Srinivasa Ramanujan discovery

ChatGPT

Srinivasa Ramanujan (1887-1920) was an Indian mathematician known for his self-taught prowess and his groundbreaking contributions to mathematical analysis, number theory, infinite series, and continued fractions. He made significant discoveries with little formal training and is hailed as a mathematical genius. Some of his notable discoveries and contributions include:

* Ramanujan Prime: A prime number that satisfies certain conditions relating to the prime counting function.
* Ramanujan's Sum: A type of q-series that has applications in number theory.
* Ramanujan Theta Functions: These are certain transcendental functions introduced by Ramanujan.
* Ramanujan's Tautology: Related to the coefficients of powers of modular forms.
* Modular Equations: He provided deep insights into modular equations and modular forms.
* Partition Function P(n): He gave a radical asymptotic series that could compute the number of ways a number can be represented as the sum of positive integers.
* Mock Theta Functions: Functions that appear similar to, but are distinct from, the well-studied theta functions. Ramanujan introduced these in his last letter to the famous mathematician G.H. Hardy.
* Continued Fractions: Ramanujan had a special fascination with continued fractions and made several contributions to this area.
* The Infinite Series for π: Perhaps one of his most famous results, Ramanujan discovered several rapidly converging infinite series to compute the value of π.
* Ramanujan's Conjecture: Later proven as a part of the Taniyama–Shimura–Weil conjecture which was instrumental in the proof of Fermat's Last Theorem by Andrew Wiles.

Ramanujan's life and work are well-documented, especially in the letters he exchanged with G.H. Hardy, a renowned British mathematician who recognized Ramanujan's genius and collaborated with him during Ramanujan's time in England. Their collaboration and friendship have also been portrayed in films and plays, with the most notable being the movie "The Man Who Knew Infinity" based on Ramanujan's biography of the same name.

It's important to note that while these discoveries were significant, they only scratch the surface of Ramanujan's vast contributions to mathematics. His notebooks, filled with results that were ahead of his time, are still being studied and understood by mathematicians today.



list his discoveries that have relation to describing, analyzing and/or forecasting data series. make a table of discovery/meaning in simple words/how it can be utilized

ChatGPT

While Srinivasa Ramanujan's contributions primarily lie within pure mathematics, some of his discoveries can be potentially related to or applied in the context of describing, analyzing, or forecasting data series. Here's a table that attempts to outline these in the context you've specified:

| **Discovery** | **Meaning in Simple Words** | **How it can be Utilized** |
| --- | --- | --- |
| Ramanujan's Sum | A method to calculate a special sum given a number and a modulus. | Used in number theory and potentially for certain algorithms that rely on modular arithmetic. |
| Ramanujan Theta Functions | Transcendental functions relating to counting problems in mathematics. | Might be used in generating and analyzing certain types of sequences or series. |
| Partition Function P(n) | Tells how many ways a number can be represented as the sum of positive integers. | Useful in combinatorics and possibly for understanding the distribution or arrangement of data points. |
| Mock Theta Functions | Functions similar to theta functions but with distinct properties. | Can potentially be used in understanding sequences or series with certain characteristics or patterns. |
| Continued Fractions | Represents numbers as fractions in a repeated manner. | Could be applied in algorithms for data approximation, and possibly in understanding patterns in certain data series. |
| Infinite Series for π | Rapidly converging series to compute the value of π. | Useful for high-precision calculations which can be beneficial in certain analytical tasks involving circular data. |

While the above table provides a rough alignment between Ramanujan's discoveries and their potential applications in data series, it's worth noting that their direct applications in data science or analytics might require significant adaptation. His work was deeply rooted in pure mathematics, and while they have profound implications, their direct applicability to practical data tasks might not always be straightforward. However, the theoretical foundation they provide can sometimes inspire techniques or solutions in applied mathematics and statistics.



can you add more to that table?

ChatGPT

Certainly! While Ramanujan's main contributions are in the realm of pure mathematics, I'll try to extend the table by looking into broader implications or more tangential applications of his discoveries that might be relevant for data analysis or forecasting:

| **Discovery** | **Meaning in Simple Words** | **How it can be Utilized** |
| --- | --- | --- |
| Ramanujan's Sum | A method to calculate a special sum given a number and a modulus. | Used in number theory and potentially for certain algorithms that rely on modular arithmetic. |
| Ramanujan Theta Functions | Transcendental functions relating to counting problems in mathematics. | Might be used in generating and analyzing certain types of sequences or series. |
| Partition Function P(n) | Tells how many ways a number can be represented as the sum of positive integers. | Useful in combinatorics and possibly for understanding the distribution or arrangement of data points. |
| Mock Theta Functions | Functions similar to theta functions but with distinct properties. | Can potentially be used in understanding sequences or series with certain characteristics or patterns. |
| Continued Fractions | Represents numbers as fractions in a repeated manner. | Could be applied in algorithms for data approximation, and possibly in understanding patterns in certain data series. |
| Infinite Series for π | Rapidly converging series to compute the value of π. | Useful for high-precision calculations which can be beneficial in certain analytical tasks involving circular data. |
| Modular Equations | Equations relating variables in the context of modular arithmetic. | Useful in cryptography. Could also be applied to periodic data or cyclic patterns. |
| Rogers-Ramanujan Identities | Specific mathematical relations between certain sums and products. | Applications in statistical mechanics, which could extend to statistical models in data analysis. |
| Hypergeometric Series | Represents functions as a power series involving ratios of factorial functions. | Useful for solving differential equations, which are prevalent in forecasting and time series analysis. |
| Elliptic Functions | Functions related to elliptic curves and integrals. | Used in cryptography, and can be relevant for modeling certain types of complex data relationships. |
| Ramanujan's Lost Notebook | Contains various results in q-series, modular forms, and other areas. | Some of these results could be explored for advanced mathematical modeling in data science. |
| Ramanujan Prime | A certain kind of prime number defined relative to the number of primes less than half its value. | Though this is more theoretical, prime numbers have applications in encryption and security algorithms. |

Please note that while this table links Ramanujan's contributions to potential data series applications, these connections might be broad or indirect. Implementing his contributions in practical data science tasks often requires a good understanding of the underlying mathematical principles and their relevance to the specific problem at hand.



OK, let's say I'm looking for new functions to create features for my data-series. a generic function template would be something that takes intraday 1-min values and represent them as 1 or more figures. Which of math findings from Ramanujan could potentially be used to write such feature-generating functions?

ChatGPT

Certainly, when thinking about transforming 1-min intraday values into more condensed or representative features, several mathematical concepts introduced or worked on by Ramanujan can be potentially explored:

* Partition Function
* �(�)
* *P*(*n*)
  + Function Idea: Count how many ways an intraday value can be represented as the sum of positive integers. This might be more abstract but can provide insights into the structure of the value.
  + Application: Breakdown intraday volumes or aggregated values into their constituent components.
* Ramanujan's Theta Functions
  + Function Idea: Transform a sequence of intraday values using the theta functions. This would provide a new perspective on the data and might be able to capture some patterns in the series.
  + Application: Extract cyclic or seasonal patterns in intraday price movements.
* Mock Theta Functions
  + Function Idea: As with the regular theta functions, mock theta functions can be used to transform sequences and potentially highlight specific patterns.
  + Application: Capture specific anomalies or non-standard patterns in the data.
* Continued Fractions
  + Function Idea: Represent a day's average value or some other metric as a continued fraction. This representation can provide insights into the hierarchy or breakdown of the number.
  + Application: Understand the structure of certain aggregated metrics. This can be especially useful if values tend to have multiple recurrent contributors.
* Infinite Series for π
  + Function Idea: Calculate the convergence rate of a day's values to π. It's abstract but might be useful if there's some underlying circular or repetitive pattern in the data.
  + Application: Investigate periodic patterns or cycles in intraday data.
* Ramanujan's Sum
  + Function Idea: Calculate Ramanujan's sum for each day's values. This sum has specific properties that might highlight patterns when applied to financial data.
  + Application: Understand the cyclic behavior of intraday price changes or returns.

When you're looking to generate features from raw data, it's often a matter of experimentation. Ramanujan's works offer rich mathematical concepts, and while they were not originally intended for time-series analysis in finance, their underlying principles can be adapted creatively. Always ensure to validate the effectiveness of such features in your predictive models before integrating them into your final solution.



give me more

ChatGPT

Certainly, let's explore some more concepts from Ramanujan's work and their potential applications to generating features for intraday data:

* Ramanujan's Modular Equations
  + Function Idea: Transform a sequence using modular arithmetic based on Ramanujan's equations.
  + Application: Analyze periodic behavior in intraday data by mapping them onto modular spaces.
* Ramanujan Primes
  + Function Idea: Check the frequency of intraday values that are close to or resemble the characteristics of Ramanujan primes.
  + Application: Analyze the rarity or uniqueness of certain intraday patterns or trends.
* Ramanujan's Trigonometric Functions
  + Function Idea: Use some of Ramanujan's unique trigonometric derivations to transform intraday values.
  + Application: This can be used to capture cyclic patterns or harmonics in intraday data, especially if the data has some seasonality or repeating patterns.
* Highly Composite Numbers
* Function Idea: Ramanujan had an interest in numbers with more divisors than any smaller number. This concept could be used to generate features representing how "composite" a value is.
* Application: Understand the breakdown or composition of aggregated metrics during the day.
* Nested Radicals
* Function Idea: Represent sequences or daily patterns using nested radicals. Ramanujan had unique formulas involving these.
* Application: Capture specific value hierarchies or structures in intraday data.
* Elliptic Functions and Integrals
* Function Idea: Use Ramanujan's insights on elliptic integrals to transform time-series data.
* Application: Identify non-linear patterns or transitions in intraday data.
* Ramanujan's TAU Function
* Function Idea: Calculate TAU values for intraday metrics and see if there are any patterns or correlations.
* Application: Investigate how data behaves relative to specific mathematical functions.
* Ramanujan's Magic Square
* Function Idea: Map intraday data onto a magic square structure and analyze its characteristics.
* Application: Understand symmetries or unique relationships within a day's data.

To reiterate, these concepts are quite abstract and weren't originally intended for the field of time series forecasting in financial markets. However, with creative thinking and by understanding the underlying mathematical principles, one can derive innovative features. It's essential, though, to validate any newly generated feature's predictive power before integrating it into a model.



how about narrowing it down. so far I've used only functions that use data from 1-min within each day. only that shape of chunks. but there may be patterns to look for that start in the end of previous day and end some time of the next day, or span certain periods within a number of days. of course we can't go combinatorically to find all those cases, we need some rich mathematical concepts that could be used, which ones?

ChatGPT

For time series data, especially intraday data, the key challenge lies in identifying and capturing patterns that persist over non-uniform time frames, as you've identified. Ramanujan's work provides several rich mathematical concepts that might be helpful in this context. Let's distill a few based on the idea of capturing patterns that span across days:

* Modular Forms and Functions
  + Idea: The concept behind modular forms is the behavior of functions under certain transformations. We can take rolling windows of time series data, with a window spanning a part of the previous day and part of the next day, and then analyze how certain statistics or features of the window change as we transform (shift) the window.
  + Application: Detect patterns that might not be evident in a static daily window but become apparent when observed in a moving window across days.
* Continued Fractions
  + Idea: Use continued fractions to represent sequences that span across days. This might allow for the representation of intraday data in a more continuous format, helping in identifying persistent patterns.
  + Application: Analyze long-term dependencies in the data by representing them in the form of continued fractions.
* Theta Functions
  + Idea: Theta functions have periodic properties and can be used to model sequences that exhibit periodic behaviors, not just within a single day but spanning across multiple days.
  + Application: Capture cyclic patterns that span more than a single trading day.
* Elliptic Functions and Integrals
  + Idea: Elliptic functions can be viewed as generalizations of the trigonometric functions. Given their non-linear nature, they could be instrumental in capturing patterns in data chunks that span irregular periods.
  + Application: Represent and analyze data that starts at the end of one day and spans into the subsequent day.
* Nested Radicals
  + Idea: Using nested radicals might be a way to capture hierarchical structures in data. For example, within a three-day span, the primary radical could represent the entire period, and nested ones could represent individual days or sub-parts of days.
  + Application: Detect multi-level patterns in data chunks that span across days.
* Partition Function
  + Idea: Ramanujan had extensive work on the partition function. In the context of time series, this could be utilized to see how many ways a given period's data can be partitioned based on some criteria.
  + Application: Understand the multiplicity or variety of patterns within chunks of data spanning multiple days.

When approaching such a problem, the key lies in experimentation. Mathematical concepts, while rich and varied, need to be adapted and tested rigorously against real-world data to determine their efficacy. The advantage of using concepts like Ramanujan's is that they introduce non-conventional ways of looking at data, which might uncover previously unseen patterns. However, the validation of their usefulness in predictive power is critical.



ok, let's take another approach. suppose I wanna make an indicator that would show visually something that is not obvious from looking at the price charts. a template would be some rolling function that have one or more outputs to display or mark some price bars in colors. which of Ramanujan concepts are applicable?

ChatGPT

Creating visual indicators for price charts using mathematical concepts can provide traders and analysts with unique perspectives on market behavior that might not be immediately apparent from raw price data. Drawing from Ramanujan's vast body of work, let's conceptualize a few potential visual indicators:

* Modular Function-Based Moving Window:
  + Idea: Similar to a moving average, but the weight given to each price in the window is determined by a modular function transformation.
  + Visualization: Display this as a line on the price chart. When the modular function-based line significantly deviates from a standard moving average, it can be a visual cue of a pattern shift or an emerging trend.
* Continued Fractions Volatility Index:
  + Idea: Represent price changes as continued fractions. The depth or length of the continued fraction could be a representation of volatility.
  + Visualization: A bar or line chart beneath the main price chart. Longer bars indicate higher volatility periods.
* Theta Function Cycles:
  + Idea: Identify cyclical patterns in the price data using theta functions.
  + Visualization: Overlay circles or ellipses on the price chart where the major axis represents the detected cycle's duration and the minor axis represents its intensity or amplitude.
* Elliptic Function Momentum:
  + Idea: Use elliptic functions to model price momentum. When price changes behave in a way similar to elliptic function transformations, it might indicate sustained momentum.
  + Visualization: Display as a histogram beneath the price chart. Positive values indicating upward momentum and negative for downward.
* Nested Radical Price Structure:
  + Idea: Capture hierarchical price structures using nested radicals. For instance, if you consider three nested periods - monthly, weekly, and daily - nested radicals could provide a composite score for price behavior across these periods.
  + Visualization: Color-code price bars. A dark hue could indicate alignment in price behavior across all periods, while a lighter hue might suggest discordance.
* Partition Function Market Diversity:
  + Idea: Use the partition function to gauge market diversity — how many different ways can a given price move be interpreted based on other market variables like volume, open interest, etc.?
  + Visualization: Shade or color the background of the price chart. A varied or 'patchy' background could suggest periods of high market diversity, indicating uncertainty or multiple driving factors.
* Ramanujan Summation Patterns:
  + Idea: Identify patterns based on Ramanujan's summation techniques, like the 1 + 2 + 3 + ... = -1/12 identity. This can be a way to detect anomalies or unusual behaviors in price sequences.
  + Visualization: Overlay dotted or dashed lines on the price chart at points where these summation patterns emerge.

Using these mathematical concepts for visual indicators would be experimental. They may reveal unseen patterns or nuances in the data, or they may serve primarily as novel visual tools without significant predictive value. However, the process of conceptualizing and testing them against real-world data can yield valuable insights.



2 / 2

Ramanujan Summation Patterns: Idea: Identify patterns based on Ramanujan's summation techniques, like the 1 + 2 + 3 + ... = -1/12 identity. This can be a way to detect anomalies or unusual behaviors in price sequences. Visualization: Overlay dotted or dashed lines on the price chart at points where these summation patterns emerge.

ChatGPT

Certainly, the Ramanujan Summation Patterns could be a novel way to approach price data, but it's important to understand the mathematical context behind the seemingly odd result

1+2+3+...=−1/12

1+2+3+...=−1/12. This result comes from the realm of analytic continuation and regularization techniques in mathematics, and it doesn't mean in a classical sense that the sum of all positive integers is

−1/12

−1/12.

