Uniwersytet Jagielloński w Krakowie
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Karolina Góra
Nr albumu: 1143418

Badanie metod wyznaczania wykładnika Hursta

Praca licencjacka
na kierunku Informatyka

Praca wykonana pod kierunkiem
dr hab. Paweł Góra prof. UJ
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Kraków 2020

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Abstract

Implementation of algorithms *Rescaled Range* (R/S), *Detrended Fluctuation Analysis* (DFA) and *Detrended Moving Average* (DMA). Observed how, for different data series, each method creates a straight, which directional factor is a value of Hurst exponent and what possible discrepancies there can appear.

All methods are programmed in Python3, with use of ready numerical library (numpy) for calculations and graphic library (matplotlib) for generating final charts.

Abstrakt

Zaimplementowano algorytmy *Przeskalowanego zasięgu (*R/S), *Zniekształconej analizy fluktuacji* (DFA) i *Zniekształconej średniej ruchomej* (DMA). Przeprowadzono obserwacje, jak dla różnych ciągów danych, poszczególne metody wyznaczają proste, których współczynniki kierunkowe odpowiadają wartości wykładnika Hursta oraz jakie rozbieżności mogą się przy tym pojawić.

Wszystkie metody zostały zaprogramowane w języku Python3, z wykorzystaniem gotowych bibliotek numerycznych (numpy) do obliczeń oraz biblioteki graficznej (matplotlib) do generowania wykresów wynikowych.

Spis treści

1. Wstęp

1.1. O wykładniku Hursta

1.1.1. Geneza

Badanie serii danych zebranych w odstępach czasowych jest kluczową kwestią dla lepszego zrozumienia zjawisk występujących w naturze. Analizujemy wszystko co da się poddać obserwacji, w fizyce, biologii, a w szczególności kwestie finansowe i ekonomiczne. Podstawowym problemem pojawiającym się w kontekście analizy jest to, czy dane mają powiązania między sobą. Istnieje wiele technik, pozwalających na rozstrzygnięcie tego.

Wykładnik Hursta nazwany został po Haroldzie Edwinie Hurstcie, który był głównym członkiem zespołu prowadzącego badania. Można spotkać się z określeniami, że wyznacznik ten, jest ‘indeksem zależności’ lub ‘indeksem długo-zasięgowej (long-range) zależności’. W geometrii fraktalnej wykładnik Hursta jest oznaczany przez H , co jest uhonorowaniem Harolda Hursta oraz Ludwiga Otto Höldera przez Benoit’a Mandelbrota. Jest to oznaczenie bezpośrednio powiązane z wymiarem fraktalnym i mierzy ‘moc’ losowości ciągu. Dzięki temu możemy oszacować prawdopodobieństwo, czy trend danego ciągu danych utrzyma się.

1.1.2. Interpretacja wartości wykładnika

Wykładnik ma wartości prawidłowe, jeżeli znajduje się w zakresie $[0,1]$ i w nim wyznaczamy trzy klasy według których określamy nasz rezultat:

- ➔ $H=0.5$ – biały szum, co oznacza, że proces zachowuje się jak błądzenie losowe i nie posiada ukierunkowanego trendu
- ➔ $0<H<0.5$ – ciąg posiada negatywną korelację. Oznacza to, że dane będą często i szybko zmieniać swój kierunek – gdy ciąg jest rosnący zacznie szybko maleć i odwrotnie.
- ➔ $0.5<H<1$ – ciąg przedstawia długo-zasięgową zależność (Long Range Dependence). Oznacza to, że gdy ciąg ma tendencję wzrostową (lub malejącą), to prawdopodobnie ten trend się utrzyma. Badane zdarzenie potrzebuje silnego bodźca, aby spowodować zmianę trendu.

1.1.3. Metody wyznaczania wykładnika Hursta

Istnieje wiele metod pozwalających na wyprowadzenie wartości wykładnika H . Podstawową metodą jest, wyprowadzona przez samego Hursta, metoda ‘Przeskalowanego zakresu’ (Rescaled Range) w skrócie R/S. W tej pracy skupimy się właśnie na miej, oraz na dwóch od niej pochodnych DFA (Detrended Fluctuation Analysis) i DMA (Detrended Moving Average). Wszystkie te algorytmy odpierają się na podobnej filozofii, dzielenia ciągu badanych danych na

mniejsze i przemieszczaniu się po nich. Celem każdego z algorytmów, jak takie przetworzenie danych, aby po przebadaniu ciągu dla różnych wielkości podciągów, wyniki ułożyły się w linii prostej, której współczynnik kierunkowy będzie odpowiadał wartości naszego wykładnika H. Zależność między współczynnikiem kierunkowym prostej a wykładnikiem Hursta można opisać na dwa sposoby.

1. Na wykresie podwójnie logarytmicznym (log-log) przedstawiany jest wynik działania algorytmu. Prosta, dopasowana do widma mocy naszych danych, ma współczynnik kierunkowy α . Wtedy $H=\frac{1-\alpha}{2}$.
2. Na wykresie log-log przedstawione dane są serią ułamkową i tworzą zależność $F(L)\sim L^{\alpha}$. Wtedy dla ułamkowego szumu Gaussa $H=\alpha$. Z tego systemu odczytywania wykładnika Hursta będziemy korzystać.

1.2. Wyjaśnienie pojęć

1.2.1. Zjawisko fluktuacji

1.2.2. Random Walk

1.2.3 Odchylenie standardowe

1.3. Technologie i narzędzia

1.3.1. Język programowania

Algorytmy, którymi się zajmujemy, są metodami numerycznymi, dlatego przy wyborze języka programowania ważne było, aby radził on sobie z przetwarzaniem dużej ilości danych.

➔ **Python3 -** posiada technologię, pozwalającą na swobodne i łatwe odczytywanie danych tekstowych. Posiada modele do gromadzenia dużych zbiorów danych i wbudowane funkcję do ich analizy. Dobrze zaimplementowane biblioteki numeryczne i graficzne dają możliwość szybkiej implementacji naszych algorytmów i wizualizacji ich wyników.

1.3.2. Biblioteki pomocnicze

W ramach języka Python3 do implementacji naszych metod wykorzystane zostały dwie biblioteki wbudowane:

➔ **numpy** – jest to rozbudowana biblioteka metod numerycznych, zaprojektowana do wykonywania działań w najkorzystniejszej złożoności obliczeniowej. Metody wykorzystane w naszym przypadku:

- **average()** – funkcja zwracająca średnią wartości z tablicy.

- **polyfit()–** funkcja jako argumenty przyjmuje: tablicę indeksów dla danych (Index[]), tablicę danych (Data[]), odpowiadającym indeksom z tablicy pierwszej i stopień równania krzywej, jaka ma zostać dopasowana do podanych punktów (Index[i], Data[i]). Zwraca współczynniki wyżej opisanego równania.
- **log()** – zwraca logarytm z podanego argumentu. Może także przekształcić tablicę z wartościami, a tablicę z logarytmami tych wartości.
- **sqrt()** – zwraca pierwiastek z podanego argumentu.

➔ **pyplot** – jest to biblioteka funkcji do tworzenia wykresów danych. Wykorzystane metody to:

- **scatter()** – tworzy zestawienie punktowe danych
- **title()** – nadaje tytuł wykresu()
- **ylabel()** – nadaje nazwę osi Y
- **xlabel()** – nadaje nazwę osi X
- **text()** – pozwala na dodanie tekstu w polu wykresu. Funkcja ta została wykorzystana do wyświetlania na wykresie współczynnika kierunkowego prostej.
- **plot()** – rysuje na wykresie prostą z punktu a do punktu b .
- **show()** – generuje wykres o określonych wyżej wymienionymi funkcjami parametrach i wyświetla po wywołaniu funkcji.

1.3.3. Środowisko pracy

Jako środowisko pracy wykorzystany został PyCharm. Program jest przystosowany specjalnie do pracy z językiem Python. Pozwala on na estetyczne pisanie kodu i jego automatyczne formatowanie. Co najbardziej pomocne w naszym wypadku, posiada on tryb pracy pozwalający na systematyczny podgląd i zapis generowanych przez nasz program wykresów.

