FQE
Financial Quants and Engineers Club

High-Frequency Trading with Fractional Brownian Motion

Presented By
Arthur F, Matt C, Amber T

Theory

Brownian Motion…

- Represents that price changes are unpredictable
- Returns are completely independent of each other and no memory
- Thus no predictions and money can be made from previous data
- Basically a random walk

Theory

Fractional Brownian Motion has memory

- Increments of the Brownian Motion are no longer independent
- This is accomplished mathematically through several steps
- The most important is the inclusion of the Hurst exponent
- $H > 0.5$ implies persistence, $h < 0.5$ implies mean reversion

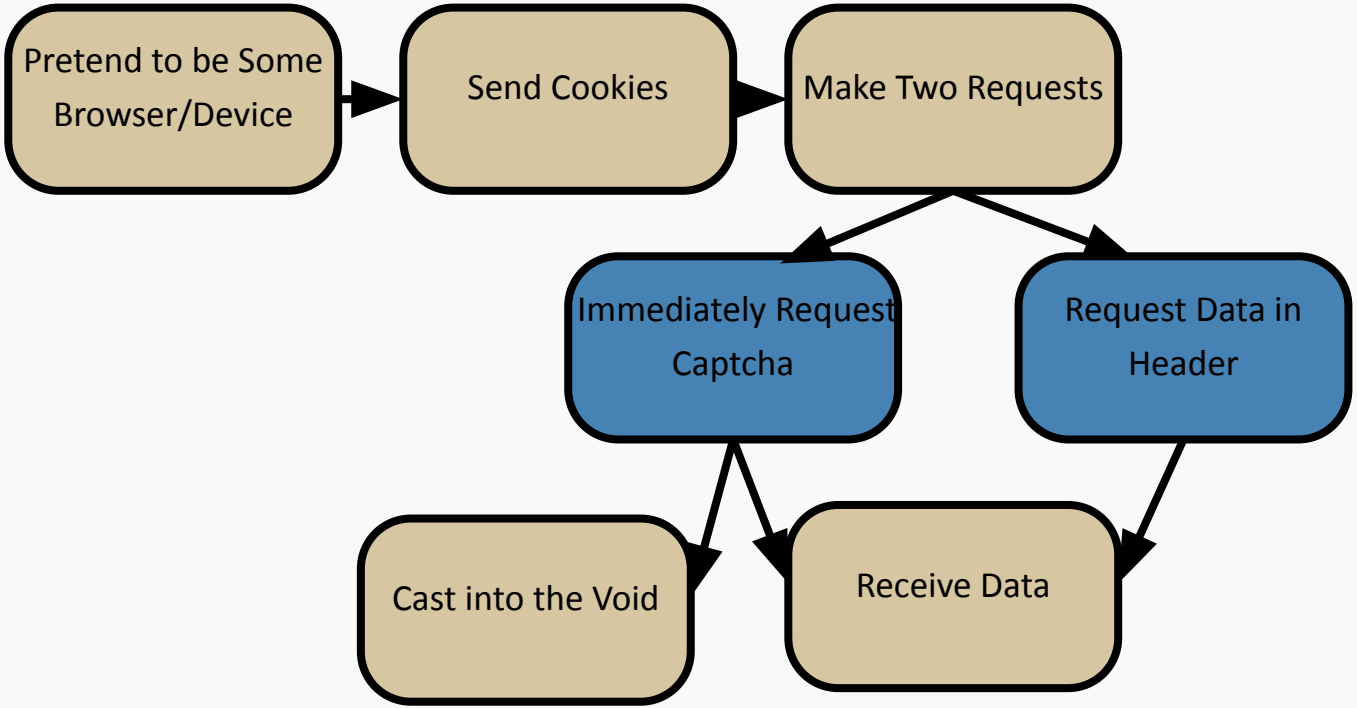
Grabbing Hourly Data

The Hurst Exponent relies on substantial amounts of data in order to function. This means their is a necessity to grab substantial amounts of hourly data quickly and efficiently.

WRDS data has a system called TAQ (Trade and Quote, or something similar) which provides ALL orders, but this would require a bunch of data normalization.

Solution: Web Scraping!

