Data Collection, Cleaning & Storage

Goals:

- Review some basics of the Python language.
- Discuss the process of collecting and cleaning data.
- Review the basic features of SQL programming.
- Discuss best practices for designing and working with databases.
- Learn how to interact with databases and files in Python.

Review: Multi-programmer environments

- When we are working as part of a team on large-scale applications we should always use git/github or a similar program.
- This gives us the ability to:
  - Keep track of all versions of the code base
  - Revert specific changes if needed
  - Merge one user's code changes with another user's.
- In my experience, the following are the most useful git keywords:
  - git status
  - git add
  - git commit
  - git push
  - git pull
  - git merge

Homework Review: Writing a simulation algorithm generically

- In your assignment you were asked to write a piece of code that simulated a stochastic process and priced an exotic option. This requires:
  - Generation of Gaussian Random Variables
  - Simulation of increments (& paths) of the Black-Scholes SDE
  - Generation of paths for the underlying asset
  - Calculation of the Payoff for a lookback
- How can we make sure that we code this generically?
  - Most importantly, we would want to separate the creation of the random variables
  - We should also separate the calculation of the payoff of the exotic option

Homework Review: Writing a simulation algorithm generically

- What objects might we use to do this efficiently?
  - Random Number Generation Class
  - Class for the Stochastic Process
  - Payoff Class
- If we write this well, we are able to substitute a different payoff class, or a different stochastic process, without re-writing code.
- One way to do this is to use **inheritance**.
- We would start by creating StochasticProcess and Payoff **base classes**
- We would then implement **child classes** for Black-Scholes Stochastic Process and Lookback Option Payoff, which would inherit from StochasticProcess and Payoff respectively.

Homework Review: Coding Guidelines

- Wherever possible, I highly recommend using pre-written (& pre-tested) modules. It is good to know how these modules work whenever possible, but it is almost never necessary to reinvent the wheel.
- When doing this, it is best practice to write a **wrapper** function or class that controls your interaction with these modules. (Why?)
  - Example: downloading data via yahoo finance, generating random numbers, etc.
- It is also best practice to make sure that your functions have checks on the data that is being passed to them.
  - Example: In your implementation of the Black-Scholes formula, you should ensure that negative vols are handled. (How?)

Review: Lifecycle of a quant software project

- Stages of a quant software project:
  - Data Parsing / Collection
  - Data Cleaning
  - Database Design & Data Storage
  - Model Implementation
  - Model Validation
  - Presentation of model results via an interface
- Today we will go through the first three phases in this process.

Review: Lifecycle of a quant software project

- The most common programming language of choice for Data Collection / Parsing scripts is Python, although there are several suitable alternatives.
- To perform these tasks in Python we need to be able to do the following:
  - Read & write text/CSV files
  - Connect to a local or remote database server
  - Insert new data into a database, or update existing data
- Luckily, Python has modules that allow us to handle these tasks with relative ease, as we will discuss later in the lecture.

What kinds of datasets do we encounter in quant finance?

- Stock/Currency Price Data
- Fixed Income Price/Yield Data
- Fundamental Corporate Data (i.e. Balance Sheet Info)
- Borrower Data (for Mortgage Backed Securities)
- Futures Data
- Options Data

Depending on the type of data that we are working with, we can expect a fairly different **frequency**, **size** and overall **data structure**.

Some types of data are inherently messier (e.g., options data).

Question: what types of data-cleaning algorithms make sense for each of these datasets?

What are the most common data related challenges that we should expect to face?

- Cleaning data in a robust, consistent way
- Handling gaps in different parts of our data
- Handling extremely large data sets
- Structuring the data in an optimal way
- Building robust data integrity checks to ensure high quality data
- What else?

Python: Basics

- Python will be a critical tool for downloading, parsing, cleaning and writing your data to a database.
- We are not going to go into much detail on how to code in Python as my assumption is you are all already capable.
- Please see the following Python documentation if you need to review any details of the Python language: Python

Python: Basic Concepts

- Conditional Statements

```
if (condition1):
    function1()
elif (condition2):
    function2()
else:
    function3()
```

Logical Operators for Conditional Statements: and / or / not

- Definite Loops

```
nums = list(range(0,100))
for x in nums:
    print(x)
```

Python: Basic Concepts

- Indefinite Loops

```
x = 0
while x < 100:
    print(x)
    x += 1
```

Python: Working with Strings

- Strings are compound data types that consists of smaller strings each containing a single character.
- Python has some handy built-in functions for working with strings

  [] operator : enables us to select part of a string

  == operator : enables us to compare strings

  + operator : concatenate two strings

  in & not in : check if another string is in a string

  find : find the start index of a string within another string

  split : split a string by a specified character

  len : determine length of a string
- Strings are **immutable**. This means that they cannot be modified and we cannot use the [] operator to change a part of a string.