However, using the idea as an inspiration for a novel indicator, here's how you might proceed:

1. Contextual Interpretation: The

−1/12

−1/12 result is tied to the idea of assigning values to divergent series under certain contexts. In the world of price charts, this could be interpreted as: How can we assign a value or meaning to a trend that seems to be running away or diverging?

2. Implementation:

a. Running Total Sequence: For each price point, compute a running total from a given start point. For a bullish trend, this would involve continually adding higher positive values, analogous to the

1+2+3+...

1+2+3+.... For bearish trends, you would continually add decreasing values.

b. Detection: Search for sequences where the running total is diverging rapidly. For instance, if the sequence grows too quickly or falls too sharply beyond a certain threshold, it could be a candidate for the pattern.

c. Normalization: Apply a regularization technique, such as averaging over a window, to assign a value to these detected sequences. The aim isn't to get

−1/12

−1/12 but to have a consistent method to assign a value to a trend that seems out of the ordinary.

3. Visualization:

Overlay dotted or dashed lines on the price chart where these summation patterns are detected. The intensity or color of the dots/dashes could be representative of how "anomalous" the pattern is. For example, a sequence that grows very rapidly might be indicated with a deep red dotted line, while a mildly diverging sequence could be pale red.

4. Application:

Such an indicator could serve to highlight areas on the chart where price action is getting too heated (either upwards or downwards) and might be due for a reversal, slowdown, or even just warrant caution. It's an unconventional approach to the data, but as with many techniques in trading, sometimes it's the novel approaches that can offer fresh insights.

However, as always with new trading strategies or indicators, backtesting is crucial. This would provide insights into how effective the Ramanujan-inspired summation pattern detection is at highlighting meaningful points in the price data.



OK, I've got some code here that draws the charts. Adapt this code to download S&P EOD data from yahoo finance and implement this indicator. It should produde html file. import pandas as pd import plotly.graph\_objects as go from plotly.subplots import make\_subplots import numpy as np # Function to calculate Heiken Ashi def heiken\_ashi(df,\_open, \_high, \_low, \_close): # this will give an error if there are duplicated columns for some reason, so make sure everything is good df['HA\_Close']=(df[\_open]+ df[\_high]+ df[\_low]+df[\_close])/4 df['HA\_Open']=(df[\_open].shift() + df[\_close].shift())/2 df['HA\_High']=df[['HA\_Open','HA\_Close',\_high]].max(axis=1) df['HA\_Low']=df[['HA\_Open','HA\_Close',\_low]].min(axis=1) return df def create\_chart(df, html\_filename, traces\_data, \_open, \_high, \_low, \_close): # Calculating Heiken Ashi df = heiken\_ashi(df.copy(), \_open, \_high, \_low, \_close) # traces\_data <-- list of lists, each lists with data # Creating subplots fig = make\_subplots(rows = len(traces\_data)+1, cols=1) row = 1 # Adding candlestick plot fig.add\_trace(go.Candlestick(x=df.index, open=df['HA\_Open'], high=df['HA\_High'], low=df['HA\_Low'], close=df['HA\_Close'], name='BTCUSDT'), row=row, col=1) \_cc = [col for col in df.columns if 'Forecasted' in col] # if it's just one model thrshld = 0.001 if len(\_cc) == 1: # then we can show arrows, otherwise not \_cc\_col = \_cc[0] df['long'] = np.nan df['short'] = np.nan ''' df[\_cc\_col].to\_csv("forecast.csv") df.loc[df[\_cc\_col] >= thrshld, 'long'] = df['low'] - 1000 df.loc[df[\_cc\_col] <= -thrshld, 'short'] = df['high'] + 1000 ''' long\_periods = [('2023-01-01', '2023-01-30'), ('2023-02-12', '2023-02-23'), ('2023-03-07', '2023-03-16'), ('2023-04-01', '2023-04-20'), \ ('2023-05-21', '2023-06-23'), ('2023-07-03', '2023-07-17')] short\_periods = [('2023-01-31', '2023-02-11'), ('2023-02-24', '2023-03-06'), ('2023-03-17', '2023-03-31'), ('2023-04-21', '2023-05-20'), ('2023-06-24', '2023-07-02')] for start\_date, end\_date in long\_periods: # Convert the start and end dates to pandas Timestamps for comparison start\_date = pd.Timestamp(start\_date) end\_date = pd.Timestamp(end\_date) # Apply conditions and set 'long' values df.loc[(df[\_cc\_col] > 0) & (df.index >= start\_date) & (df.index <= end\_date), 'long'] = df['low'] - 3000 for start\_date, end\_date in short\_periods: # Convert the start and end dates to pandas Timestamps for comparison start\_date = pd.Timestamp(start\_date) end\_date = pd.Timestamp(end\_date) df.loc[(df[\_cc\_col] < 0) & (df.index >= start\_date) & (df.index <= end\_date), 'short'] = df['high'] + 3000 fig.add\_trace(go.Scatter(x=df.index, y=df['long'], mode='markers', name='Long', marker=dict(symbol='triangle-up', color='green', size=10)), row=row, col=1) fig.add\_trace(go.Scatter(x=df.index, y=df['short'], mode='markers', name='Short', marker=dict(symbol='triangle-down', color='red', size=10)), row=row, col=1) for trace in traces\_data: row += 1 # one row per list of fields for col\_name in trace: if any(val in col\_name for val in ['volume','MK','Forecast','pnl']) and '\_cum' not in col\_name: fig.add\_trace(go.Bar(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1) elif "best" in col\_name: # paint this specific line as dashed (for visual comparison) fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name, line={"dash": "dot"}), row=row, col=1) else: fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1) fig.update\_layout(xaxis\_rangeslider\_visible=False, hovermode='x unified') fig.update\_traces(xaxis='x', hoverinfo='all') # for vertical line #fig.show() # Saving as HTML fig.write\_html(html\_filename)

ChatGPT

Certainly! To adapt the code:

* We'll first integrate the yfinance library to fetch S&P EOD data from Yahoo Finance.
* Implement the Ramanujan-inspired summation pattern detection (as a simple example).
* Modify the create\_chart function to visualize the indicator.

python

Copy code

import yfinance as yf

def fetch\_data():

# Fetch data for S&P 500 from Yahoo Finance

df = yf.download('^GSPC', start="2022-01-01", end="2023-12-31", progress=False)

return df

def ramanujan\_indicator(df, \_close):

window\_size = 5 # Define a suitable window size for the rolling sum

df["Rolling\_Sum"] = df[\_close].rolling(window=window\_size).sum()

df["Ramanujan\_Indicator"] = np.where(df["Rolling\_Sum"].diff() > df[\_close].rolling(window=window\_size).std(), 1, 0)

# This is a very basic example, you'd likely want to refine this to be more sophisticated

return df

df = fetch\_data()

df = ramanujan\_indicator(df, "Close")

# This could be your main method to execute everything:

if \_\_name\_\_ == '\_\_main\_\_':

traces\_data = [['Close', 'Ramanujan\_Indicator']]

create\_chart(df, "SP500\_with\_Ramanujan.html", traces\_data, 'Open', 'High', 'Low', 'Close')

In this code:

* We used the yfinance library to fetch data.
* We implemented a basic version of the Ramanujan indicator. This is a toy example; in a real-world scenario, you would need to iterate, refine, and backtest the logic to get meaningful results.
* The create\_chart function visualizes the fetched data and our newly created Ramanujan indicator.

Remember, the Ramanujan-inspired indicator here is a basic implementation and may not give you trading signals right away. However, it's a starting point to think creatively about how his work might inspire new technical indicators.



super, it works! rewrite this code, so 1) it download data from 2010 2) I can pass a list of windows sizes 5, 8,13,21,44,65,111,111+65 3) it will overlay resulting values on one indicator sub-charts chart

ChatGPT

Of course! Here's a version of the code that implements the changes you requested:

python

Copy code

import yfinance as yf

import pandas as pd

import plotly.graph\_objects as go

from plotly.subplots import make\_subplots

import numpy as np

def fetch\_data():

# Fetch data for S&P 500 from Yahoo Finance from 2010 onwards

df = yf.download('^GSPC', start="2010-01-01", end="2023-12-31", progress=False)

return df

def ramanujan\_indicator(df, \_close, windows):

for window in windows:

rolling\_sum\_col = f"Rolling\_Sum\_{window}"

indicator\_col = f"Ramanujan\_Indicator\_{window}"

df[rolling\_sum\_col] = df[\_close].rolling(window=window).sum()

df[indicator\_col] = np.where(df[rolling\_sum\_col].diff() > df[\_close].rolling(window=window).std(), 1, 0)

return df

def create\_chart(df, html\_filename, traces\_data, \_open, \_high, \_low, \_close):

# ... [rest of the `create\_chart` function remains the same]

# Main execution

if \_\_name\_\_ == '\_\_main\_\_':

df = fetch\_data()

windows = [5, 8, 13, 21, 44, 65, 111, 176] # 111 + 65 = 176

df = ramanujan\_indicator(df, "Close", windows)

indicator\_columns = [f"Ramanujan\_Indicator\_{window}" for window in windows]

traces\_data = [indicator\_columns]

create\_chart(df, "SP500\_with\_Ramanujan.html", traces\_data, 'Open', 'High', 'Low', 'Close')

This updated version:

* Downloads data from 2010.
* Applies the Ramanujan-inspired indicator for multiple window sizes you specified.
* Overlays all resulting indicators onto a single sub-chart.



this is amazing! I can already see some patterns!

ChatGPT

That's fantastic to hear! It's always exciting when you can derive meaningful insights from data, especially when using a blend of financial data and mathematical concepts inspired by a legendary mathematician like Ramanujan.

Do remember, though, that while historical patterns can be indicative, they're not predictive. Always use a combination of tools, methodologies, and intuition when making decisions based on such indicators. Also, consider testing the effectiveness of this indicator on out-of-sample data or through backtesting before considering it in any live trading scenarios.

If you have any more questions or need further assistance with refining or expanding upon this tool, don't hesitate to ask. Happy data diving!



yeah, I want to have more indicators like this. you got the approach - we plot the sub-windows with Ramanujan's indicator function of these rolling periods.

ChatGPT

Absolutely, using mathematical concepts to generate unique features or indicators is a fascinating approach. Let's continue with this approach and derive more indicators inspired by mathematical concepts, including more from Ramanujan's discoveries.

1. Continued Fraction Patterns: Ramanujan was known for his work with continued fractions. We can generate a continued fraction representation of the average price in the window and see if it reveals any unique patterns.

2. Nested Radicals: Ramanujan had unique insights into nested radicals. We can experiment with creating an indicator using nested radicals of the high, low, and closing prices, or their combinations.

3. Ramanujan's Theta Functions: Theta functions can be applied to rolling windows of price data to detect anomalies or recurring patterns. This would be a more advanced implementation, but it could yield unique insights.

4. Partition Function: Ramanujan's insights into the partition function can be utilized as a volatility or momentum measure. How many ways can the price change within a given window, given certain constraints?

5. Modular Forms and Equations: Ramanujan's work on modular forms can be adapted to reflect on how price changes conform to certain patterns or structures.

For each of these concepts, the approach would be:

* Define a function that computes the desired value or pattern for a given window of data.
* Apply this function as a rolling window over your price data.
* Overlay or plot the result as a new indicator on your chart.

Would you like to explore one of these ideas further, or do you have a specific mathematical concept in mind you'd like to dive into?



OK, this is my template, write code for all of these indicators import yfinance as yf import pandas as pd import plotly.graph\_objects as go from plotly.subplots import make\_subplots import numpy as np import algo\_html\_charts as charts def fetch\_data(ticker,start="2010-01-01") : # Fetch data for S&P 500 from Yahoo Finance from 2010 onwards df = yf.download(ticker, end="2023-12-31", progress=False) return df def ramanujan\_indicator(df, \_close, windows): for window in windows: rolling\_sum\_col = f"Rolling\_Sum\_{window}" indicator\_col = f"Ramanujan\_Indicator\_{window}" df[rolling\_sum\_col] = df[\_close].rolling(window=window).sum() df[indicator\_col] = np.where(df[rolling\_sum\_col].diff() > df[\_close].rolling(window=window).std(), 1, 0) return df if \_\_name\_\_ == '\_\_main\_\_': ticker = "BTC-USD" df = fetch\_data(ticker, '2015-01-01') windows = [5, 8, 13, 21, 44, 65, 111, 176] # 111 + 65 = 176 df = ramanujan\_indicator(df, "Close", windows) indicator\_columns = [f"Ramanujan\_Indicator\_{window}" for window in windows] traces\_data = [indicator\_columns] charts.create\_chart(df, f"{ticker}\_Ramanujan.html", traces\_data, 'Open', 'High', 'Low', 'Close')

ChatGPT

Great! Let's continue with the defined structure and integrate the proposed Ramanujan-inspired indicators. Below is a combination of your template with the new indicators:

python

Copy code

import yfinance as yf

import pandas as pd

import plotly.graph\_objects as go

import numpy as np

import algo\_html\_charts as charts

def fetch\_data(ticker,start="2010-01-01") :

# Fetch data for S&P 500 from Yahoo Finance from 2010 onwards

df = yf.download(ticker, start=start, end="2023-12-31", progress=False)

return df

def ramanujan\_indicator(df, \_close, windows):

for window in windows:

# Original Indicator

rolling\_sum\_col = f"Rolling\_Sum\_{window}"

indicator\_col = f"Ramanujan\_Indicator\_{window}"

df[rolling\_sum\_col] = df[\_close].rolling(window=window).sum()

df[indicator\_col] = np.where(df[rolling\_sum\_col].diff() > df[\_close].rolling(window=window).std(), 1, 0)

# Continued Fraction Patterns

rolling\_avg\_col = f"Rolling\_Avg\_{window}"

continued\_fraction\_col = f"Continued\_Fraction\_{window}"

df[rolling\_avg\_col] = df[\_close].rolling(window=window).mean()

df[continued\_fraction\_col] = df[rolling\_avg\_col] / (1 + df[rolling\_avg\_col])

# Nested Radicals (Simplified for demonstration)

nested\_radical\_col = f"Nested\_Radical\_{window}"

df[nested\_radical\_col] = np.sqrt(df[\_close] + np.sqrt(df[\_close]))

# Theta Function Patterns (Simplified for demonstration)

theta\_function\_col = f"Theta\_Function\_{window}"

df[theta\_function\_col] = (df[\_close].diff()).rolling(window=window).apply(lambda x: np.sin(np.sum(x)) / np.sum(x), raw=True)

# Partition Function (Simplified for demonstration)

partition\_function\_col = f"Partition\_Function\_{window}"

df[partition\_function\_col] = df[\_close].rolling(window=window).apply(lambda x: len(np.unique(x)), raw=True)

# Modular Forms (Simplified for demonstration)

modular\_col = f"Modular\_Forms\_{window}"

df[modular\_col] = df[\_close].rolling(window=window).apply(lambda x: np.mean(x) % np.std(x) if np.std(x) != 0 else 0, raw=True)

return df

if \_\_name\_\_ == '\_\_main\_\_':

ticker = "BTC-USD"

df = fetch\_data(ticker, '2015-01-01')

windows = [5, 8, 13, 21, 44, 65, 111, 176] # 111 + 65 = 176

df = ramanujan\_indicator(df, "Close", windows)

# Create a dynamic list of indicator columns based on the windows

indicator\_prefixes = ["Ramanujan\_Indicator", "Continued\_Fraction", "Nested\_Radical", "Theta\_Function", "Partition\_Function", "Modular\_Forms"]

traces\_data = [[f"{prefix}\_{window}" for window in windows] for prefix in indicator\_prefixes]

charts.create\_chart(df, f"{ticker}\_Ramanujan.html", traces\_data, 'Open', 'High', 'Low', 'Close')

This revised code includes the new Ramanujan-inspired indicators based on our discussions. Note that some of the mathematical representations, especially the theta functions and modular forms, are simplified and are for demonstration. Actual implementations might require more complex mathematical derivations or libraries.

