Comput Econ
https://doi.org/10.1007/s10614-018-9813-x

Programming Language Choices for Algo Traders: The Case of Pairs Trading

Pedro Vergel Eleuterio[1] · **Lovjit Thukral**[2]

Accepted: 15 April 2018

Abstract In the last 20 years, relative value strategies have increased in popularity in various asset classes, including equity and commodity markets. Due to an increase in market participants, more sophisticated algorithms than those used in the past are now required to generate excess returns in pairs trading strategies. Sophisticated algorithms can cause an increase in complexity which, in-turn, increases computational run time. In our pairs trading example, C++ provides the best performance, however, it is also the most time consuming to implement. Among the languages that allow for faster development, Cython provides the best balance between run times and ease of prototyping.

Keywords Programming language · Performance · Pairs trading · Trading strategies

JEL Classification G10 · G11 · G12 · G14

Investing in asset pairs has increased in popularity in the last 20 years in various asset. Classes, including equities and commodities. Pairs trading is a strategy based on exploiting financial markets that are not in equilibrium, i.e. taking advantage from deviations in prices and expecting that, over time, rational equilibrium will be established. Conceptually, the strategy is fairly simple. It consists of taking a long position

✉ Pedro Vergel Eleuterio
pedro_vergel@hotmail.com

Lovjit Thukral
Lovjit.s.thukral@jpmorgan.com

[1] Birkbeck, University of London, Malet St, Bloomsbury, London WC1E 7HX, UK

[2] JP Morgan Asset Management, 25 Bank St, Canary Wharf, London E14 5JP, UK

Published online: 02 May 2018

(i.e. buying) in a security and, simultaneously, taking a short position (i.e. selling) in another security. The securities are selected with the expectation that the spread between the two will narrow. Econometric tools, such as co-integration, are used in the selection process. Investing in a pair of co-integrated stocks has the advantage of providing returns that are not dependent on the general direction of the market.

In 2010, Binh and Faff conclude that the simple pairs trading strategies studied extensively throughout the academic literature (see, for example, Goetzmann and Rouwenhorst 1998; Elliott et al. 2005; Erb and Harvey 2006; Gatev et al. 2006) have experienced a downward trend in profitability. This has given rise to a wide attempt by both academics and practitioners to update the strategies by using different tools to increase profitability. Among such literature, Miao (2014), uses a two-stage correlation and co-integration approach, Fallahpour et al. (2015), use a reinforcement learning method and Huang et al. (2015) benefit from the use of genetic algorithms. This race to increase the profitability of the pairs trading strategy has led to the increase of the computational power required.

For the prototyping of a pairs trading strategy, the choice of programming language has become more critical than ever. The goal is to have a programming language which has a simple syntax while also being computationally fast and robust in order to find alpha generating opportunities.

The history of academic publications on the comparisons of programming languages is large; however, there has only been one paper to date which looks at programming comparisons for investment strategies. Ceccon et al. (2016) show how different programming languages perform when looking at simple momentum based strategies. They find that C++ was very fast as expected, but Cython and Julia were interesting candidates due to their simplicity and performance. We extend this work by looking at a more complicated pairs trading strategy which requires more computational power due to the calculation of various different factors alongside the backtest itself.

Prechelt (2000) produces a comprehensive comparison of C, C++, Java, Perl and Python, measuring productivity as a function of properties such as run time, memory consumption, program structure, reliability and the amount of effort. The author shows that scripting languages such as Perl and Python are more productive than conventional languages such as C++, C and Java. Aruoba and Fernández-Villaverde (2014) compare programming languages applied to problems in economics, comparing the run-time for solving an economic model called "The Stochastic Neoclassical Growth Model" using C++, FORTRAN, Java, Julia, Python, Matlab, Mathematica and R.

This paper answers the following question: which programming languages are most appropriate for pairs trading strategies? The question has been motivated by our own practical applications where we have required our trading tools to be fast, yet easy to understand. We test the run time of our pair trading strategies on S&P 500 data using Python, Julia, Cython, and C++. We find Cython to be the strongest candidate due to its simplicity and high performance.