1.4. Dane

1.4.1. Typ danych

Dane, na których możemy wyznaczać współczynnik Hursta, muszą spełniać dwa założenia.

1. Zbiór danych musi być ciągiem, gdzie pomiary zostały wykonane w równych odstępach czasowych.
2. Jeśli to możliwe, dane powinny być ciągiem ułamkowym, dla zwiększenia dokładności wykonywanych obliczeń.

1.4.2. Wybrane zbiory danych

Wszystkie algorytmy, które zostaną przedstawione w drugiej części pracy, zostały wykonane dla różnych zbiorów danych i a w części trzeciej porównano wyniki, jakie dają poszczególne metody dla tych samych danych. Wybrane dane zapisane zostały w plikach:

- ➔ nile.txt – dane na temat wylewu rzeki Nil
- ➔ births.txt – dane na temat zmian ilości narodzin w czasie
- ➔ temp.txt –
- ➔ zurich.txt –
- ➔ assigment4.txt – dane zebrane specjalnie pod badanie algorytmu DFA

2. Implementacje

2.1. Funkcje pomocnicze

2.1.1. prepareData() - przygotowanie tablic z danymi to przetwarzania

Każdy z zaimplementowanych algorytmów korzysta z funkcji przygotowującej dane do przetwarzania.

```
#!/usr/bin/python
# -*- coding: iso-8859-2 -*-


def prepareData(file, start_length, end_length):
    array = []
    indexes = []
    with open(file) as data:
        for line in data:
            i, value = line.split()
            indexes.append(int(i))
            array.append(float(value))

    L = []
    if start_length:
        w = start_length
    else:
        w = len(array)

    while w / 2 > end_length:
        if w % 2 == 0:
            L.append(int(w / 2))
            w = w / 2
        else:
            w -= 1

    return indexes, array, L

```

funkcja przygotowująca dane tekstowe do przetwarzania przez algorytmy

Funkcja przyjmuje trzy atrybuty:

- ➔ file – nazwa pliku tekstowego, w którym zapisane są dane
- ➔ start_length – maksymalna długość przedziału, na które podzielimy dane
- ➔ end_length – minimalna długość przedziału, na jakie podzielimy dane

i zwraca trzy tablice:

- ➔ indexes[] – tablica indeksów, według których posortowane są badane dane
- ➔ array[] – tablica zebranych danych, które poddamy obserwacji
- ➔ L[] – tablica długości badanych przedziałów

Korzystając z możliwości importowania funkcji między plikami, dzięki *from prepareFiles import ** importujemy wyżej zaimplementowaną funkcję do użytku każdego z naszych algorytmów.

2.2. Algorytm R/S (Rescaled Range)

Algorytm *R/S*, jest to podstawowa metoda wyznaczania wykładnika Hursta. Opiera się ona na podziałach danych na równe części i modyfikacji danych w obrębie tych podziałów. Jak nazwa (*ang. rescaled range*) wskazuje, algorytm bada zachowanie asymptotycznego przeskalowania zakresu jako funkcji na seriach czasowych. Zależność między funkcją R/S, a wykładnikiem H przedstawia wzór:

$$\frac{R(n)}{S(n)} = C\,n^{H} \qquad n \to \infty \qquad C = const.$$

gdzie:

- **n** – to czas obserwacji (ilość punktów danych z serii czasowej), długość badanego aktualnie przedziału; przyjmuje się, że szukamy takich n, aby ilość mieszczących się w naszym zbiorze przedziałów o tej długości, była potęgą liczby 2;
- **R(n)** – to zakres *n* skumulowanych odchyleń od średniej;
- **S(n)** – jest odchyleniem standardowym wielkości (*n*) dla badanego segmentu podziału.

1. Aby rozpocząć procedurę algorytmu, musimy zacząć od przygotowania danych. W tej metodzie potrzebujemy tablicy ze zbiorem danych $X[\,]$ oraz tablicę $L[\,]$ z długościami *n*, dla których będziemy powtarzać algorytm.

```
9    #0
10   indexes, X, L = prepareData('nile.txt', None, 2)        # pobranie danych
11   N = len(X)                                               # N = ilość badanych danych
12   AVG = []                                                 # tablica, w której zbieramy końcowe wyniki dla każdego n
```

Implementacja deklaracji danych wyjściowych

Dla powyższego przykładu, w którym wykorzystujemy plik z ciągami danych zebranych na temat wylewów nilu, otrzymamy je w postaci:

$X = [1157.0, 1088.0, 1169.0, 1169.0, 984.0, \ldots, 1205.0, 1054.0, 1151.0, 1108.0]$ o długości $N = 662$ pomiarów, oraz wybrane długości (n) segmentów jako tablicę $L = [331, 165, 82, 41, 20, 10, 5]$.

Istotną częścią podzielenia naszych danych na przedziały o długości *n*, jest fakt, że nie mogą one na siebie zachodzić, czyli muszą być rozłączne. (1) Z tak przygotowanymi danymi, wykonujemy kolejne kroki w pętli po tablicy L.

```
for n in L:                                  # (1)
    R_S = []                                 # wartości R/S dla serii o długości n
    R = []                                   # zbiór największych różnic odchyleń w przedziałach długości n
    S = []                                   # zbiór wartości odchyleń standardowych dla przedziałów o długości n
    i = 0
    while i <= N - n:                        # (2)
        segment = X[i:i+n]                   # wybranie kolejnego segmentu o długości n
        m = (np.average(segment))            # wyliczenie średniej dla wybranego segmentu o długości n
        Y = []                               # Seria odchyleń dla danego segmentu
        Z = []                               # tablica sum odchyleń dla wszystkich serii o długości n
        for s in range(i, i+n):              # (3)
            Y.append(X[s] - m)
            Z.append(np.sum(Y))              # zapisanie pełnego odchylenia średniej dla przedziału

        R.append(max(Z) - min(Z))            # (6) Największa rónica odchyleń dla zbadanego podziału
        S.append(satndardDeviation(n, Y))    # (4) Odchylenie standardowe dla wyznaczonego przedziału
                                             # (5)
        i += n                               # wybranie początku następnego przedziału o długości n

    for r, s in zip(R, S):                   # (7)
        if s != 0:
            R_S.append(r/s)                  # (8) wyznaczenie R/S dla każdego przedziału o długości n

    AVG.append(np.average(R_S))              # (9) zapisanie średniej ze wszystkich zebranych wartości R_S[n]
```

Implementacja kroków od 2 do 10

2. (2) Dla każdego przedziału o długości n w serii danych X, obliczamy średnią: $m=\frac{1}{n}\sum_{i=1}^{n} X_i$. Rozpatrzmy część działań, gdy badamy ciąg dla $n=82$. Średnia dla pierwszego fragmentu danych będzie wynosić $m=1153.62$ *.
3. (3) W tym kroku następuje tytułowe przeskalowanie, gdzie dane w każdym przedziale regulujemy wyliczoną dla niego średnią: $Y_t=X_t-m$. Otrzymamy tym sposobem nowy pierwszy przedział o wartościach: $Y=[32.05,-36.95,44.05,\ \ ...,\ -43.95,71.05,71.05]$ *, które wcześniej stanowiły 82 pierwsze wartości ciągu X.
4. Na podstawie tablicy Y, budujemy ciąg sum odchyleń, których składnikami są wartości z tablicy Y o indeksach od 0 do t (pozycja aktualnie wyliczanego elementu tablicy Z): $Z_t=\sum_{i=1}^{t} Y_i$. Obie tablice Y i Z mają długość n oraz $Y_0 = Z_0$. Odnosząc się dalej do naszego przykładu z nilem $Z=[32.05,-4.90,39.15,\ \ ...,\ -142.10,-71.05,4.55e\text{-}13]$ *.
5. Obliczamy standardowe odchylenie S: $S(n)=\sqrt{\frac{1}{n}\sum_{i=1}^{n} Y_i^2}$. W tym celu mamy koleją funkcję pomocniczą *standardDeviation (10)*, aby kod był bardziej czytelny. Funkcja ta zwraca wartość odchylenia standardowego dla wyregulowanego przez nas ciągu X. Funkcja jako argumenty przyjmuje:
   - ➔ **size** – rozmiar przedziału, czyli size = n.
   - ➔ **array** – tablica o rozmiarze *n*, zawierająca wartości wybranego, uzgodnionego przy pomocy średniej przedziału, ciągu danych; wyliczony w punkcie 3 ciąg Y.

```
51    def satndardDeviation(size, array):
52        cumulative_sum = 0
53        for i in range(0, size):
54            cumulative_sum += array[i] * array[i]                # (10)
55        return np.sqrt(cumulative_sum / size)
```

Implementacja funkcji standardDeviation

Po tym jak funkcja zostanie wywołana (4), zwracaną przez nią wartość zapisywana jest w tabeli S, jako odchylenie standardowe dla badanego przedziału. Dla naszego przykładu S osiągnie wartość $S=89.05$ *.