However, it should provide a starting point, and you can further refine these indicators based on their performance and insights they provide.



super, it works, now rewrite this to generate 1 html file per one indicator

ChatGPT

2 / 2

Of course! In order to create one HTML file per indicator, we will modify the main execution loop to iterate over the indicator prefixes and create a chart for each of them separately. Here's the modified code:

python

Copy code

import yfinance as yf

import pandas as pd

import plotly.graph\_objects as go

import numpy as np

import algo\_html\_charts as charts

def fetch\_data(ticker,start="2010-01-01") :

# Fetch data for S&P 500 from Yahoo Finance from 2010 onwards

df = yf.download(ticker, start=start, end="2023-12-31", progress=False)

return df

def ramanujan\_indicator(df, \_close, windows):

for window in windows:

# Original Indicator

rolling\_sum\_col = f"Rolling\_Sum\_{window}"

indicator\_col = f"Ramanujan\_Indicator\_{window}"

df[rolling\_sum\_col] = df[\_close].rolling(window=window).sum()

df[indicator\_col] = np.where(df[rolling\_sum\_col].diff() > df[\_close].rolling(window=window).std(), 1, 0)

# Continued Fraction Patterns

rolling\_avg\_col = f"Rolling\_Avg\_{window}"

continued\_fraction\_col = f"Continued\_Fraction\_{window}"

df[rolling\_avg\_col] = df[\_close].rolling(window=window).mean()

df[continued\_fraction\_col] = df[rolling\_avg\_col] / (1 + df[rolling\_avg\_col])

# Nested Radicals (Simplified for demonstration)

nested\_radical\_col = f"Nested\_Radical\_{window}"

df[nested\_radical\_col] = np.sqrt(df[\_close] + np.sqrt(df[\_close]))

# Theta Function Patterns (Simplified for demonstration)

theta\_function\_col = f"Theta\_Function\_{window}"

df[theta\_function\_col] = (df[\_close].diff()).rolling(window=window).apply(lambda x: np.sin(np.sum(x)) / np.sum(x), raw=True)

# Partition Function (Simplified for demonstration)

partition\_function\_col = f"Partition\_Function\_{window}"

df[partition\_function\_col] = df[\_close].rolling(window=window).apply(lambda x: len(np.unique(x)), raw=True)

# Modular Forms (Simplified for demonstration)

modular\_col = f"Modular\_Forms\_{window}"

df[modular\_col] = df[\_close].rolling(window=window).apply(lambda x: np.mean(x) % np.std(x) if np.std(x) != 0 else 0, raw=True)

return df

if \_\_name\_\_ == '\_\_main\_\_':

ticker = "BTC-USD"

df = fetch\_data(ticker, '2015-01-01')

windows = [5, 8, 13, 21, 44, 65, 111, 176] # 111 + 65 = 176

df = ramanujan\_indicator(df, "Close", windows)

# Create a dynamic list of indicator columns based on the windows

indicator\_prefixes = ["Ramanujan\_Indicator", "Continued\_Fraction", "Nested\_Radical", "Theta\_Function", "Partition\_Function", "Modular\_Forms"]

for prefix in indicator\_prefixes:

indicator\_columns = [f"{prefix}\_{window}" for window in windows]

traces\_data = [indicator\_columns]

charts.create\_chart(df, f"{ticker}\_{prefix}.html", traces\_data, 'Open', 'High', 'Low', 'Close')

Now, when you run this code, it will generate a separate HTML file for each indicator, named by the ticker and the indicator's prefix.



3 / 3

super, now rewrite this so every file has a separate sub-plot of indicator's values for Open, High, Low, Close, Volume (5 sub-plots). so one file per indicator type and it should have 5 subplots, each subplot for each window size.

ChatGPT

Certainly! I'll adjust the code so that each indicator type creates one HTML file, and within that file, it'll have 5 subplots (one for each of Open, High, Low, Close, Volume) with their respective window size values. Here's how you can achieve this:

python

Copy code

import yfinance as yf

import pandas as pd

import plotly.graph\_objects as go

import numpy as np

import algo\_html\_charts as charts

def fetch\_data(ticker,start="2010-01-01") :

# Fetch data for S&P 500 from Yahoo Finance from 2010 onwards

df = yf.download(ticker, start=start, end="2023-12-31", progress=False)

return df

def ramanujan\_indicator(df, columns, windows):

for col in columns:

for window in windows:

# Original Indicator

rolling\_sum\_col = f"Rolling\_Sum\_{col}\_{window}"

indicator\_col = f"Ramanujan\_Indicator\_{col}\_{window}"

df[rolling\_sum\_col] = df[col].rolling(window=window).sum()

df[indicator\_col] = np.where(df[rolling\_sum\_col].diff() > df[col].rolling(window=window).std(), 1, 0)

# ... You can continue this pattern for other indicators ...

return df

if \_\_name\_\_ == '\_\_main\_\_':

ticker = "BTC-USD"

df = fetch\_data(ticker, '2015-01-01')

windows = [5, 8, 13, 21, 44, 65, 111, 176] # 111 + 65 = 176

columns = ["Open", "High", "Low", "Close", "Volume"]

df = ramanujan\_indicator(df, columns, windows)

# Create a dynamic list of indicator columns based on the windows

indicator\_prefixes = ["Ramanujan\_Indicator"] # You can add more prefixes here for other indicators

for prefix in indicator\_prefixes:

for col in columns:

traces\_data = [[f"{prefix}\_{col}\_{window}" for window in windows]]

charts.create\_chart(df, f"{ticker}\_{prefix}\_{col}.html", traces\_data, 'Open', 'High', 'Low', 'Close')

Now, when you run this code, each indicator type will generate a separate HTML file for every column (Open, High, Low, Close, Volume) and within each file, it'll have subplots for each window size.



I mean f"{ticker}\_{indficator\_name}.html" and inside the file it should have a subplot per each of open/high/low/close/volume, but each subplot will have indicator values for every window size provided

ChatGPT

Ah, I understand your request better now. I'll adjust the code so that each indicator type creates one HTML file, and within that file, there'll be 5 subplots (for Open, High, Low, Close, Volume), with each subplot containing the indicator values calculated for all provided window sizes.

Let's implement it:

python

Copy code

import yfinance as yf

import pandas as pd

import plotly.graph\_objects as go

import numpy as np

import algo\_html\_charts as charts

def fetch\_data(ticker, start="2010-01-01"):

# Fetch data for S&P 500 from Yahoo Finance from 2010 onwards

df = yf.download(ticker, start=start, end="2023-12-31", progress=False)

return df

def ramanujan\_indicator(df, columns, windows):

for col in columns:

for window in windows:

# Original Indicator

rolling\_sum\_col = f"Rolling\_Sum\_{col}\_{window}"

indicator\_col = f"Ramanujan\_Indicator\_{col}\_{window}"

df[rolling\_sum\_col] = df[col].rolling(window=window).sum()

df[indicator\_col] = np.where(df[rolling\_sum\_col].diff() > df[col].rolling(window=window).std(), 1, 0)

# ... You can continue this pattern for other indicators ...

return df

if \_\_name\_\_ == '\_\_main\_\_':

ticker = "BTC-USD"

df = fetch\_data(ticker, '2015-01-01')

windows = [5, 8, 13, 21, 44, 65, 111, 176] # 111 + 65 = 176

columns = ["Open", "High", "Low", "Close", "Volume"]

df = ramanujan\_indicator(df, columns, windows)

# Create a dynamic list of indicator columns based on the windows

indicator\_prefixes = ["Ramanujan\_Indicator"] # You can add more prefixes here for other indicators

for prefix in indicator\_prefixes:

for col in columns:

traces\_data = [[f"{prefix}\_{c}\_{window}" for window in windows] for c in columns]

charts.create\_chart(df, f"{ticker}\_{prefix}.html", traces\_data, 'Open', 'High', 'Low', 'Close')

Now, this code will generate a separate HTML file for each indicator type. Each file will contain 5 subplots (for Open, High, Low, Close, Volume), with each subplot plotting the indicator values for all provided window sizes.



2 / 2

import yfinance as yf import pandas as pd import plotly.graph\_objects as go import numpy as np import algo\_html\_charts as charts def fetch\_data(ticker,start="2010-01-01") : # Fetch data for S&P 500 from Yahoo Finance from 2010 onwards df = yf.download(ticker, start=start, end="2023-12-31", progress=False) return df def ramanujan\_indicator(df, \_close, windows): for window in windows: # Original Indicator rolling\_sum\_col = f"Rolling\_Sum\_{window}" indicator\_col = f"Ramanujan\_Indicator\_{window}" df[rolling\_sum\_col] = df[\_close].rolling(window=window).sum() df[indicator\_col] = np.where(df[rolling\_sum\_col].diff() > df[\_close].rolling(window=window).std(), 1, 0) # Continued Fraction Patterns rolling\_avg\_col = f"Rolling\_Avg\_{window}" continued\_fraction\_col = f"Continued\_Fraction\_{window}" df[rolling\_avg\_col] = df[\_close].rolling(window=window).mean() df[continued\_fraction\_col] = df[rolling\_avg\_col] / (1 + df[rolling\_avg\_col]) # Nested Radicals (Simplified for demonstration) nested\_radical\_col = f"Nested\_Radical\_{window}" df[nested\_radical\_col] = np.sqrt(df[\_close] + np.sqrt(df[\_close])) # Theta Function Patterns (Simplified for demonstration) theta\_function\_col = f"Theta\_Function\_{window}" df[theta\_function\_col] = (df[\_close].diff()).rolling(window=window).apply(lambda x: np.sin(np.sum(x)) / np.sum(x), raw=True) # Partition Function (Simplified for demonstration) partition\_function\_col = f"Partition\_Function\_{window}" df[partition\_function\_col] = df[\_close].rolling(window=window).apply(lambda x: len(np.unique(x)), raw=True) # Modular Forms (Simplified for demonstration) modular\_col = f"Modular\_Forms\_{window}" df[modular\_col] = df[\_close].rolling(window=window).apply(lambda x: np.mean(x) % np.std(x) if np.std(x) != 0 else 0, raw=True) return df if \_\_name\_\_ == '\_\_main\_\_': ticker = "^GSPC" df = fetch\_data(ticker, '2000-01-01') #windows = [5, 8, 13, 21, 44, 65, 111, 176] # 111 + 65 = 176 windows = [376,188,94,47,23] df = ramanujan\_indicator(df, "Close", windows) # Create a dynamic list of indicator columns based on the windows indicator\_prefixes = ["Ramanujan\_Indicator", "Continued\_Fraction", "Nested\_Radical", "Theta\_Function", "Partition\_Function", "Modular\_Forms"] for prefix in indicator\_prefixes: indicator\_columns = [f"{prefix}\_{window}" for window in windows] traces\_data = [indicator\_columns] charts.create\_chart(df, f"{ticker}\_{prefix}.html", traces\_data, 'Open', 'High', 'Low', 'Close') OK, this code is perfect, but it considers only Close, rewrite it so it calculates the same set of indicators for each data\_source from ['Open', 'High', 'Low', 'Close', 'Volume'] it will add prefixes for columns names for each data\_source in each column of [indicator\_columns] and in the end will call create\_chart like this code below charts.create\_chart(df, f"{ticker}\_{prefix}.html", [traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close, traces\_data\_volume ], 'Open', 'High', 'Low', 'Close')

ChatGPT

Certainly! Let's extend the indicators to be calculated for each data source (['Open', 'High', 'Low', 'Close', 'Volume']). I'll make sure to modify the code to generate prefixes for column names based on each data source and then collate the traces data for the create\_chart function.

Here's the modified code:

python

Copy code

import yfinance as yf

import pandas as pd

import plotly.graph\_objects as go

import numpy as np

import algo\_html\_charts as charts

def fetch\_data(ticker, start="2010-01-01"):

# Fetch data from Yahoo Finance

df = yf.download(ticker, start=start, end="2023-12-31", progress=False)

return df

def ramanujan\_indicator(df, data\_sources, windows):

for \_close in data\_sources:

for window in windows:

# Original Indicator

rolling\_sum\_col = f"Rolling\_Sum\_{\_close}\_{window}"

indicator\_col = f"Ramanujan\_Indicator\_{\_close}\_{window}"

df[rolling\_sum\_col] = df[\_close].rolling(window=window).sum()

df[indicator\_col] = np.where(df[rolling\_sum\_col].diff() > df[\_close].rolling(window=window).std(), 1, 0)

# Continued Fraction Patterns

rolling\_avg\_col = f"Rolling\_Avg\_{\_close}\_{window}"

continued\_fraction\_col = f"Continued\_Fraction\_{\_close}\_{window}"

df[rolling\_avg\_col] = df[\_close].rolling(window=window).mean()

df[continued\_fraction\_col] = df[rolling\_avg\_col] / (1 + df[rolling\_avg\_col])

# ... similarly, continue for other indicators ...

return df

if \_\_name\_\_ == '\_\_main\_\_':

ticker = "^GSPC"

df = fetch\_data(ticker, '2000-01-01')

windows = [376,188,94,47,23]

data\_sources = ['Open', 'High', 'Low', 'Close', 'Volume']

df = ramanujan\_indicator(df, data\_sources, windows)

indicator\_prefixes = ["Ramanujan\_Indicator", "Continued\_Fraction", "Nested\_Radical", "Theta\_Function", "Partition\_Function", "Modular\_Forms"]

for prefix in indicator\_prefixes:

traces\_data\_open = [f"{prefix}\_Open\_{window}" for window in windows]

traces\_data\_high = [f"{prefix}\_High\_{window}" for window in windows]

traces\_data\_low = [f"{prefix}\_Low\_{window}" for window in windows]

traces\_data\_close = [f"{prefix}\_Close\_{window}" for window in windows]

traces\_data\_volume = [f"{prefix}\_Volume\_{window}" for window in windows]

charts.create\_chart(df, f"{ticker}\_{prefix}.html", [traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close, traces\_data\_volume], 'Open', 'High', 'Low', 'Close')

With this setup, each indicator will now have its own HTML file. Inside each file, there will be five subplots, corresponding to 'Open', 'High', 'Low', 'Close', and 'Volume'. Each subplot will display indicator values for all the window sizes specified.



Tell me more about the idea begind this code implementing theta function # Theta Function Patterns (Simplified for demonstration) theta\_function\_col = f"Theta\_Function\_{\_close}\_{window}" df[theta\_function\_col] = (df[\_close].diff()).rolling(window=window).apply(lambda x: np.sin(np.sum(x)) / np.sum(x), raw=True)

ChatGPT

Certainly!

The Theta function, in the realm of number theory and complex analysis, is a special function with numerous applications in various areas of mathematics. It typically has the property of being a solution to certain heat equations and plays an important role in the theory of modular forms. The classical Jacobi theta functions are a collection of functions that describe the number of ways an integer can be represented by a sum of integers subject to certain conditions.

However, in the context of the provided code, the name "Theta Function Patterns" seems to be more of a heuristic or descriptive name, as the implementation doesn't match the true mathematical definition of the theta functions. Let's break down what this part of the code does:

* Difference Computation:
* python
* Copy code

df[\_close].diff()

* The code starts by computing the difference between consecutive values of \_close (which can be 'Open', 'High', 'Low', 'Close', or 'Volume'). This results in a new series where each value is the change from the previous data point.
* Rolling Window:
* python
* Copy code

...rolling(window=window)...

* A rolling window is applied to this difference series. Essentially, for each data point, the code considers the window number of previous values (including the current one). For instance, if window is 5, then for the data point at index i, the code looks at the values from index i-4 to i.
* Apply Lambda Function:
* python
* Copy code

...apply(lambda x: np.sin(np.sum(x)) / np.sum(x), raw=True)

* Within each of these rolling windows, the code applies a lambda function. The lambda function first sums all values within the window (resulting in a cumulative change over the window), takes the sine of that sum, and then divides the result by the sum itself. Conceptually, this process captures the oscillatory behavior of the changes in the \_close values over the window.

The resulting series, named "Theta\_Function\_{\_close}\_{window}", captures this oscillatory behavior for each window size and for each data source ('Open', 'High', 'Low', 'Close', or 'Volume').

This heuristic might help in identifying cyclical patterns in the price or volume changes over different window sizes. It's essential to note that while the name "Theta Function" is used, this is not a rigorous implementation of any well-known mathematical theta function but rather a heuristic or descriptive name for a custom, experimental indicator.



can explain in more details what exactly it identifies if we see a spike of this function's value in the indicator window?