Programming in Python: Writing user-defined functions

- In Python we can write our own user-defined standalone functions.
- We can also write functions within user-defined classes, as we will see later.
- The following is an example of a Black-Scholes function written as a user-defined function:

```
def callPx(s_0, k, r, sigma, tau):
    sigmaRtT = (sigma * math.sqrt(tau))
    rSigTerm = (r + sigma * sigma/2.0) * tau
    d1 = (math.log(s_0/k) + rSigTerm) / sigmaRtT
    d2 = d1 - sigmaRtT
    term1 = s_0 * cdf(d1)
    term2 = k * math.exp(-r * tau) * cdf(d2)
    return  term1 - term2
```

Programming in Python: Working with Tuples

- Tuples provide us with a convenient data structure to store data of different types.

```
tuple1 = ('SPY', 290.31)
```

- There can be more than two items in a tuple:

```
tuple1 = ('SPY', '2018-09-06', 290.31)
```

- And we can also create nested or multi-dimensional tuples:

```
tuple1 = ('SPY', 290.31)
tuple2 = ('QQQ', 181.24)
tuples = (tuple1, tuple2)
```

Programming in Python: Working with Tuples

- We can access a particular element in a tuple using the [] operator.
- However, like strings, tuples are immutable, meaning we cannot use the [] operator to modify a specific value in a tuple.
- Tuples are particularly useful as a return value from a function as they provide a means for returning somewhat unstructured data that the caller can then unpack.

Programming in Python: Working with Data Frames

- DataFrames are one important component of the pandas module.
- They are tabular data structures where both the rows and columns can be named.
- DataFrames are also the return value for many pandas file reading functions.
- If you have worked with DataFrames in R then much of their functionality will be familiar to you.
- Some of the most useful features of DataFrames are:
  - The ability to join to another DataFrame
  - The ability to group and aggregate data in the DataFrame

Data Collection: Overview

- The first step in the data collection process is to identify a proper source for the data that you need. In some cases this process will be trivial, and in other cases it will be quite onerous.
- We then need to build a script in Python or a similar language that will parse this data source and format the results in an appropriate manner (and then eventually write them to a database).
- In some cases this might just require reading a text file or CSV file, or connecting to another database where the data is alread stored.
- In other cases, we may need to parse a more complicated file, such as a website.
- We will also need to know how to work with different files, such as connections to local and remote files, FTP servers and websites.
- In this course we only discuss the basics of this process...

Data Collection: Data Sources

- Yahoo Finance is probably the most popular source of data for students.
- Ken French's website also has some useful data on equity factor returns.
- FRED also has a lot of good free data, including some economic data on economic indicators.
- Treasury.gov has free historical yield curve data.
- Quandl is another good free dataset, for example for futures data.
- HistData is also a good source of free FX data.

Data Collection: Data Sources

- Free historical options data is hard to find, however there are many sources that are low cost, such as iVolatility.
- Bloomberg is also a great source for data, and you should take advantage of your free student access.
- *Python has modules that make interfacing with most of these data sources seemless*
- Data for some asset classes (i.e. mortgage-backed-securities, exotic options) is extremely hard to come by while others is readily available and easy to pull into Python. Keep that in mind as you choose a dataset for your course project.

Data Collection: Reading & Writing files in Python

- The pandas module is extremely helpful for working with text/csv files.
- For example, the **pandas.read_csv** file can be used to read a CSV.
- It also has other functions to handle other formats, such as **pandas.read_table** and **pandas.read_html**.
- Generally speaking for every reader function that pandas has, there is a corresponding writer function, which would enable you to write an object, such as a DataFrame to a CSV or other file type.
- The pandas documentation has a lot more information on what types of files you can parse and what the modules capability is.