Scraping Hourly Data



Scraping Data Live

```python
Cookie_db_proxy = _peewee.Proxy()
class ISODateTimeField(_peewee.DateTimeField):
    # Ensure Python datetime is read & written correctly for sqlite,
    # because user discovered peewee allowed an invalid datetime
    # to get written.
    def db_value(self, value):
        if value and isinstance(value, _dt.datetime):
            return value.isoformat()
        return super().db_value(value)
    def python_value(self, value):
        if value and isinstance(value, str) and 'T' in value:
            return _dt.datetime.fromisoformat(value)
        return super().python_value(value)
class _CookieSchema(_peewee.Model):
    strategy = _peewee.CharField(primary_key=True)
    fetch_date = ISODateTimeField(default=_dt.datetime.now)

    # Which cookie type depends on strategy
    cookie_bytes = _peewee.BlobField()
```

```python
def _is_this_consent_url(self, response_url: str) -> bool:
    """
    Check if given response_url is consent page

    Args:
        response_url (str) : response.url

    Returns:
        True : This is cookie-consent page
        False : This is not cookie-consent page
    """
    try:
        return urlsplit(response_url).hostname and urlsplit(
            response_url
        ).hostname.endswith("consent.yahoo.com")
    except Exception:
        return False

def _accept_consent_form(
    self, consent_resp: requests.Response, timeout: int
) -> requests.Response:
    """
    Click 'Accept all' to cookie-consent form and return response object.

    Args:
```

```python
@classmethod
def _initialise(cls, cache_dir=None):
    if cache_dir is not None:
        cls._cache_dir = cache_dir

    if not _os.path.isdir(cls._cache_dir):
        try:
            _os.makedirs(cls._cache_dir)
        except OSError as err:
            raise _ISINCacheException(f"Error creating ISINCache folder: '{cls._cache_dir}' reason: {err}")
    elif not (_os.access(cls._cache_dir, _os.R_OK) and _os.access(cls._cache_dir, _os.W_OK)):
        raise _ISINCacheException(f"Cannot read and write in ISINCache folder: '{cls._cache_dir}'")

    cls._db = _peewee.SqliteDatabase(
        _os.path.join(cls._cache_dir, 'isin-tkr.db'),
        pragmas={'journal_mode': 'wal', 'cache_size': -64}
    )
```

Hurst Equation

Math to program

For each such time series of length n, $X = X_1, X_2, \ldots, X_n$, the rescaled range is calculated as follows:[6][7]

1. Calculate the mean;

$$m = \frac{1}{n}\sum_{i=1}^{n} X_i\,.$$

2. Create a mean-adjusted series;

$$Y_t = X_t - m \quad \text{for } t = 1, 2, \ldots, n\,.$$

3. Calculate the cumulative deviate series Z;

$$Z_t = \sum_{i=1}^{t} Y_i \quad \text{for } t = 1, 2, \ldots, n\,.$$

4. Compute the range R;

$$R(n) = \max(Z_1, Z_2, \ldots, Z_n) - \min(Z_1, Z_2, \ldots, Z_n).$$

5. Compute the standard deviation S;

$$S(n) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (X_i - m)^2}.$$

6. Calculate the rescaled range $R(n)/S(n)$ and average over all the partial time ser

```python
def calculate_hurst_exponent(data: pd.DataFrame, window_size: int) -> float:
    windows = split_windows(data, window_size)
    hurst_exponents = []

# then in windows, slice returns
    for window in windows:
        # need to copy to avoid warnings
        window = window.copy()
        # convert to log returns
        window['Log Return'] = np.log(window['Close'] / window['Close'].shift(1))
        window = window.dropna(subset=['Log Return'])
        # calculate the mean
        mean = window['Log Return'].mean()
        # mean adjusted series, just subtract the mean from the log return
        window['Log Return'] = window['Log Return'] - mean
        # calculate cumulative deviate series (haha cumsum)
        window['Cumulative Deviate'] = window['Log Return'].cumsum()
        # compute range
        range = window['Cumulative Deviate'].max() - window['Cumulative Deviate'].min()
        # caculate std from the og
        std = window['Log Return'].std()
        # calculate the hurst exponent
        hurst_exponents.append(range / std)
    return np.mean(hurst_exponents)
```

Hurst Equation

How much of a random walk is the market?

