Journal of Statistical Software

August 2022, Volume 103, Issue 14. doi: 10.18637/jss.v103.i14

HighFrequencyCovariance: A Julia Package for Estimating Covariance Matrices Using High Frequency Financial Data

Stuart Baumann

Margaryta Klymak
University of Oxford

Abstract

High frequency data typically exhibit asynchronous trading and microstructure noise, which can bias the covariances estimated by standard estimators. While a number of specialized estimators have been proposed, they have had limited availability in open source software. **HighFrequencyCovariance** is the first Julia package which implements specialized estimators for volatility, correlation and covariance using high frequency financial data. It also implements complementary algorithms for matrix regularization. This paper presents the issues associated with exploiting high frequency financial data and describes the volatility, covariance and regularization algorithms that have been implemented. We then demonstrate the use of the package using foreign exchange market tick data to estimate the covariance of the exchange rates between different currencies. We also perform a Monte Carlo experiment, which shows the accuracy gains that are possible over simpler covariance estimation techniques.

Keywords: covariance estimation, correlation, volatility, high frequency financial data, Julia.

1. Introduction

A common problem in financial econometrics is the estimation of covariance and correlation matrices between the price series of different assets. These matrices are frequently used by financial institutions. For example, they allow traders and investors to understand the risk characteristics of the portfolios they hold and what assets to buy or sell to better hedge their position. These matrices can also be applied in algorithmic trading strategies, in pricing algorithms for derivatives, in mean-variance portfolio selection (Markowitz 1952) or for better understanding the macroeconomic relationships between different assets in the economy. Despite their wide array of applications, few open source software packages exist to consistently estimate covariance matrices using high frequency data. This package aims to fill that gap.

Covariance and correlation matrices can be more accurately estimated utilizing high frequency data than is possible with lower frequency data. If observations of covariance are observed over long intervals such as an hour there are few observations of price movements per trading day. On the other hand, if price movements can be measured in the space of a few seconds then a daily covariance matrix can be estimated using thousands of observations and thus can achieve more accurate estimates. There are however three major statistical complications with exploiting high frequency financial data to estimate covariance matrices. The first is that we observe updated prices (typically obtained from realized trades or updated quotes) for different assets asynchronously and at different frequencies. If one uses a short duration between returns, some assets might not have an updated price within some intervals which leads to bias in estimating covariance. The simplest remedy of using longer return durations forgoes the efficiency gains that high frequency data makes possible. The second difficulty is "microstructure" noise. If trade prices are used, there is generally some noise depending on where an asset trades between the bid and the ask price. If quote prices are used (for instance using the mid price halfway between the bid and the ask), then there may be small jumps as the bid and ask move by discrete amounts (or "ticks") rather than moving continuously. Adding to this issue is the possibility that microstructure noise may be correlated with the underlying price process which can introduce additional bias. The third difficulty is that an estimated matrix may not be positive semidefinite (PSD) which can happen for several reasons. These reasons include the use of different datasets that estimate covariances between different asset pairs as well as the application of econometric techniques that do not guarantee a PSD result. These cases require regularization techniques to ensure that the resultant covariance matrix does not imply mutually impossible correlations.

The **HighFrequencyCovariance** package (Baumann and Klymak 2021) is the first package in Julia (Bezanson, Edelman, Karpinski, and Shah 2017) that aims at averting all of these issues thus allowing for consistent and efficient estimation of correlation matrices and volatilities. First, we implement two algorithms to estimate volatility and the variance of microstructure noise. Second, we include five covariance estimation techniques that can be used in the face of microstructure noise and nonsynchronous trading. Third, our package includes four regularization techniques. Finally, the package offers several convenience functions to work with covariance matrices. These include functions for combining different covariance matrices as well as functions for estimating covariances between different asset pairs separately and then combining the result into a unified PSD covariance matrix.

To our knowledge, there are no other open source packages implemented in Julia that provide high frequency covariance estimation techniques. For some of the algorithms implemented, we believe **HighFrequencyCovariance** provides the first open source implementation in any language. The closest package to ours is **highfrequency** in R (Boudt, Cornelissen, Payseur, Kleen, and Sjoerup 2022). While the goal of **highfrequency** appears to be similar to our package's goal, which is to support the estimation of covariance and volatility using high frequency data, the set of functionalities implemented by each package is quite different. On the one hand, **highfrequency** offers eigenvalue regularization, but it does not implement Higham's (2002) nearest correlation matrix technique or Ledoit and Wolf's (2001) identity matrix regularization. It also does not implement the spectral local-method-of-moments covariance estimation method or a convenient way to perform blockwise estimation of covariance matrices. While **highfrequency** offers several statistical tests for jumps (Aït-Sahalia and Jacod 2009), our Julia package currently does not.

This paper first defines the covariance matrices that we seek to estimate in Section 2. We then describe the algorithms that have been implemented in **HighFrequencyCovariance** for overcoming the econometric issues in volatility and covariance estimation in Section 3 and Section 4. We focus on providing some brief intuition to guide the choice of what algorithm to use and avoid mathematical and convergence proofs (readers interested in a more formal presentation can find these details in the cited papers). Section 5 discusses the implemented regularization algorithms. Section 6 discusses the software design choices of the package. The use of the package is then illustrated first in the simple setting of using Monte Carlo generated data in Section 7 and is then demonstrated on real foreign exchange (FX) data in Section 8. Section 9 uses a Monte Carlo experiment to show the accuracy gains (and consistency) of the covariance estimation techniques that **HighFrequencyCovariance** implements. Section 10 concludes the paper.

2. The covariance estimation problem

To clearly define the goal of our estimation and our notation, we will consider that the data generation process can be modeled by the stochastic integral:

$$\mathbf{Y_t} = \int_{t_0}^{T} \sigma_{\mathbf{t}} d\mathbf{W_t},$$

where $\mathbf{W_t}$ is a vector of independent Brownian motions, $\sigma_{\mathbf{t}}$ is a càdlàg volatility matrix process and t is the time index.

$\mathbf{Y}_t$ reflects a vector of "underlying" prices for every asset; however, it cannot be observed directly. Instead, we can see a price for each asset corrupted by market microstructure noise at asynchronous times. If we observe an asset (with index k) at a tick[1] (with index j so it is the jth observation for this asset) at time t_j^k then we observe the price X_j^k:

$$X_j^k = Y_{t_j^k}^k + \epsilon_j^k,$$

where ϵ_j^k is an expectation zero random variable that reflects microstructure noise. It is subscripted by j rather than t_j^k reflecting that it is different for each tick and does not move continuously with time as $\mathbf{Y}_t$ does. For an asset k the time of the first tick is denoted t_1^k, the number of ticks is n_k and the time of the last tick is $t_{n_k}^k$. We will define the first and last times across all assets as $t_0 = \min_k t_1^k$ and $T = \max_k t_{n_k}^k$ respectively.

Our goal is to estimate the integrated covariance matrix over the time period for which we have data:

$$\Sigma = \int_{t_0}^{T} \sigma_{\mathbf{t}} \sigma_{\mathbf{t}}^{\top} dt.$$

We can then estimate each element of the correlation matrix (with row and column indices r, c) by:

$$\text{Corr}_{r,c} = \frac{\Sigma_{r,c}}{\sqrt{\Sigma_{r,r}}\sqrt{\Sigma_{c,c}}}. \quad (1)$$

[1] In the literature the same word "tick" is used to denote the minimum increment for prices in the orderbook and also for individual observations in a time series database. We will generally mean the latter meaning of "tick" throughout this paper except for mentioning the effect of minimum quote increments as a source of microstructure noise.

Finally, the volatility, ν_q, for the asset q can be estimated by:

$$\nu_q = \frac{\sqrt{\Sigma_{q,q}}}{\sqrt{T - t_0}} \tag{2}$$

Note that our calculated volatilities will be scaled to the same time units as the input data.

