Formulaic Alphas Documented

HangukQuant[1, 2*]

July 27, 2023

[1] [2]

Abstract

This paper notes the design, use and interpretation for formulaic alphas. They also serve as documentation for the alpha reports published on hangukquant.com.

Contents

[1] ***1**: hangukquant@gmail.com, hangukquant.substack.com

[2] ***2**: DISCLAIMER: the contents of this work are not intended as investment, legal, tax or any other advice, and is for informational purposes only. It is illegal to make unauthorized copies, forward to an unauthorized user or to post this article electronically without express written consent by HangukQuant.

0.1 A Note of Precaution

There is absolutely no warranty or guarantee implied with this product. Use at your own risk. I provide no guarantee that it will be functional, destructive or constructive in any sense of the word. Trading is a risky operation.

1 Primitives

The formulaic alphas are built from a set of primitives $\mathcal{P}$. These primitives consist of terminals $\mathcal{T}$ and functions $\mathcal{F}$. The set $\mathcal{P} = \mathcal{T} \cup \mathcal{F}$. Terminals are generally quantitative or categorical exhibits of market data or constants. Functions are computed with terminals as inputs (if any). Some functions have arguments/parameters/children, but this is not compulsory. Terminals and functions are discussed in Section 1.1 and Section 1.2 respectively.

1.1 Terminals

Terminals are represented by the set $\mathcal{T}$, and contain variables we want to learn the distribution or forecasting power of. Typically, in formulaic alphas, these are the market variables and/or fundamental data. For the purposes of formulaic alphas, all (possibly degenerate) random variables represented by terminals are time-series processes. A valid terminal set would be

$$\{open, high, low, close, volume, pb\_ratio, mcap, executive\_compensation\} \cup \mathbb{R}$$

and so on. The terminal set need not only contain terminals of interpretive nature. They could be arbitrary features labelled $feature1, feature2$, so and so. When the terminal described is ambiguous or describes a set in the formulaic alpha representation, the terminal is appended with an identifier (we call this the *space*). For instance, the variable *close* is non-ambiguous. On the other hand, *const* points to the set of real numbers $\mathbb{R}$. When used in formulaic alpha representations, we need to append it with the identifier/space. We do this with the naming convention *terminal_space*. For instance, the variable *const_3* specifies the constant terminal $3 \in \mathbb{R}$.

1.2 Functions

[3] Functions $\mathcal{F}$ act upon the terminals. They can take any number of nonnegative inputs. A function is often parameterized by multiple arguments. For instance, the correlation function is ambiguous. When we write $correlation(open, close)$, there is ambiguity about the duration used in computing the sample statistic. A more complete parameterization would be of the form $correlation(open, close, t = 12)$ to specify the number of rolling data samples used in computing the correlation sample statistic. We want to separate between the time parameter and the input variables. For readability, the identifier/space (if any) is used with naming convention $function\_space(\cdots)$ to specify a non-ambiguous function f from the function family set F. For instance, a twelve-day correlation function between the volume transacted and closing prices may be encoded $cor\_12(volume, close)$. The function universe $\mathcal{F}$ is then a union of these function families, that is $\mathcal{F} = \cup_i F_i$, where F_i is some function family and $f \in F_i$ is a specific function (in the example $cor = F$, $f = cor\_12 \in cor$). The word 'space' is used to indicate that the different members of a function set F and their collection $\mathcal{F}$, together with the terminals $\mathcal{T}$ specify a constrained hypothesis space for formulaic alphas specifying rules for market-neutral return predictability. The hypothesis space is constrained via the semantic and syntactic validity of the formulaic alpha composed of the primitives $\mathcal{P}$. Their specific usage and examples are given Section 3 and Section 4 respectively. Function outputs may be multidimensional or univariate. For instance, the cross sectional rank function maps a vector of inputs to the rank of the elements in the vector. The time series rank function takes a time series process and outputs the rank of the most recent entry. Note that the *space* is not compulsory if the function set $F = \{f\}$ is a singleton - there is no ambiguity. Additionally, to specify unique function $f \in F$, the *space* does not necessarily have to be a number. $skew\_12$ specifies a twelve day skew. $neutralize\_industry$ specifies a function that subtracts from the input the industry mean; $neutralize\_sector$ specifies a function that subtracts from the input the sector mean. Here the function set $F = neutralize$ has members that can be identified by string constants 'sector', 'industry' and so on.