6. Kiedy mamy już wyliczoną wartość S, do równania R/S brakuje nam już tylko wartości R. Jest ona równa największej różnicy odchyleń w badanym przedziale. Do jej wyprowadzenia używamy tablicy Z, którą zbudowaliśmy w kroku 4 i korzystając ze wzoru $R=max(Z)-min(Z)$, dodajemy naszą wartości do tablicy ze zbiorem R dla badanego n.
7. Punkty od 2-6 wykonujemy dla wszystkich kolejnych, rozłącznych przedziałów o długości $n \rightarrow$ czyli będzie ich $\lfloor \frac{N}{n} \rfloor$. Poniżej zaprezentowany jest zbiór wyników z powyższych kroków dla wszystkich przedziałów o $n=82$ mieszczących się w zbiorze X:

n = 82				
i	Badany zakres	Średnia przedziału [m] *	Największa różnica odchyleń w przedziale [R_i] *	Odchylenie standardowe [S_i] *
0	0 - 81	1153.62	1077.77	90.64022154676363
1	82 - 163	1073.99	2033.66	79.13
2	164 - 245	1129.02	1541.54	89.34
3	246 - 327	1141.97	1502.19	75.34
4	328 - 409	1125.08	1288.69	77.20
5	410 - 491	1190.05	1389.27	74.56
6	492 - 573	1226.57	1481.23	64.48
7	574 - 655	1143.27	64.48	69.20

Tabela 1: Wyniki kroków od 2-7 dla danych nilu, przy serii o przedziałach wielkości 82 danych czasowych

8. Kiedy mamy już zapełnioną tabelę z punktu 7, dla każdej pary R_i i S_i, obliczamy interesującą nas wartość R/S (8) i zapisujemy ją w tablicy i zapisujemy w tablicy R_S, dla danego wymiaru n. Tym krokiem możemy rozszerzyć naszą wcześniejszą tabelę, o wartości R/S:

n = 82				
i	Badany zakres	Największa różnica odchyleń w przedziale $[R_i]^*$	Odchylenie standardowe $[S_i]^*$	$[R/S]_i^*$
0	0 - 81	1077.77	90.64	11.89
1	82 - 163	2033.66	79.12	25.70
2	164 - 245	1541.54	89.34	17.25
3	246 - 327	1502.19	75.34	19.94
4	328 - 409	1288.69	77.20	16.69
5	410 - 491	1389.27	74.55	18.63
6	492 - 573	1481.23	64.47	22.97
7	574 - 655	64.48	69.20	20.89

Tabela 2: Wynik kroku 8 algorytu z obliczeniem kluczowej wartości RS

9. Na koniec obliczamy nasz ostateczny wynik dla wymiaru n, który jest równy wartości średniej obliczonych w punkcie wyżej R/S. Wartość tę zapisujemy w tablicy AVG[] (9).
10. Kroki 2-9 powtarzamy dla każdej wartości *n* z tablicy L. Ostatecznie otrzymamy tablicę wynikową AVG[], z wartościami odpowiadającymi każdemu z badanych n:

X		
j	$L_j = n$	$\frac{R/S}{n} = AVG_j^*$
0	331	72.89
1	165	41.37
2	82	19.25
3	41	11.02
4	20	5.92
5	10	3.41
6	5	1.99

Tabela 3: Zestawienie danych końcowych agorytmu, na podstawie których odczytujemy wniosek

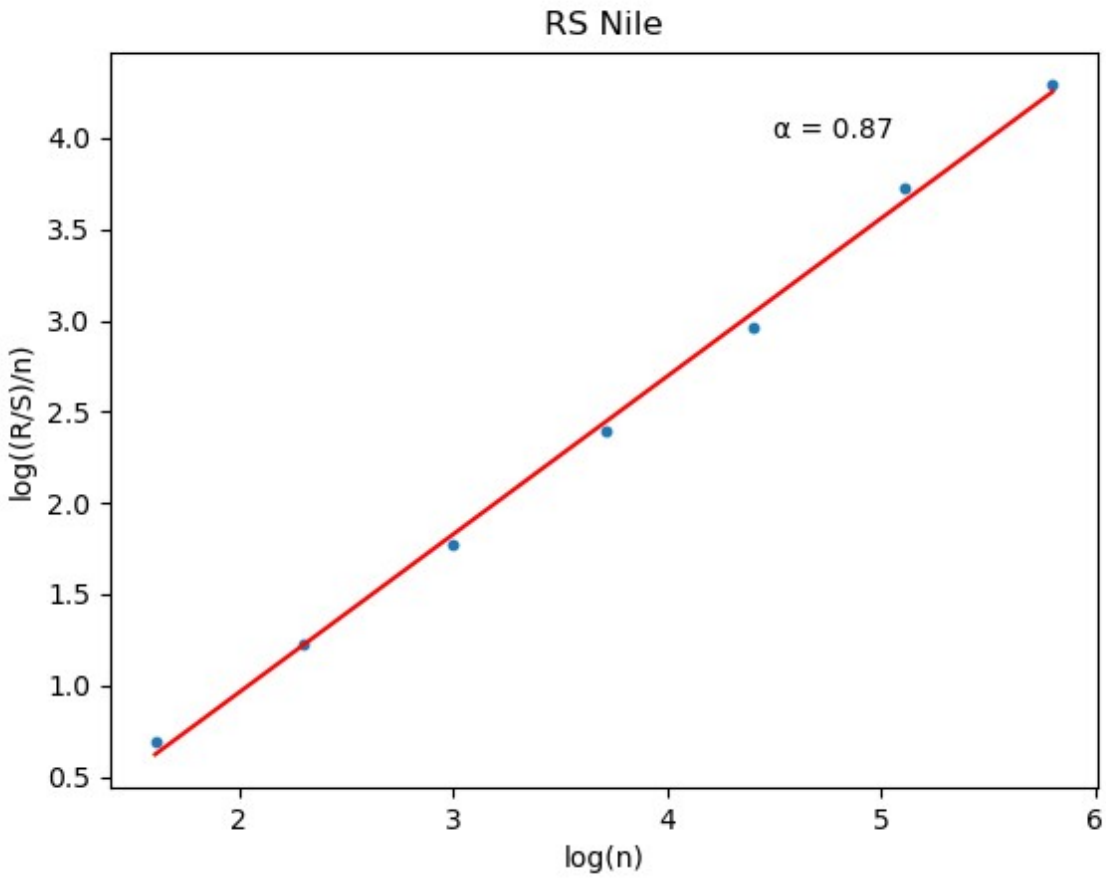
Aby odczytać z wykonanych obliczeń nasz wykładnik H, tworzymy wykres podwójnie logarytmiczny, dla długości segmentów danych (n) i średniej R/S jaką dla niej uzyskaliśmy. Poniżej

przedstawiony fragment kodu odpowiada za generowanie wizualizacji danych przy pomocy biblioteki pyplot.

```
38    plt.scatter(np.log(L), np.log(AVG), s=10)
39    plt.title('RS Nile')
40    plt.ylabel('log((R/S)/n)')
41    plt.xlabel('log(n)')
42    result = np.polyfit(np.log(L), np.log(AVG), 1)
43
44    plt.text(4.5, 4, '\u03B1 = {}'.format(round(result[0], 2)))
45    x1 = np.log(L[0])
46    x2 = np.log(L[-1])
47    plt.plot([np.log(L[0]), np.log(L[-1])], [result[0] * x1 + result[1], result[0] * x2 + result[1]], 'red')
48    plt.show()
```