ChatGPT

3 / 3

Certainly! To understand the significance of a spike in this function's value, let's break down the computations of the custom "Theta Function Patterns" indicator once more:

Given the implementation:

python

Copy code

...apply(lambda x: np.sin(np.sum(x)) / np.sum(x), raw=True)

* Compute Difference: First, the difference between consecutive values of \_close is computed, which yields the day-to-day changes (positive, negative, or zero).
* Rolling Window Summation: For each data point, the sum of these changes over a specified rolling window is computed. This cumulative change can be thought of as the "net movement" over that window.
* Sine Transformation: The sine function is then applied to this cumulative change. The sine function oscillates between -1 and 1. Significant cumulative changes (in either direction: positive or negative) will push the argument of the sine function towards its periodic extremes.
* Normalization: The sine value is then divided by the cumulative change (net movement) itself.

Given this process, the result can be interpreted as follows:

* Near Zero Value: If the net movement over the rolling window is close to zero, the indicator value will also be near zero, as both the numerator (sine of the sum) and denominator (sum itself) will be close to zero. This indicates that the \_close values over the window were relatively stable with little net movement either up or down.
* Spike in Positive Value: If there's a significant positive net movement in \_close values within the window, the sine function's output will approach its maximum (close to 1), but then this gets divided by the sum itself. The resulting ratio, in absolute terms, signifies how pronounced this upward movement is relative to its magnitude. A spike here indicates a relatively consistent and pronounced upward trend in the \_close values over the window.
* Spike in Negative Value: Similarly, if there's a significant negative net movement, the sine function will approach its minimum (close to -1). A negative spike in the indicator would indicate a consistent and pronounced downward trend in the \_close values over the window.
* Oscillatory Behavior: Given the periodic nature of the sine function, if the \_close values oscillate back and forth without a clear trend within the window, the indicator would also oscillate. This can be a sign of volatility or uncertainty.

In essence, a spike in the indicator's value (either positive or negative) indicates a strong, consistent movement in the \_close values over the rolling window, while oscillations or values near zero suggest more erratic or flat behavior.



can this function be made even more sensitive?

ChatGPT

Certainly! The sensitivity of an indicator can be adjusted in various ways to capture smaller changes or react more strongly to given data patterns. Here are some ways to increase the sensitivity of the Theta Function Patterns indicator:

* Shorten the Window Size: Reducing the rolling window size can make the indicator more sensitive to recent price changes. Instead of capturing trends over a longer period, a shorter window will react more quickly to recent movements. However, this can also introduce more noise into the indicator.
* Modify the Difference Calculation:
  + Using the percentage change (pct\_change()) instead of the absolute difference (diff()) can highlight relative changes, making the indicator more sensitive to proportional shifts.
  + Another way is to use a two-period difference, where the difference between today's price and the price from two days ago is calculated.
* Weighted Rolling Sum: Instead of using a simple rolling sum, you could use an exponentially weighted moving sum, which gives more importance to recent changes. This can be achieved using pandas' ewm function.
* Enhance the Sine Transformation:
  + Apply a multiplier to the cumulative change before taking the sine. This will push the sine function to its periodic extremes more frequently.
  + Use another periodic function that has sharper transitions, like tanh, which can accentuate changes.
* Adjust the Normalization:
  + Instead of dividing by the cumulative change, divide by the absolute value of the cumulative change. This will make the indicator always positive and might magnify the spikes.
  + Consider removing the normalization entirely, allowing the output of the sine function to dominate the indicator's behavior.
* Combine Multiple Measures: Create an aggregate of the indicators derived from different window lengths or transformations. This can be the average, maximum, or a weighted sum of multiple Theta Function Patterns indicators.

Here's a revised version of the Theta Function Patterns with some of the mentioned sensitivity adjustments:

python

Copy code

theta\_function\_col = f"Theta\_Function\_{\_close}\_{window}"

df['Cumulative\_Percentage\_Change'] = df[\_close].pct\_change().rolling(window=window).sum()

df[theta\_function\_col] = np.sin(2 \* np.pi \* df['Cumulative\_Percentage\_Change']) / np.abs(df['Cumulative\_Percentage\_Change'])

This version uses percentage change, multiplies the cumulative change by

2�

2*π* before taking the sine (to get a full oscillation over a unit change), and normalizes using the absolute value of the cumulative change.

When making the indicator more sensitive, it's essential to ensure that the enhanced sensitivity doesn't simply lead to more noise and false signals. Always validate the effectiveness of the modified indicator with backtesting and real-world scenarios.



write a version that uses 2 window sizes (the one given and int(size/2)). so it uses pct\_chg, weighted rolling sum and tanh

ChatGPT

Alright, let's integrate the modifications:

* Using the percentage change to capture relative shifts in the data.
* Using the exponentially weighted moving sum for a weighted rolling sum.
* Using the hyperbolic tangent function (tanh) to accentuate changes and also to provide an output that's constrained between -1 and 1.
* Introducing a secondary window of size int(window/2) to capture intermediate-term changes and perhaps produce a more nuanced indicator.

Here's the code:

python

Copy code

def enhanced\_theta\_function(df, \_close, window):

# Calculate the percentage change

df['Pct\_Change'] = df[\_close].pct\_change()

# Calculate the weighted sum for the main window

main\_window\_col = f"Weighted\_Sum\_{\_close}\_{window}"

df[main\_window\_col] = df['Pct\_Change'].ewm(span=window).mean()

# Calculate the weighted sum for the half window

half\_window = int(window / 2)

half\_window\_col = f"Weighted\_Sum\_{\_close}\_{half\_window}"

df[half\_window\_col] = df['Pct\_Change'].ewm(span=half\_window).mean()

# Create the enhanced theta function using tanh

theta\_function\_col\_main = f"Theta\_Function\_{\_close}\_{window}"

theta\_function\_col\_half = f"Theta\_Function\_{\_close}\_{half\_window}"

df[theta\_function\_col\_main] = np.tanh(df[main\_window\_col])

df[theta\_function\_col\_half] = np.tanh(df[half\_window\_col])

return df

# Usage

df = enhanced\_theta\_function(df, "Close", 20) # Example with window size 20

This enhanced\_theta\_function will add four columns to the dataframe:

* Weighted sum for the main window size
* Weighted sum for half the window size
* Theta function values using tanh for the main window
* Theta function values using tanh for half the window size

The tanh function will push the values between -1 and 1, thus standardizing the output. The two window sizes allow the indicator to capture both short-term and intermediate-term dynamics in the data.



write code to check if for each column in each dataframe [traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close] if there are more than 1 colums with name 'Theta\_Function2\*' is both >0 or <0, make 'Forecasted' column to be 1 (>0) or -1 (<0)

ChatGPT

The request is to identify periods when all "Theta\_Function2\*" columns for each dataset (Open, High, Low, Close) have the same sign (either all positive or all negative) and, based on this observation, assign values to a new column named "Forecasted".

To achieve this, you'll iterate through each list of columns corresponding to each dataset (Open, High, Low, Close), and then for every dataset, you'll check the sign consistency of the "Theta\_Function2\*" columns. If they're all positive or all negative, you'll update the 'Forecasted' column accordingly.

Here's how you can implement this:

python

Copy code

def check\_theta\_columns\_sign(df, traces\_data\_list):

# Initialize 'Forecasted' column with 0s

df['Forecasted'] = 0

for traces\_data in traces\_data\_list:

# Filter only columns that start with 'Theta\_Function2'

theta\_columns = [col for col in traces\_data if col.startswith('Theta\_Function2')]

if not theta\_columns: # If there are no such columns, skip

continue

# Condition for all columns to have positive values

positive\_condition = (df[theta\_columns] > 0).all(axis=1)

# Condition for all columns to have negative values

negative\_condition = (df[theta\_columns] < 0).all(axis=1)

df.loc[positive\_condition, 'Forecasted'] = 1

df.loc[negative\_condition, 'Forecasted'] = -1

return df

# Define traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close with relevant columns

traces\_data\_open = [...]

traces\_data\_high = [...]

traces\_data\_low = [...]

traces\_data\_close = [...]

df = check\_theta\_columns\_sign(df, [traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close])

Now, the dataframe df will have a 'Forecasted' column where:

* 1 represents all the Theta\_Function2\* columns being positive for that date.
* -1 represents all the Theta\_Function2\* columns being negative for that date.
* 0 indicates that the columns have mixed signs or no such columns exist.



to check that at least 2 columns with theta have positive values ornegative values

ChatGPT

Alright! The modification is straightforward.

You just need to adjust the conditions to check for situations where at least 2 of the "Theta\_Function2\*" columns have the same sign. We can use the .sum(axis=1) method for this, as it will give the sum of Boolean values (True=1, False=0). A sum of 2 or more would indicate that at least two columns satisfy the condition.

Here's the updated function:

python

Copy code

def check\_theta\_columns\_sign(df, traces\_data\_list):

# Initialize 'Forecasted' column with 0s

df['Forecasted'] = 0

for traces\_data in traces\_data\_list:

# Filter only columns that start with 'Theta\_Function2'

theta\_columns = [col for col in traces\_data if col.startswith('Theta\_Function2')]

if len(theta\_columns) < 2: # If there are fewer than 2 such columns, skip

continue

# Condition for at least 2 columns to have positive values

positive\_condition = (df[theta\_columns] > 0).sum(axis=1) >= 2

# Condition for at least 2 columns to have negative values

negative\_condition = (df[theta\_columns] < 0).sum(axis=1) >= 2

df.loc[positive\_condition, 'Forecasted'] = 1

df.loc[negative\_condition, 'Forecasted'] = -1

return df

# Define traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close with relevant columns

traces\_data\_open = [...]

traces\_data\_high = [...]

traces\_data\_low = [...]

traces\_data\_close = [...]

df = check\_theta\_columns\_sign(df, [traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close])

This function now checks if at least 2 of the "Theta\_Function2\*" columns have positive values and sets 'Forecasted' to 1, or if at least 2 have negative values, it sets 'Forecasted' to -1. Otherwise, it remains 0.



rewrite this code to show arrows regardless the long\_periods and short\_periods \_cc = [col for col in df.columns if 'Forecasted' in col] # if it's just one model thrshld = 0.001 if len(\_cc) == 1: # then we can show arrows, otherwise not \_cc\_col = \_cc[0] df['long'] = np.nan df['short'] = np.nan ''' df[\_cc\_col].to\_csv("forecast.csv") df.loc[df[\_cc\_col] >= thrshld, 'long'] = df['low'] - 1000 df.loc[df[\_cc\_col] <= -thrshld, 'short'] = df['high'] + 1000 ''' long\_periods = [('2023-01-01', '2023-01-30'), ('2023-02-12', '2023-02-23'), ('2023-03-07', '2023-03-16'), ('2023-04-01', '2023-04-20'), \ ('2023-05-21', '2023-06-23'), ('2023-07-03', '2023-07-17')] short\_periods = [('2023-01-31', '2023-02-11'), ('2023-02-24', '2023-03-06'), ('2023-03-17', '2023-03-31'), ('2023-04-21', '2023-05-20'), ('2023-06-24', '2023-07-02')] for start\_date, end\_date in long\_periods: # Convert the start and end dates to pandas Timestamps for comparison start\_date = pd.Timestamp(start\_date) end\_date = pd.Timestamp(end\_date) # Apply conditions and set 'long' values df.loc[(df[\_cc\_col] > 0) & (df.index >= start\_date) & (df.index <= end\_date), 'long'] = df[\_low] - 3000 for start\_date, end\_date in short\_periods: # Convert the start and end dates to pandas Timestamps for comparison start\_date = pd.Timestamp(start\_date) end\_date = pd.Timestamp(end\_date) df.loc[(df[\_cc\_col] < 0) & (df.index >= start\_date) & (df.index <= end\_date), 'short'] = df[\_high] + 3000 fig.add\_trace(go.Scatter(x=df.index, y=df['long'], mode='markers', name='Long', marker=dict(symbol='triangle-up', color='green', size=10)), row=row, col=1) fig.add\_trace(go.Scatter(x=df.index, y=df['short'], mode='markers', name='Short', marker=dict(symbol='triangle-down', color='red', size=10)), row=row, col=1)

ChatGPT

If you want to display arrows regardless of the long\_periods and short\_periods, you can simplify the code by just using conditions based on the \_cc\_col values without considering the date range.

Here's a refactored version of the code:

python

Copy code

\_cc = [col for col in df.columns if 'Forecasted' in col] # if it's just one model

thrshld = 0.001

if len(\_cc) == 1: # then we can show arrows, otherwise not

\_cc\_col = \_cc[0]

df['long'] = np.nan

df['short'] = np.nan

# Define 'long' arrows for values greater than threshold

df.loc[df[\_cc\_col] >= thrshld, 'long'] = df['low'] - 3000

# Define 'short' arrows for values less than negative threshold

df.loc[df[\_cc\_col] <= -thrshld, 'short'] = df['high'] + 3000

fig.add\_trace(go.Scatter(x=df.index, y=df['long'], mode='markers', name='Long', marker=dict(symbol='triangle-up', color='green', size=10)), row=row, col=1)

fig.add\_trace(go.Scatter(x=df.index, y=df['short'], mode='markers', name='Short', marker=dict(symbol='triangle-down', color='red', size=10)), row=row, col=1)

This simplified version of the code removes the conditions based on long\_periods and short\_periods and just checks the \_cc\_col values to decide where to place the arrows.



from here make a list of symbol tickers as python list Symbol Name Last Price Change % Change Volume Intraday High/Low 52 Week Range Day Chart ^GSPC S&P 500 4,404.33 -33.53 -0.76% 2.001B ^DJI Dow Jones Industrial Average 34,765.74 -180.65 -0.52% 336.59M ^IXIC NASDAQ Composite 13,474.63 -156.42 -1.15% 5.87B ^NYA NYSE COMPOSITE (DJ) 15,818.11 -93.20 -0.59% 0 ^XAX NYSE AMEX COMPOSITE INDEX 4,346.09 -32.31 -0.74% 0 ^BUK100P Cboe UK 100 733.39 -3.45 -0.47% 0 ^RUT Russell 2000 1,871.52 -24.24 -1.28% 0 ^VIX CBOE Volatility Index 16.78 +0.32 +1.94% 0 ^FTSE FTSE 100 7,356.88 -32.76 -0.44% 0 ^GDAXI DAX PERFORMANCE-INDEX 15,789.45 +22.17 +0.14% 0 ^FCHI CAC 40 7,260.25 -7.45 -0.10% 0 ^STOXX50E ESTX 50 PR.EUR 4,284.27 -4.30 -0.10% 0 ^N100 Euronext 100 Index 1,350.00 -4.34 -0.32% 0 ^BFX BEL 20 3,644.18 -28.22 -0.77% 0 IMOEX.ME MOEX Russia Index 2,222.51 -4.14 -0.19% 0 ^N225 Nikkei 225 31,489.62 -277.20 -0.87% 0 ^HSI HANG SENG INDEX 18,329.30 -251.81 -1.36% 0 000001.SS SSE Composite Index 3,150.13 -26.05 -0.82% 2.403B 399001.SZ Shenzhen Index 10,579.56 -100.17 -0.94% 116.067M ^STI STI Index 3,213.58 -19.16 -0.59% 0 ^AXJO S&P/ASX 200 7,120.80 -74.40 -1.03% 0 ^AORD ALL ORDINARIES 7,334.60 -77.20 -1.04% 0 ^BSESN S&P BSE SENSEX 65,539.42 +137.50 +0.21% 0 ^JKSE IDX COMPOSITE 6,900.54 -14.56 -0.21% 0 ^KLSE FTSE Bursa Malaysia KLCI 1,463.51 +3.23 +0.22% 0 ^NZ50 S&P/NZX 50 INDEX GROSS ( GROSS 11,628.33 -134.78 -1.15% 0 ^KS11 KOSPI Composite Index 2,489.46 -36.18 -1.43% 198,336 ^TWII TSEC weighted index 16,446.78 -8.02 -0.05% 0 ^GSPTSE S&P/TSX Composite index 19,899.07 -0.72 -0.00% 191.253M ^BVSP IBOVESPA 115,591.52 -579.90 -0.50% 0 ^MXX IPC MEXICO 53,827.71 +259.39 +0.48% 334.271M ^IPSA S&P/CLX IPSA 5,058.88 0.00 0.00% 0 ^MERV MERVAL 38,390.84 +233.89 +0.61% 0 ^TA125.TA TA-125 1,864.23 -7.04 -0.38% 0 ^CASE30 EGX 30 Price Return Index 18,062.50 +97.20 +0.54% 220.479M ^JN0U.JO Top 40 USD Net TRI Index 3,911.01 -22.35 -0.57% 0

ChatGPT

To create a list of symbol tickers from the provided data, you can extract the symbols given before the description of each entry:

Here's a Python list containing the symbol tickers:

python

Copy code

symbol\_tickers = [

"^GSPC", "^DJI", "^IXIC", "^NYA", "^XAX", "^BUK100P", "^RUT", "^VIX",

"^FTSE", "^GDAXI", "^FCHI", "^STOXX50E", "^N100", "^BFX", "IMOEX.ME",

"^N225", "^HSI", "000001.SS", "399001.SZ", "^STI", "^AXJO", "^AORD",

"^BSESN", "^JKSE", "^KLSE", "^NZ50", "^KS11", "^TWII", "^GSPTSE", "^BVSP",

"^MXX", "^IPSA", "^MERV", "^TA125.TA", "^CASE30", "^JN0U.JO"

]

print(symbol\_tickers)

This will output a list of all the symbol tickers provided.



can this trace be of fixed vertical height?like 1/4 of screen height # Adding candlestick plot fig.add\_trace(go.Candlestick(x=df.index, open=df['HA\_Open'], high=df['HA\_High'], low=df['HA\_Low'], close=df['HA\_Close'], name='BTCUSDT'), row=row, col=1)

ChatGPT

To fix the vertical height of a trace in a subplot, you can set the height of the rows using the fig.update\_layout method and the row\_heights attribute.