Data Collection: Reading & Writing files in Python

- If the functionality built into pandas for some reason isn't flexible enough (for example if you are parsing a file with more complex formatting), then you can open it as a file_object and read the file line by line parsing each line with a custom function.

```
f = open(fname, 'r'):
line = f.readline()
while line:
    parse_line()
    line = f.readline()
f.close()
```

- The 'r' parameter specifies that we are opening the file read-only.
- The File Manipulation Page has more info on reading and writing files.

Data Cleaning: Overview

- Once we have identified our data source and created a script that parses the data, our next step will be to **clean** the data.
- Depending on the type of data that you are working with, this process might be quite simple or quite complex.
- The data cleaning process consists of:
  - Filling in missing data
  - Identifying data anomalies and inconsistencies
  - Finding and correcting implied arbitrage opportunities in the data

Data Cleaning: Filling in Missing Data

- Let's say we are given a dataset of stock prices for the last 10 years with gaps in price data a few of the underlying stocks.
- If we want to analyze the data consistently, then it will be critical to have a dataset with equal length for all stocks. To get the intuition behind this, think about computing a correlation with missing data or datasets of different lengths.
- How can we fill in the data?
  - Use the last good value
  - Use linear interpolation
  - Bootstrap the missing data
  - Use regression to fill in the missing values (i.e. infer the change in the missing stock from the changes of the other stocks)
- We should not **look ahead** when we are filling in the data. (Why?)

Missing Data: Bootstrapping Approach

- Bootstrapping is a non-parametric technique for creating sample paths from an empirical distribution.
- The idea behind bootstrapping is that each realization is in actuality just a draw from some unobserved density.
- Bootstrapping provides a method for sampling from that density with minimal assumptions.
- It is also a very simple and intuitive technique, but is quite powerful nonetheless.
- In practice it involves simulating returns, and paths of asset prices by scrambling the historical data, or sampling various historical returns and creating series of them in random order.
- In the context of missing data, it enables us to fill in the missing data by sampling from the historical distribtution.

Bootstrapping: Procedure

A bootstrapping algorithm can be summarized as follows:

- Create a matrix of historical returns for the universe of assets.
- Generate a sequence of random integers between 1 and N, where N is the number of historical days of data.
- Look up the return data for all assets that corresponds to that randomly chosen date.
- Apply these returns to the entire asset universe.
- Continue this for many time steps until the end of a path is reached. Repeat until a large number of paths are created.
- Calculate whatever payoffs, risk measures or other analytics desired from the full set of simulated paths.

Missing Data: Regression Approach

- For simplicity, let's assume that we are operating in a single-factor, CAPM world (as you did in your homework).
- In this case, our model for a stock's return is:

$$r_{i,t} = \beta r_{\text{mkt},t} + \epsilon_{i,t} \tag{1}$$

- In order to use this approach to fill in missing data we would calculate the β of the stock with missing data over the entire dataset for all days where ther is data for both the market and the stock.
- We would then calculate the stock's return on the missing day's using the market return according to (1)

Data Cleaning Example: Arbitrage Checks for Options Data

- In order to prevent arbitrage in our options data, we must have:
  - Call (put) prices that are monotonically decreasing (increasing) in strike.
  - Call (put) prices whose rate of change is greater than 0 (-1) and less than 1 (0)
  - Call and put prices that are convex with respect to changes in strike.
- Question: Why do these rules have to hold to prevent arbitrage?
- As a general rule, I also tend to discard one-sided quotes (that is, with 0 bid or offer).
- Question: What types of check would we do for futures data? Yield curve data?

Working with Databases: Overview

- **Database Design:** The first step in working with a database. It involves creating a set of *tables*, often called a *schema*.
  - Generally, we will want the schema to be relational and will want to define the relationships between the various tables.
  - We will also want to provide rules that the data must adhere to.

    For example, we might require that all stock prices be non-negative.

    We also might require there is only one stock price per date.
- **Database Programming:** The next step involves writing queries that help us define how to insert, update and access the data.
  - Best practice is to use stored procedures and views in order to achieve these tasks.

Working with Databases: Overview

- **Database Administration / Maintenance:** Once we have built the schema and the corresponding set of queries, we will need to continually monitor the database to ensure continued high performance.
  - We will not discuss this aspect of working with databases in this course, but it is good to be aware of.