We can also specify compound functions. These are functions that are implicit representations

[3] HangukQuant would appreciate any errata and correction in the formulas. You may reach HangukQuant at hangukquant@gmail.com. Readers can also submit their own functions/compound functions by specifying the string formula or implementation, and HangukQuant will consider it for addition to the formulaic alphas.

of other functions in $\mathcal{F}$. See Section 4 for compound functions.

2 Operator Types

When combining together different market variables and arbitrary features, the operator type needs to be specified in relation to the desirable behavior of the formulaic alpha program. For instance, a function may be defined on time series variables or as a cross section of market variables at some time instance. Additionally, when features are arbitrary, the sampling frequency of the features w.r.t the measurement process needs to be made compatible for a legitimate operation. For instance 'free cash flow' (FCF) statistic may only be measured quarterly via financial statements, while the 'close' statistic is measured every trading day (or other specified granularities). If we specify a rule $plus(close, fcf)$, then how is the operation defined? If there are multiple operands and some missing data, how do we compute the output of a function? In this section, we specify general operator types to handle the ambiguity presented by these challenges. Each function $f \in \mathcal{F}$ belong to one of these classes of operators.

For some N-vector random variable $X = (X_1, \cdots, X_N)$ and T-sized estimation window, we may indicate some entry X_{it} as the $t \in [T]$ measurement for the i-th asset. The functions are categorized according to the indexing and aligning method of the variables of different sampling frequency.

Note that here, $t = 1, 2, \cdots, T$ is not a common time index. It depends on the function specification. For instance, when the sampling frequency is quarterly, it is possible that $T = 4$ refers to a span of one trading year, while it refers to just four trading days when the sampling frequency is daily. We make clear this distinction by the different operators types. As placeholder for arbitrary values, we use the letters $a \sim z$. For convenience, we assume that when a letter is lexicographically greater than another letter, we assume the value represented by the bigger letter is larger. That is, the value represented by letters $a < b < c \cdots < z$. When we want to represent the same value for different variables, we add other notations to the same letter ($b = b' = b^* = \tilde{b} = \bar{b}$ and so on). The entries '-' are representative of null/NaN/None (or some other equivalent datatype) values. Rank functions are in increasing order, such that $rank(a) < rank(b) < rank(c) < \cdots < rank(z)$. Rank counts start from one, such that $rank([a, b, c]) = [1, 2, 3]$. Additionally, tie-policy with rank functions is the averaging method. That is, $rank([a, a, a]) = [2, 2, 2]$, since 2 is the average of $1 + 2 + 3$. In this paper, for data structures presented for sequential measurements of random variables, the convention is *oldest* → *newest*. That is, given time series measurements in a list $[a, b, c]$, entry c is the most recent observable and a is the most dated observable of some random variable. '?' indicates entries that have insufficient information to compute.

Definition 1 (Union Index Operator)**.** *This computation is performed on each asset separately. The union index operator (UIO) is a function class that aligns instrument sampling frequency by the union of their timestamps. The function is then applied to this unified index, where the operands are applied on a last known-to-date basis. To make clear this example, suppose for over some ten day period, we have the following entries for random variables X_1, X_2:*

```
X_1 = [a - - b - - c - - d ]
X_2 = [e - - - - f g - - - ]
```

,then the unified last-known-to date index assumes the variables take values

```
X_1 = [a - - b - b c - - d ]
X_2 = [e - - e - f g - - g ]
```

. An operator that takes on the UIO class in our formulaic alphas is the plus operator. When we employ plus(X_1, X_2), then the resulting time series process is

```
X_3 = [a+e - - b+e - b+f c+g - - d+g ]
```

The union index operator helps to align two (or more) different time series variables when their sampling frequencies are different and in scenarios where we would like to compute the alpha value on an anytime basis using the most recent data available for each of the operands.