The central structure implemented in our package, `CovarianceMatrix`, stores the correlation matrix and the vector of volatilities separately. This is because the variance of the stochastic price process increases with time and in most applied settings a covariance matrix will be desired over a different duration than the training data. In our package, an actual covariance matrix can be calculated for any desired duration by combining the correlation matrix and volatilities stored in the `CovarianceMatrix` structure.

3. Volatility estimation

For every covariance estimation technique that we implement, we can get an estimate of the volatility from the covariance matrix through Equation 2. In addition to this, **HighFrequencyCovariance** implements two techniques that purely estimate volatility. These can be used as an alternative way of estimating volatility. The most basic method (implemented in the `simple_volatility` method) is to add up the quadratic variation of the logarithmic price series. Thus, if we have a series $\mathbf{X}$, then we get the quadratic variation as $\sum_{j=1}^{N-1}(X^k_{j+1} - X^k_j)^2$. We then convert it to volatility by dividing it by the duration of the returns $t^k_{n_k} - t^k_1$ and then taking the square root.

However, microstructure noise causes severe biases in estimating volatilities with this simple method. As the time between two ticks goes to zero, the variation in the continuous price process can be small, but it will still have the microstructure noise. In the extreme case when there are infinitely many ticks in a finite time interval, the simple volatility estimation technique is a consistent estimator for the variance of the noise process $E[(\epsilon^k)^2]$ rather than the volatility of the clean price process (Zhang, Mykland, and Aït-Sahalia 2005).

The simplest solution to this problem is tick subsampling so that there is a price every 15 minutes (or some suitable duration). Over this longer period, the impact of microstructure noise is hopefully negligible compared to the continuous variation in Y^k_t. This has the disadvantage, however, of an efficiency loss through throwing away data. In addition, there is no natural estimate of the variance of the microstructure noise, which is useful in its own right and is also an input to some covariance estimation techniques.

Zhang *et al.* (2005) provide a different solution for estimating volatility. In their method, two time scales are used to estimate volatility, one using all the ticks (a short time scale) and one using different subsamplings of the tick data at a longer timescale. They then combine these two estimates to provide a consistent estimator for the volatility of the price process.

Given an accurate estimate of the volatility of the price process, the quadratic variation that would theoretically be expected over the duration of the data can be determined. The amount that measured quadratic variation in X^k_j exceeds this theoretical quadratic variation can be ascribed to microstructure noise. We can thus use this to find an estimate of the variance of the noise process.

This algorithm is implemented in the `two_scales_volatility` method of the package which returns estimates of both the volatility of the price process and the variance of the noise.

4. Covariance estimation

The first covariance estimation technique implemented in **HighFrequencyCovariance** is the `simple_covariance` method. In this method, we loop over each pair of assets k, k' with prices as of some grid of time periods that we choose $t_1, t_2, t_3, ..., t_N$ and estimate the covariance as:

$$\mathsf{COV}(X^k, X^{k'}) = \frac{1}{N}\sum_{l=1}^{N}(X^k_{l^k} - \bar{X^k})(X^{k'}_{l^{k'}} - \bar{X^{k'}}),$$

where $l^k := \text{argmax}_h(t^k_h)1_{t^k_h<t_l}$ is the most recent tick for asset k before time t_l.

This method has several advantages. It is easy to implement and is a fast running algorithm. The resultant matrix is guaranteed to be PSD and this method is already known to most people with basic statistical training.

There are several issues associated with high frequency financial data that can lead to covariance estimates from the simple method being inconsistent or inefficient. The first issue is that using too fine a time grid with this standard technique will result in downwards biased correlation estimates due to assets trading asynchronously. This is sometimes termed the Epps (1979) effect and it is best illustrated with a simple example. Consider that we have two perfectly correlated assets and we observe both prices at time 0. We observe the first asset's price at time 1 at which time it has returned α. Then at time 2, we observe the second asset which has returned $\alpha\beta$ over the period between 0 and 2. If we use returns between times 0 and 1 then the first asset returned α while the second returned 0. Between 1 and 2 we measure the first asset as having returned zero while the second returned $\alpha\beta$. While these assets are perfectly correlated, we will estimate a correlation of zero in this case which comes about entirely due to a lack of synchronicity of observations. The second issue is that microstructure noise can bias covariance estimates in the same way as it can bias volatilities. To address these shortcomings, we implemented four additional covariance estimation techniques that are more robust to asynchronous observations and microstructure noise.

4.1. Multivariate realized kernel

The second estimation method is the multivariate realized kernel (Barndorff-Nielsen, Hansen, Lunde, and Shephard 2011). Sometimes called the BNHLS method after the authors, the multivariate realized kernel is an algorithm designed to provide consistent PSD covariance estimates despite settings where there is microstructure noise (that may not be independent of the underlying price process) and asynchronously traded assets. It is a refinement of an earlier algorithm, the univariate realized kernel estimator (Barndorff-Nielsen, Hansen, Lunde, and Shephard 2008), which is faster converging but relies on an assumption of independence between microstructure noise and the underlying price process.

The BNHLS estimator uses refresh time sampling. This means we take a time grid defined by a recursive pattern where each subsequent time is defined as the time by which every asset has an updated price since the previous time:

$$t_{l+1} = \max_k \left(t^k_{l,\min}\right) \qquad \text{where} \qquad t^k_{l,\min} = \min_j t^k_j | t^k_j > t_l.$$

Given this time grid, we then estimate returns based on the most recent price for each asset at each refresh time. We construct a h-th realized autocovariance matrix:

$$\Gamma_h = \sum_{j=h+1}^{n} X_j X_{j-h}^\top \qquad \text{for} \qquad h \geq 0,$$
$$\Gamma_h = \Gamma_{-h}^\top \qquad \text{for} \qquad h < 0.$$

This is implemented in the `bnhls_covariance` method which uses a weighted summation of these realized autocovariance matrices with a kernel function to determine how much to weigh each one. By default **HighFrequencyCovariance** uses the Parzen kernel, but the quadratic spectral, Fejer, Tukey-Hanning and BNHLS (2008) kernels that are discussed in Barndorff-Nielsen *et al.* (2011, Table 1) are also available.

The use of refresh time sampling here means that the number of returns that can be used to calculate a covariance is governed by the slowest asset to update. As a result, it may be natural to pair this estimator with a blocking and regularization technique (Hautsch, Kyj, and Oomen 2012). In this case, we bunch assets by their trading frequency. The covariances between rapidly trading assets can be calculated at a high frequency while calculating covariances involving a slow trading asset at a slower frequency. **HighFrequencyCovariance** supports this procedure with the `put_assets_into_blocks_by_trading_frequency` and `blockwise_estimation` functions. The problem is that this procedure by necessity will use a different dataset for different covariances and as a result regularization may be required to get a PSD estimate of the covariance matrix.

4.2. Preaveraging of returns

The third estimation method we implement is the preaveraging method by Christensen, Podolskij, and Vetter (2013). These authors argue that refresh time sampling throws away data in the cases when some assets trade multiple times within a refresh time interval. While blocking is one answer to this, the suggestion of Christensen *et al.* (2013) is that more use of the data can be made if multiple returns can be aggregated together. They first assemble preaveraged returns for each asset by averaging together different tick-to-tick returns within a sliding time window. The covariance between two assets k, k' can then be estimated using the preaveraged returns. Specifically, we calculate the integrated covariance between two assets by multiplying together returns from each asset that are in a time window that overlaps. Then we add up the products of all such return pairs that share a window before we rescale this sum for the size of the time window and the weighting function used in preaveraging.

This is implemented in the `preaveraged_covariance` method. It has the advantage of reducing the impact of microstructure noise through averaging. It also allows the use of all data points and can consistently estimate the covariance matrix despite the Epps effect. A disadvantage of this method is that the resultant covariance matrix may not be PSD and thus regularization would be required. In small samples, it is also possible to estimate covariance matrices that have negative variances. In addition, the use of preaveraging means that volatility estimates from preaveraged returns are downwards biased. As a result of this last point, the **HighFrequencyCovariance** implementation uses the preaveraging technique's implied correlation matrix but reports volatilities yielded from the two scales volatility method.

