

Brownian motion - symulacja Monte Carlo  
Sprawozdanie

Szymon Oplatek

29 maja 2022

Spis treści

1	Wstęp	1
2	Opis implementacji	2
2.1	Wykorzystane technologie . . . . .	2
2.2	Analiza kodu . . . . .	2
2.2.1	Importowanie bibliotek . . . . .	2
2.2.2	Funkcja darken_color() . . . . .	2
2.2.3	Klasa Particle . . . . .	3
2.2.4	Animacja . . . . .	4
2.2.5	Agregacja statystyk . . . . .	6
3	Analiza danych	8
3.1	Analiza geometryczno-probabilistyczna . . . . .	10
3.2	Wnioski . . . . .	11

1 Wstęp

Projekt polegał na zaprogramowaniu w języku Python symulacji ruchu Browna cząsteczki zgodnie z następującymi założeniami:

- 1. Cząsteczka, której ruch będziemy śledzić, znajduje się w początku układu współrzędnych  $(0, 0)$ ;
- 2. W każdym ruchu cząsteczka przemieszcza się o stały wektor o wartości 1;
- 3. Kierunek ruchu wyznaczać będziemy losując kąt z zakresu  $[0, 2\pi]$ ;
- 4. Współrzędne kolejnego położenia cząsteczki wyliczać będziemy ze wzorów:

$$x_n = x_{n-1} + r \cdot \cos(\phi)$$
$$y_n = y_{n-1} + r \cdot \sin(\phi)$$

Gdzie:

$r$  — długość jednego kroku

$\phi$  — kąt wskazujący kierunek ruchu w odniesieniu do osi OX

- 5. Końcowy wektor przesunięcia obliczymy ze wzoru:

$$|s| = \sqrt{x^2 + y^2}$$

## 2 Opis implementacji

Z założeń wynika, że symulacja będzie przeprowadzana w przestrzeni dwuwymiarowej. Polegać będzie ona na sekwencyjnym losowaniu kolejnych pozycji cząsteczki (tj. par  $(x, y)$ , gdzie zmienne  $x$  oraz  $y$  są typu float) aby po każdej iteracji obliczyć metrykę pod postacią długości wektora przesunięcia cząsteczki względem początku układu współrzędnych, czyli wartości typu float równej  $\sqrt{x_n^2 + y_n^2}$ , gdzie  $x_n$  oraz  $y_n$  stanowią współrzędne pozycji cząsteczki po  $n$  ruchach. Przedstawiona symulacja jest więc formą implementacji metody Monte Carlo.

Mówiąc o wykonaniu symulacji będę mieć od teraz na myśli wykonanie ustalonej, skończonej liczby ruchów cząsteczki. Każda nowa symulacja rozpoczyna się w punkcie  $(0, 0)$ . Długości wektorów przesunięć, oznaczane od teraz  $s_i(n)$ , gdzie  $i$  odpowiada  $i$ -tej symulacji, a  $n$  odpowiada  $n$ -temu krokowi cząsteczki, stanowiąc będą podstawę przeprowadzanej na koniec projektu analizy. Zmagazynowane zostaną one w zmiennych typu DataFrame (obiektach pochodzących z pakietu pandas) gdzie każda  $i$ -ta kolumna danych będzie zawierać wartości  $s_i(n)$  dla danej symulacji. Takie szeregi danych zostaną przeliczone na np. średnie

$$S_n = \sum_{i \in I} \frac{s_i(n)}{|I|}$$

Przeprowadzimy sto symulacji po sto ruchów oraz setkę symulacji tysiąca ruchów. Odczytane wyniki statystyczne oraz powstałe w trakcie analizy geometrycznej problemu obiekty otrzymają reprezentację graficzną.

Ponadto w ramach wizualizacji zdecydowałem się sporządzić animację kolejnych ruchów cząsteczki w symulacji.

### 2.1 Wykorzystane technologie

Kod napisany jest w środowisku Python 3.0 i wykorzystuje biblioteki takie jak:

1. Numpy - Pochodzą z niej wykorzystywane przeze mnie obiekty matematyczne takie jak przybliżenie liczby  $\pi$ , operacja pierwiastkowania czy próbka ze zmiennej losowej o rozkładzie jednostajnym.
2. Pandas - Dostarcza klasę DataFrame, za której pomocą dokonuję analizy danych.
3. Matplotlib - Umożliwia tworzenie reprezentacji graficznych dla zgromadzonych danych.
4. Colorsys - Zawiera funkcje `rgb_to_hls` oraz `hls_to_rgb`, pozwalające na konwersję między formatami cyfrowego zapisu koloru.