Implementacja generowania wizualnego przedstawienia wyniku algorytmy RS

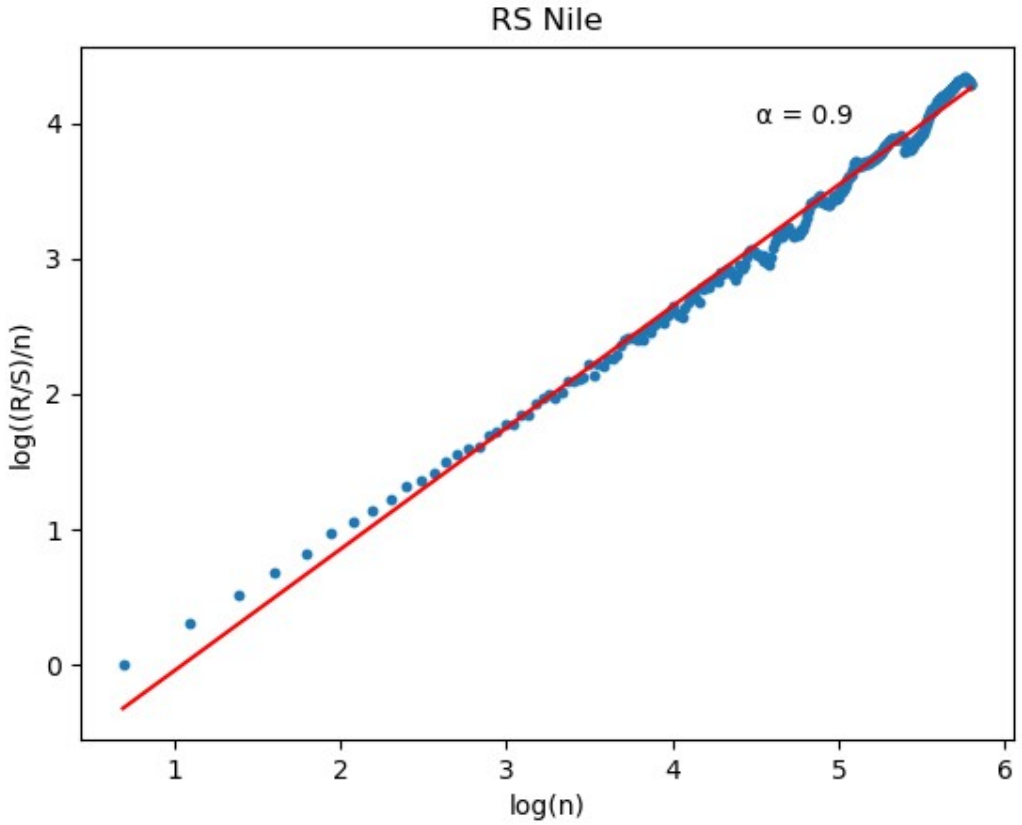
W opisany sposób otrzymujemy wykres zamieszczony poniżej, zawierający ostateczny wynik analizy danych nilu, które zebraliśmy w tabeli w kroku 10 naszego postępowania.



Zobrazowanie danych wynikowych nilu, dla wyżej określonych długości serii danych

Interesujące nas H, czyli współczynnik α prostej stworzonej przy pomocy regresji liniowej dla naszych końcowych danych wynosi ~0.87, czyli należy do zakresu od 0.5 do 1, co pozwala założyć, że w przyszłości trend danych utrzyma się.

Poprawność działania algorytmu możemy zaobserwować uruchomiając go dla dużo większej ilości badanych przedziałów. Interesującą obserwacją jest, że wynik nie zostaje zaburzony, dane wynikowe nadal układają się w niemalże idealnej prostej, a współczynnik zbliżył się do wartości 0.9, wskazując na jeszcze silniejszą możliwość zachowania trendu. Poniżej zamieszczony zostaje wykres, gdzie do obliczeń wykorzystano przedziały o kolejnych długościach $n \in [2, 331]$ przy czym $n \in \mathbb{Z}$, co daje 329 serii obliczeniowych.



Zobrazowanie danych wynikowych nilu dla 329 badanych serii

2.3. Algorytm DFA (Detrended Fluctuation Analysis)

Algorytm DFA jest, jak nazwa wskazuje, metodą analizy fluktuacji danych. Kluczowymi momentami w jego procesie są:

- ➔ modyfikacja danych, korzystając z tak zwanego Random Walk-u;
- ➔ wyznaczenie wartości fluktuacji dla podciągów danych i wyliczenie jej średniej.

1. Przygotowanie danych (0) wygląda podobnie jak w algorytmie R/S. Korzystając z funkcji pomocniczej *prepareData* odczytujemy:
   - ➔ **indeksy** jakimi oznaczone są zebrane w we wskazanym pliku pomiary. W przeciwieństwie do algorytmu RS, teraz będą one odgrywać istotną rolę w obliczeniach i prezentacji danych;
   - ➔ Tablicę z wartościami pomiarów, które będziemy badać. Dla uniknięcia późniejszej kolizji oznaczeń, nazwiemy ją ***D;***
   - ➔ Tablicę **L** z wybranymi długościami przedziałów, na jakie będziemy dzielić dane w poszczególnych seriach badawczych. Podobnie jak w metodzie RS, przyjmujemy założenie, że przy N równym ilości wszystkich wczytanych danych (długość tablicy D), szukamy takich długości n, że ilość podziałów $\lfloor \frac{N}{n} \rfloor$ jest potęgą 2.

   Odnosząc się do naszego przykładu z Nilem, przygotowane dane będą takie same jak w przypadku algorytmu RS.

   Pierwsza operacja jaką wykonujemy jest przygotowaniem do zamiany danych na Random Walk. W tym celu obliczamy średnią przygotowanego zbioru danych D (1). Dla naszego przykładu średnia ciągu wyniesie $avg = 1148.20$ [*].

```
 8   def DFA():
 9       # 0
10       indexes, X, L = prepareData('nile.txt', None, 2)
11       N = len(X)
12
13       # 1 średnia wszystkich danych
14       avg = np.average(X)
```

Implementacja przygotowania danych do przetwarzania

2. Zaopatrzeni w dane z punktu pierwszego, możemy przygotować zbiór danych D do obróbki, zamieniając go na Random Walk, czyli ruch losowy lub błądzenie losowe. Działanie to ma na celu pozwolić na ułożenie danych, które potrafią być mocno rozproszone i przedstawić je w mniej chaotycznej postaci. Sama ta operacja pozwoli zaobserwować potencjalne zachowanie trendu.

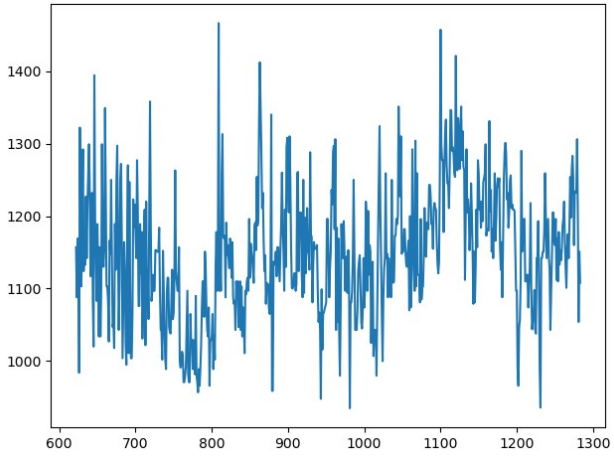
   Do uzyskania wspomnianego przedstawienia danych stosujemy prosty wzór, który opisuje, że każdemu elementowi ciągu D, przypisujemy jego sumę z poprzedzającymi go danymi, pomniejszonymi o wyliczoną wcześniej średnią całego ciągu.