4.3. Spectral local method of moments

The fourth estimation method we implement is the spectral local method of moments technique (Bibinger, Hautsch, Malec, and Reiss 2014, 2019). This algorithm starts by breaking the trading period into equally sized subintervals. Given each subinterval, we compute a spectral statistic matrix by using a weighted summation of the returns within that interval. We calculate these weights by means of an orthogonal sine function with some spectral frequency j. Then we gather many different spectral statistic matrices by doing this repeatedly with various spectral frequencies. Our estimate of the covariance matrix is then calculated as the average[2] of the spectral statistic matrices plus an adjustment for microstructure noise.[3] If a covariance matrix is desired calculated from the entire day (rather than a subinterval), this can be done by averaging over the `CovarianceMatrix` estimates for each subinterval.[4] We implement this in the `spectral_covariance` method. It has a similar advantage as the preaveraging method in that it uses all observations. However, the resultant matrix is not guaranteed to be PSD.

4.4. Two scales covariance

The final method of this package infers correlation between two assets based on their linear combination. Given two time series $\mathbf{X}^{\mathbf{k}}$ and $\mathbf{X}^{\mathbf{k}'}$ we have the identity:

$$\mathsf{COV}(\mathbf{X}^{\mathbf{k}}, \mathbf{X}^{\mathbf{k}'}) = \frac{1}{4\gamma(1-\gamma)} \left(\mathsf{VAR}(\gamma \mathbf{X}^{\mathbf{k}} + (1-\gamma)\mathbf{X}^{\mathbf{k}'}) - \mathsf{VAR}(\gamma \mathbf{X}^{\mathbf{k}} - (1-\gamma)\mathbf{X}^{\mathbf{k}'}) \right) \quad (3)$$

for some $0 < \gamma < 1$. This can be converted to a correlation with the equation:

$$\mathrm{Corr}(\mathbf{X}^{\mathbf{k}}, \mathbf{X}^{\mathbf{k}'}) = \frac{\mathsf{COV}(\mathbf{X}^{\mathbf{k}}, \mathbf{X}^{\mathbf{k}'})}{\sqrt{\mathsf{VAR}(\mathbf{X}^{\mathbf{k}})}\sqrt{\mathsf{VAR}(\mathbf{X}^{\mathbf{k}'})}}$$

This is the approach that Aït-Sahalia, Fan, and Xiu (2010) take. Rather than directly estimating covariances, they generated different combinations of the time series for different assets and use that to infer the covariance. This package implements a similar method, `two_scales_covariance`, to estimate the correlation matrix. The key difference between this package's implementation and Aït-Sahalia *et al.*'s (2010) method is that they calculate variances using a maximum likelihood technique while this paper uses the two scales volatility technique. We made this change to avoid the complications of iterative optimization. Aït-Sahalia *et al.* (2010) also suggest that rather than using refresh time sampling of the respective time series, instead the refresh times should be calculated and then using an arbitrary (rather than the final) tick between subsequent refresh times. This is followed in the **HighFrequencyCovariance** implementation.

[2] While the paper finds that equal weighting will consistently estimate the covariance matrix, it recommends on efficiency grounds to optimally weight them according to the Fisher information. This is not implemented in this package as it requires matrix inversions which have a substantial computational cost and present stability challenges.

[3] A `Dict` describing the microstructure noises can be input to the algorithm if it is known. If this is not provided the two scales volatility estimates of microstructure noise will be used.

[4] This is one advantage of storing correlation matrices separately from volatilities. If integrated covariance matrices were stored directly then in order to average two together the durations over which each was estimated would need to be accounted for. This complication does not arise with correlations and volatilities stored separately as long as the time units of the volatility are accounted for.

5. Regularization

Many of the covariance estimators in this package do not come with a guarantee that the result will be PSD. Thus, we employ regularization techniques in the event an estimated correlation matrix is not PSD.

One of the simplest methods for regularising a covariance matrix that is to linearly interpolate it with the identity matrix. This is the approach of Ledoit and Wolf (2001) who provide expressions for mixing between an estimated matrix and the identity matrix. These expressions give an optimal solution in the sense of minimising the expected distance (in the squared Frobenius norm) between the estimated matrix and the true covariance matrix. This is implemented in the `identity_regularisation` method.

Eigenvalue filtering is another approach that has been considered in the literature (Laloux, Cizeau, Bouchaud, and Potters 1999; Tola, Lillo, Gallegati, and Mantegna 2008). This is implemented by first taking an eigenvector decomposition of the sample covariance matrix. We then calculate the Marchenko-Pastur distribution of eigenvalues we would expect given a random matrix of equal dimensions to the estimated matrix. Given this distribution, we choose a cutoff along the same lines as suggested by Hautsch, Kyj, and Malec (2015). We do not change any eigenvalues of greater size than this cutoff. For all eigenvalues below this cutoff, we average the positive eigenvalues and replace all of the eigenvectors below this cutoff with this average. This process reduces the impact of the corresponding eigenvectors while averaging them (rather than setting them to zero) means that we still get a full rank correlation matrix when we reassemble the eigenvalues and eigenvectors. This is implemented in the `eigenvalue_clean` method.

One solution for regularising correlation matrices is to project them to the nearest (by some metric) valid correlation matrix in the space of valid correlation matrices. This is the approach of Higham (2002) who suggests an iterative algorithm where we map a matrix to the nearest PSD matrix. We then map that matrix to the nearest unit diagonal matrix with off-diagonal entries less than one (in absolute value). We then map the result to the nearest PSD matrix and so on. Eventually, we converge to the nearest valid PSD correlation matrix. This is implemented in the `nearest_correlation_matrix` method. The function `nearest_psd_matrix` is also implemented which just maps a `Hermitian` matrix to the nearest PSD matrix (without then mapping to the space of unit diagonal correlation matrices). This can be used for covariance matrices.

HighFrequencyCovariance allows regularization to be applied in two places. The first is that for every covariance estimation method a regularization method can be specified which will be applied to map an estimated covariance matrix to a regularized one. The second is that regularization can be applied to the `CovarianceMatrix` structure in which case it can optionally be applied to the correlation matrix or the covariance matrix.[5] A user might want to do this to reduce noise. For this objective, the first two regularization algorithms are well suited. One justification for mixing with the identity matrix is that this shrinks a correlation matrix towards a Bayesian prior of no correlation (Ledoit and Wolf 2001). Eigenvalue filtering is also a widely used technique for reducing noise by effectively deleting the impact of the least informative eigenvectors in a matrix.

[5] In this case the covariance matrix is constructed from the correlations and volatilities, regularized, then the regularized covariance matrix is split into correlations and volatilities and placed in a `CovarianceMatrix` structure.

6. HighFrequencyCovariance design

There are two main structures (also referred to as structs) that are used in the package. The first is the `SortedDataFrame` which wraps a `DataFrame` containing the price update ticks for a collection of assets. The purpose of a `SortedDataFrame` is to increase the speed of subsequent calculations by pre-sorting the ticks and segmenting them by asset. In an applied setting with real data, we would have a `DataFrame` with numeric columns for time and price and a symbol column to identify the asset. This can be converted to a `SortedDataFrame` using the constructor with the names of the columns containing the time, asset and price information being input to the constructor along with the `DataFrame` itself and a `Dates.Period` object which describes the temporal units of the numeric time column.[6] This struct is immutable.

The second main struct is `CovarianceMatrix` which contains four members. The first is a `Hermitian` correlation matrix. The second is a vector of volatilities and the third is a vector of symbols that label the correlation matrix and volatility vector.[7] The final member is a `Dates.Period` object which details the length of time that the volatilities correspond to. The covariance estimation functions of the package all return results in the form of a `CovarianceMatrix` struct.