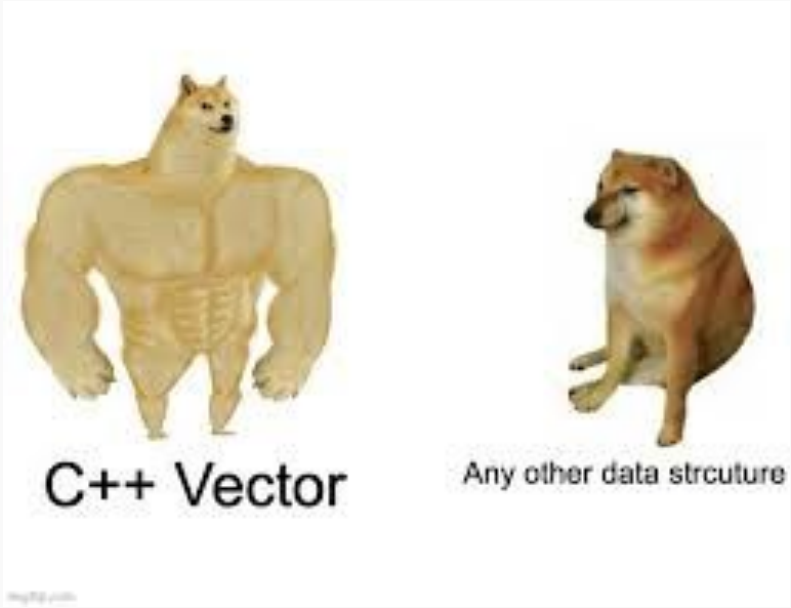
```
data = grab_data("ORCL", "2025-01-01", "2025-04-01")[['Date', 'Close']]
window_sizes = [50, 100, 150, 200, 250]
print(calculate_hurst_fitted_exponent(data, window_sizes))
rng = np.random.default_rng()
random_integer = rng.integers(low=-1, high=1, size=10000)
init = 100000
data = [init]
for i in random_integer:
    if i == 0:
        init -= 1
    else:
        init += 1
    data.append(init)
data = pd.DataFrame({'Close': data})
print(calculate_hurst_fitted_exponent(data, window_sizes))
```

The market implements a random walk better than an actual random walk generator

```
(venv) xct@Arthurs-MacBook-Pro-2 High-Frequency-Trading-with-Fractional-Brownian-Motion % pyt
hon -m src.grab_data.hurst_calculator
0.49813332838667146
0.5261702366763307
```

C++

Hurst Equation is Slow



We attempted to create a C++ piece of code to more quickly perform calculations and data reads

Problem: Not all of us are super familiar with C++

Solution: Pybind! Combine the two languages.

Pybind is a pain, Pandas is too good

Hurst Equation is Slow

C++

```
brew install pybind11
cmake -DCMAKE_PREFIX_PATH="$(brew --prefix pybind11)" .

if downloaded to a virtual environment, may also have to run build with venv running
if in doubt, can always try copying over the module into scripts directory

cp build/price_module.cpython-313-darwin.so scripts/
python3 scripts/index.py

Run Application

direct python to the module's executables in the build directory

PYTHONPATH=build python3 scripts/index.py
```

```
from datetime import datetime as dt
from ..historicaldata impor
import matplotlib.pyplot as

def grab_data(ticker: str,
    historical_data = Histo
    return historical_data.history
```

(property) history: DataFrame

Returns the historical data as a pandas DataFrame.