$$D_n = \sum_{k=1}^{n} \left(d_k - \langle d_n \rangle \right) \quad \leftarrow \quad \langle d_n \rangle = avg \quad (2)$$

```
16      # 2 zmiana danych na random walk
17      randomWalk = []
18      cumulative_sum = 0.0
19      for i in range(0, N):
20          cumulative_sum += array[i] - avg
21          randomWalk.append(cumulative_sum)
```

Implementacja zamiany danych na tzw. Random Walk

Poniżej przedstawione zostało zestawienie, jak graficznie prezentowały się dane oryginalne (rys.1), a na jakich danych będziemy wykonywać dalsze obliczenia, dzięki zamianie postaci danych na Random Walk.



rys 1. Dane oryginalne [X]

rys 2. Dane zmodyfikowane - random walk

3. Z tak przygotowanymi danymi, zaczynamy działania należące do algorytmu DFA.

```
23      # petla do wybierania długości segmentów
24      F_avg = []
25      for segment_size in L:
26          # 3
27          temp_array = randomWalk.copy()
28          X = indexes.copy()
29          i = 0
30          k = indexes[0]
31          F = []
32          while i <= N - segment_size:
33              # znalezienie prostej w segmencie: line[0]=a; line[1]=b;
34              line = np.polyfit(X[0:segment_size], temp_array[0:segment_size], 1)
35
36              del temp_array[0:segment_size]
37              del X[0:segment_size]
38
39              # 4 wyliczenie F
40              F.append(calculateF(line, segment_size, i, k, randomWalk))
41              k = k + segment_size
42              i = i + segment_size
43
44          # 5 obliczenie sredniej fluktuacji dla danej dlugosci segmentu
45          F_avg.append(np.average(F))
46          print(segment_size, np.average(F))
```

Implementacja obliczeń algorytmu DFA

Ponieważ działanie algorytmu opera się na badaniu przedziałów, dzielimy zbiór danych na podzbiory o długości n, gdzie n jest wartością przygotowanej w koku 1 tabeli L. Tak jak wcześniej, rozpatrzymy działanie algorytmy dla $n=82$.

Dodatkowym manewrem jaki zostaje wykonany, jest utworzenie przy każdej nowej serii badawczej kopii tablicy z wartościami RandomWalku w tablicy Y (sugerując się nazewnictwem osi układu kartezjańskiego), ponieważ są to wartości oraz kopii tablicy indeksów, analogicznie do tablicy X, ponieważ dane te odpowiadają osi X. Wykonujemy te operacje, ponieważ w czasie działania algorytmu, będziemy usuwać przebadane już ciągi danych, a ponieważ algorytm wykonujemy wielokrotnie, nie chcemy utracić oryginalnego zbioru danych.

Ostatnią zmienną jaką przygotowujemy dla danej serii, jest tablica F, w której zbierane będą wartości fluktuacji dla poszczególnych przedziałów wielkości n.

4. Z takim przygotowaniem dla serii, rozpoczynamy badanie jej przedziałów. W każdym przedziale musimy odnaleźć lokalny trend. W tym celu do każdego segmentu o długości n, dopasowujemy prostą.

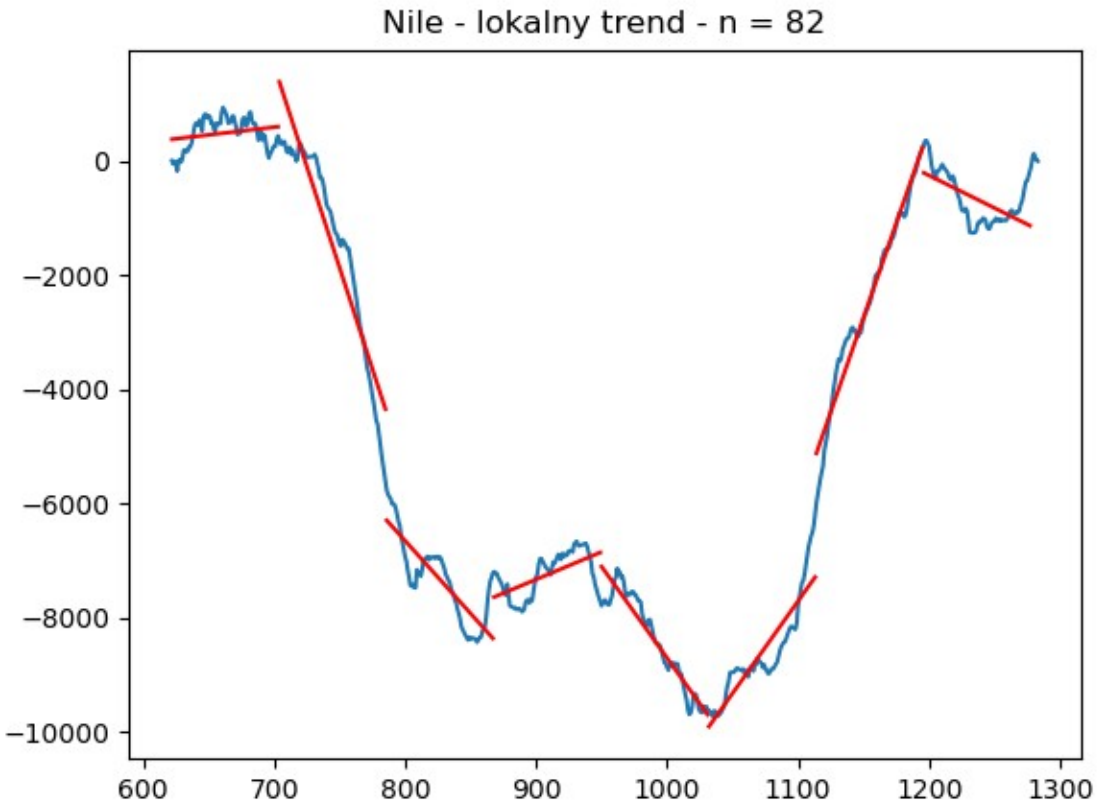
$$Y_i = a \cdot i + b \quad \leftarrow \quad i \in X \quad (4)$$

Po uzyskaniu interesujących nas współrzędnych, przechodzimy do kroku 5, w którym na ich podstawie obliczymy fluktuację badanego przedziału. Kroki 4-5 wykonujemy dla wszystkich segmentów o długości n. Zestawienie segmentów i współrzędnych prostych określających ich trendy można zobaczyć w tabeli poniżej.

n = 82					
id	Badany przedział	Zakres indeksów badanego przedziału [X]	Wartości skrajne przedziału [Y] *	a *	b *
0	0 - 81	622 - 703	8.80, 444.40	2.72	-1311.41
1	82 - 163	704 - 785	372.20, -5641.20)	-70.82	51250.62
2	164 - 245	786 - 867	-5767.40, -7213.79	-25.50	13750.07
3	246 - 327	868 - 949	-7196.00, -7724.79	9.64	-1.60
4	328 - 409	950 - 1031	-7791.59, -9620.00	-32.05	23341.44
5	410 - 491	1032 - 1113	-9626.19, -6188.60	3.23	-4.32
6	492 - 573	1114 - 1195	-5990.79, -237.81	6.61	-7.88
7	574 - 655	1196 - 1277	298.61, -166.78	-1.14	1.34

Tabela 4: Wartości prostych określających trend lokalny w przedziałach o wielkości 82 danych

Dzięki pracy na kopiach danych, utworzonych w kroku 3, po wyliczeniu współrzędnych prostej dla danego przedziału, możemy usunąć go ze zbiorów X i Y. Sprawia to, że kolejny przedział, który będziemy badać znajdzie się na początku tablic X i Y, a to pozwala nam na przechodzenie kroku 4 zawsze wybierając pierwsze n wartości z tych tablic, dzięki czemu zmniejszamy prawdopodobieństwo pomyłki przy szukaniu właściwych indeksów.