The design philosophy of this package prioritizes obtaining estimates easily and without requiring users to have a deep understanding of the underlying algorithms. In most cases, the default parameters will provide good covariance estimates without the need for fine tuning. Advanced users might want to fine tune parameters to improve their estimates. This is possible as most parameters detailed in the papers proposing each algorithm can be customized in **HighFrequencyCovariance**.

For each estimation technique, two user interfaces are provided. The first is provided by the `estimate_volatility`, `estimate_microstructure_noise`, `estimate_covariance` and `regularise` functions. These allow for estimation and regularization using one function with the method being specified by a symbol argument of these functions. Alternatively and equivalently, each method can also be called directly using a different function for each method. These method specific functions have the same name as the method and include `two_scales_volatility` and `spectral_covariance` among others.

7. Using HighFrequencyCovariance

HighFrequencyCovariance is a registered package and can be installed using Julia standard package manager **Pkg.jl** (Julia Programming Language 2022):

```
julia> using Pkg
julia> Pkg.add("HighFrequencyCovariance")
```

Before introducing the core estimation functions, it is worthwhile to discuss an inbuilt function that facilitates Monte Carlo generation of time series data. The package has a function,

[6]For instance, a period of one hour indicates that one unit in the time column corresponds to one hour.

[7]While the microstructure noise variance can also be estimated using the `two_scale_volatility` method this is intentionally not included in the `CovarianceMatrix` struct as it is an artifact of the data rather than of the assets. For instance, we might expect that the level of noise would decrease if we moved from using trade data to using quote data, but this will not affect the correlations or the volatilities of the true underlying price processes.

`generate_random_path` which generates a Monte Carlo timeseries of prices for a chosen number of assets. The assets have random volatilities (from a uniform distribution) and a random correlation matrix (from an inverse Wishart distribution).[8] Each asset has a different rate of trading which is encapsulated by an exponential distribution (with a random rate as drawn from a uniform distribution) modeling the waiting time until the subsequent tick. There is also random microstructure noise for each asset (each noise term is drawn from a normal distribution with standard deviation drawn, for each asset, from a uniform distribution).[9]

We now use this function to make some example data with four assets to show the capabilities and user interface of the package.[10]

```
julia> using DataFrames, Dates, LinearAlgebra, StableRNGs
julia> using HighFrequencyCovariance
julia> dims = 4
julia> ticks = 100000
julia> rng = StableRNG(100)
julia> time_period_per_unit  = Second(1)
julia> ts_data, true_covar, true_micro_noise, true_update_rates =
         generate_random_path(dims, ticks; rng = rng,
           time_period_per_unit  = time_period_per_unit)
```

where `ts_data` is a `SortedDataFrame` struct, `true_covar` is a `CovarianceMatrix` struct, `true_micro_noise` and `true_update_rates` are `Dicts` containing microstructure noise variances and the update rates. This time series data can be visualized by using the `show` method of `SortedDataFrame`:

```
julia> show(ts_data)

SortedDataFrame with 100000 rows.
    Row  Time         Name     Value
         Float64      Symbol   Float64

      1      1.06098  asset_1   0.000605886
      2      1.89955  asset_4  -0.000531824
      3      2.43674  asset_4  -5.28007e-5

  99998  51667.3      asset_4  -0.000475006
  99999  51668.5      asset_4   0.000206461
 100000  51668.7      asset_4   8.27579e-5
```

We start the analysis by estimating the volatility of each asset with both methods implemented by the package.

[8] After drawing a PSD matrix from the Wishart distribution, we do a similar procedure as in Equation 1 to get a correlation matrix.

[9] The function default distributions of volatilities, microstructure noise variances and update rates have been set to roughly resemble large cap stocks with annualized volatility between 10% and 50%. There are new ticks for each asset every 0.5-5 seconds (in expectation). Microstructure noise variance is by default set to roughly equal to the variance of 5 minutes returns given the input volatility distribution.

[10] Note that while we use a `StableRNG` for reproducibility reasons, `MersenneTwister` can also be used from the **Random** module.

```
julia> assets     = get_assets(ts_data)
julia> simple     = estimate_volatility(ts_data, assets, :simple_volatility)
julia> two_scales = estimate_volatility(ts_data, assets,
        :two_scales_volatility)
```

The level of microstructure noise can also be estimated for each asset. Note that this microstructure noise is a variance per tick and not a volatility per unit time.

```
julia> micro_noise = estimate_microstructure_noise(ts_data, assets)
```

Now we can print a table showing each of these estimated values together with the true values for both volatility and microstructure noise.

```
julia> println(vcat(
        DataFrame(sort(simple)...,(:estimation=> "Simple Method")),
        DataFrame(sort(two_scales)..., (:estimation=> "Two Scales")),
        DataFrame(Dict(true_covar.labels .=> true_covar.volatility)...,
                                (:estimation => "True Values"))))

3x5 DataFrame
 Row  asset_1      asset_2      asset_3      asset_4      estimation
      Float64      Float64      Float64      Float64      String

   1  0.000282689  0.000256263  0.000262538  0.000105329  Simple Method
   2  0.000188386  0.000127892  0.00014807   0.000143707  Two Scales
   3  9.23524e-5   7.61032e-5   9.63553e-5   5.08742e-5   True Values
```

The two scales method here outperforms the simple method overall. The simple method is particularly inaccurate for assets 1,2 and 3. A reason for this might be that this asset's price is measured with higher levels of microstructure noise relative to others. In this Monte Carlo setting we can investigate for this:

```
julia> println(vcat(DataFrame(sort(micro_noise)...,
        (:estimation => "Est. Microstructure Noise")),
        DataFrame(true_micro_noise...,
        (:estimation => "True Microstructure Noise"))))

2x5 DataFrame
 Row  asset_1    asset_2    asset_3    asset_4    estimation
      Float64    Float64    Float64    Float64    String

   1  5.3443e-6  3.3526e-6  3.9558e-6  1.2152e-6  Est. Microstructure Noise
   2  5.2941e-6  3.3103e-6  3.9466e-6  1.2148e-6  True Microstructure Noise
```

It can be seen here that we have a large degree of accuracy in estimating microstructure noise. We can also see that microstructure noise is high for assets 1, 2 and 3. The low level of microstructure noise in asset 4 is likely why the simple method was effective in this case.

Now we can also estimate covariance matrices. We will do this with several of the methods of the package. In the below function calls we set the regularization option to missing, which means regularization will not be performed. A regularization method can alternatively be specified here, in which case regularization is applied after estimation.

```
julia> simple = estimate_covariance(ts_data, assets, :simple_covariance;
         regularisation = missing)
julia> preav = estimate_covariance(ts_data, assets, :preaveraged_covariance;
         regularisation = missing)
julia> two_scales = estimate_covariance(ts_data, assets,
         :two_scales_covariance;
         regularisation = missing)
julia> bnhls = estimate_covariance(ts_data, assets, :bnhls_covariance;
         regularisation = missing)
```

We can examine one of the covariance matrices. This can be done using the `show` function (where the second/third arguments give how many significant figures/decimal places to print for volatility/correlation respectively).

```
julia> show(two_scales, 4, 4)

Volatilities per time interval of 1 second
  :asset_1   :asset_2   :asset_3   :asset_4
 0.0001884  0.0001279  0.0001481  0.0001437

Correlations
 :___         :asset_1    :asset_2    :asset_3    :asset_4
 :asset_1   1.0         -0.1178      0.0277      0.0613
 :asset_2  -0.1178       1.0        -0.2271     -0.0582
 :asset_3   0.0277      -0.2271      1.0        -0.1617
 :asset_4   0.0613      -0.0582     -0.1617      1.0
```

We may be concerned that one of the estimates, for instance the `bnhls` estimate, is not PSD. We can test this:

```
julia> valid_correlation_matrix(bnhls)

false
```

Unfortunately, this `bnhls` estimate is not PSD. However, we can regularize it using the nearest correlation matrix method of the `regularise` function:

```
julia> regularised_bnhls = regularise(bnhls, ts_data,
         :nearest_correlation_matrix)
```

For the sake of exposition, we will proceed considering the user has a preference for the preaveraging and two scales estimation techniques and only wants to use these two estimates. The user would like to take an elementwise combination of both estimates which is easy to achieve by using the `combine_covariance_matrices` function.

```
julia> matrices = [preav, two_scales]
julia> weights  = [1,1]
julia> combined = combine_covariance_matrices(matrices, weights)
```

This averaging is done elementwise given our input weights for each `CovarianceMatrix`.