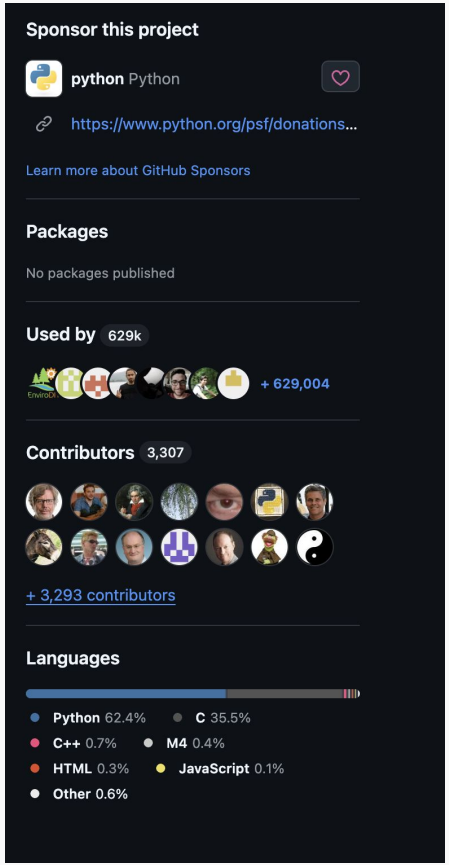
Returns:
pd.DataFrame: The historical stock data.


```
template <typename T>
    requires std::same_as<T, std::size_t> || std::same_as<T, int>
PriceRow PriceHistory::get_row(T index) const {
    PriceRow row{permno[index], date[index], ticker[index], permco[index],
                 bidlo[index], askhi[index], prc[index], vol[index],
                 ret[index],   shrout[index]};
    return row;
}

void PriceHistory::head() const {
    std::cout << std::setw(8) << "PERMNO" << std::setw(12) << "DATE"
              << std::setw(8) << "TICKER" << std::setw(8) << "PERMCO"
              << std::setw(10) << "BIDLO" << std::setw(10) << "ASKHI"
              << std::setw(10) << "PRC" << std::setw(10) << "VOL"
              << std::setw(10) << "RET" << std::setw(10) << "SHROUT" << "\n";
    for (size_t i{0}; i <= 5; ++i) {
        std::cout << get_row(i);
    }
}
```