Zobrazowanie trendu na badanych przedziałach

5. Na podstawie wyprowadzonego równia prostej dla aktualnego przedziału, wyprowadzamy wartość fluktuacji F, korzystając z poniższego wzoru:

$$F(n)=\sqrt{\frac{1}{n}\sum_{i=i_0;\,j=k}^{i_0+n-1;\,j=k+n-1}\left(RW_j-i\cdot a-b\right)^2} \quad (8)$$

$$i_0=X_0;\quad k=id\cdot n;\quad RW-RandomWalk$$

Dla przykładu, biorąc pierwszy ciąg 82 danych, równanie wyglądało by w następujący sposób:

$$F(82)=\sqrt{\frac{1}{82}\sum_{i=622;\,j=0}^{i=703;\,j=81}\left(RW_j-i\cdot 2.72+1211.41\right)^2}$$

W tym momencie znów korzystamy z pierwotnie przygotowanego do obliczeń ciągu danych, czyli Random Walk-u. Ponownie mamy zaimplementowaną funkcję pomocniczą *calculateF*, która zwraca nam wartość fluktuacji dla określonego przedziału. Funkcja przyjmuje 5 argumentów:

➔ **line** – tablica ze współrzędnymi *a* i *b*, które otrzymaliśmy w kroku 4 przy użyciu funkcji *polyfit*, do wyprowadzenia równania prostej;

➔ **n –** wielkość badanego przedziału

➔ **i –** określa początkową wartość x, w równaniu prostej, dla badanego przedziału danych

➔ **k –** określa indeks początkowy, od którego należy zacząć pobierać dane z tablicy danych RW;

➔ **RW** – pełna tablica danych – Random Walk

```
77  # 8
78  def calculateF(line, n, i, k, RW):
79      segment_sum = 0
80      for n in range(i, i + n):
81          segment_sum += (RW[n] - (line[0] * k) - line[1]) * (RW[n] - (line[0] * k) - line[1])
82          k = k + 1
83
84      F = np.sqrt((1 / n) * segment_sum)
85      return F
```

Implementacja funkcji pomocniczej wyliczającej wartość fluktuacji

Wartość fluktuacji dla poszczególnych segmentów o wielkości *n* zbieramy w tablicy *F* (5).

n = 82					
id	a *	b *	i_0	k	Wartość fluktuacji $[F_{id}]$ *
0	2.72	-1311.41	622	0	270.09
1	-70.82	51250.62	704	82	587.09
2	-25.50	13750.07	786	164	403.90
3	9.64	-1.60	868	246	313.87
4	-32.05	23341.44	950	328	249.05
5	3.23	-4.32	1032	410	380.49
6	6.61	-7.88	1114	492	290.24
7	-1.14	1.34	1196	574	364.27

6. Po przejściu przez wszystkie segmenty i uzyskaniu pełnej tablicy F, wyprowadzamy średnią fluktuację dla podziałów o wielkości *n*.

```
57          # 6 obliczenie sredniej fluktuacji dla danej dlugosci segmentu
58          F_avg.append(np.average(F))
```

Wyliczenie średniej fluktuacji dla podziałów o wielkości n

Jest to wartość końcowa jaką chcemy otrzymać z naszych obliczeń dla wybranego n. Dodajemy ją do tablicy F_avg (6), w której zbieramy średnie wartości fluktuacji, dla każdej badanej wielkości n.

7. Kroki od 3-6 powtarzamy dla każdej wielkości n, która zawiera się w tablicy L.
8. Po wykonaniu całego algorytmu dla każdej wartości z tablicy L, tworzymy zestawienie w postaci wykresu podwójnie logarytmicznego (*log-log*), gdzie przedstawiamy zależność długości (n) segmentów do średniej fluktuacji dla niej obliczonej (7).

D		
j	$n = L_j$	$F(L) = F\_avg_j^*$
0	331	1592.46
1	165	854.18
2	82	357.37
3	41	226.09
4	20	104.35
5	10	59.97
6	5	29.31

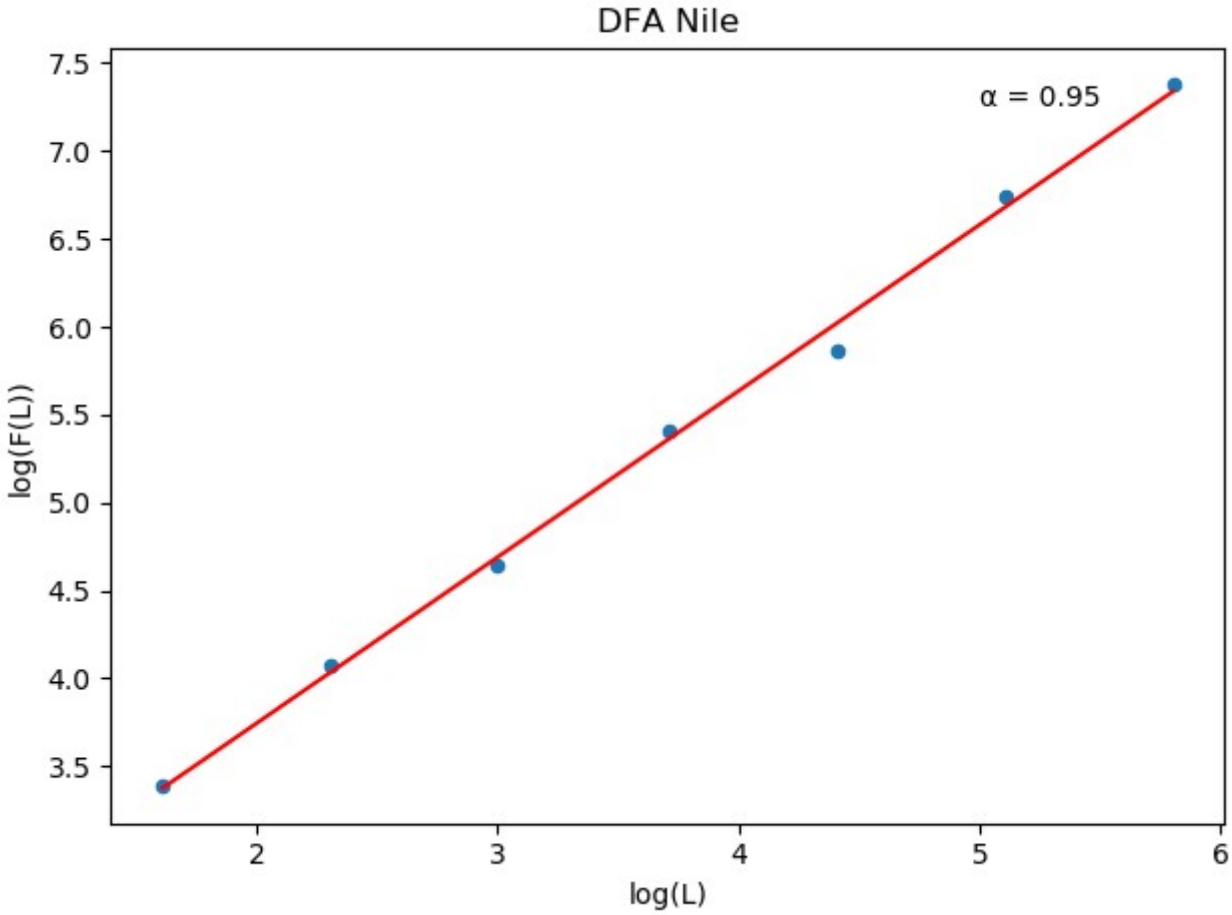
Tabela 5: Zestawienie zależności średniej fluktuacji przedziałów od ich długości n

Z tak przygotowanych danych generujemy wizualizację wyników, przy pomocy poniższej implementacji,

```
48    # 6 double logaritmic plot
49    plt.scatter(np.log(L), np.log(F_avg), s=20)
50    plt.title('DFA nile')
51    plt.ylabel('log(F(L))')
52    plt.xlabel('log(L)')
53
54    result = np.polyfit(np.log(L), np.log(F_avg), 1)
55    print('alfa = ', result[0])
56    print(result)
57
58    plt.text(5, 7.25, '\u03B1 = {}'.format(round(result[0], 2)))
59    x1 = np.log(L[0])
60    x2 = np.log(L[-1])
61    plt.plot([np.log(L[0]), np.log(L[-1])], [result[0]*x1+result[1], result[0]*x2+result[1]], 'red')
62    plt.show()
```