Working with Databases: Overview

- Most common Database Engines:
  - SQL Server
  - PostgreSQL
  - Oracle
  - MySQL
  - MongoDB
  - Others?
- MongoDB is referred to as a NoSQL database, and is a bit different than the rest in that it supports less rigid data structures and is not relational in nature.
- In this course we will only talk about relational databases, and most examples will either be SQL Server or PostgreSQL based.

Database Design: Key Concepts

- Primary Keys: Unique representation of each row in your table.
- Foreign Keys: Enforces a link between two tables
- Constraints:
  - Not Null
  - Unique
- Column Data Types:
  - Integer: int, bigint, smallint, tinyint
  - String: char, varchar
  - Numeric: float, decimal
  - Large Unstructured Data: TEXT, BLOB
  - Date: date, datetime
  - Boolean: bit

Database Design: Sample Schema First Attempt

- instrument_data
  - instrument_ticker
  - start_date
  - end_date
  - quote_date
  - timestamp
  - mkt_price

Database Design: Sample Schema Second Attempt

- instrument_info
  - instrument_id
  - instrument_ticker
  - start_date
  - end_date
- instrument_prices
  - instrument_id
  - quote_date
  - timestamp
  - mkt_price

Database Design: Sample Schema Third Attempt

- instrument_info
  - instrument_id
  - instrument_ticker
  - start_date
  - end_date
- instrument_prices
  - instrument_id
  - data_run_id
  - quote_date
  - mkt_price
- data_runs
  - data_run_id

  - data_run_date
  - data_run_time

- Why might we prefer the second schema?
- Where should we put primary keys? Foreign keys?

Database Programming: Key Commands

- There are 4 main types of SQL commands:
  - SELECT
  - UPDATE
  - INSERT
  - DELETE
- SELECT queries define how our stored data is made available to our model and other applications.
- INSERT, UPDATE and DELETE queries define how our data storage process work and how our data collection process writes to our database.
- In this course we will focus on SELECT statements.

Basics of SQL Programming: Example INSERT, UPDATE and DELETE Statements

- INSERT

```
INSERT INTO prices (ticker, close_price)
VALUES ('MSFT', 100.0);
```

- UPDATE

```
UPDATE prices SET close_price = 105.0
WHERE ticker = 'MSFT';
```

- DELETE

```
DELETE FROM prices
WHERE ticker = 'MSFT';
```

Basics of SQL Programming: Example INSERT, UPDATE and DELETE Statements

- NOTE: UPDATE and DELETE commands are somewhat risky and should be executed carefully. It is a best practice to always run the query initially as a SELECT statement to ensure that the number of rows and content of the rows is what you expect.

Basics of SQL Programming: Querying results from a table

- The above queries would return **ALL** rows from the Equity_Prices table. We can use a **WHERE** clause to restrict the rows returned:

```
1  SELECT ticker, price
2  FROM Equity_Prices
3  WHERE quote_date = 'today'
```

- A **WHERE** clause can be arbitrarily complex, and as we will see later can even include another query.

Basics of SQL Programming: Ordering query results

- An **ORDER BY** clause can be used to sort your query results by one or more columns.

```
1          SELECT ticker, shares, mkt_value
2          FROM positions
3          WHERE quote_date = 'today'
4          ORDER BY mkt_value DESC
```

- An order by clause can reference a column either by name or by number, so the following query is equivalent:

```
1          SELECT ticker, shares, mkt_value
2          FROM positions
3          WHERE quote_date = 'today'
4          ORDER BY 3 DESC
```

- These queries will show all positions with quote_date equal to the current date and will display them in order from highest market value to lowest.
- The default sort order is ascending. The **DESC** keyword will sort the results in descending order.
- If the **DESC** keyword was removed from the query, or replaced with **ASC**, then the results would be return in order from lowest market value to highest.

Basics of SQL Programming: Querying results from multiple tables

- The JOIN keyword can be used to constuct a query that pulls data from multiple tables:

```
1 SELECT px.ticker, price, shares
2 FROM prices px
3     JOIN positions pos ON px.ticker = pos.ticker
```

- Notice that this query uses *aliases* in the FROM clause.
- Also note that in the SELECT portion of the query the alias is added to the first column (ticker). This is because the column is defined in both tables in the FROM clause.
- It is best practice to JOIN on int data types rather than strings.
- This query will only return rows where the ticker exists in both the prices and positions tables.