The remainder of the paper is as follows. We outline our empirical methodology in Sect. 1 and present results in Sect. 2. We conclude with Sect. 3.

1 Empirical Methods and Assumptions

In order to test each programming language, we implement our own proprietary trading algorithm. The algorithm is based on a number of econometric and statistical tests, which have been enhanced to provide the required level of risk adjusted returns in a live trading environment.

Each of the econometric and statistical tests is calculated for each pair of stocks in our sample. For the purpose of this paper, we calculate a total of five factors with various levels of complexity. Each factor is aimed to measure a characteristic of the relationship. Our factors include variants of correlation, co-integration, speed of mean reversion metrics, and a measure of dislocation of both time series. The most computationally demanding factors are those that require several steps, such as co-integration tests. Co-integration tests are meant to differentiate from short and long term variations.

Classic methodologies for testing co-integrated relationships are those of Engle and Granger (1987) and Johansen (1991). A prerequisite for testing co-integration is that both series are I(1) processes—they have unit roots and their first differences are stationary. If co-integration exists, both time series move together over a long period with fluctuations in the short term. Engle and Granger propose a two-step procedure. Firstly, a linear regression is estimated:

$$y_t = \alpha + \beta x_t + e_t. \tag{1}$$

In theory, due to the asymptotic properties of the Engle and Granger test, the choice of the dependent variable affects the coefficients but not the distribution of the test statistics (see Eq. 1). However, in practice, the procedure is repeated with each of the variables as the dependent variable (Geman and Vergel Eleuterio 2015). Secondly, the estimated residuals are calculated and tested for stationarity (see Eq. 2) using a Dickey Fuller test:

$$\hat{e}_t = y_t - \hat{\alpha} - \hat{\beta} x_t. \tag{2}$$

Additionally, a co-integrated relationship can be expressed as an Error Correction Model (see Eq. 3), which provides the magnitude of the correction in the short term, i.e. the speed at which the spread between both stocks narrows, θ. Using first differences, we write the model as:

$$\Delta y_t = \Delta x_t + \theta \hat{e}_{t-1} + u_t \tag{3}$$

If the coefficient θ is negative, it indicates a narrowing of the spread and therefore the existence of co-integration.

The dataset comprises daily quotes on the constituents of the S&P 500 from 2010 to 2015. Each implementation loads the prices from a SQLite database. This step is dependent on the database library used, hence it is not included when considering the performance of each language. The implementations in each language all follow the same structure. We start by loading the data in memory and computing each of the

factors separately. Additionally, we also compute all the factors together to obtain a global benchmark for the whole strategy.

Our main objective is to get the best performance out of our implementation reducing the development times required from a pure C++ implementation. In order to provide a meaningful overview of the speed of the languages, we run each factor multiple times and we collect a set of summary statistics related to run time. In order to provide a good benchmark, we avoid compilers optimizations, such as dead code elimination, hence we made sure to use the results of each computation.

We now describe implementation details that are specific to each language. These details should help the trader to understand the implications of writing high performance programs in each language. In order to keep the language comparison fair, we follow best practice as much as possible for each implementation.

The C++ implementation is the one that requires by far the most time to implement. This is due to the fact that, although the language allows the trader great flexibility, it also requires careful control over memory allocation and access. We test a number of different compiler flags to see which one obtained the best performance. Unlike the other languages in this comparison, the resulting C++ program does not require further work to obtain acceptable performance. We concur with Ceccon et al. (2016), who show that implementing the algorithms in C++ takes three times longer than in Python. Cython development times are just slightly higher than those of Python, while Julia is by far the fastest language to implement. These relationships can also be observed during code maintenance, although maintenance times can be greatly reduced by following best practices in commenting and documenting existing code.

In the Python implementation we use the NumPy library to handle vector operations. NumPy implements vectors operations in C code, hence, using NumPy as much as possible in the Python code is the best way to obtain good performance. Moreover, unlike the other languages considered in this paper, vectorizing the algorithms results in improved performance. Using NumPy functions, to the best of our knowledge and experience, over pure Python functions can increase speed by a factor of 100 to a 1000 times, depending on algorithm complexity.