Implementacja generowania wizualnego przedstawienia wyniku algorytmy DFA

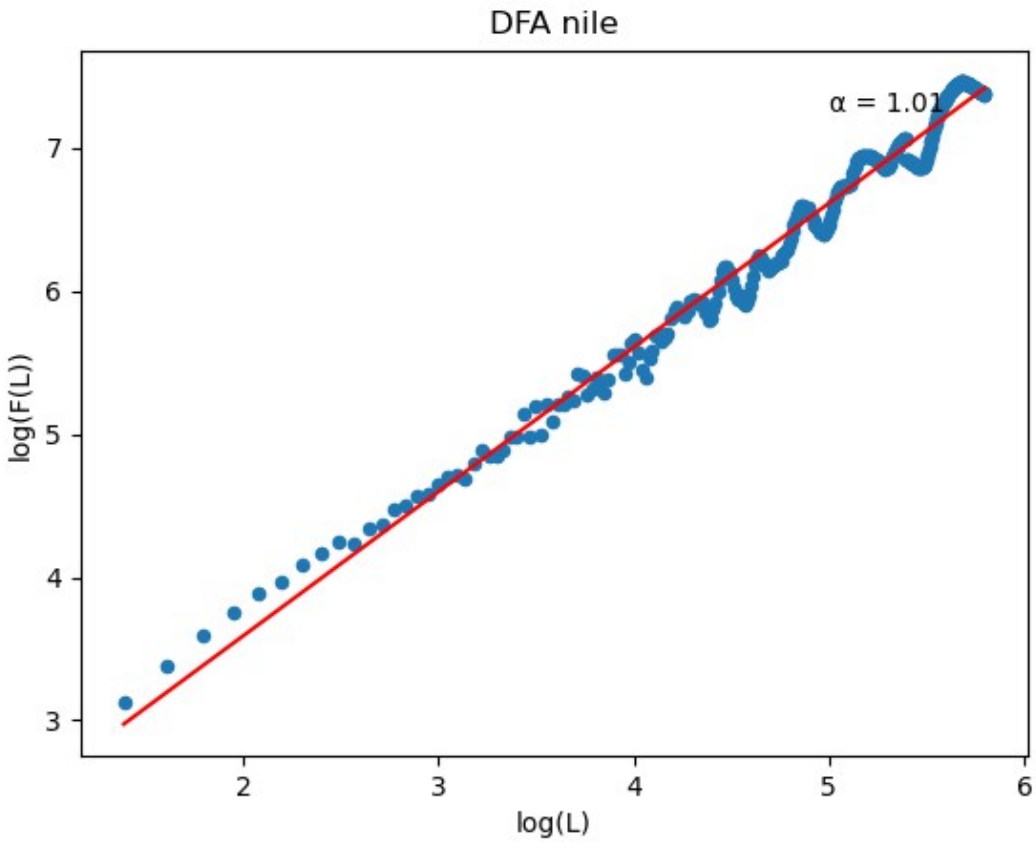
której efekt można zobaczyć na wykresie poniżej.



Wynik algorytmy DFA dla danych nilu, przy określonych wyżej przedziałach czasowych

Możemy z niego wyczytać, że wykładnik Hursta wynosi 0.95. Oznacza to wysokie prawdopodobieństwo, że trend danych utrzyma się, ponieważ wynik znajduje się w zakresie (0.5, 1]. Co dobre, wynik tej metody nie odstaje mocno od wyniku jaki uzyskaliśmy z metody RS. Możemy zatem założyć, że obie metody działają poprawnie.

Przeprowadzając podobne doświadczenie jak w metodzie RS i wywołując algorytm DFA dla 327 kolejnych, rożnych przedziałów czasowych, zaobserwujemy zbliżenie się wykładnika Hursta do 1, a nawet jego przekroczenie. Zasadniczo takie rezultaty nie powinny się pojawiać, ale obserwując widoczne zmiany w układzie danych, możemy założyć, że przyszły ruch danych obniży ponownie współczynnik, jednak będzie on stale oscylował w okolicy 1, dopóki nie nastąpi gwałtowna zmiana zachowania danych, współczynnik zacznie spadać i trend danych się zmieni.



Rezultat algorytmu DFA dla danych nilu, przy segmentach wielkości od 4 do 331

2.4. Algorytm DMA (Detrended Moving Average)

Algorytm DMA jest widocznie najprościej zbudowany i łatwy do implementacji. Dane ulegają wygładzeniu za pomocą tytułowej ruchomej średniej. Cel tej operacji jest taki sam jak w przypadku stosowania błądzenia losowego przy algorytmie DFA – uzależnienie od siebie danych i ich uporządkowanie.

O ile jest to metoda w implementacji banalna, w samym zastosowaniu już nie. Pierwszą czynnością, która może sprawić kłopot w wypadku tego algorytmu, jest wybór zakresu wielkości n segmentu, okienka, którym będzie się przemieszczać po wczytanych danych. Druga sprawa, to dobór danych, na których przeprowadzamy obliczenia.

1. Do uruchomienia algorytmu przygotowujemy się w podobny sposób jak wcześniej – pobieramy dane z pliku. Interesują nas wartości zebranych obserwacji w tablicy **X**, oraz tablica **L** ze specjalnie wybranymi wielkościami, powiedzmy, okienka badawczego, którym będziemy przemieszczać się po szeregu czasowym. Tym razem jednak wartości te będą następującymi po sobie kolejnymi wielkościami, czyli np. zbiór $L=[300,301,...,400]$.

```
8   def DMA():
9       # 0
10      indexes, X, L = prepareData('nile.txt', 20, 4)
11      N = len(X)
12
13      modifiedStandardDeviation = []
14      for n in L:
15          # 1 srednia ruchoma dla przedziałów dlugosci n
16          i = n-1
17          movingAverage = X.copy()
18          while i < N:
19              cumulative_sum = 0.0
20              for k in range(0, n):
21                  cumulative_sum += X[i-k]
22              movingAverage[i] = cumulative_sum/n
23              i += 1
24
25          #2
26          sum = 0.0
27          for i in range(n, N):
28              sum += (X[i-1] - movingAverage[i-1]) * (X[i-1] - movingAverage[i-1])
29          modifiedStandardDeviation.append(np.sqrt(sum / (N - n)))
```

Implementacja algorytmu DMA

2. W algorytmie DMA nie dzielimy zbioru danych X na przedziały o równej długości, lecz wybieramy długość segmentu, na obszarze którego w danym kroku będziemy liczyć średnią i przesuwać się z każdym krokiem o jeden element, aż dojdziemy do końca naszej tablicy z danymi.

   Celem tego kroku, jest obliczenie średnich, dla elementów od n do N w tablicy X. Pętlę taką wykonujemy stosując wzór, który opisuje, że elementom tablicy X, zaczynając od elementu

i=n (przy założeniu, że tablicę indeksujemy od 1), przypisujemy średnią $\tilde{x}_i$ jako średnią z sumy wyrazu i-tego i n-1 poprzedzających go elementów.

$$\tilde{x}(i)=\frac{1}{n}\sum_{k=0}^{n} X(i-k) \quad (1)$$

```
16          # 1 srednia ruchoma dla przedziałów dlugosci n
17          i = n-1
18          movingAverage = X.copy()
19          while i < N:
20              cumulative_sum = 0.0
21              for k in range(0, n):
22                  cumulative_sum += X[i-k]
23              movingAverage[i] = cumulative_sum/n
24              i += 1
```

Implementacja obliczania średniej ruchomej ale segmentu o wielkości n

Czyli np. dla n=300, pierwszą wartość średniej ruchomej wyliczymy dla X(299), sumując wartości od X(0) do X(299) i dzieląc tę sumę przez 300. Kolejno obliczymy średnią ruchomą dla X(300) sumując wartości od X(1) do X(300) i dzieląc przez 300. Tak przesuwając się, dojdziemy to wyliczania ostatniej wartości średniej dla X(661), gdzie sumę elementów od X(360) do X(661) podzielimy przez 300.