However, setting an exact fraction of the screen height can be challenging directly within Plotly. What you can do is define relative heights of rows in multi-row subplots. If your figure has only one row, then the trace will take up the entire height, and you won't need to specify row heights.

Assuming you have a 4-row subplot and you want the candlestick trace (on the first row) to take up a quarter (1/4) of the height, here's how you can adjust:

python

Copy code

import plotly.graph\_objects as go

# Sample data

# df = ...

# Create subplot with 4 rows and 1 column

fig = make\_subplots(rows=4, cols=1)

# Adding candlestick plot

fig.add\_trace(go.Candlestick(x=df.index,

open=df['HA\_Open'],

high=df['HA\_High'],

low=df['HA\_Low'],

close=df['HA\_Close'], name='BTCUSDT'), row=1, col=1)

# Assuming the first row (with the candlestick) should be 1/4 of the height

# and the rest three should share the remaining height equally:

fig.update\_layout(row\_heights=[0.25, 0.25, 0.25, 0.25])

fig.show()

Note that you'll need to adjust this code based on the actual number of rows in your subplot and their content. The given code assumes you have 4 rows and each should be of equal height. Adjust the values in the row\_heights list accordingly.



the first trace should be fixed size (300 px) and others the default size, is this possible? # Creating subplots fig = make\_subplots(rows = len(traces\_data)+1, cols=1) row = 1 # Adding candlestick plot fig.add\_trace(go.Candlestick(x=df.index, open=df['HA\_Open'], high=df['HA\_High'], low=df['HA\_Low'], close=df['HA\_Close'], name='BTCUSDT'), row=row, col=1)

ChatGPT

If you want to have a fixed height for the first subplot (in pixels) and let other subplots take up the remaining space, you can't directly specify a pixel value for a subplot height using the standard make\_subplots and update\_layout functions.

However, you can achieve this by using dcc.Graph from Dash (a web application framework by Plotly). Here's a workaround using Dash:

* Create a Dash app.
* Add the graph as a dcc.Graph component.
* Use CSS to specify a fixed height for the first subplot.

Here's a simplified example:

python

Copy code

# Ensure you have Dash installed

# !pip install dash

import dash

import dash\_core\_components as dcc

import dash\_html\_components as html

import plotly.graph\_objects as go

# Sample data

# df = ...

# Initialize the Dash app

app = dash.Dash(\_\_name\_\_)

# Create the figure with subplots

fig = make\_subplots(rows=2, cols=1)

# Add traces

fig.add\_trace(go.Candlestick(x=df.index,

open=df['HA\_Open'],

high=df['HA\_High'],

low=df['HA\_Low'],

close=df['HA\_Close'], name='BTCUSDT'), row=1, col=1)

# Assuming you have other traces to add to the other subplots

# ...

# Define the app layout

app.layout = html.Div([

dcc.Graph(id='my-graph', figure=fig, style={'height': '600px'}),

])

# Run the app

if \_\_name\_\_ == '\_\_main\_\_':

app.run\_server(debug=True)

The style={'height': '600px'} in dcc.Graph will set the total height for the graph. The height of the individual subplots can be managed with additional CSS.

If you're not familiar with Dash, it might take a bit of effort to get the exact styling you want. Otherwise, for an easier solution, you might need to use a different plotting library or adjust the relative heights as mentioned in the previous answer.



is there a simple way like this: fig.update\_layout(row\_heights=[0.25, and calc other accordingly ])

ChatGPT

There isn't a direct row\_heights parameter in the make\_subplots or update\_layout functions as of my last update in September 2021. However, you can use the specs argument of the make\_subplots function to set relative heights using the rowspan property.

Here's a way you can do it:

Let's say you want 4 rows in total:

* The first row takes 25% of the height (i.e., 0.25 of the entire plot height).
* The remaining 75% is divided equally among the other 3 rows, so each of those rows takes 25% of the entire plot height.

Here's how you can set it up:

python

Copy code

from plotly.subplots import make\_subplots

import plotly.graph\_objects as go

# Number of rows excluding the main row

num\_other\_rows = 3

# Create the subplots with specs

fig = make\_subplots(

rows=num\_other\_rows + 1,

cols=1,

# Specify the relative height for the first row. The other rows take default heights.

specs=[[{"rowspan": num\_other\_rows \* 0.25}]] + [[{}] for \_ in range(num\_other\_rows)]

)

# Add the candlestick trace for the first row

fig.add\_trace(go.Candlestick(

# ... your data here ...

name='BTCUSDT'

), row=1, col=1)

# Continue with adding other traces to the other rows ...

# Show the figure

fig.show()

In this code:

* The specs argument specifies the configuration of each subplot.
* The rowspan property sets the relative height of a subplot row. We set the height of the first row to be 3 times the default height (hence taking 25% of the total height), and the remaining rows take the default height.

This should achieve the result you want!



2 / 2

modify this code to this please import pandas as pd import plotly.graph\_objects as go from plotly.subplots import make\_subplots import numpy as np # Function to calculate Heiken Ashi def heiken\_ashi(df,\_open, \_high, \_low, \_close): # this will give an error if there are duplicated columns for some reason, so make sure everything is good df['HA\_Close']=(df[\_open]+ df[\_high]+ df[\_low]+df[\_close])/4 df['HA\_Open']=(df[\_open].shift() + df[\_close].shift())/2 df['HA\_High']=df[['HA\_Open','HA\_Close',\_high]].max(axis=1) df['HA\_Low']=df[['HA\_Open','HA\_Close',\_low]].min(axis=1) return df def create\_chart(df, html\_filename, traces\_data, \_open, \_high, \_low, \_close): # Calculating Heiken Ashi df = heiken\_ashi(df.copy(), \_open, \_high, \_low, \_close) # traces\_data <-- list of lists, each lists with data # Creating subplots fig = make\_subplots(rows = len(traces\_data)+1, cols=1) row = 1 # Adding candlestick plot fig.add\_trace(go.Candlestick(x=df.index, open=df['HA\_Open'], high=df['HA\_High'], low=df['HA\_Low'], close=df['HA\_Close'], name='BTCUSDT'), row=row, col=1) for i in [1,2,3]: \_cc = [col for col in df.columns if f'Forecasted{i}' in col] # if it's just one model thrshld = 0.001 if len(\_cc) == 1: # then we can show arrows, otherwise not \_cc\_col = \_cc[0] df[f'long{i}'] = np.nan df[f'short{i}'] = np.nan # Define 'long' arrows for values greater than threshold df.loc[df[\_cc\_col] >= thrshld, f'long{i}'] = df[\_low] - (2500+500\*i) # Define 'short' arrows for values less than negative threshold df.loc[df[\_cc\_col] <= -thrshld, f'short{i}'] = df[\_high] + (2500+500\*i) fig.add\_trace(go.Scatter(x=df.index, y=df[f'long{i}'], mode='markers', name=f'Long{i}', marker=dict(symbol='triangle-up', color='green', size=5\*i)), row=row, col=1) fig.add\_trace(go.Scatter(x=df.index, y=df[f'short{i}'], mode='markers', name=f'Short{i}', marker=dict(symbol='triangle-down', color='red', size=5\*i)), row=row, col=1) for trace in traces\_data: row += 1 # one row per list of fields for col\_name in trace: if any(val in col\_name for val in ['volume','MK','Forecast','pnl']) and '\_cum' not in col\_name: fig.add\_trace(go.Bar(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1) elif "best" in col\_name: # paint this specific line as dashed (for visual comparison) fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name, line={"dash": "dot"}), row=row, col=1) else: fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1) fig.update\_layout(xaxis\_rangeslider\_visible=False, hovermode='x unified') fig.update\_traces(xaxis='x', hoverinfo='all') # for vertical line fig.show() # Saving as HTML fig.write\_html(html\_filename)

ChatGPT

You mentioned earlier that you'd like the main candlestick chart to occupy a fixed height (e.g., 300 pixels) and the other traces to use the default size.

As I explained in the previous response, there isn't a direct way to specify fixed pixel heights for subplot rows in Plotly, but you can use relative row heights to achieve a similar effect.

Given the code you provided, here's how you can modify the creation of the subplots to set a larger height for the main candlestick chart and distribute the remaining height among the other traces:

python

Copy code

import pandas as pd

import plotly.graph\_objects as go

from plotly.subplots import make\_subplots

import numpy as np

# ... (rest of the code) ...

def create\_chart(df, html\_filename, traces\_data, \_open, \_high, \_low, \_close):

# ... (rest of the code) ...

# Determine relative height for the main candlestick chart

main\_chart\_height = 3 # for example, 3 times the height of other rows

num\_other\_rows = len(traces\_data)

total\_rows = main\_chart\_height + num\_other\_rows

# Create subplots with relative heights

fig = make\_subplots(

rows=total\_rows,

cols=1,

specs=[[{"rowspan": main\_chart\_height}]] + [[{}] for \_ in range(num\_other\_rows)]

)

row = 1

# ... (rest of the code) ...

# This will create a candlestick chart that occupies the height of the first three rows.

# The other traces will each occupy one row.

# Adjust the value of main\_chart\_height to achieve the desired effect.

# ... (rest of the code) ...

This modification uses the rowspan property to specify a relative height for the main candlestick chart. The value of main\_chart\_height determines how many rows the candlestick chart will span. You can adjust this value as needed to achieve the desired appearance.



pls write the full code

ChatGPT

Alright, I will integrate the relative height logic into the code you provided:

python

Copy code

import pandas as pd

import plotly.graph\_objects as go

from plotly.subplots import make\_subplots

import numpy as np

# Function to calculate Heiken Ashi

def heiken\_ashi(df,\_open, \_high, \_low, \_close):

df['HA\_Close']=(df[\_open]+ df[\_high]+ df[\_low]+df[\_close])/4

df['HA\_Open']=(df[\_open].shift() + df[\_close].shift())/2

df['HA\_High']=df[['HA\_Open','HA\_Close',\_high]].max(axis=1)

df['HA\_Low']=df[['HA\_Open','HA\_Close',\_low]].min(axis=1)

return df

def create\_chart(df, html\_filename, traces\_data, \_open, \_high, \_low, \_close):

# Calculating Heiken Ashi

df = heiken\_ashi(df.copy(), \_open, \_high, \_low, \_close)

# Determine relative height for the main candlestick chart

main\_chart\_height = 3 # for example, 3 times the height of other rows

num\_other\_rows = len(traces\_data)

total\_rows = main\_chart\_height + num\_other\_rows

# Create subplots with relative heights

fig = make\_subplots(

rows=total\_rows,

cols=1,

specs=[[{"rowspan": main\_chart\_height}]] + [[{}] for \_ in range(num\_other\_rows)]

)

row = 1

# Adding candlestick plot

fig.add\_trace(go.Candlestick(x=df.index,

open=df['HA\_Open'],

high=df['HA\_High'],

low=df['HA\_Low'],

close=df['HA\_Close'], name='BTCUSDT'), row=row, col=1)

for i in [1, 2, 3]:

\_cc = [col for col in df.columns if f'Forecasted{i}' in col]

thrshld = 0.001

if len(\_cc) == 1:

\_cc\_col = \_cc[0]

df[f'long{i}'] = np.nan

df[f'short{i}'] = np.nan

# Define 'long' arrows for values greater than threshold

df.loc[df[\_cc\_col] >= thrshld, f'long{i}'] = df[\_low] - (2500+500\*i)

# Define 'short' arrows for values less than negative threshold

df.loc[df[\_cc\_col] <= -thrshld, f'short{i}'] = df[\_high] + (2500+500\*i)

fig.add\_trace(go.Scatter(x=df.index, y=df[f'long{i}'], mode='markers', name=f'Long{i}', marker=dict(symbol='triangle-up', color='green', size=5\*i)), row=row, col=1)

fig.add\_trace(go.Scatter(x=df.index, y=df[f'short{i}'], mode='markers', name=f'Short{i}', marker=dict(symbol='triangle-down', color='red', size=5\*i)), row=row, col=1)

for trace in traces\_data:

row += 1

for col\_name in trace:

if any(val in col\_name for val in ['volume', 'MK', 'Forecast', 'pnl']) and '\_cum' not in col\_name:

fig.add\_trace(go.Bar(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1)

elif "best" in col\_name:

fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name, line={"dash": "dot"}), row=row, col=1)

else:

fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1)

fig.update\_layout(xaxis\_rangeslider\_visible=False, hovermode='x unified')

fig.update\_traces(xaxis='x', hoverinfo='all') # for vertical line

fig.show()

# Saving as HTML

fig.write\_html(html\_filename)

This code will ensure that the main candlestick chart spans the height of three rows while the other traces each occupy one row. Adjust the main\_chart\_height variable as needed to change the relative size of the main chart.



Exception has occurred: ValueError The 'specs' argument to make\_subplots must be a 2D list of dictionaries with dimensions (8 x 1). Received value of type <class 'list'>: [[{'rowspan': 3}], [{}], [{}], [{}], [{}], [{}]] File "C:\Users\Administrator\Documents\distribution forecasting\algo\_html\_charts.py", line 27, in create\_chart specs=[[{"rowspan": main\_chart\_height}]] + [[{}] for \_ in range(num\_other\_rows)] File "C:\Users\Administrator\Documents\distribution forecasting\theta\_function.py", line 149, in <module> charts.create\_chart(df, f"{ticker}\_{prefix}\_{str(windows)}.html", [traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close, traces\_data\_volume], 'Open', 'High', 'Low', 'Close') ValueError: The 'specs' argument to make\_subplots must be a 2D list of dictionaries with dimensions (8 x 1). Received value of type <class 'list'>: [[{'rowspan': 3}], [{}], [{}], [{}], [{}], [{}]]

ChatGPT

I understand the problem. The error is due to the fact that the specs list must be 2D and equal to the number of rows we've mentioned in make\_subplots.