We can compare how close each of the estimates is to the true correlation matrix. We do this by examining the mean absolute elementwise difference between the true and estimated correlations and volatilities.

```
julia> calculate_mean_abs_distance(true_covar, combined)

(Correlation_error = 0.17717743, Volatility_error = 7.309e-5)

julia> calculate_mean_abs_distance(true_covar, simple)

(Correlation_error = 0.44381242, Volatility_error = 0.00013228)

julia> calculate_mean_abs_distance(true_covar, preav)

(Correlation_error = 0.12252518, Volatility_error = 7.309e-5)

julia> calculate_mean_abs_distance(true_covar, two_scales)

(Correlation_error = 0.26168133, Volatility_error = 7.309e-5)
```

In this case, the correlation matrix calculated with preaveraging performed the best, and the two scales covariance method performed second best. The simple method was particularly inaccurate, which is likely a result of the need to throw away many ticks in order to avert the high frequency bias issues related in this paper's introduction.[11]

Before moving on to an example with real data, we will demonstrate how to use one of our estimated `CovarianceMatrix`s to calculate an actual covariance matrix over some interval. For an 8 hour interval we do this with the following code:

```
julia> covariance_interval = Hour(8)
julia> covar = covariance(combined, covariance_interval, combined.labels)
julia> covar

4x4 Hermitian{Float64, Matrix{Float64}}:
  0.00102209   -0.000125527   3.3856e-5     9.08782e-5
 -0.000125527   0.000471065  -0.000209126  -5.02195e-5
  3.3856e-5    -0.000209126   0.00063143   -0.000176454
  9.08782e-5   -5.02195e-5   -0.000176454   0.000594767
```

[11]Note the volatility error of the preaveraging and the two scales method are the same, as both use the two scales volatility method for volatilities.

Note that while the `CovarianceMatrix` contains volatilities that are appropriate for one second and while the returns in the tick data are over various durations, we can use the `covariance` method to estimate matrices over any duration. The labelling of this `Hermitian` covariance matrix is as per the `combined.labels` vector input to the `covariance` function.

8. Estimating covariance between FX rates

Now we consider the case of using the package to estimate covariances using real FX data. We will use publicly available FX tick data from Dukascopy (2021).[12] We use all quote ticks in the TickStory database for all FX pairs that include the United States Dollar (USD). Our data will be from 9:00 until 17:00 on 2021-06-21.[13] Once we have our data, we start by loading some packages and reading the data:

```
julia> using CSV, DataFrames, Dates, Distributions, KernelDensity, Statistics
julia> using HighFrequencyCovariance
julia> path = @__DIR__
julia> data = CSV.read(joinpath(path, "data.csv"), DataFrame)
```

As we loaded from a CSV, we do some conversions of data types before starting our analysis.

```
julia> data[!,:ticker] = Symbol.(data[:,:ticker])
julia> assets = sort(unique(data[:,:ticker]))
julia> dateformat = DateFormat("yyyy-mm-dd HH:MM:SS.sss")
julia> data[!,:stamp] = DateTime.(data[:,:stamp], Ref(dateformat))
julia> data[!,:Times] = Time.(data[:,:stamp])
julia> data[!,:Dates] = Date.(data[:,:stamp])
```

Before we start estimating a `CovarianceMatrix`, we first check that there are ticks throughout our entire interval of time. If some assets did not have ticks for some hours of the trading interval, then it might make sense to segment the time interval and perform separate estimations.

```
julia> data[!,:Hours] = parse.(Int, Dates.format.(data[:,:Times], "H"))
julia> data = transform(groupby(data, :ticker), nrow => :all_ticks)
julia> data = transform(groupby(data,[:ticker, :Hours]),nrow => :hour_ticks)
julia> plotdata = combine(groupby(data, [:ticker, :Hours, :all_ticks]),
         nrow => :hour_ticks)
julia> plotdata[!,:HourDensity] = plotdata[:,:hour_ticks] ./
         plotdata[:,:all_ticks]
julia> using Gadfly
```

[12] This is accessible through Tickstory (2021) which is a tick database that provides free access to the data.

[13] This data can be assembled by downloading a CSV file for each FX pair including the USD in the TickStory GUI. Once each asset is selected you can use the option to "export to file" and then select the Output Format "Generic tick format (comma delimited)". Once all CSVs are downloaded they can be combined (by vertical concatenation) and a `mid` column calculated as `0.5 * bid_price + 0.5 * ask_price` and a `ticker` column containing the name of each FX pair. The resulting `DataFrame` will be equivalent to the data contained in the file `data.csv` in this code block.

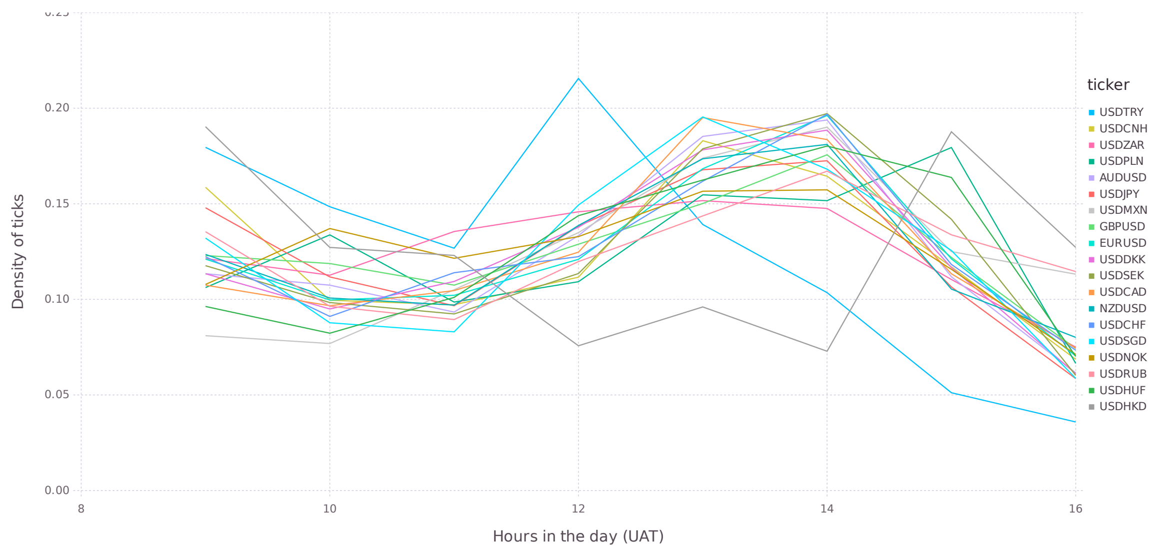

Figure 1: Density of ticks over the trading day.

```
julia> plt = Gadfly.plot(plotdata, x=:Hours, y=:HourDensity, color=:ticker,
         Geom.line, Guide.xlabel("Hours in the day (UAT)"),
         Guide.ylabel("Density of ticks"),
         style(key_position = :right))
```

The plot generated in the above code is replicated in Figure 1. We can see that there are ticks for all assets over the whole period. Before it is possible to input this `DataFrame` into a `SortedDataFrame` we must perform two tasks. The first task is to make a `Real` column to represent time. The second task is to create a `Dates.Period` object that defines what duration each unit in the numeric time column corresponds to. We will make a numeric time with a time interval of a second measured from the time of the earliest tick in our sample.[14]

```
julia> min_time = minimum(data[:,:stamp])
julia> data[!,:SecsFromBase] = map(x -> x.value,
         (data[:,:stamp] .- min_time))./1000
julia> timeperiod = Second(1)
```

Movements in FX rates (along with many financial assets) tend to be multiplicative rather than additive. As a result, we will take the log of the FX rate and calculate correlations and covariance with respect to log returns.

```
julia> data[!,:logmid] = log.(data[:,:mid])
```

Now we create a `SortedDataFrame` by putting the names of the variables we have created together with the time period into the `SortedDataFrame` constructor.

```
julia> ts = SortedDataFrame(data, :SecsFromBase, :ticker, :logmid,
         timeperiod)
```

[14]Note that subtracting `DateTimes` results in milliseconds so we divide by 1000 to get seconds.

