.price_inc:
                # consumer leaves market if they fail consistently
                # however, they must be close to the bottom of their price range
                continue
            else:
```

```
                new_consumers.append(consumer)

    new_producers = []
    # once transactions are all made producer set prices
    for num, producer in enumerate(producers):
        if num >= index:
            producer.success = False
        producer.set_price()

        if producer.attempts < threshold:
            new_producers.append(producer)
        else:
            if (producer.price - producer.min_price) <= producer.price_inc:
                # producer leaves market if they fail consistently
                # however, they must be close to the bottom of their price range
                continue
            else:
                new_producers.append(producer)

    if not prices_today:
        # no successful transactions means the list is empty
        transaction_prices.append(np.nan)
    else:
        prices_today = np.array(prices_today)
        transaction_prices.append(prices_today.mean())

    return new_producers, new_consumers, transaction_prices


def reset_price_inc(producers, consumers, price):
    """
    Reset price increment because of anticipated shock.
    :param producers: list of Producer objects
    :param consumers: list of Consumer objects
    :param price: anticipated price change
    :return: producers, consumers
    """
    for consumer in consumers:
        consumer.price_inc = max(price // 2, consumer.price_inc)
    for producer in producers:
        producer.price_inc = max(price // 2, producer.price_inc)
    return producers, consumers


def increase_price(producers, consumers, which="consumer", price=500):
    """
    Increase max_price for Consumers, min_price for Producers
```

```
    :param producers: list of Producer objects
    :param consumers: list of Consumer objects
    :param which: whether consumers or producers increase prices
    :param price: price change upwards
    :return: producer, consumers
    """
    if which == "consumer":
        for consumer in consumers:
            consumer.max_price += price
    elif which == "producer":
        for producer in producers:
            producer.min_price += price
    else:
        raise ValueError("Variable 'which' must be either 'consumer' or␣
↪'producer'")

    producers, consumers = reset_price_inc(producers, consumers, price)
    return producers, consumers


def decrease_price(producers, consumers, which="consumer", price=500):
    """
    Decrease max_price for Consumers, min_price for Producers
    :param producers: list of Producer objects
    :param consumers: list of Consumer objects
    :param which: whether consumers or producers increase prices
    :param price: price change downwards
    :return: producer, consumers
    """
    if which == "consumers":
        for consumer in consumers:
            consumer.max_price -= price
    elif which == "producers":
        for producer in producers:
            producer.min_price -= price
    else:
        raise ValueError("Variable 'which' must be either 'consumer' or␣
↪'producer'")

    producers, consumers = reset_price_inc(producers, consumers, price)
    return producers, consumers


def main():
    """
    Main function
    """
```

```
    consumers = populate_consumers(750, 200, 50)
    producers = populate_producers(500, 150, 30)

    plot_supply_demand(producers, consumers)

    # the simulation
    price_history = []  # initialise
    for day in range(0, days):
        if day == panic_day:
            producers, consumers = increase_price(producers, consumers,␣
↪which="consumer", price=500)
        if day == overcompensate_day:
            extra_producers = populate_producers(400, 100, 30, time=day)
            producers += extra_producers
            reset_price_inc(producers, consumers, price=400)
        if day == more_consumers_day:
            extra_consumers = populate_consumers(1000, 300, 20, time=day)
            consumers += extra_consumers
            reset_price_inc(producers, consumers, price=400)

        producers, consumers, price_history = simulate_market(producers,␣
↪consumers, price_history)

    # get mean producer price
    producer_price = []  # initialise
    for producer in producers:
        print(producer)
        producer_price.append(producer.price_list)
    producer_price = np.array(producer_price)
    print(producer_price)
    mean_producer_price = np.nanmean(producer_price, axis=0)

    # get mean consumer price
    consumer_price = []  # initialise
    for consumer in consumers:
        print(consumer)
        consumer_price.append(consumer.price_list)
    consumer_price = np.array(consumer_price)
    mean_consumer_price = np.nanmean(consumer_price, axis=0)

    plt.subplot(2, 1, 1)
    plt.plot(mean_consumer_price, color="r", linestyle="--", linewidth=0.7,␣
↪label="Mean consumer price")
    plt.plot(mean_producer_price, color="b", linestyle="--", linewidth=0.7,␣
↪label="Mean producer price")
    plt.ylabel('Price')
    plt.legend()
```

```
    plt.subplot(2, 1, 2)
    plt.plot(price_history, color="black", linestyle=":", linewidth=0.7,␣
 ↪label="Market price", marker=".", ms=0.7)

    plt.xlabel("Time/days")
    plt.ylabel("Price")
    plt.legend()

    plt.suptitle("Market Price Against Time")
    plt.show()


main()
```