Definition 2 (Slow Index Operator)**.** *This computation is performed on each asset separately. The slow index operator (SIO) is a function class that aligns the instrument sampling frequency by the* *slowest sampling variable. The function is then applied to this slower index, where the operands are applied on a last known-to-date basis. To make clear this example, suppose for over some ten day period, we have the following entries for random variables X_1, X_2:*

```
X_1 = [a b c - - - - d e f ]
X_2 = [g - - h - - i - - - ]
```

. In X_1, the random variable is sampled more frequently (six times) while X_2 is a slower sampling random variable (three times). The alignment matches the data to index of X_2, such that the function is operated on a new index

```
X_1 = [a - - c - - c - - - ]
X_2 = [g - - h - - i - - - ]
```

. An operator that takes on the SIO class in our formulaic alphas is the cor (correlation) operator. This class of operators is helpful when we want to update the formulaic alpha output estimate only when the slowly moving variable (which usually contains more information from an information theoretic stance) is newly sampled. This is also helpful when the degeneracy of a random variable in a sample causes problems in the evaluation of a function output. See that the correlation statistic of a degenerate random variable (constant value) is undefined, since it has zero standard deviation. This alignment helps to make operations such as $cor\_12(open, analyst\_target)$ *a valid operation.*

Definition 3 (Self Index Operator)**.** *This computation is performed on each asset separately. The self index operator (SEIO) is a function class that aligns an instrument index by its own sampling frequency. The function is applied on a univariate time series. To make clear this example, suppose for over some ten day period, we have the following entries for random variable X_1:*

```
X_1 = [a - b - c d - e f - ]
```

.It is not assumed that the data are forward filled. Instead, the function computation is done w.r.t only the available data. An example of a SEIO function is the sum operator, which adds all values

in the defined window. Suppose the sum operator is defined for window size two, then when this is applied to X_1*, we would obtain*

```
X_2 = [? - a+b - b+c c+d - d+e e+f - ]
```

. This class of operators is helpful when the relevant information pertaining to the time series is its evolution over time. It can also be helpful in specifying path-dependent functions, such as tsrank (time-series rank). Additionally, when the measurement is not over fixed time periods but with irregular sampling frequencies (such as the bid-ask quotes), this indexation allows us to extract time-agnostic changes in a random variable over sequential measurements. See that this alignment helps to create formulaic alphas that encode feature information over variable time periods. For instance, the function $plus(tsrank\_4(close), tsrank\_4(quarterly\_earnings))$ *would contain information over four trading days in the left operand and over a financial year in the right operand.*

Definition 4 (Self Index Operator 2)**.** *This computation is performed on each asset separately. The self index operator 2 (SEIO2) is a function class that aligns an instrument index by its own sampling frequency, but instead of taking in a vector input, it only takes as input a single measurement. There is no change or alignment to the instrument index. Some instances are the log or sign function.*

Definition 5 (All Index Operator)**.** *This computation is performed on a cross-sectional measurement of the asset universe w.r.t to some random variable at specific time* $t \in [T]$*. It takes as input a vector of size matching the asset universe, and outputs a vector of the same size. The all index operator (AIO) is a function class that aligns all instrument indices by a common sampling frequency* $1, 2, \cdots T$*. The function is then applied to this unified index, where the operands are applied on a last known-to-date basis. To make clear this example, suppose for over some ten day period, we have the following entries for random variable* X_1, X_2, X_3*:*

```
X_1 = [a - b - c d - e f - ]
X_2 = [g - h i j - - - - - ]
X_3 = [k - l - - - - - - - ]
```

. The last known-to-date basis of the random variables are

```
X_1 = [a a b b c d d e f f ]
X_2 = [g g h i j j j j j j ]
X_3 = [k k l l l l l l l l ]
```

. Let these be stacked in a matrix, such that we have

```
X = [
    [a,g,k], [a,g,k], [b,h,l], [b,i,l], [c,j,l],
    [d,j,l], [d,j,l], [e,j,l], [f,j,l], [f,j,l]
]
```

. Then the function is applied cross-sectionally, such that we have

```
X’= [
    f([a,g,k]), f([a,g,k]), f([b,h,l]), ...
    ...                                f([f,j,l]), f([f,j,l])
]
```

This class of operators is helpful when the relevant information is in the comparison of the statistic in relation to other measurements across the asset universe. An example is the csrank (cross-sectional rank) function.