Next, we will estimate covariance with the preaveraging method, the two scales method and the spectral method. Default regularization will be applied. This default regularization is the `nearest_psd_matrix` in the case of the preaveraging and spectral method (as regularization is done at the level of the covariance matrix), and the `nearest_correlation_matrix` in the case of the two scales method (as regularization is done directly on the correlation matrix).

```
julia> preav_estimate     = estimate_covariance(ts, assets,
                                :preaveraged_covariance;
                                regularisation = :default)
julia> twoscales_estimate = estimate_covariance(ts, assets,
                                :two_scales_covariance;
                                regularisation = :default)
julia> spectral_estimate  = estimate_covariance(ts, assets,
                                :spectral_covariance;
                                regularisation = :default)
```

Now as a starting point we may want to see if all three `CovarianceMatrix`s are PSD and generate similar predictions:

```
julia> valid_correlation_matrix(preav_estimate)

true

julia> valid_correlation_matrix(twoscales_estimate)

true

julia> valid_correlation_matrix(spectral_estimate)

true

julia> calculate_mean_abs_distance(preav_estimate, twoscales_estimate)

(Correlation_error = 0.11493703, Volatility_error = 0.0)

julia> calculate_mean_abs_distance(spectral_estimate, twoscales_estimate)

(Correlation_error = 0.06101978, Volatility_error = 6.1e-7)

julia> calculate_mean_abs_distance(preav_estimate, spectral_estimate)

(Correlation_error = 0.14429151, Volatility_error = 6.1e-7)
```

We can see that all three estimation techniques deliver reasonably similar correlation and volatility estimates which is reassuring. In addition, they are all PSD. Without any reliable method of determining which of these estimates is the most accurate, we will average over all three:

```
julia> covar = combine_covariance_matrices([preav_estimate,
         twoscales_estimate, spectral_estimate])
```

Now that we have an estimate, we will perform several sanity checks to ensure our estimated `CovarianceMatrix` is credible. As it is not possible to fit the entire `CovarianceMatrix` on the page we will use the `rearrange` function to subset the matrix and show only five pairs:

```
julia> show(rearrange(covar,[:USDDKK, :EURUSD, :USDHKD, :GBPUSD, :AUDUSD]),
          4, 4)

Volatilities per time interval of 1 second
  :USDDKK   :EURUSD   :USDHKD   :GBPUSD   :AUDUSD
 1.463e-5  1.454e-5  8.686e-7  1.842e-5  2.228e-5

Correlations
 :___        :USDDKK    :EURUSD    :USDHKD    :GBPUSD    :AUDUSD
 :USDDKK   1.0        -0.9936    -0.0043    -0.5787    -0.6728
 :EURUSD  -0.9936      1.0        0.0123     0.5824     0.6854
 :USDHKD  -0.0043      0.0123     1.0       -0.0833    -0.0404
 :GBPUSD  -0.5787      0.5824    -0.0833     1.0        0.6518
 :AUDUSD  -0.6728      0.6854    -0.0404     0.6518     1.0
```

It can be seen that the correlation between USDDKK and EURUSD is quite large in magnitude at -0.9936. While this is a valid correlation, in many settings a correlation like this may raise concerns. In this setting however, there is an economic justification for such a strongly negative correlation. The Danish Krone is pegged by the Danish central bank at a rate such that 746 DKK is worth 100 EUR (Danmarks Nationalbank 2022). This means that when the USDDKK increases (so that DKK is worth less in terms of USD) then the EURUSD should go down (so the EUR is also worth less in terms of USD). Therefore, the closeness of this correlation to -1.0 is encouraging for the accuracy of the correlation matrix. Another implication of this peg is that the volatility of the USDDKK should be close to the volatility of the EURUSD. We can see that this is the case, with the volatility of the USDDKK being slightly higher (which may reflect volatility of the Danish Krone around its peg). A further implication is that the pairwise correlation between EURUSD and another pair should be close to the negative of the correlation between USDDKK and that other pair. This can be tested:

```
julia> USDDKK_correlations = get_correlation.(Ref(covar), :USDDKK,
                              setdiff(assets, [:USDDKK, :EURUSD]))
julia> EURUSD_correlations = get_correlation.(Ref(covar), :EURUSD,
                              setdiff(assets, [:USDDKK, :EURUSD]))
julia> Statistics.cor(USDDKK_correlations, EURUSD_correlations)

-0.9998844432559166
```

This results in the expected strongly negative correlation between these two series of pairwise correlations.

One other notable peg is the peg of the Hong Kong Dollar to the United States Dollar. This implies that the volatility of the USDHKD will be small as the Hong Kong Monetary Authority intervenes to target a rate of 7.75 HKD per USD (Hong Kong Monetary Authority 2021). We can see that this is the case with the volatility of the USDHKD pair being around ten times lower than the next least volatile pair which is USDCNH. We can also see that the pairwise correlations between USDHKD and other assets are small. This makes sense if deviations in USDHKD are driven by market frictions and changes in exchange rate peg credibility. These forces are likely to be orthogonal to the macroeconomic factors driving changes in the other exchange rates.

While the preceding sanity checks have exploited information about the underlying assets' fundamental relationships, there are more general sanity checks we can also conduct. We may want to test how our covariance matrix performs out of sample. To do this, we take all half-hourly returns between 9:00 to 17:00 from 2021-06-22 to 2021-06-26, the days after our training date. These returns are from the first tick after 9:00, to the last tick before 9:30 for each asset and each day. For the next time interval, the returns are from the first tick after 9:30 to the last tick before 10:00 for each asset and day and so on. As the total duration is quite close to 30 minutes for each return, we will ignore any small incongruencies and take the same 30 minute duration for the analysis of all returns.

```
julia> future_returns= CSV.read(joinpath(path,"data_nextday.csv"),DataFrame)
julia> future_returns[!, :ticker] = Symbol.(future_returns[:, :ticker])
julia> println(first(future_returns, 4))
```

```
4x4 DataFrame
 Row  ticker  interval                    log_return  duration
      Symbol  String31                    Float64     String31

   1  AUDUSD  2021-06-22 09:00:00.000000    0.000227  0 days 00:29:59.504000
   2  AUDUSD  2021-06-22 09:30:00.000000   -0.000147  0 days 00:29:59.685000
   3  AUDUSD  2021-06-22 10:00:00.000000    0.000906  0 days 00:29:59.251000
   4  AUDUSD  2021-06-22 10:30:00.000000    0.000233  0 days 00:29:59.518000
```

Now that our data is loaded, we might first want to see how the volatility of the 30 minute returns corresponds to the volatilities implied by our covariance matrix. We also want to see if our estimated correlations correspond to realized correlations over subsequent days. We can do this by using the 30 minute returns to calculate the realized covariance matrix via the simple method with the function `simple_covariance_given_returns`. Then we can split it into volatilities and correlations as per Equation 1 and Equation 2 in the `cov_to_cor` function.

```
julia> future_rets_wide = unstack(future_returns, :interval, :ticker,
                                       :log_return)
julia> future_rets_mat = Float64.(Matrix(future_rets_wide[:,covar.labels]))
julia> future_rets_covar = simple_covariance_given_returns(future_rets_mat)
julia> correl, sds = cov_to_cor(future_rets_covar)
julia> vols = sds / sqrt(30*60)
```

For testing the volatilities we use:

```
julia> Statistics.cor(vols, covar.volatility)

0.8850266961604603

julia> Statistics.mean(vols ./ covar.volatility)

0.9695290869008333
```

Similarly, for testing correlations:

```
julia> Statistics.mean(abs.(correl .- covar.correlation))

0.13124848487418406
```

We can see that our estimated volatilities are highly correlated with the realized volatilities from subsequent days. The scale of our estimated volatilities also matches. Our correlations are also similar to the realized correlations from these subsequent days.