Also Pybind was difficult to initialize and use consistently

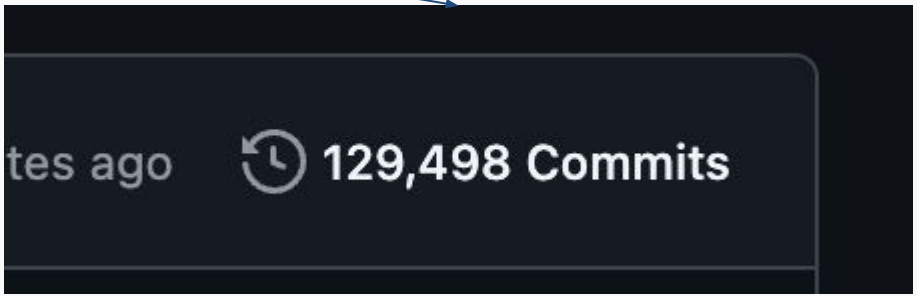
Python got hands tho ngl



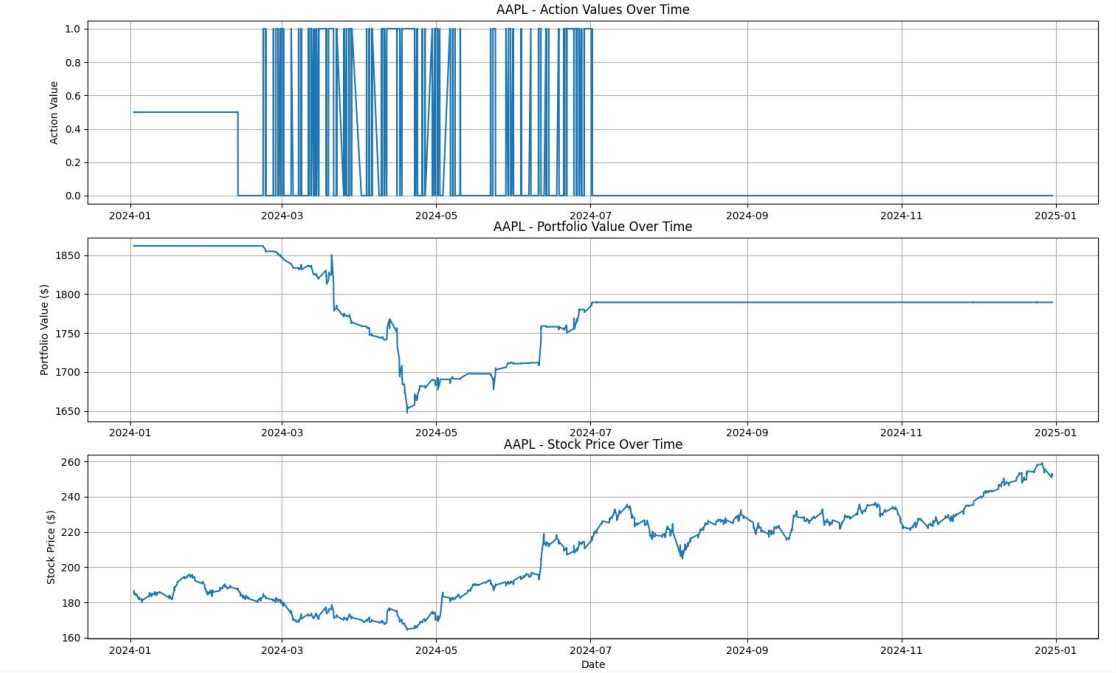
Python is not that Slow

- Python is mostly just C with essentially wrappers
- This means that actual time gains are more negligible than you may expect
- C and C++ have a bunch of control and good libraries which is why they're used, not necessarily time boost

Terrifying



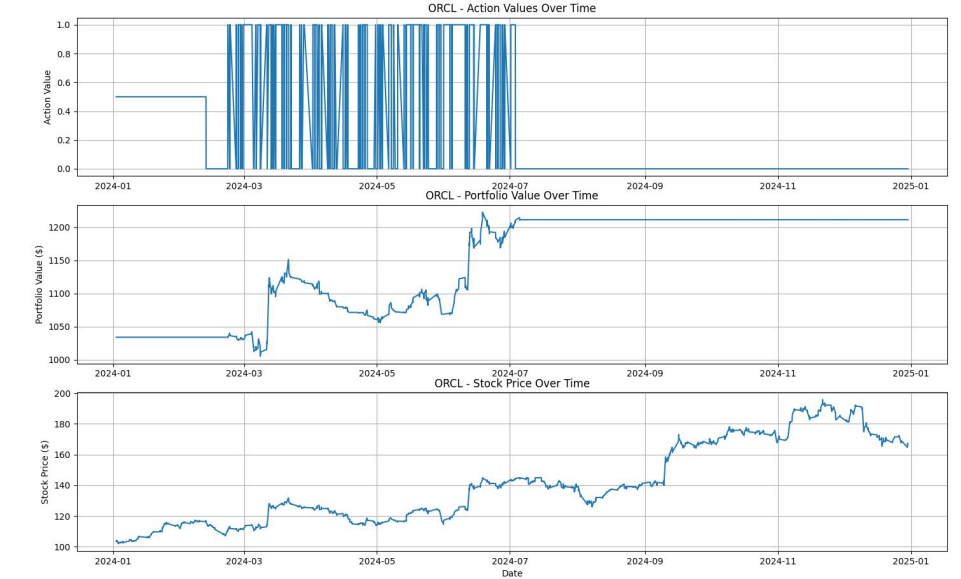
Backtests



Don’t mind the
Panda warnings lmao

```
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/xct/dev_projects/brownian_motion/High-Frequency-Trading-with-Fractional-Brownian-Motio
n/venv/lib/python3.13/site-packages/numpy/_core/_methods.py:144: RuntimeWarning: invalid valu
e encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
{'initial_budget': np.float64(1861.8989562988281), 'final_value': np.float64(1789.51180262756
72), 'return': np.float64(-0.03887813214910197), 'time_seconds': 37.44910788536072}
Backtest completed in 37.45 seconds
```

Backtests




```
e encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
{'initial_budget': np.float64(1033.8999938964844), 'final_value': np.float64(1211.24379018402
46), 'return': np.float64(0.17152896540716703), 'time_seconds': 36.92596888542175}
Backtest completed in 36.93 seconds
^CTraceback (most recent call last):
```

FQE
Financial Quants and Engineers Club

Questions?