### 2.2 Analiza kodu

#### 2.2.1 Importowanie bibliotek

Importuję wypisane w podsekcji "Wykorzystane technologie" biblioteki. `matplotlib.pyplot` służy wyświetlaniu grafów, podczas gdy `matplotlib.colors` pozwala zarządzać zdefiniowanymi w `matplotlib` kolorami. Do tego obiekt `FuncAnimation` pochodzący z `matplotlib.animation` posłuży animowaniu wykresów.

```
%matplotlib notebook

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.colors as mc
from colorsys import rgb_to_hls, hls_to_rgb
from matplotlib.animation import FuncAnimation
```

Listing 1: Import bibliotek.

Warto nadmienić, że pojawiający się u samej góry napis `%matplotlib notebook` to komenda magiczna zapisująca w ustawieniach, że aktualnym środowiskiem do wyświetlania grafiki programu jest JupyterNotebook. Deklaracja jest konieczna do zapewnienia prawidłowego wyświetlania wykresów i ich animacji.

#### 2.2.2 Funkcja `darken_color()`

Funkcja `darken_color()` wykorzystywana jest na etapie animacji do progresywnego zmieniania koloru punktów już narysowanych. W podsekcji "Animacja" opisuję jej zastosowanie w sposób bardziej szczegółowy.

```
def darken_color(color, amount=1.5):
    """
    Funkcja zmieniająca jasność zadanego w pierwszym parametrze koloru o
    stopień zadany w drugim parametrze.
    Kolor podawany jest w postaci trójki (R, G, B) reprezentującej
    format RGB lub jako string (słowny lub w formacie #RGB) reprezentujący
    kolor w bibliotece matplotlib.
    Współczynnik o wartości < 1.0 rozjaśnia, dla > 1.0 pociemnia.
    Zwracany jest kolor w postaci trójki (R,G,B)
    """

    #Konwersja możliwego słownego przedstawienia koloru w format RGB
    try:
        c = mc.cnames[color]
    except:
        c = color
    c = rgb_to_hls(*mc.to_rgb(c))

    #Zabezpieczenie utrzymujące wartość nasycenia w przedziale [0,1]
    temp = 1 - amount * (1 - c[1])
    temp = temp if 0 <= temp <= 1 else 0

    c = hls_to_rgb(c[0], temp, c[2])
    return c
```

Listing 2: Deklaracja funkcji `darken_color()`.

Obliczenie nowego koloru polega na konwersji aktualnego koloru do formatu HLS, gdzie człon L - odpowiadający za jasność - zostaje pomniejszony zgodnie ze wzorem:

$$L' = 1 - a \cdot (1 - L)$$

gdzie  $L$  stanowi poprzednią wartość jasności,  $L'$  to obliczana nowa wartość, a  $a$  jest współczynnikiem podawanym przy wywołaniu funkcji. Widać, że dla wartości  $a > 1.0$  wartość  $L$  będzie ulegać zmniejszeniu i analogicznie zwiększeniu w przypadku przeciwnym.

### 2.2.3 Klasa Particle

Klasa Particle ma służyć ułatwieniu procesu losowania i śledzenia kolejnych pozycji cząsteczki. Ponadto upraszcza ona znakomicie kod, co widać doskonale w listingu nr 8. W założeniu obiekt tej klasy ma być iterowalny, a z każdą iteracją (których ilość będzie domyślnie ograniczona z możliwością dostosowania limitu) zwracać parę  $(p, s)$ , gdzie  $p$  będzie parą prezentującą aktualną pozycję cząstki, a  $s$  jej aktualny dystans od punktu startowego. Za pomocą takiej klasy w ramach wizualizacji wylosowanych zostanie 50 kroków cząsteczki oraz wykonanych zostanie łącznie 200 symulacji w ramach wspomnianego we wstępie zbierania danych.