Basics of SQL Programming: Different types of JOINs

- But what if we wanted to return all prices regardless of whether we have a corresponding position row? We would need to use a different type of JOIN
- SQL supports the following types of JOIN's
  - INNER
  - LEFT OUTER
  - RIGHT OUTER
  - FULL OUTER
- The default JOIN type is INNER JOIN
- A LEFT JOIN in the query above would allow us to accomplish our objective.

Basics of SQL Programming: Aggregating data using a GROUP BY clause

- In some cases, we will want to write an aggregate query:

```
1  SELECT ticker, SUM(shares)
2  FROM positions
3  GROUP BY ticker
```

- The **SUM** function is the most common aggregate function. Other common aggregate functions are COUNT, MAX, MIN.
- The list of columns in the **GROUP BY** clause define how the results will be aggregated.
- In this example our query will return a single row per ticker and sum the shares row all rows in the table for each ticker.
- If we were to add another column to the **GROUP BY** clause, say quote_date, then one row would be returned per day per ticker.

Basics of SQL Programming: Aggregating data using a GROUP BY clause

- We can use a **HAVING** clause to restrict the groups returned.

```
1          SELECT ticker, SUM(shares) as shrs
2          FROM positions
3          GROUP BY ticker
4          HAVING shrs > 0
```

- This query only displays groups where the SUM of the shares column is positive (long-only positions). We could replace shrs with abs(shrs) to return all groups with non-zero positions.
- The DISTINCT keyword can be used instead of a GROUP BY clause to group by all columns in the SELECT list

```
1          SELECT DISTINCT ticker, SUM(shares)
2          FROM positions
```

Advanced SQL Programming: Joining to a sub-query

- If we chose to, we can write SQL queries that are far more complex.
- In particular we can write queries where one or more of the tables in the FROM clause is a separate sub-query:

```
1          SELECT *
2          FROM prices px
3              JOIN (SELECT *
4                        FROM positions) a
5              ON px.ticker = a.ticker
```

Advanced SQL Programming: Joining to a sub-query

- Similarly, we can write queries where the WHERE clause has a condition that involves a separate sub-query:

```
1          SELECT *
2          FROM prices px
3          WHERE ticker IN (SELECT ticker
4          FROM positions)
```

Advanced SQL Programming: Pivoting data with CASE statements

- CASE statements can be used to add conditional statements to SQL queries:

```
1        SELECT ticker,
2                    CASE WHEN shares > 0 then 1
3                                ELSE 0
4                                END as is_long
5        FROM positions
6        WHERE quote_date = 'today'
```

Advanced SQL Programming: Pivoting data with CASE statements

- We can use these conditional statements to help us pivot data:

```
SELECT ticker,
SUM(CASE WHEN MONTH(qd) = 1 then 1 ELSE 0 END) as JAN,
SUM(CASE WHEN MONTH(qd) = 2 then 1 ELSE 0 END) as FEB,
SUM(CASE WHEN MONTH(qd) = 3 then 1 ELSE 0 END) as MAR,
...
SUM(CASE WHEN MONTH(qd) =12 then 1 ELSE 0 END) as DEC

FROM positions
GROUP BY ticker
```

- Without using this pivot method, we could easily group the data by month but would not be able to create a separate column for each month.

Advanced SQL Programming: Stored Procedures & Views

- **Stored procedures** and **Views** are both just stored SQL queries.
- Storing the queries enables them to be more efficient as the database is able to pre-calculate it's execution plan.
- Views are SELECT statements that return some result set to the user.
- Stored procedures are more flexible.
  - They can return a result set or nothing
  - They can execute SELECT statements as well as INSERT/UPDATE/DELETE statements.
- It is best practice to use these objects rather than create SQL queries dynamically in your Python/C++ code.

Indices & Query Optimization

- Indices are like an appendix or table of contents for a book. They guide the query engine to the right rows efficiently as the query runs.
- There are two types of indices: Clustered and Non-clustered
- Clustered indices control how the data is actually stored.
- Queries can be optimized by ensuring that WHERE clauses contain column that are part of the table's indices.
- If we are frequently running a query against columns not part of the index, then we might want to create a new index that uses this and other columns.
- While indices speed up select statements, they may also cause INSERT/UPDATE/DELETE statements to slow down. (Why?)