3 Interpretations and Usage

Here we discuss the interpretations for the functions used in the formulaic alphas. The operator types UIO, SIO, SEIO, SEIO2 and AIO are defined as in Definitions 1, 2, 3, 4 and 5 respectively. The modules *np,ss* refer to *numpy* and *scipy* modules of the Python library respectively. No dynamic type casting is assumed.

```
-------------------------------------------------------------------
```

Define the following utility functions:

```
ctrmoment=lambda x,k: np.mean((x - np.mean(x))**k)
stdmoment=lambda x,k: ctrmoment(x,k)/ctrmoment(x,2)**(k/2)
```

1. abs(x): the absolute value of x.

   TYPE: SEIO2

   CODE:

   ```
   np.abs(x)
   ```

2. neg(x): the negation of x.

   TYPE: SEIO2

   CODE:

   ```
   -1*x
   ```

3. log(x): the natural log of x.

   TYPE: SEIO2

   CODE:

   ```
   np.log(x)
   ```

4. sign(x): 1 if $x > 0$ else -1 if $x < 0$ else 0.

   TYPE: SEIO2

   CODE:

   ```
   np.sign(x)
   ```

5. recpcal(x): $\frac{1}{x}$

   TYPE: SEIO2

   CODE:

   ```
   1/x
   ```

6. pow_a(x): x^a

   TYPE: SEIO2

   CODE:

```
np.power(x,a)
```

7. csrank(x): cross sectional rank of vector x.

   TYPE: AIO

   CODE:

   ```
   scipy.stats.rankdata(x, method="average", nan_policy="omit")
   ```

8. cszscre(x): cross sectional z-score of vector x.

   TYPE: AIO

   CODE:

   ```
   (x-np.mean(x))/np.std(x)
   ```

9. delta_a(x): change in variable x over the last a days.

   TYPE: SEIO

   CODE:

   ```
   x[-1]-x[-(a+1)]
   ```

   NOTE: Recall that the data structure x has inputs placed in chronological order. If we want to observe the $a =$1-day delta, then we would take the difference between the last entry (most recent) and the penultimate entry (second most recent) value in x.

10. delay_a(x): lag a time series variable x by a measurements.

    TYPE: SEIO

    CODE:

    ```
    x[-(a+1)]
    ```

    NOTE: Recall that the data structure x has inputs placed in chronological order. If we want to take a $a =$1-day lagged data, then we would take the penultimate entry (second most recent) value $x[-(1+1)]$.

11. sum_a(x): sum measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    np.sum(x)
```

12. prod_a(x): multiply measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    np.prod(x)
```

13. mean_a(x): take the mean of measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    np.mean(x)
```

14. ewma_a(x): take the exponentially-weighted mean of measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    ewma=x[0]
    for i in x[1:]:
        ewma=0.36*i+0.64*ma

```

15. median_a(x): take the median of measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    np.median(x)
```

16. std_a(x): take the standard deviation of measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    ctrmoment(x,2)**(0.5)
```

17. var_a(x): take the variance of measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    ctrmoment(x,2)
```

18. skew_a(x): take the skew of measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    stdmoment(x,3)
```

19. kurt_a(x): take the (excess) kurtosis of measurements in x over a measurements.

    TYPE: SEIO

    CODE:

```
    stdmoment(x,4)-3
```

20. tsrank_a(x): take the time-series rank of the most recent entry in x over a measurements.

    TYPE: SEIO

    CODE:

```
    scipy.stats.rankdata(x, method="average", nan_policy="omit")[-1]
```

21. tsmax_a(x): take the time-series maximum of the most recent a measurements.

    TYPE: SEIO

    CODE:

```
    np.max(x)
```

22. tsmin_a(x): take the time-series minimum of the most recent a measurements.

    TYPE: SEIO

    CODE:

```
    np.min(x)
```

23. tsargmax_a(x): take the index of the time-series maximum of the most recent a measurements.

    TYPE: SEIO

    CODE:

```
    np.argmax(x)
```

24. tsargmin_a(x): take the index of the time-series minimum of the most recent a measurements.

    TYPE: SEIO

    CODE:

```
    np.argmin(x)
```

25. tszscre_a(x): take the time-series z-score of the most recent entry in x over a measurements.

    TYPE: SEIO

    CODE:

```
    (x[-1]-np.mean(x))/np.std(x)
```

26. max(x,y,$\cdots$): take the maximum over the arguments presented.