We can see how much of an improvement this method is compared to if we had estimated covariance with the simple covariance method.[15]

```
julia> simple_estimate = estimate_covariance(ts, assets, :simple_covariance;
         regularisation = :default)
julia> Statistics.cor(vols, simple_estimate.volatility)

0.8933722898678813

julia> Statistics.mean(vols ./ simple_estimate.volatility)

0.9457959505163285

julia> Statistics.mean(abs.(correl .- simple_estimate.correlation))

0.1517847273323817
```

We can see that the volatilities and correlations estimated with the high frequency techniques are more correlated with the realized volatilities and correlations for subsequent days. When it is considered that some of the gaps between training values and subsequent day realized values is likely to be due to intertemporal changes in these quantities (which is irreducible error), it appears that a substantial fraction of the reducible error is reduced by the use of high frequency techniques.

8.1. Financial applications

There are countless financial applications for covariance matrices. For a given portfolio we can use a `CovarianceMatrix` to estimate the variance of the portfolio's value. First, we make a struct to represent our portfolio:

[15] Note that the default covariance method uses a subsample of the ticks to avoid the Epps effect. Therefore, these results should be similar to the results of the common tactic of estimating covariance over 15 or 30 minute returns.

```
julia> struct Portfolio{R<:Real}
         weights::Vector{R}
         labels::Vector{Symbol}
       end
```

Now assume we have exposure to three futures contracts on `[:EURUSD, :GBPUSD, :AUDUSD]`[16] and our portfolio weights are `[1,1,1]`. We can now estimate the variance of our portfolio over an 8 hour trading day:

```
julia> function portfolio_variance(port::Portfolio, cov::CovarianceMatrix,
                                   duration::Dates.Period)
         cov2 = covariance(rearrange(cov, port.labels), duration)
         return transpose(port.weights) * cov2 * port.weights
       end
julia> our_portfolio = Portfolio([1,1,1], [:EURUSD, :GBPUSD, :AUDUSD])
julia> trading_day_duration = Hour(8)
julia> portfolio_variance(our_portfolio, preav_estimate, trading_day_duration)

6.923304662274563e-5
```

We may also be interested in determining the optimal portfolio to hedge one asset (for instance `:USDHUF`) given a portfolio of other assets (for instance `:GBPUSD`, `:AUDUSD`, `:USDJPY` and `:USDNOK`).

We can do this by using the conditional distribution of the multivariate Gaussian. This gives us the expected return of our traded asset after conditioning on the realized returns of other assets. As the expression for this expectation is a weighted sum of the returns of the other assets, we can take the negative of these weights to find a hedging portfolio.

```
"""
Given a CovarianceMatrix, an asset of interest and returns for assets to
condition on, this function estimates the (univariate) distribution of the
asset of interest after conditioning on the returns of the other assets.
"""
julia> function get_conditional_distribution(covar::CovarianceMatrix,
                           asset::Symbol,
                           conditioning_assets::Vector{Symbol},
                           conditioning_asset_returns::Vector{<:Real},
                           data_return_interval = covar.time_period_per_unit)
                 covariance_labels = covar.labels
                 covar_matrix = covariance(covar, data_return_interval)
                 asset_index =  findall(asset .== covariance_labels)
                 conditioning_indices = map(x -> findfirst(==(x), covariance_labels),
             conditioning_assets)
          # Segmenting the covariance matrix.
          sigma11 = covar_matrix[asset_index,asset_index]
```

[16]For simplicity we ignore any drift rates and any collateral posting.

```
        sigma12 = covar_matrix[asset_index,conditioning_indices]
        sigma21 = covar_matrix[conditioning_indices,asset_index]
        sigma22 = covar_matrix[conditioning_indices,conditioning_indices]
        mu1     = zeros(length(asset_index))
        mu2     = zeros(length(conditioning_indices))
        weights = sigma12 / sigma22
        conditional_mu = mu1 + weights * (conditioning_asset_returns - mu2)
        conditional_sigma = sigma11 -  weights * sigma21
        dist = Normal(conditional_mu[1], sqrt(conditional_sigma[1,1]))
        return dist, weights
    end
```

The code below calculates such a hedging portfolio with the result we should go long on `:GBPUSD`, `:AUDUSD` and short on `:USDJPY`, `:USDNOK` to the indicated weights.

```
julia> asset = :USDHUF
julia> other_assets = [:GBPUSD, :AUDUSD, :USDJPY, :USDNOK]
julia> covar_subsetted = rearrange(covar, vcat(asset, other_assets))
```

Now we only want the weights and they do not depend on the correlated asset returns we will feed in zeros for this argument.

```
julia> _, weights = get_conditional_distribution(covar_subsetted, asset,
          other_assets,
          zeros(length(other_assets)))
julia> hedging_portfolio = -weights

1x4 Matrix{Float64}:
 0.226729  0.200199  -0.117004  -0.197586
```

9. Accuracy

We assess the accuracy of each implemented covariance matrix by repeating the general Monte Carlo procedure from Section 7 and comparing the estimated correlation matrices and volatilities to their true values. We start by generating data exhibiting asynchronous price updates with microstructure noise. We do this two times, once for 4 assets and once for 16 assets. In each case and for each technique, the default estimation and regularization settings are applied.

We measure the accuracy of each algorithm by the mean absolute difference between the corresponding elements of the estimated and the true correlation matrices and volatilities. The results are presented in Figure 2 where each point gives the average accuracy from the 4 asset and the 16 asset Monte Carlo samples. The top panels show accuracy in estimating correlations while the bottom panels show accuracy in estimating volatilities. In each panel, the x axis shows how many ticks of data are used and the y axis shows the average (across 100 generated paths) mean absolute error for each estimated volatility and correlation.

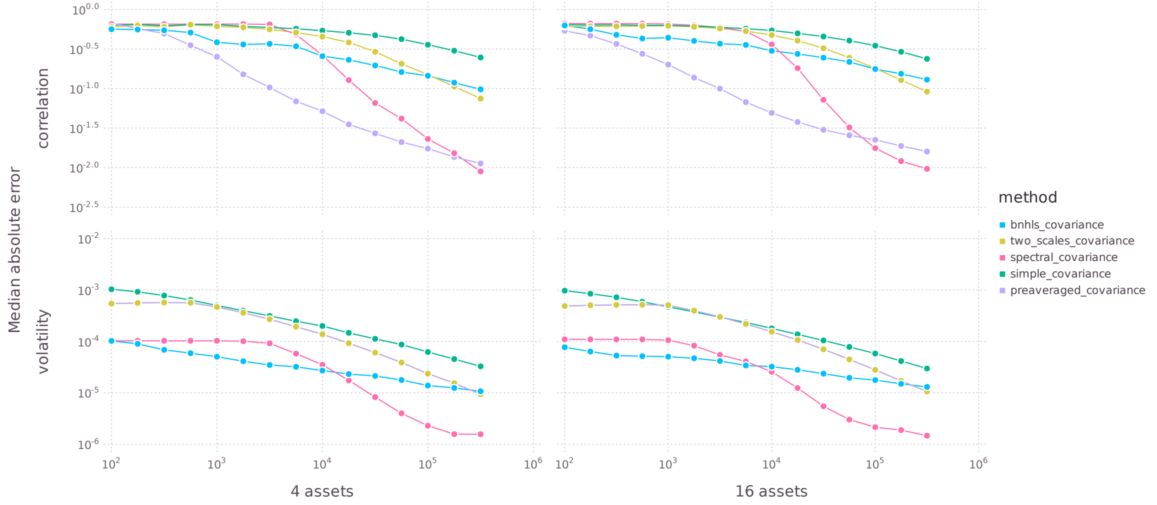

Figure 2: Accuracy of correlation and volatility estimates. In each panel, the x axis shows the average number of updates per assets and the y axis shows the average (across 100 generated paths) mean absolute error for each estimated volatility and correlation.