With regards to Cython, programming is similar to writing code in pure Python. Once the functions are implemented in Cython, they are called from the main Python program. Calling the Cython modules from Python simulates the real world usage of Cython, i.e. computationally expensive functions are called from the main Python program.

In order to make our code more efficient in Python, we disable Python division, bounds check, and wrap-around indexing. Disabling these features in Python requires the trader to be more careful about accessing arrays and doing divisions, but the result involves less operations at run time. The interaction between C and Python code results in an additional cause degradation in performance but, fortunately, Cython provides a means of highlighting these interactions through annotations in the source code.

Julia is a programming language has a similar syntax to programs such as Matlab and R. However, non-vectorized code results in better performance in Julia compared with Matlab and R. Another advantage for the user is that Julia allows the trader to use for loops without penalizing performance. Writing for loops is a more familiar way to

Table 1 Run times

Factors:	Statistics:	Python	Cython	Julia	C++
All factors	Minimum	114.758	0.228	1.397	0.150
	Maximum	148.133	0.232	1.428	0.203
	Mean	139.677	0.230	1.416	0.164
	SD	1.21E+01	1.04E−03	1.21E−02	2.18E−02
Correlation	Minimum	0.008	0.009	0.019	0.048
	Maximum	0.009	0.011	0.023	0.050
	Mean	0.009	0.009	0.021	0.048
	SD	1.54E−04	5.19E−04	1.02E−03	6.57E−04
Co-integration	Minimum	107.577	0.132	0.422	0.078
	Maximum	158.212	0.133	0.426	0.082
	Mean	129.295	0.132	0.425	0.080
	SD	1.79E+01	5.53E−04	1.32E−03	1.31E−03
Speed of mean reversion 1	Minimum	4.029	0.053	0.526	0.062
	Maximum	4.269	0.054	0.597	0.064
	Mean	4.106	0.054	0.560	0.063
	SD	7.03E−02	2.51E−04	2.09E−02	5.84E−04
Speed of mean reversion 2	Minimum	3.297	0.025	0.407	0.052
	Maximum	3.690	0.025	0.423	0.053
	Mean	3.426	0.025	0.416	0.052
	SD	1.23E−01	8.71E−05	5.14E−03	4.52E−04
Dispersion	Minimum	1.794	0.009	0.021	0.040
	Maximum	1.873	0.011	0.027	0.042
	Mean	1.817	0.009	0.025	0.041
	SD	2.06E−02	5.25E−04	2.20E−03	5.95E−04

express operations on vectors for most users, such as those traders that generally use VBA, Matlab or R in their day to day activities.

Moreover, Julia provides excellent tools that allow for the writing of optimized versions of programs in a short time period. Among Julia's recommendations for optimizations are to disable bounds checking on a function-by-function basis and to avoid using abstract types, especially for containers to avoid the use of array pointers—using the @code_warntype macro to highlight these abstract types makes this easy to perform.

2 Results

Results for each factor in addition to results for the whole strategy are presented in Table 1, including descriptive statistics of the running times for each language.

Table 2 Average relative running time relative to C++

Factors:	Python	Cython	Julia	C++
All factors	852.462	1.406	8.644	1.00
Correlation	0.177	0.185	0.435	1.00
Co-integration	1625.469	1.664	5.338	1.00
Speed of mean reversion 1	65.525	0.858	8.933	1.00
Speed of mean reversion 2	65.266	0.474	7.924	1.00
Dispersion	44.395	0.219	0.607	1.00

This table contains minimum, maximum, mean and standard deviation of running times in seconds for each factor and the total of all factors for each programming language, i.e. Python, Cython, Julia and C++. Lower values indicate better performance.