Database Connections in Python

- Python has modules to help connect to many database platforms.
- The **pypyodbc** module in particular can be used to connect to a database via an ODBC connection.
- Here's a code snippet that would allow us to connect to a SQL Server database:

```
1     import pypyodbc
2     import pandas as pd
3
4     cnxn = pypyodbc.connect(
5             "Driver={SQL Server Native Client 11.0};"
6             "Server=server_name;"
7             "Database=db_name;"
8             "uid=User;pwd=password")
9     qry = 'select * from table'
10    df = pd.read_sql_query(qry, cnxn)
```

Database Connections in Python

- The variable **cnxn** defines a connection to the database which can execute many queries.
- The pandas module provides a function which takes a database connection and a query string, and executes the query and returns the results in a DataFrame object.
- NOTE: It is best practice to close all database connections once your code is finished running. This can be done by calling **cnxn.close()**.
- It is also best practice to use a single database conenction through your entire application.
  - Opening multiple connections can harm performance of your database, and there is a significant amount of overhead required to open and close each connection.

Permissions of Database Connections in Python

- This syntax can be used to execute any SQL query, whether it is a **SELECT** query or a **DELETE** query that empties the entire database.
- *It is best practice to restrict the permissions of applications or other user interfaces, such as Python as much as possible.*
- For example, we might give Python only **SELECT** permission on our database so that an application or user cannot accidently alter the underlying data.
- If we wanted the application to be able to **DELETE**, **INSERT** or **UPDATE** in a controlled way, we would then give the application permission to execute **STORED PROCEDURES** that performed these tasks in a robust way,

Review: Factor Modeling

- In your last homework, you looked at a single factor / CAPM model approach to sector ETF returns.
- While there is a good bit of theory that tells us that the CAPM model *should work* in practice most people use multi-factor models.
- What other factors might we add? There are two general approaches to this...
  - Choose an expanded number of named, fundamental/economic factors. A well known example of this is the Fama-French factor model.
  - Choose the factors statistically

Review: Fama French Factors

- In the Fama French model we define the factors as:
  - Market Return
  - Size (Market Capitalization)
  - Book to market (Value)
- That is:

$$r_{i,t} = \beta_1 r_{\text{mkt},t} + \beta_2 r_{\text{size},t} + \beta_3 r_{\text{btm},t} + \epsilon_{i,t} \tag{2}$$

- Of course, there could be other factors that impact the return of a stock and asset managers are always looking for the next factor that might help them predict stock returns.

Review: Black-Scholes Greeks

- As we saw in the last lecture, the Black-Scholes model has a closed-form solution for European call and put option prices:

$$c_0 = \tilde{\mathbb{E}}\left[e^{-rT}\left(S_T - K\right)_+\right] = \Phi(d_1)S_0 - \Phi(d_2)Ke^{-rT} \tag{3}$$

- Here, Φ is the CDF of the standard normal distribution and

$$\begin{aligned} d_1 &= \frac{1}{\sigma\sqrt{T}}\left(\ln\frac{S_0}{K} + \left(r + \frac{\sigma^2}{2}\right)T\right), \\ d_2 &= d_1 - \sigma\sqrt{T} \end{aligned} \tag{4}$$

Review: Black-Scholes Formula

- If we differentiate (3) with respect to various parameters, we will find that the sensitivities, or Greeks, in the Black-Scholes mode also have a closed-form solution.
- For example, if we differentiate (3) with respect to the underlying asset we can see that the delta of a call option is:

$$\Delta = \Phi(d_1) \tag{5}$$

- Note that this derivative is not trivial as S_0 appears in the $\Phi(d_2)$ as well as the first term.
- Similar analytical expressions can be derived for vega, gamma, theta and rho.

Review: Calculation of Greeks for Exotic Options or more complex Stochastic Processes

- As our stochastic process becomes more complex, or the option we are pricing becomes more complex, we should not expect analytical formulas for prices or Greeks.
- For example, in the case of a lookback option, we needed to use *simulation* to get price under the Black-Scholes model.
- As a result, we must resort to calculating Greeks numerically. We can do this by **shifting** the input parameter and re-calculating the input price. For example, for Δ we would have:

$$\Delta \approx \frac{c_0(S_0 + \epsilon) - c_0(S_0 - \epsilon)}{2\epsilon} \tag{6}$$

where ϵ is some pre-defined shift amount.