Deklarowane z samego początku są atrybuty startpos oraz steplen, odpowiadające kolejno pozycji startowej (domyślnie para (0,0)) oraz długości kroku (domyślnie 1). Deklaruje się też miejsce na parametry aktualnego dystansu od pozycji startowej, limit iteracji oraz aktualny licznik iteracji.

```
class Particle:
    """
    Klasa reprezentująca jedną cząsteczkę.
    Parametrami początkowymi są pozycja początkowa podawana pod postacią
    dwójki (float, float) oraz długości kroku domyślnie równego 1.
    Klasa poddawana iterowaniu zwraca kolejne pozycje cząsteczki, której
    ruch polega na wykonaniu kroku o zadanej długości w losowym kierunku
    z przedziału [0, 2pi].
    Obiekt tej klasy można wywołać z parametrem liczby naturalnej n,
    ograniczając w rezultacie ilość iteracji do zadanej liczby.
    Obiekt bez uprzedniego wywołania z parametrem n wykonuje się domyślnie
    10 razy.
    """

    def __init__(self, startpos, steplen = 1):
        self.startpos = startpos
        self.pos = startpos
        self.len = steplen
        self.dist = 0
        self.iterlimit = 10
        self.itercount = 0

    def currentpos(self):
        """
        Funkcja publiczna zwracająca aktualną pozycję cząsteczki.
        """
        return self.pos
```

```

def currentdist(self):
    """
    Funkcja publiczna zwracająca aktualną odległość cząsteczki od punktu
    startowego.
    """
    return self.dist

def reset(self):
    """
    Funkcja resetująca pozycję, dystans, aktualny krok i limit iteracji
    do wartości początkowych.
    """
    self.pos = self.startpos
    self.dist = 0
    self.iterlimit = 10
    self.itercount = 0
    return self

```

Listing 3: Deklaracja klasy Particle cz. 1.

Zachowanie zdefiniowanego obiektu tej klasy w przypadku iteracji prezentuje poniższy listing. Wywołując obiekt z parametrem  $n$  możemy samodzielnie zmienić limit iteracji na dowolną liczbę naturalną. W każdej z nich losowana jest liczba z zakresu  $[0, 2\pi]$ , następnie zgodnie z założeniami obliczony oraz zapisany w miejsce aktualnej pozycji zostaje nowy punkt.

```

def __iter__(self):
    self.itercount = 0
    return self

def __call__(self, n):
    self.iterlimit = n if n > 0 else self.iterlimit
    return self

def __next__(self):
    phi = np.random.uniform(low=0.0, high=2*np.pi)
    newpos = (self.pos[0] + self.len*np.cos(phi), self.pos[1] + \
    + self.len*np.sin(phi))
    self.pos = newpos

    a,b = self.startpos
    self.dist = np.sqrt((a-newpos[0])**2 + (b-newpos[1])**2)
    if self.itercount >= self.iterlimit:
        raise StopIteration
    self.itercount += 1
    return (self.pos, self.dist)

```

Listing 4: Deklaracja klasy Particle cz. 2.

Wyjątek powodujący zakończenie iteracji wszczynany jest, gdy liczba iteracji zrówna się z ich limitem. Po każdej iteracji zwracana jest para zawierająca kolejno koordynaty cząsteczki oraz jej dystans od punktu startowego, a wewnętrzny licznik iteracji obiektu zwiększa się.

## 2.2.4 Animacja

Wykorzystując pakiet matplotlib tworzę animację ruchu cząsteczki po płaszczyźnie dwuwymiarowej. Stałe odpowiadające za kolory i rozmiary osi oraz planszy są deklarowane z początku pod postacią słowników, które potem łatwo rozpakowuję (za pośrednictwem operatora `**`) jako parametry z kluczami do odpowiednich funkcji rysujących.

```

GRIDSIZE = 1      #Stała rozmiaru kratownicy.
SIZE = 10         #Stała długości osi (w jednym kierunku).

#Parametry graficzne kratownicy
gridprop = {'color' : '#610f39',
            'linestyle' : '-.',
            'linewidth' : 0.5}

#Parametry graficzne cząsteczki
particleprop = {'color' : '#34a33f',
                'linestyle' : '-.',
                'linewidth' : 0.3,
                'marker' : '+',
                'markeredgecolor' : '#f03607',
                'markeredgewidth' : 2}

axcolor = '#DAA520'      #Kolor osi
bkgcolor = '#221333'     #Kolor tła

ticks = np.around(np.arange(-SIZE, SIZE+0.1, GRIDSIZE), 3)
minorticks = (ticks+GRIDSIZE/2)[: -1]

```

Listing 5: Deklaracja stałych przed animacją.