Run time is one of the most important metrics for programming pair trading strategies. These strategies are usually built to work in several time frames and dislocations from equilibrium can be short lived, hence, in some cases speed of execution can make the difference between a profitable strategy and a losing one. Cython run times for Correlation are comparable to Python's, but it does much better in the remaining algorithms. This does not come as a surprise, since by using Cython we delegate most of the computing to the generated C library. The best performance in Python is achieved by delegating most of the work to NumPy, in this way all operations are executed by the library C code. Additionally, when the algorithms are not implemented only using NumPy, or in the case of the co-integration test when we use the external library StatsModels, performance degrades to unacceptable levels. Julia performs better than Python in each and all factors except in the Correlation factor with mean running times of 0.021 and 0.009 s respectively.

The standard deviation of running times in Python is low when using NumPy functions, while for pure Python functions the standard deviation is larger. Julia has a relatively high standard deviation due to the presence of a garbage collector, rather than by the just-in-time (JIT) compiler, since the first function call is not included in the benchmark. Cython and C++ both have a very low standard deviation in running times. A reason for this could be the existence of a more predictable memory model for both languages.

Table 2 shows the relative performance of each language relative to C++. Again, lower values indicate better performance. Previous results are highlighted, and clearly show that, performance-wise, Cython is the standout performer after C++.

3 Conclusion

In this paper we looked at four popular languages used by quantitative researchers and traders to program their models—C++, Python, Cython, and Julia—in terms of their performance and ease of obtaining programs that run in an acceptable amount of time.

In conclusion, C++ provides the best performance for prototyping quantitative trading strategies; however, it is the most time consuming to implement. Among the languages that allow for faster development times, Cython is clearly the one that provides the best balance between run time and ease of use in the case of pairs trading strategies. We find Julia to be an interesting programming language due to its sheer simplicity, and although implementing the code is relatively simple in Python, its run times with the standard statistic libraries are the slowest among the four programming languages.

References

Aruoba, S. B., & Fernández-Villaverde, J. (2014). *A comparison of programming languages in economics (No. w20263)*. London: National Bureau of Economic Research.

Bin, D., & Faff, R. (2010). Does simple pairs trading still work? *Financial Analysts Journal, 66*(4), 83–95.

Ceccon, F., Thukral, L., & Vergel Eleuterio, P. (2016). Momentum strategies: Comparison of programming language performance. *Journal of Trading, 11*(2), 49–53.

Elliott, R. J., Van Der Hoek, J., & Malcolm, W. P. (2005). Pairs trading. *Quantitative Finance, 5*(3), 271–276.

Engle, R., & Granger, C. (1987). Co-integration and error correction: Representation, estimation and testing. *Econometrica, 55*(2), 251–276.

Erb, C. B., & Harvey, C. R. (2006). The strategic and tactical value of commodity futures. *Financial Analysts Journal, 62*(2), 69–97.

Fallahpour, S., Hakimian, H., Taheri, K., & Ramezanifar, E. (2015). Pairs trading strategy optimization using the reinforcement learning method: A co-integration approach. Available at SSRN 2624328.

Gatev, E. G., Goetzmann, W. N., & Rouwenhorst, K. G. (2006). Pairs trading: Performance of a relative value arbitrage rule. *The Review of Financial Studies, 19*(3), 797–827.

Geman, H., & Vergel Eleuterio, P. (2015). Live Cattle as a new frontier in commodity markets. *Journal of Agriculture and Sustainability, 7*(1), 39–71.

Goetzmann, W., & Rouwenhorst, K. G. (1998). *Pairs trading: Performance of a relative value arbitrage rule (No. ysm3)*. New Haven: Yale School of Management.

Huang, C. F., Hsu, C. J., Chen, C. C., Chang, B. R., & Li, C. A. (2015). An intelligent model for Pairs trading using genetic algorithms. *Computational Intelligence and Neuroscience, 501,* 939606.

Johansen, S. (1991). Estimation and hypothesis testing of co-integration vectors in gaussian vector autoregressive models. *Econometrica, 59*(6), 1551–1580.

Miao, G. J. (2014). High frequency and dynamic pairs trading based on statistical arbitrage using a two-stage correlation and co-integration approach. *International Journal of Economics and Finance, 6*(3), p96.

Prechelt, L. (2000). An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl. *IEEE Computer, 33*(10), 23–29.