We can see that the `simple_covariance` method generally performs poorly. While it does improve with more data,[17] it is generally always outperformed by the other methods both in volatility and correlation estimation.

Looking at the more advanced methods, we can see that the number of ticks that we have is important in choosing a method and that no method is dominant in all situations. In correlation estimation, the best performing technique is the `preaveraged_covariance` method for a lower number of ticks, but this is overtaken by the `spectral_covariance` method for a higher number of ticks.[18]

When we consider accuracy in estimating volatilities, the most effective technique appears to be the `bnhls_covariance` technique for when there are few ticks; however, this is again overtaken by the `spectral_covariance` when there is a large amount of data.

10. Conclusion

HighFrequencyCovariance is the first openly available Julia package that provides algorithms to estimate covariance matrices using high frequency data. For many of these algorithms, it is the first open source package implementation in any language. In total, we implement two volatility estimators, five covariance estimators and four regularization techniques. The package has a simple interface that enables users to quickly generate reasonable covariance matrix estimates without a detailed knowledge of the package or the underlying algorithms.

[17] Note that the `simple_covariance` by default throws away many ticks so that it has appropriately long intervals. As a result, it does not succumb to the biases discussed in the introduction.

[18] Note that the `spectral_covariance` case failed to generate predictions in this case for 16 dimensions when it had less than 500 observations (on average) per asset.

References

Aït-Sahalia Y, Fan J, Xiu D (2010). "High-Frequency Covariance Estimates with Noisy and Asynchronous Financial Data." *Journal of the American Statistical Association*, **105**(492), 1504–1517. doi:10.1198/jasa.2010.tm10163.

Aït-Sahalia Y, Jacod J (2009). "Testing for Jumps in a Discretely Observed Process." *The Annals of Statistics*, **37**(1), 184–222. doi:10.1214/07-aos568.

Barndorff-Nielsen OE, Hansen PR, Lunde A, Shephard N (2008). "Designing Realized Kernels to Measure the Ex Post Variation of Equity Prices in the Presence of Noise." *Econometrica*, **76**(6), 1481–1536. doi:10.3982/ecta6495.

Barndorff-Nielsen OE, Hansen PR, Lunde A, Shephard N (2011). "Multivariate Realised Kernels: Consistent Positive Semi-Definite Estimators of the Covariation of Equity Prices with Noise and Non-Synchronous Trading." *Journal of Econometrics*, **162**(2), 149–169. doi:10.1016/j.jeconom.2010.07.009.

Baumann S, Klymak M (2021). ***HighFrequencyCovariance.jl***. Julia package version 0.3.2, URL https://github.com/s-baumann/HighFrequencyCovariance.jl.

Bezanson J, Edelman A, Karpinski S, Shah VB (2017). "Julia: A Fresh Approach to Numerical Computing." *SIAM Review*, **59**(1), 65–98. doi:10.1137/141000671.

Bibinger M, Hautsch N, Malec P, Reiss M (2014). "Estimating the Quadratic Covariation Matrix from Noisy Observations: Local Method of Moments and Efficiency." *The Annals of Statistics*, **42**(4), 1312–1346. doi:10.1214/14-aos1224.

Bibinger M, Hautsch N, Malec P, Reiss M (2019). "Estimating the Spot Covariation of Asset Prices – Statistical Theory and Empirical Evidence." *Journal of Business & Economic Statistics*, **37**(3), 419–435. doi:10.1080/07350015.2017.1356728.

Boudt K, Cornelissen J, Payseur S, Kleen O, Sjoerup E (2022). ***highfrequency**: Tools for Highfrequency Data Analysis*. R package version 0.9.4, URL https://CRAN.R-project.org/package=highfrequency.

Christensen K, Podolskij M, Vetter M (2013). "On Covariation Estimation for Multivariate Continuous Itô Semimartingales with Noise in Non-Synchronous Observation Schemes." *Journal of Multivariate Analysis*, **120**, 59–84. doi:10.1016/j.jmva.2013.05.002.

Danmarks Nationalbank (2022). "Monetary and Exchange-Rate Policy." URL https://www.nationalbanken.dk/en/monetarypolicy/implementation/Pages/default.aspx.

Dukascopy (2021). *Historical Data Feed.* URL https://www.dukascopy.com/swiss/english/cfd/range-of-markets/.

Epps TW (1979). "Comovements in Stock Prices in the Very Short Run." *Journal of the American Statistical Association*, **74**, 291–296. doi:10.1080/01621459.1979.10482508.

Hautsch N, Kyj LM, Malec P (2015). "Do High-Frequency Data Improve High-Dimensional Portfolio Allocations?" *Journal of Applied Econometrics*, **30**(2), 263–290. doi:10.1002/jae.2361.

Hautsch N, Kyj LM, Oomen RCA (2012). "A Blocking and Regularization Approach to High-Dimensional Realized Covariance Estimation." *Journal of Applied Econometrics*, **27**(4), 625–645. doi:10.1002/jae.1218.

Higham NJ (2002). "Computing the Nearest Correlation Matrix – A Problem From Finance." *IMA Journal of Numerical Analysis*, **22**(3), 329–343. doi:10.1093/imanum/22.3.329.

Hong Kong Monetary Authority (2021). "How Does the LERS Work?" URL https://www.hkma.gov.hk/eng/key-functions/money/linked-exchange-rate-system/how-does-the-lers-work/.

Julia Programming Language (2022). ***Pkg.jl***. Julia package version 1.7.1, URL https://github.com/JuliaLang/Pkg.jl.

Laloux L, Cizeau P, Bouchaud JP, Potters M (1999). "Noise Dressing of Financial Correlation Matrices." *Physical Review Letters*, **83**(7), 1467–1470. doi:10.1103/physrevlett.83.1467.

Ledoit O, Wolf M (2001). "A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices." *Journal of Multivariate Analysis*, **88**(2), 365–411. doi:10.1016/s0047-259x(03)00096-4.

Markowitz H (1952). "Portfolio Selection." *The Journal of Finance*, **7**(1), 77–91. doi:10.1111/j.1540-6261.1952.tb01525.x.

Tickstory (2021). *Tickstory: The Trader's Database*. Tick data provider, URL https://tickstory.com/.

Tola V, Lillo F, Gallegati M, Mantegna RN (2008). "Cluster Analysis for Portfolio Optimization." *Journal of Economic Dynamics and Control*, **32**(1), 235–258. doi:10.1016/j.jedc.2007.01.034.

Zhang L, Mykland PA, Aït-Sahalia Y (2005). "A Tale of Two Time Scales: Determining Integrated Volatility with Noisy High-Frequency Data." *Journal of the American Statistical Association*, **100**(472), 1394–1411. doi:10.1198/016214505000000169.

Affiliation:

Stuart Baumann
E-mail: Stuart@StuartBaumann.com

Margaryta Klymak
Somerville College
Department of Economics
University of Oxford
10 Manor Rd, Oxford OX1 3UQ, United Kingdom
E-mail: margaryta.klymak@economics.ox.ac.uk

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
August 2022, Volume 103, Issue 14 *Submitted:* 2021-10-16
doi:10.18637/jss.v103.i14 *Accepted:* 2022-06-21