    TYPE: UIO

    CODE:

```
    np.maximum.reduce([df.values for df in [x,y,...]])
```

27. plus(x,y): addition of operands.

    TYPE: UIO

    CODE:

```
    x+y
```

28. minus(x,y): subtraction of operands.

    TYPE: UIO

    CODE:

```
    x-y
```

29. mult(x,y): multiplication of operands.

    TYPE: UIO

    CODE:

```
    x*y
```

30. div(x,y): division of operands.

    TYPE: UIO

    CODE:

```
x/y
```

NOTE: Both $\infty, -\infty$ are replaced with NaN values.

31. and(x,y): logical AND operator.

    TYPE: UIO

    CODE:

    ```
    np.logical_and(x,y)
    ```

    NOTE: Both x, y must be time series of boolean data type entries.

32. or(x,y): logical OR operator.

    TYPE: UIO

    CODE:

    ```
    np.logical_or(x,y)
    ```

    NOTE: Both x, y must be time series of boolean data type entries.

33. eq(x,y): logical EQUALS operator.

    TYPE: UIO

    CODE:

    ```
    x.eq(y)
    ```

    NOTE: Even though the operators are semantically valid when the operands are numerical data types, our formulaic alphas assume they are of boolean data type entries. Here the *.eq* function is from the *pandas.DataFrame* module.

34. gt(x,y): ‘greater than’ operator.

    TYPE: UIO

    CODE:

    ```
    x>b
    ```

35. lt(x,y): ’lesser than’ operator.

    TYPE: UIO

    CODE:

```
x<y
```

36. ite(x,y,z): if x then y else z.

    TYPE: UIO

    CODE:

    ```
    x.fillna(0).astype(int)*y + (~x.astype(bool)).fillna(0).astype(int)*z
    ```

    NOTE: x must be time series of boolean data type entries.

37. cor_a(x,y): a-measurement pearson correlation between x and y.

    TYPE: SIO

    CODE:

    ```
    np.corrcoef(x,y)[0][1]
    ```

38. kentau_a(x,y): a-measurement kendall-tau correlation between x and y.

    TYPE: SIO

    CODE:

    ```
    scipy.stats.kendalltau(x,y)[0]
    ```

39. cov_a(x,y): a-measurement covariance between x and y.

    TYPE: SIO

    CODE:

    ```
    np.cov(x,y)[0][1]
    ```

4 Compound Functions as Examples

Some well known compound functions, indicators and strategies are given using the primitives discussed in Section 1 and their interpretations given in Section 3. These compound functions may also be found in the formulaic alphas.

---

1. grssret_a(): gross-returns.

```
    div(close,delay_a(close))
```

2. logret_a(): log-returns.

```
    log(grssret_a())
```

3. netret_a(): net-returns.

```
    minus(grssret_a(),const_1)
```

4. volatility_a(): volatility.

```
    std_a(minus(logret_1(),mean_a(logret_1())))
```

5. rsi_a(): relative strength-index.

```
  minus(
    const_100,
    div(
      const_100,
      plus(
        const_1,
        div(
          mean_a(max(logret_1(),const_0)),
          mean_a(max(neg(logret_1()),const_0))
        )
      )
    )
  )
```

6. mvwap_a(): moving volume-weighted average price.

```
  div(
    sum_a(
      mult(
        div(
          plus(open,plus(low,close)),
          const_3
        ),
        volume
        )
    ),
    sum_a(volume)
)
```

7. obv_a(): on balance volume.

```
sum_a(
    mult(
        sign(grssret_1()),
        volume
    )
)
```

8. atr_a(): average true range.

```
mean_a(
    max(
        minus(high,low),
        abs(minus(high,delay_1(close)))
    )
)
```

9. adx_a(): average directional index.

```
mean_a(
    mult(
        const_100,
        div(
            abs(
                minus(
                    mult(const_100,div(mean_a(delta_1(high)),atr_a())),
                    mult(const_100,div(mean_a(neg(delta_1(low))),atr_a()))
                )
            ),
            abs(
                plus(
                    mult(const_100,div(mean_a(delta_1(high)),atr_a())),
                    mult(const_100,div(mean_a(neg(delta_1(low))),atr_a()))
                    )
                )
            )
        )
    )
```

References

-