Let's correct that.

python

Copy code

specs\_list = [[{"rowspan": main\_chart\_height}]] + [[{}] for \_ in range(num\_other\_rows - 1)] # -1 because the main chart already occupies multiple rows

You need to replace the line:

python

Copy code

specs=[[{"rowspan": main\_chart\_height}]] + [[{}] for \_ in range(num\_other\_rows)]

with the new specs\_list:

python

Copy code

specs=specs\_list

So the relevant part of the code should be:

python

Copy code

# Determine relative height for the main candlestick chart

main\_chart\_height = 3 # for example, 3 times the height of other rows

num\_other\_rows = len(traces\_data)

total\_rows = main\_chart\_height + num\_other\_rows

specs\_list = [[{"rowspan": main\_chart\_height}]] + [[{}] for \_ in range(num\_other\_rows - 1)]

# Create subplots with relative heights

fig = make\_subplots(

rows=total\_rows,

cols=1,

specs=specs\_list

)

I hope this clears things up!



specs\_list [[{...}], [{}], [{}], [{}], [{}]]

ChatGPT

It seems that the generated specs\_list is shorter than the total number of rows that you specified in make\_subplots.

The list structure for specs has the following meaning:

* [[{"rowspan": main\_chart\_height}]]: This will make the main chart span main\_chart\_height rows.
* [[{}] for \_ in range(num\_other\_rows - 1)]: This creates a single-row specification for each additional chart.

To solve the problem, let's make sure that the specs\_list contains a 2D list with dimensions matching the total number of rows you intend for your subplots.

Here's how you should construct the specs\_list:

python

Copy code

# Determine the number of additional rows you'll need

extra\_rows\_required = main\_chart\_height - 1 # As one row is already accounted for in the main chart

specs\_list = [[{"rowspan": main\_chart\_height}]] + [[{}] for \_ in range(num\_other\_rows + extra\_rows\_required - 1)]

Now you can use this specs\_list when creating the subplots:

python

Copy code

fig = make\_subplots(

rows=total\_rows,

cols=1,

specs=specs\_list

)

If you follow this, the dimensions of specs\_list will match with the total rows specified in make\_subplots.



no, still the same issue, pls try doing it another way

ChatGPT

Alright, let's simplify the problem step by step. The error message is clearly indicating that the dimensions of the 'specs' list isn't matching what make\_subplots is expecting.

* First, let's determine how many rows are required in total. We know that the main candlestick chart occupies one row, and len(traces\_data) additional rows are needed for the additional traces.
* We'll then create the specs list based on these numbers.

Here's how we can do this:

python

Copy code

# Calculate total rows

total\_rows = 1 + len(traces\_data)

# First, set the candlestick row to occupy a specific height (e.g., 300 pixels)

specs\_list = [[{"rowheight": 300}]]

# Append default specs for the rest of the rows

for \_ in range(total\_rows - 1):

specs\_list.append([{}])

# Now, create the subplots

fig = make\_subplots(rows=total\_rows, cols=1, shared\_xaxes=True, vertical\_spacing=0.02, specs=specs\_list)

I added shared\_xaxes=True and vertical\_spacing=0.02 to ensure the x-axes are aligned and to reduce the space between subplots. Adjust the vertical\_spacing value if required.

Give this a shot and see if it works for you. If this doesn't solve the problem, then there might be an external factor in your environment causing the issue, and we'll have to investigate further.



please write the full code

ChatGPT

Alright, I'll integrate the change into your provided code. Here's the modified code:

python

Copy code

import pandas as pd

import plotly.graph\_objects as go

from plotly.subplots import make\_subplots

import numpy as np

# Function to calculate Heiken Ashi

def heiken\_ashi(df,\_open, \_high, \_low, \_close):

df['HA\_Close']=(df[\_open]+ df[\_high]+ df[\_low]+df[\_close])/4

df['HA\_Open']=(df[\_open].shift() + df[\_close].shift())/2

df['HA\_High']=df[['HA\_Open','HA\_Close',\_high]].max(axis=1)

df['HA\_Low']=df[['HA\_Open','HA\_Close',\_low]].min(axis=1)

return df

def create\_chart(df, html\_filename, traces\_data, \_open, \_high, \_low, \_close):

df = heiken\_ashi(df.copy(), \_open, \_high, \_low, \_close)

# Calculate total rows

total\_rows = 1 + len(traces\_data)

# Create the specs list

specs\_list = [[{"rowheight": 300}]]

for \_ in range(total\_rows - 1):

specs\_list.append([{}])

# Create the subplots

fig = make\_subplots(rows=total\_rows, cols=1, shared\_xaxes=True, vertical\_spacing=0.02, specs=specs\_list)

# Adding candlestick plot

fig.add\_trace(go.Candlestick(x=df.index,

open=df['HA\_Open'],

high=df['HA\_High'],

low=df['HA\_Low'],

close=df['HA\_Close'], name='BTCUSDT'), row=1, col=1)

for i in [1,2,3]:

\_cc = [col for col in df.columns if f'Forecasted{i}' in col]

thrshld = 0.001

if len(\_cc) == 1: # then we can show arrows, otherwise not

\_cc\_col = \_cc[0]

df[f'long{i}'] = np.nan

df[f'short{i}'] = np.nan

# Define 'long' arrows for values greater than threshold

df.loc[df[\_cc\_col] >= thrshld, f'long{i}'] = df[\_low] - (2500+500\*i)

# Define 'short' arrows for values less than negative threshold

df.loc[df[\_cc\_col] <= -thrshld, f'short{i}'] = df[\_high] + (2500+500\*i)

fig.add\_trace(go.Scatter(x=df.index, y=df[f'long{i}'], mode='markers', name=f'Long{i}', marker=dict(symbol='triangle-up', color='green', size=5\*i)), row=1, col=1)

fig.add\_trace(go.Scatter(x=df.index, y=df[f'short{i}'], mode='markers', name=f'Short{i}', marker=dict(symbol='triangle-down', color='red', size=5\*i)), row=1, col=1)

row = 2

for trace in traces\_data:

for col\_name in trace:

if any(val in col\_name for val in ['volume','MK','Forecast','pnl']) and '\_cum' not in col\_name:

fig.add\_trace(go.Bar(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1)

elif "best" in col\_name: # paint this specific line as dashed (for visual comparison)

fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name, line={"dash": "dot"}), row=row, col=1)

else:

fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1)

row += 1

fig.update\_layout(xaxis\_rangeslider\_visible=False, hovermode='x unified')

fig.update\_traces(xaxis='x', hoverinfo='all') # for vertical line

fig.show()

# Saving as HTML

fig.write\_html(html\_filename)

# Example call

# df = your\_dataframe\_here

# create\_chart(df, "output.html", [[...your\_trace\_data\_here...]], 'Open', 'High', 'Low', 'Close')

The central change made is the creation of the specs\_list and the updated call to make\_subplots. Adjust as needed and ensure that other parts of your workflow are compatible with this change.



Exception has occurred: ValueError Invalid key specified in an element of the 'specs' argument to make\_subplots: 'rowheight' Valid keys include: ['type', 'secondary\_y', 'colspan', 'rowspan', 'l', 'r', 'b', 't'] File "C:\Users\Administrator\Documents\distribution forecasting\algo\_html\_charts.py", line 26, in create\_chart fig = make\_subplots(rows=total\_rows, cols=1, shared\_xaxes=True, vertical\_spacing=0.02, specs=specs\_list) File "C:\Users\Administrator\Documents\distribution forecasting\theta\_function.py", line 149, in <module> charts.create\_chart(df, f"{ticker}\_{prefix}\_{str(windows)}.html", [traces\_data\_open, traces\_data\_high, traces\_data\_low, traces\_data\_close, traces\_data\_volume], 'Open', 'High', 'Low', 'Close') ValueError: Invalid key specified in an element of the 'specs' argument to make\_subplots: 'rowheight' Valid keys include: ['type', 'secondary\_y', 'colspan', 'rowspan', 'l', 'r', 'b', 't']

ChatGPT

I apologize for the oversight. The rowheight property is not a valid attribute for the make\_subplots function.

However, adjusting the height of a subplot in plotly is a bit trickier than it sounds. We don't have a direct way to specify the height of a row, but you can achieve it by manipulating the relative proportions of the subplot rows using the row\_heights parameter of make\_subplots.

Let's proceed with a workaround. We will create a row\_heights list with values in such a way that the first row (for your candlestick) gets a disproportionate amount of space compared to others. You can adjust the factor of 3 to any value that gives you the desired visualization.

Here's a refactored version of your create\_chart function:

python

Copy code

def create\_chart(df, html\_filename, traces\_data, \_open, \_high, \_low, \_close):

df = heiken\_ashi(df.copy(), \_open, \_high, \_low, \_close)

# Calculate total rows

total\_rows = 1 + len(traces\_data)

# Create the row heights list to disproportionately allocate space to the first plot

main\_chart\_factor = 3 # adjust this factor for desired height

remaining\_space = (total\_rows - 1 + main\_chart\_factor - 1)

row\_heights = [main\_chart\_factor/remaining\_space] + [1/remaining\_space] \* (total\_rows - 1)

# Create the subplots

fig = make\_subplots(rows=total\_rows, cols=1, shared\_xaxes=True, vertical\_spacing=0.02, row\_heights=row\_heights)

#... [rest of your code remains unchanged]

This will make the first row (main\_chart\_factor times) taller than the rest. Adjust the main\_chart\_factor as needed.



perfect, awesome!

ChatGPT

I'm glad to hear that worked for you! If you have any more questions or need further assistance, feel free to ask. Happy coding and best of luck with your project! 😊



df = yf.download(ticker, start=start, end="2023-08-16", progress=False) - this for weekly bar

ChatGPT

To download weekly data using the yfinance library, you would set the interval parameter to '1wk'. Here's how you would modify your code:

python

Copy code

import yfinance as yf

ticker = 'AAPL' # for example

start = '2020-01-01' # example start date

df = yf.download(ticker, start=start, end="2023-08-16", interval='1wk', progress=False)

Make sure to adjust the ticker and start variables accordingly. This will provide you with weekly data for the specified ticker and date range.



translate this into pine script def ramanujan\_indicator(df, data\_sources, windows): for \_close in data\_sources: for window in windows: theta\_function\_col2 = f"Theta\_Function2\_{\_close}\_{window}" df['Cumulative\_Percentage\_Change'] = df[\_close].pct\_change().rolling(window=window).sum() df[theta\_function\_col2] = np.sin(2 \* np.pi \* df['Cumulative\_Percentage\_Change']) / np.abs(df['Cumulative\_Percentage\_Change']) return df

ChatGPT

Converting the provided Python code into Pine Script involves several steps:

* Pine Script doesn't use DataFrames like Python. Instead, it uses series and built-in functions for calculations.
* We need to make sure we're using a loop-compatible method for calculations since Pine Script doesn't have typical loops.

Here's a translation of your function into Pine Script:

pinescript

Copy code

//@version=4

study("Ramanujan Indicator", shorttitle="RI", overlay=true)

// Define input parameters

windows = input(int, "Window Length", 14)

\_close = close // You can customize this to other series if needed

// Calculate Cumulative Percentage Change

cumulativePctChange = cum(close / close[1] - 1)

thetaFunction2 = na(cumulativePctChange) ? na : sin(2 \* pi \* cumulativePctChange) / abs(cumulativePctChange)

plot(thetaFunction2, color=color.red, title="Theta Function")

Note:

* Pine Script is quite different from Python in terms of syntax and data handling capabilities. Ensure you thoroughly test any translated code before using it in any trading activities.
* The given Python code seems to calculate for multiple \_close values and windows. However, Pine Script doesn't support nested loops or multiple plots from within a loop in the same way. The provided Pine Script translates the core logic for one window and one close value. If you want to adapt for multiple windows, you'll need to replicate the logic and adjust accordingly.



Save & Submit

Cancel

ChatGPT

To display a horizontal line on each subplot in Pine Script, you can use the `



OK, I have this code row = 2 for trace in traces\_data: for col\_name in trace: if any(val in col\_name for val in ['volume','MK','Forecast','pnl']) and '\_cum' not in col\_name: fig.add\_trace(go.Bar(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1) elif "best" in col\_name: # paint this specific line as dashed (for visual comparison) fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name, line={"dash": "dot"}), row=row, col=1) else: fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1) row += 1 fig.update\_layout(xaxis\_rangeslider\_visible=False, hovermode='x unified') fig.update\_traces(xaxis='x', hoverinfo='all') # for vertical line I need to add horizontal line on 0 for every sub-plot

ChatGPT

To add a horizontal line to each subplot in Plotly, you can make use of the add\_shape method.

In the context of your code, you can add a shape (horizontal line) for each subplot right after adding the traces for that subplot. Here's how you can modify the code to add a horizontal line at y=0 for each subplot:

python

Copy code

row = 2

for trace in traces\_data:

for col\_name in trace:

if any(val in col\_name for val in ['volume','MK','Forecast','pnl']) and '\_cum' not in col\_name:

fig.add\_trace(go.Bar(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1)

elif "best" in col\_name: # paint this specific line as dashed (for visual comparison)

fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name, line={"dash": "dot"}), row=row, col=1)

else:

fig.add\_trace(go.Scatter(x=df.index, y=df[col\_name], name=col\_name), row=row, col=1)

# Add a horizontal line at y=0 for the current subplot (row)

fig.add\_shape(

type="line",

x0=min(df.index),

x1=max(df.index),

y0=0,

y1=0,

yref=f"y{row}", # this refers to the y-axis of the current subplot

line=dict(color="gray", width=1),

row=row,

col=1

)

row += 1

fig.update\_layout(xaxis\_rangeslider\_visible=False, hovermode='x unified')

fig.update\_traces(xaxis='x', hoverinfo='all') # for vertical line

In the provided code, after plotting each trace for a specific subplot, a horizontal line at y=0 is drawn for that subplot using fig.add\_shape(). The key parameter here is yref which is set to the y-axis of the current subplot to ensure the line is drawn on the correct subplot.



code to create one directory if it does not exist

ChatGPT

To create a directory if it does not exist, you can use the os module in Python. Here's a simple code to do that:

python

Copy code

import os

directory\_name = "my\_directory" # replace with your desired directory name

if not os.path.exists(directory\_name):

os.makedirs(directory\_name)

This code checks if a directory with the given name exists, and if it doesn't, it creates one. The os.makedirs() function is used instead of os.mkdir() because os.makedirs() can create any intermediate directories that are missing, ensuring that the full path exists.



3 / 3

I thinking of adding smoothing before taking the functions of ohlcv, can we do something like "sliding window normalization", so we're not dependent on specific price edges and get pretty much unified picture?

ChatGPT

Yes, sliding window normalization (or local normalization) is a great way to transform the data to focus on relative movements rather than absolute values. This technique can help algorithms better detect patterns that are invariant to the scale or level of the data.

Here's a basic idea of how to implement sliding window normalization for a time series:

* Sliding Window: For each point in your time series, consider a window of n previous data points (including the current point).
* Compute Statistics: For each window, compute the mean and standard deviation (or min and max if you prefer a [0,1] range).
* Normalize: Subtract the mean and divide by the standard deviation for each point in the window.

Here's a basic implementation using pandas:

python

Copy code

import pandas as pd

def sliding\_window\_normalize(series, window\_size):

mean = series.rolling(window=window\_size).mean()

std = series.rolling(window=window\_size).std()

return (series - mean) / std

# Example

df = pd.DataFrame({

'Close': [1,2,3,4,5,6,7,8,9,10]

})

df['Normalized\_Close'] = sliding\_window\_normalize(df['Close'], window\_size=3)

print(df)

In the example above, a rolling window of size 3 is used to normalize the 'Close' prices.

Do note that:

* This will introduce NaN values for the first few data points (where the rolling window doesn't have enough data). You might want to handle these NaN values based on your application.
* The choice of window size is crucial. A too-small window might over-normalize and remove genuine trends, while a too-large window might under-normalize and retain absolute price level effects.

The approach normalizes prices based on their local context, rather than their global context. This can be very useful in financial data analysis, where you're more interested in local trends and patterns than absolute price levels.



rewrite this for open,high,low,close inside the sliding\_window\_normalize

ChatGPT

Sure, to apply the sliding window normalization to Open, High, Low, and Close columns, you can extend the function to handle each of these columns.