3. Kolejnym krokiem, jest wyliczenie zmodyfikowanego odchylenia standardowego. Jest to połączenie modyfikacji danych wejściowych o wyliczoną w kroku 2 średnią i na ich podstawie obliczania odchylenia dla aktualnie rozpatrywanego n. Działanie to przedstawia wzór:

$$\sigma_{DMA}(n)=\sqrt{\frac{1}{N-n}\sum_{i=n}^{N}(x_i-\tilde{x}_i)^2¿}$$

Jak dobrze widać, wyrażenie po którym sumujemy, jest częścią równania, która odpowiada za modyfikację szeregu danych X o wyliczoną wyżej średnią.

```
26          #2
27          sum = 0.0
28          for i in range(n, N):
29              sum += (X[i-1] - movingAverage[i-1]) * (X[i-1] - movingAverage[i-1])
30          modifiedStandardDeviation.append(np.sqrt(sum / (N - n)))
```

Implementacja wyprowadzania zmodyfikowanego odchylenia standardowego dla segmentu o wielkości n

4. Kroki 2 i 3 powtarzamy dla kolejnych wielkości n z tablicy L.

   Wykonując algorytm dla danych Nilu, przy n z zakresu [300, 400), otrzymamy poniższe wyniki:

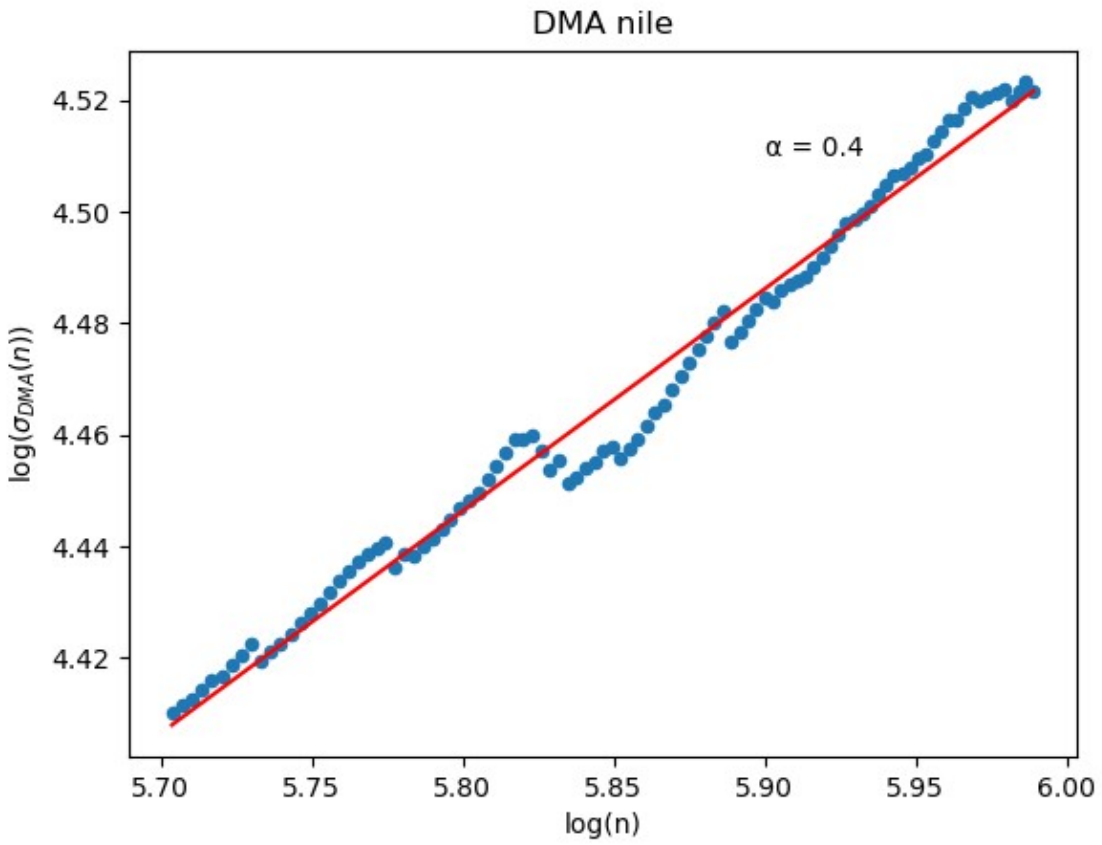
$n \in L$	$\sigma(n)$ *
300	82.29
301	82.41

302	82.47
303 - 397	...
398	92.13
399	91.10

5. Na koniec przedstawiamy dane na wykresie podwójnie logarytmicznym (3).

```
34      # 3 double logaritmic plot
35      plt.scatter(np.log(L), np.log(modifiedStandardDeviation), s=20)
36      plt.title('DMA nile')
37      plt.ylabel(r'log($\sigma_{DMA}(n)$)')
38      plt.xlabel('log(n)')
39
40      result = np.polyfit(np.log(L), np.log(modifiedStandardDeviation), 1)
41      print('alfa = ', result[0])
42      print(result)
43
44      plt.text(4, 4.46, '\u03B1 = {}'.format(round(result[0], 2)))
45      x1 = np.log(L[0])
46      x2 = np.log(L[-1])
47      plt.plot([np.log(L[0]), np.log(L[-1])], [result[0] * x1 + result[1], result[0] * x2 + result[1]], 'red')
48      plt.show()
```

Implementacja generowania wizualizacji wyniku algorytmu DMA



Wynik algorytmu DMA dla danych nilu przy n z zakresu 300-399

Przy pierwszym spojrzeniu rzuca się w oczy różnica wartości współrzędnej α i chaotyczność układu danych, w porównaniu z wynikami metod RS i DFA. Tak jak wspomniałam wcześniej, jest to metoda w zastosowaniu niełatwa. Ciężko bowiem jest dobrać rozmiary okienka badawczego do zebranych przez nas danych. Mimo, że jest to metoda oficjalnie uznawana za jedną z metod do wyznaczanie wykładnika Hursta, jest także tą najczęściej odrzucaną, ponieważ jej rezultat zazwyczaj odstaje silnie od innych algorytmów. Jak widać w omawianym tutaj przykładzie, przy algorytmie DMA wylądowaliśmy wręcz po przeciwnej stronie wartości 0.5, kiedy przy RS i DFA dążyliśmy do okrągłej wartości 1.

Dodatkową wadą rezultatów jakie daje ta metoda, jest brak ścisłej zależności w przedstawionych danych. Wybierając okna badawcze o różnych rozpiętościach i w różnej ilości, możemy zbliżyć się do $\alpha=0$, a nawet zejść poniżej, ale równocześnie jesteśmy w stanie osiągnąć wyniki z kategorii $\alpha>4$ gwałtownie wychodzące poza oczekiwany przez nas zakres.

3. Analiza różnych szeregów czasowych

Żeby wyżej opisane implementacje nie sprawiały wrażenia, iż działają specjalnie pod nasz wiodący przykład danych o wylewach Nilu, w tym rozdziale przyjrzymy się ich działaniu dla innych szeregów czasowych. Po algorytmach RS i DFA oczekujemy mniej lub bardziej zgodności, co do wartości do której zdążają. Przy algorytmie DMA, jako tym najbardziej zawodnym, spróbujemy odnaleźć takie wielkości okien badawczych, które zbliżą wartość wykładnika do rezultatów wcześniejszych metod.

4. Podsumowanie

Przypisy

* – wszystkie podane wyniki są w przybliżeniu do 2 miejsc po przecinku.

Literatura

1. WYKORZYSTANIE WYKŁADNIKA HURSTA DO PROGNOZOWANIA ZMIAN CEN NA GIEŁDZIE PAPIERÓW WARTOŚCIOWYCH
2. STATISTICAL PROPERTIES OF OLD AND NEW TECHNIQUES IN DETRENDED ANALYSIS OF TIME SERIES
3. Wikipedia – Hurst Exponent
4. UJ wykłady Paweł Góra
5. Metody Analizy długozasięgwej
6. Detrending Moving Average variance: a derivation of the scaling law
7. https://blogs.cfainstitute.org/investor/2013/01/30/rescaled-range-analysis-a-method-for-detecting-persistence-randomness-or-mean-reversion-in-financial-markets/
8. https://quantdare.com/demystifying-the-hurst-exponent/