Here's an implementation:

python

Copy code

import pandas as pd

def sliding\_window\_normalize(df, window\_size):

for col in ['Open', 'High', 'Low', 'Close']:

mean = df[col].rolling(window=window\_size).mean()

std = df[col].rolling(window=window\_size).std()

normalized\_col\_name = f"Normalized\_{col}"

df[normalized\_col\_name] = (df[col] - mean) / std

return df

# Example

data = {

'Open': [1,2,3,4,5,6,7,8,9,10],

'High': [2,3,4,5,6,7,8,9,10,11],

'Low': [0,1,2,3,4,5,6,7,8,9],

'Close': [1.5,2.5,3.5,4.5,5.5,6.5,7.5,8.5,9.5,10.5]

}

df = pd.DataFrame(data)

df = sliding\_window\_normalize(df, window\_size=3)

print(df)

The sliding\_window\_normalize function now accepts a DataFrame df and a window\_size. It computes the normalized values for each of Open, High, Low, and Close columns and adds these normalized values as new columns to the DataFrame.



what would be the method to have a sliding window of n bars which identifies 2 most dominant (high amplitude and high persistence) cycles that also look like 2nd or third harmonics of each other?

ChatGPT

Identifying dominant cycles with a sliding window approach involves some advanced signal processing techniques, particularly within the field of spectral analysis. Here's a generalized approach to find the two dominant cycles in a given data sequence:

* Hilbert-Huang Transform (HHT) or Fourier Transform: The most common way to identify dominant frequencies (or cycles) in a dataset is to use the Fourier Transform. However, Fourier Transform assumes that the data is stationary (i.e., its statistical properties do not change over time), which is rarely the case with financial time series. An alternative is the Hilbert-Huang Transform, which is adaptive and can handle non-stationary and non-linear data.
* Sliding Window: Implement a sliding window over the data. For each window, compute its Empirical Mode Decomposition (EMD) if you're using HHT, or simply compute the Fourier Transform if you're using that approach.
* Identify Dominant Cycles: For each window:
  + With the Fourier Transform: Look at the magnitude of the Fourier coefficients. The frequencies corresponding to the largest coefficients are the dominant cycles.
  + With the HHT: Examine the Intrinsic Mode Functions (IMFs) obtained from the EMD and their Hilbert spectrum.
* Check for Harmonic Relationships: Once you've identified dominant cycles, you can check if they are harmonics of one another. If f1 and f2 are the frequencies of two cycles, then f2 is the nth harmonic of f1 if f2 = n \* f1. For 2nd or 3rd harmonics, you would check if f2 is approximately 2 or 3 times f1.
* Amplitude & Persistence:
  + Amplitude: This is straightforward if using the Fourier Transform - the dominant cycles have the highest amplitudes. With the HHT, you can derive amplitude information from the Hilbert spectrum of each IMF.
  + Persistence: You can define this in several ways. One approach might be to see how consistently a particular cycle appears as dominant across several sliding windows.
* Filtering: Depending on how noisy the data is, you may need to filter out cycles that don't meet a certain threshold of amplitude to ensure that they are truly dominant and not artifacts of noise.

Implementing this is not trivial, especially if you're unfamiliar with signal processing techniques. Python libraries like numpy, scipy, and PyEMD can be of help.

If you intend to perform this kind of analysis on financial time series data, remember that such data has a lot of noise, and it can be very challenging to accurately identify persistent, meaningful cycles. Always be cautious when interpreting results and avoid overfitting to historical data.



pls write a function that does it using HHT. input is a df and set\_of\_columns, the output should be df added 2 columns per column in set\_of\_columns, named like column\_cycle\_1, columns\_cycle\_2

ChatGPT

Certainly! The Hilbert-Huang Transform (HHT) mainly consists of two steps: Empirical Mode Decomposition (EMD) and Hilbert Spectral Analysis.

Here's a simple function using the PyEMD package for EMD and numpy for the Hilbert Transform. The function identifies two dominant cycles for each column in set\_of\_columns:

python

Copy code

import numpy as np

import pandas as pd

from PyEMD import EMD

def dominant\_cycles(df, set\_of\_columns):

# Helper function to find the dominant frequency

def find\_dominant\_freqs(imf):

hilbert\_transformed = np.abs(np.hilbert(imf))

freqs = np.fft.fftfreq(len(hilbert\_transformed))

magnitudes = np.abs(np.fft.fft(hilbert\_transformed))

# Return indices of two largest magnitudes

return freqs[np.argsort(magnitudes)[-2:]]

for col in set\_of\_columns:

series\_data = df[col].values

# Apply Empirical Mode Decomposition

emd = EMD()

IMFs = emd(series\_data)

# For this example, let's just take the first IMF

# Typically, you would analyze all IMFs or a subset of them

imf1 = IMFs[0]

dominant\_freqs = find\_dominant\_freqs(imf1)

# Add dominant cycles to the dataframe

# Note: The actual cycles are the inverse of the frequencies

df[col + '\_cycle\_1'] = 1/dominant\_freqs[0]

df[col + '\_cycle\_2'] = 1/dominant\_freqs[1]

return df

# Example usage:

# df = pd.DataFrame({'data': np.sin(np.linspace(0, 100, 500)) + np.sin(np.linspace(0, 50, 500))})

# set\_of\_columns = ['data']

# result = dominant\_cycles(df, set\_of\_columns)

Note:

* The outputted dominant cycles are approximations. The precision depends on the length of the dataset and the characteristics of the signal itself.
* The IMF (Intrinsic Mode Function) selection here is simple and uses the first IMF. Depending on the data, one may want to inspect and select the most relevant IMFs or even combine information from multiple IMFs.
* Financial time series can be very noisy, and this approach may or may not work well depending on the nature of the data.

It's crucial to visualize and understand the IMFs produced and the cycles/frequencies detected to ensure the results make sense for the data being analyzed.



OK, but this takes all data, however I need to recalculate cycle1 and cycle2 on every bar taking a sliding window of 300 bars

ChatGPT

Sure, I understand. Here's the function adapted for a rolling window analysis:

python

Copy code

import numpy as np

import pandas as pd

from PyEMD import EMD

def dominant\_cycles(df, set\_of\_columns, window\_size=300):

# Helper function to find the dominant frequency

def find\_dominant\_freqs(imf):

hilbert\_transformed = np.abs(np.hilbert(imf))

freqs = np.fft.fftfreq(len(hilbert\_transformed))

magnitudes = np.abs(np.fft.fft(hilbert\_transformed))

# Return indices of two largest magnitudes

return freqs[np.argsort(magnitudes)[-2:]]

for col in set\_of\_columns:

cycle1 = []

cycle2 = []

for i in range(len(df) - window\_size + 1):

window\_data = df[col].iloc[i:i + window\_size].values

# Apply Empirical Mode Decomposition

emd = EMD()

IMFs = emd(window\_data)

# For this example, let's just take the first IMF

# Depending on the data, you might want to select a different IMF

imf1 = IMFs[0]

dominant\_freqs = find\_dominant\_freqs(imf1)

# Note: The actual cycles are the inverse of the frequencies

cycle1.append(1/dominant\_freqs[0])

cycle2.append(1/dominant\_freqs[1])

# Fill initial values with NaNs to align with the size of df

cycle1 = [np.nan] \* (window\_size - 1) + cycle1

cycle2 = [np.nan] \* (window\_size - 1) + cycle2

# Add dominant cycles to the dataframe

df[col + '\_cycle\_1'] = cycle1

df[col + '\_cycle\_2'] = cycle2

return df

# Example usage:

# df = pd.DataFrame({'data': np.sin(np.linspace(0, 100, 500)) + np.sin(np.linspace(0, 50, 500))})

# set\_of\_columns = ['data']

# result = dominant\_cycles(df, set\_of\_columns)

This function will apply the EMD and Hilbert transform within a rolling window of the specified size (window\_size). The resulting DataFrame will have the dominant cycles appended as new columns. The first window\_size - 1 values of these new columns will be NaN because there isn't enough data for a full window at those points.



Now I have this function that uses 2 fixed periods (window and 1/2 window), rewrite so it 1) calls dominant\_cycles function to update 2 periods every bar 2) uses the two cycles as window sizes to do calculations def enhanced\_theta\_function(df, data\_sources, windows): for \_close in data\_sources: for window in windows: theta\_function\_col2 = f"Theta\_Function2\_{\_close}\_{window}" df['Cumulative\_Percentage\_Change'] = df[\_close].pct\_change().rolling(window=window).sum() df[theta\_function\_col2] = np.sin(2 \* np.pi \* df['Cumulative\_Percentage\_Change']) / np.abs(df['Cumulative\_Percentage\_Change']) # Calculate the percentage change df['Pct\_Change'] = df[\_close].pct\_change() # Calculate the weighted sum for the main window main\_window\_col = f"Weighted\_Sum\_{\_close}\_{window}" df[main\_window\_col] = df['Pct\_Change'].ewm(span=window).mean() # Calculate the weighted sum for the half window half\_window = int(window / 2) half\_window\_col = f"Weighted\_Sum\_{\_close}\_{half\_window}" df[half\_window\_col] = df['Pct\_Change'].ewm(span=half\_window).mean() # Create the enhanced theta function using tanh theta\_function\_col\_main = f"Neat\_Theta\_Function\_{\_close}\_{window}" theta\_function\_col\_half = f"Neat\_Theta\_Function\_{\_close}\_{half\_window}" df[theta\_function\_col\_main] = np.tanh(df[main\_window\_col]) df[theta\_function\_col\_half] = np.tanh(df[half\_window\_col]) return df

ChatGPT

To modify the function to use the dominant cycles, we will:

* Use the dominant\_cycles function to get the two dominant cycles for each column of interest.
* Replace the fixed window sizes (window and 1/2 window) with the calculated dominant cycles.

Here's how you can adapt the function:

python

Copy code

import numpy as np

import pandas as pd

def enhanced\_theta\_function(df, data\_sources):

# Calculate dominant cycles

df = dominant\_cycles(df, data\_sources)

for \_close in data\_sources:

window = df[f"{\_close}\_cycle\_1"].dropna().astype(int).iloc[-1] # get last non-NaN value

half\_window = df[f"{\_close}\_cycle\_2"].dropna().astype(int).iloc[-1] # get last non-NaN value

theta\_function\_col2 = f"Theta\_Function2\_{\_close}\_{window}"

df['Cumulative\_Percentage\_Change'] = df[\_close].pct\_change().rolling(window=window).sum()

df[theta\_function\_col2] = np.sin(2 \* np.pi \* df['Cumulative\_Percentage\_Change']) / np.abs(df['Cumulative\_Percentage\_Change'])

# Calculate the percentage change

df['Pct\_Change'] = df[\_close].pct\_change()

# Calculate the weighted sum for the main window

main\_window\_col = f"Weighted\_Sum\_{\_close}\_{window}"

df[main\_window\_col] = df['Pct\_Change'].ewm(span=window).mean()

# Calculate the weighted sum for the half window

half\_window\_col = f"Weighted\_Sum\_{\_close}\_{half\_window}"

df[half\_window\_col] = df['Pct\_Change'].ewm(span=half\_window).mean()

# Create the enhanced theta function using tanh

theta\_function\_col\_main = f"Neat\_Theta\_Function\_{\_close}\_{window}"

theta\_function\_col\_half = f"Neat\_Theta\_Function\_{\_close}\_{half\_window}"

df[theta\_function\_col\_main] = np.tanh(df[main\_window\_col])

df[theta\_function\_col\_half] = np.tanh(df[half\_window\_col])

return df

# Remember to define the dominant\_cycles function as provided in the previous response

This version of the function will:

* Compute the two dominant cycles for each column in data\_sources (assuming each column contains time-series data).
* Replace the fixed window sizes with the calculated dominant cycles to compute the enhanced theta functions.

The function makes use of the last non-NaN values of the dominant cycles columns to obtain the window sizes. Make sure the length of your dataframe is greater than the window size you provided in the dominant\_cycles function to ensure the computed cycles are meaningful.



No module named 'PyEMD'

ChatGPT

It seems you're missing the PyEMD module, which is necessary for Empirical Mode Decomposition (EMD), a step in the Hilbert-Huang Transform (HHT). To use the functions I've provided previously, you need to install this module.

To install PyEMD, you can use pip:

Copy code

pip install EMD-signal

Note: The package name in the PyPI repository is EMD-signal, but it's imported in Python using from PyEMD import EMD.

Once installed, you should be able to use the provided functions without any issue.



Exception has occurred: ValueError `y` must contain only finite values. File "/home/edward/Downloads/theta/theta\_function\_dyn\_periods.py", line 32, in dominant\_cycles IMFs = emd(window\_data) File "/home/edward/Downloads/theta/theta\_function\_dyn\_periods.py", line 57, in enhanced\_theta\_function df = dominant\_cycles(df, data\_sources) File "/home/edward/Downloads/theta/theta\_function\_dyn\_periods.py", line 171, in <module> df = enhanced\_theta\_function(df, data\_sources) ValueError: `y` must contain only finite values.

ChatGPT

The error you're seeing indicates that the data you're trying to process with the Empirical Mode Decomposition (EMD) contains non-finite values (e.g., NaN or infinite values). Before applying EMD or any other signal processing method, you need to ensure that the data doesn't have any NaN or infinite values.

To handle this issue:

* Check and Handle NaN or Infinite Values: Before processing with EMD, make sure to drop or fill these values.
* python
* Copy code

df = df.dropna() # Drop rows with NaN values

* Alternatively, you can fill NaN values using various methods, such as forward fill, backward fill, or using a specific value.
* python
* Copy code

df = df.fillna(method='ffill') # Forward fill

* Ensure Window Data is Finite: Before processing a window of data, ensure it contains only finite values.
* python
* Copy code

if not np.all(np.isfinite(window\_data)):

continue # Skip this window if it contains NaN or infinite values

Update the dominant\_cycles function by adding the above check right before applying the EMD:

python

Copy code

def dominant\_cycles(df, data\_sources):

# ... (rest of the function)

for idx in range(window\_size, len(df)):

window\_data = df.iloc[idx-window\_size:idx][\_close].values

# Check for finite values

if not np.all(np.isfinite(window\_data)):

cycle1.append(np.nan)

cycle2.append(np.nan)

continue

# Extract IMFs and the residue (trend) using EMD

IMFs = emd(window\_data)

# ... (rest of the function)

This adjustment ensures that any window containing non-finite values will be skipped and will assign NaN values for the dominant cycles for that specific window.



pls rewrite the whole function

ChatGPT

Certainly! Let's integrate the necessary checks and handling for NaN or infinite values directly into the enhanced\_theta\_function and its sub-functions. I'll rewrite the function by adding these checks:

* Drop NaN values from the entire dataframe.
* Check if a sliding window contains any NaN or infinite values before processing it.

Here's the updated function:

python

Copy code

import numpy as np

from scipy.signal import hilbert

from PyEMD import EMD

def dominant\_cycles(df, data\_sources):

emd = EMD()

window\_size = 300 # Define a fixed window size

for \_close in data\_sources:

cycle1 = []

cycle2 = []

for idx in range(window\_size, len(df)):

window\_data = df.iloc[idx-window\_size:idx][\_close].values

# Check for finite values

if not np.all(np.isfinite(window\_data)):

cycle1.append(np.nan)

cycle2.append(np.nan)

continue

# Extract IMFs and the residue (trend) using EMD

IMFs = emd(window\_data)

if len(IMFs) < 2:

cycle1.append(np.nan)

cycle2.append(np.nan)

continue

# Use Hilbert transform to get instantaneous phase, then differentiate to get instantaneous frequency

analytic\_signal\_1 = hilbert(IMFs[-1])

analytic\_signal\_2 = hilbert(IMFs[-2])

inst\_phase\_1 = np.unwrap(np.angle(analytic\_signal\_1))

inst\_phase\_2 = np.unwrap(np.angle(analytic\_signal\_2))

inst\_freq\_1 = np.diff(inst\_phase\_1)

inst\_freq\_2 = np.diff(inst\_phase\_2)

# Get dominant cycle periods

cycle1.append(2 \* np.pi / np.median(inst\_freq\_1))

cycle2.append(2 \* np.pi / np.median(inst\_freq\_2))

# Fill the initial part with NaNs due to the nature of sliding window

cycle1 = [np.nan]\*window\_size + cycle1

cycle2 = [np.nan]\*window\_size + cycle2

df[f"{\_close}\_cycle1"] = cycle1

df[f"{\_close}\_cycle2"] = cycle2

return df

def enhanced\_theta\_function(df, data\_sources):

df = df.dropna() # Drop rows with NaN values to handle any NaNs or Infinite values

df = dominant\_cycles(df, data\_sources) # Update dominant cycles every bar

for \_close in data\_sources:

# Calculate the percentage change

df['Pct\_Change'] = df[\_close].pct\_change()

for idx, window\_col in enumerate([f"{\_close}\_cycle1", f"{\_close}\_cycle2"]):

window = int(df[window\_col].mean()) # Convert the cycle into an integer value for window

theta\_function\_col = f"Theta\_Function\_{idx}\_{\_close}\_{window}"

df['Cumulative\_Percentage\_Change'] = df[\_close].pct\_change().rolling(window=window).sum()

df[theta\_function\_col] = np.sin(2 \* np.pi \* df['Cumulative\_Percentage\_Change']) / np.abs(df['Cumulative\_Percentage\_Change'])

# Calculate the weighted sum

weighted\_sum\_col = f"Weighted\_Sum\_{\_close}\_{window}"

df[weighted\_sum\_col] = df['Pct\_Change'].ewm(span=window).mean()

# Create the enhanced theta function using tanh

theta\_function\_col = f"Neat\_Theta\_Function\_{\_close}\_{window}"

df[theta\_function\_col] = np.tanh(df[weighted\_sum\_col])

return df

Here are the steps:

* Drop NaNs from the entire dataframe at the start of enhanced\_theta\_function.
* Check if any window contains NaN or infinite values in dominant\_cycles and skip processing those windows. This will prevent any EMD processing on non-finite data.



so the length of my df is 1279 rows, but in cycle\_1 and cycle\_2 columns I get cycles of over 300 and even negative cycles, how can this be possible?

ChatGPT

Given your issue, there are a few points to consider:

* Instantaneous Frequency: The calculation for instantaneous frequency after Hilbert transformation might yield values that, when turned into periods (using 2 \* np.pi / frequency), result in periods longer than the data window or even negative periods.
* Median Calculation: We're using the median of the instantaneous frequency to represent the dominant cycle. If instantaneous frequencies contain outliers or unexpected values, the median might yield unexpected results.
* Use of Cycles: We're turning cycles (which are real numbers) into integers for use as windows. Rounding can sometimes lead to values being more than the original window.

Here are the steps to refine the approach:

* Bound the Values: Ensure that the resulting cycle is bounded between 1 and window\_size.
* Check Sign: Negative cycles don't make sense in this context, so it's necessary to ensure that the resulting cycle is always positive.

Let's integrate these checks in the dominant\_cycles function:

python

Copy code

def dominant\_cycles(df, data\_sources):

emd = EMD()

window\_size = 300 # Define a fixed window size

for \_close in data\_sources:

cycle1 = []

cycle2 = []

for idx in range(window\_size, len(df)):

window\_data = df.iloc[idx-window\_size:idx][\_close].values

# Check for finite values

if not np.all(np.isfinite(window\_data)):

cycle1.append(np.nan)

cycle2.append(np.nan)

continue

# Extract IMFs and the residue (trend) using EMD

IMFs = emd(window\_data)

if len(IMFs) < 2:

cycle1.append(np.nan)

cycle2.append(np.nan)

continue

# Use Hilbert transform to get instantaneous phase, then differentiate to get instantaneous frequency

analytic\_signal\_1 = hilbert(IMFs[-1])

analytic\_signal\_2 = hilbert(IMFs[-2])

inst\_phase\_1 = np.unwrap(np.angle(analytic\_signal\_1))

inst\_phase\_2 = np.unwrap(np.angle(analytic\_signal\_2))

inst\_freq\_1 = np.diff(inst\_phase\_1)

inst\_freq\_2 = np.diff(inst\_phase\_2)

# Get dominant cycle periods, bound the values, and ensure they're positive

period1 = max(1, min(window\_size, abs(2 \* np.pi / np.median(inst\_freq\_1))))

period2 = max(1, min(window\_size, abs(2 \* np.pi / np.median(inst\_freq\_2))))

cycle1.append(period1)

cycle2.append(period2)

# Fill the initial part with NaNs due to the nature of sliding window

cycle1 = [np.nan]\*window\_size + cycle1

cycle2 = [np.nan]\*window\_size + cycle2

df[f"{\_close}\_cycle1"] = cycle1

df[f"{\_close}\_cycle2"] = cycle2

return df

This should ensure that the detected cycles are between 1 and 300 (the window size) and are always positive.



"Check Sign: Negative cycles don't make sense in this context, so it's necessary to ensure that the resulting cycle is always positive." - but what is the physical meaning of those negative cycles?

ChatGPT

In the context of the Hilbert Transform and Empirical Mode Decomposition, a negative instantaneous frequency doesn't have a clear physical interpretation when analyzing financial time series or most other types of real-world data.

However, the negative values can emerge due to several reasons:

* Mathematical Artifacts: When we differentiate the unwrapped phase of the analytic signal obtained after Hilbert transform, we get the instantaneous frequency. Depending on how the phase unwraps and how closely spaced these values are, the differentiation might yield negative values due to noise or other artifacts.
* Phase Wrapping: Phase is usually represented as a value between
* −�
* −*π* and
* �
* *π*. The np.unwrap() function tries to mitigate the issue of phase jumps (from, say,
* �
* *π* to
* −�
* −*π*), but it's not always perfect. Sometimes these jumps, when differentiated, can result in negative frequencies.
* Signal Noise: Financial data can be very noisy. High frequency components or sudden jitters in the price can affect the IMF decomposition and, in turn, influence the derived instantaneous frequency.

In the physical world, when dealing with signal processing for things like radio signals or acoustics, a negative frequency often has a clear meaning and interpretation. For instance, in the realm of digital signal processing, a negative frequency corresponds to a clockwise rotation in the complex plane, while a positive frequency is a counter-clockwise rotation. But such an interpretation doesn't readily translate to financial time series.

For your application, it's best to treat these negative frequencies as artifacts and handle them accordingly, either by taking their absolute values or by imposing constraints to ensure the derived cycles remain positive.



also there are a lot of cycles that are fractional, is this acceptable for a rolling window in theta function?

ChatGPT

Yes, having fractional cycles in the context of Hilbert-Huang Transform (HHT) and the Empirical Mode Decomposition (EMD) is both common and expected. The very nature of EMD is to adaptively decompose a signal into its intrinsic mode functions (IMFs), and the instantaneous frequencies derived from these IMFs can be fractional. This is especially true for non-stationary and non-linear data like financial time series, where inherent patterns and frequencies might not fit neatly into integer-based cycles.

However, when applying this to the theta function within a rolling window, a few considerations should be kept in mind:

* Interpretability: Integer cycles (like "a 7-day cycle" or "a 30-day cycle") are easier to interpret and align more naturally with how we think of time-based patterns in finance. Fractional cycles might be harder to explain or act upon.
* Practicality: If you're using the cycles for any form of prediction or modeling, consider whether the fractional difference would meaningfully impact the results. For example, the difference between a 30-day cycle and a 30.5-day cycle might be negligible in practice.
* Discretization: If needed, you can round the fractional cycles to the nearest integer for practical purposes. This, however, might introduce some bias, so you should be aware of potential implications.
* Validation: Regardless of whether the cycles are fractional or integer, it's crucial to validate the effectiveness and stability of these derived cycles in modeling or trading. This can be done through out-of-sample testing, where the effectiveness of the cycles derived from a training period is tested on a subsequent unseen period.

In summary, while fractional cycles are perfectly acceptable from a mathematical standpoint, their utility and practicality in the context of your specific application should be carefully considered. If they offer improved results or insights, then there's no inherent reason to avoid them. If not, you might choose to simplify by rounding or opting for integer cycles.



OK, this works, but it seems to be a little lagging on some periods, either the cycles are too big or we have to imply some volatility to select bigger cycles or smaller ones depending on that. any ideas?

ChatGPT

The challenge you've described is common in cycle analysis, especially in the context of financial time series data. Due to non-stationarity in financial markets, cycles may appear, disappear, or shift in period length over time. Here are a few approaches to address this challenge:

* Volatility-Adjusted Cycles: If you observe that dominant cycles are more pronounced during periods of high volatility, you could adjust your analysis to give more weight to recent data when volatility is high. This might mean using a shorter rolling window during high volatility periods and a longer one during low volatility periods. One way to measure volatility is by using the Average True Range (ATR) or the standard deviation of returns.
* Adaptive Window Size: Instead of using a fixed rolling window size, you could use an adaptive size based on recent market conditions. This could be based on volatility or other market metrics. The idea is to respond faster to recent market changes by using a smaller window during turbulent times and a larger window during stable periods.
* Multiple Cycle Analysis: Instead of relying on one or two dominant cycles, you could look at multiple cycles simultaneously. This might involve deriving a composite signal based on the contributions of several significant cycles. This approach is more complex but can be more robust.
* Combine Cycles with Other Indicators: To improve the reliability of your cycle-based analysis, consider combining it with other technical or fundamental indicators. This can serve as a confirmation signal, ensuring that you're not solely reliant on cycle information.
* Decomposition Methods: Apart from EMD, there are other time-series decomposition methods like STFT (Short-Time Fourier Transform), WT (Wavelet Transform), etc. Each has its own strengths and weaknesses, and sometimes combining insights from multiple decomposition methods can provide a more holistic view of the dominant patterns in the data.
* Filtering Noise: Before extracting cycles, you might consider filtering out the high-frequency noise in your data. Techniques like Lowess smoothing can be beneficial for this.
* Feedback Mechanism: Consider building a feedback mechanism that adjusts the cycle extraction process based on its recent performance. For example, if the cycle-derived signals have been consistently lagging the actual market movements, the mechanism could shorten the window size or adjust other parameters to make the analysis more responsive.

Always remember, the goal is not just to detect cycles but to use them effectively, which often means combining them with other insights and being adaptive to changing market conditions.



how can we add the feedback mechanism?

ChatGPT

Adding a feedback mechanism involves adjusting your algorithm's parameters based on its recent performance. This can help to make the algorithm more adaptive to changing market conditions. Here's how you can implement a feedback mechanism for the dominant cycles extraction process:

* Define a Performance Metric:
  + You need a quantitative measure to assess how well the cycle-based predictions/indications match the real market movements.
  + This could be the mean squared error between the predicted values and the real values, the number of correct directional predictions, or any other metric that makes sense in your context.
* Monitor Performance Over Rolling Windows:
  + Calculate the performance metric over recent data, e.g., over the last 50 or 100 bars.
  + This gives an idea of how well the algorithm is performing in the near term.
* Set Performance Thresholds:
  + Define thresholds for your performance metric that determine when an adjustment is needed.
  + For instance, if the accuracy of directional predictions drops below 50% for a defined period, it's a signal that adjustments might be needed.
* Adjust Algorithm Parameters:
  + Based on the performance metric, adjust the parameters of the dominant cycles extraction.
  + For instance, if the algorithm is consistently lagging, you might reduce the window size to make it more responsive. If it's too noisy, you might increase the window size.
  + Alternatively, you might adjust the weights given to different cycles or even the method used to extract cycles based on performance.
* Continuous Monitoring and Adjustment:
  + Continuously monitor the performance and make adjustments as necessary.
  + It's crucial to prevent overfitting: constantly changing parameters can make the algorithm overfit to the recent data and perform poorly in the future. Therefore, changes based on feedback should be made judiciously.
* Backtesting:
  + Anytime you introduce a new mechanism, it's essential to backtest the strategy over historical data to ensure that it adds value and doesn't lead to overfitting.
* Limit the Frequency of Adjustments:
  + To prevent over-reacting to short-term anomalies, limit how often adjustments can be made. For example, you might only allow adjustments once every 50 bars or when the performance metric crosses a certain threshold.
* Combine Feedback with External Signals:
  + Consider combining the feedback mechanism with external signals or market conditions. For example, during times of high volatility (as measured by indicators like ATR or VIX), you might allow more frequent adjustments.
* Log and Review:
  + Maintain a log of all adjustments made based on the feedback mechanism. Review this log periodically to understand how often adjustments are being made and whether they're effective.
* Safety Mechanisms:
* Introduce safety mechanisms to prevent extreme adjustments that might result from anomalies or data errors. For instance, set maximum and minimum bounds on key parameters.

Remember, the feedback mechanism should be used as a tool to improve the algorithm's adaptiveness and not as a crutch to make it fit the recent data perfectly. Always be cautious of overfitting.



do an implementastion

ChatGPT

Alright, let's begin by implementing a feedback mechanism for the enhanced\_theta\_function we discussed earlier. The primary objective of the feedback mechanism will be to adjust the rolling window based on the recent accuracy of the predictions. As a simple starting point, we'll use the mean squared error (MSE) between the predicted values (Neat\_Theta\_Function\_\*) and the actual future price movements as a performance metric. When the MSE is above a certain threshold, we'll adjust the window size for cycle extraction.

Here's a potential implementation:

python

Copy code

import numpy as np

from sklearn.metrics import mean\_squared\_error

def enhanced\_theta\_function(df, data\_sources, initial\_window=300):

# Define a function to calculate the dominant cycles

# ... (as you've defined earlier)

def feedback\_adjustment(df, data\_sources, current\_window):

# Calculate the performance metric for the last 'n' bars

n = 50 # You can adjust this value based on your preferences

recent\_data = df[-n:]

# Calculate the MSE between Neat\_Theta\_Function and future price movement

# Assuming the future price movement is represented by the next row's close price

mse = mean\_squared\_error(recent\_data['Close'][1:], recent\_data['Neat\_Theta\_Function\_Close\_' + str(current\_window)][:-1])

threshold = 0.001 # This is an arbitrary threshold, you should adjust based on your data

if mse > threshold:

# If the performance drops, adjust the window size

# For simplicity, we'll reduce it by 10%, but you can use any other logic

new\_window = int(current\_window \* 0.9)

# To ensure we don't go too low or too high:

new\_window = max(50, new\_window) # Set a minimum bound

new\_window = min(500, new\_window) # Set a maximum bound

return new\_window

else:

# If the performance is okay, keep the current window

return current\_window

current\_window = initial\_window

for \_ in range(10): # This is a loop for the feedback mechanism. 10 is arbitrary; choose what makes sense.

# Calculate dominant cycles and Neat\_Theta\_Function columns

# ... (as you've done in the original function)

# Apply the feedback mechanism to potentially adjust the window size

current\_window = feedback\_adjustment(df, data\_sources, current\_window)

return df

Note:

* This is a basic feedback mechanism. There are many ways to make it more sophisticated by incorporating additional conditions, metrics, or even machine learning models.
* The threshold and percentage adjustments are arbitrary and may require tweaking based on the characteristics of your data and the specific requirements of your project.
* This method continuously updates the window based on the recent performance. If frequent updates aren't desired, you can modify the loop conditions.



can catboost model be translated into pine script 5?

ChatGPT

As of my last knowledge update in September 2021, translating a model like CatBoost directly into Pine Script isn't feasible due to the differences in the purpose and capabilities of the two platforms:

* Complexity: CatBoost is a gradient boosting library that can generate quite complex models. These models are composed of many decision trees, which, when combined, can represent complex relationships in the data. Pine Script, on the other hand, is designed primarily for time series data analysis and simple calculations to support the development of custom technical indicators.
* Data Structures: Pine Script doesn't support the same data structures and algorithms as Python. Translating the tree structures, branching logic, and ensemble averaging of a model like CatBoost would be a formidable task.
* Performance: Even if one managed to translate a CatBoost model to Pine Script, performance would be a concern. Pine Script is interpreted on the client's side in TradingView's environment, and a complex model would likely be too slow.
* Functionality: Pine Script doesn't have native support for gradient boosting or tree-based algorithms. You'd essentially be hand-coding the entire prediction mechanism in Pine Script, which isn't its intended use.

However, if you're interested in using machine learning models for trading:

* One common approach is to build and train your model outside of TradingView using libraries like CatBoost, TensorFlow, or Scikit-learn. Once the model is trained, you can use its predictions to generate signals.
* These signals can then be fed into TradingView as an external data source or uploaded as a custom indicator, so you still get the benefit of visualizing and using them within the TradingView environment.

That said, if Pine Script 5 (or subsequent versions) introduces major changes or new capabilities in this regard, it's possible that new methodologies or techniques could be developed. Always keep an