Zmienna `ticks` stanowi listę zaokrąglonych do trzeciego miejsca po przecinku liczb rozłożonych z odstępem `GRIDSIZE` w przedziale  $[-SIZE, SIZE+0.1]$  (dodanie wartości 0.1 zapewnia włączenie wartości brzegowych do przedziału). Będą one stanowić punkty głównej podziałki osi `OX` i `OY`, o których odgórnie zakładam równą długość. Zmienna `minorticks` to lista prezentująca podpodziałkę o tych samych odstępach, jednakże przesuniętą o pół długości odstepu. W ten sposób odstęp między elementami obu podziałek są równe.

W drugiej kolejności ustalane są parametry wykresu takie jak kolory tła i osi, umiejscowienie obu osi na środku wykresu oraz narysowanie kratki.

```
fig, ax = plt.subplots(figsize=(10,10), constrained_layout=True)

#Ustawienia podziałek
ax.tick_params(labelsize=5, colors = axcolor)
ax.set_yticks(ticks, minor=False)
ax.set_yticks(minorticks, minor=True)
ax.set_xticks(ticks, minor=False)
ax.set_xticks(minorticks, minor=True)

#Ustawienia kratownicy
ax.yaxis.grid(True, which='minor', **gridprop)
ax.xaxis.grid(True, which='minor', **gridprop)

#Ustawienia osi
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.spines['bottom'].set_position(('data', 0))
ax.spines['bottom'].set_color(axcolor)
ax.spines['left'].set_position(('data', 0))
ax.spines['left'].set_color(axcolor)

ax.set_facecolor(bkgcolor)    #Ustawienie tła
```

Listing 6: Ustawienie parametrów wykresu.

Tworzony jest obiekt klasy `Particle` o wpisanych dla czytelności parametrach domyślnych. Kolejno tworzona jest lista 50 ruchów cząsteczki oraz lista zawierająca graf (element listy `steps`) na każdą z 50 klatek planowanej animacji.

Funkcje `init()` oraz `update()` wykorzystywane są przez funkcję `FuncAnimation` w celu odpowiednio inicjalizacji środowiska do rysowania, gdzie ustalam limity osi z dodatkowym marginesem (wartość 0.2) na wypisanie oznaczeń osi, oraz aktualizacji kolejnych klatek. W trakcie tego drugiego procesu poprzez parametr `frame` dostarczane są pod postacią pary dwóch wartości typu `float` kolejne współrzędne punktów, które dopisane zostają do list `xdata` i `ydata`. Następnie wykonywana jest pętla iterująca po wszystkich dotychczas narysowanych pozycjach cząsteczki.

Chcąc uzyskać efekt ciemnienia i zanikania znaczników położenia punktu wraz z rysowaniem nowych kroków nie mogłem wykorzystać pojedynczego obiektu grafu, gdzie dorysowywane byłyby kolejne punkty, gdyż ustawienia jego stylu mają zastosowanie całościowe (tj. zmiana koloru w trakcie iteracji zmienia go wszystkim zaznaczonym już punktom). W poniższym kodzie każdy graf odpowiada jednej parze punktów i łączącej je linii, a w każdej klatce animacji są na siebie nakładane. Rozwiązanie takie pozwala na dynamiczną zmianę kolorów i rozmiarów zaznaczonych już dotychczas punktów.

```
#Zdefiniowanie i interakcja obiektu klasy Particle
Particle_one = Particle((0,0),1)
partpos = [Particle_one.currentpos()] + [info[0] for info in Particle_one(50)]
xdata, ydata = [], []
#Generowanie grafów na każdy krok cząsteczki
steps = [plt.plot([], [], **particleprop)[0] for _ in partpos]

def init():
    ax.set_xlim(-SIZE-0.2, SIZE+0.2)
    ax.set_ylim(-SIZE-0.2, SIZE+0.2)
    return steps

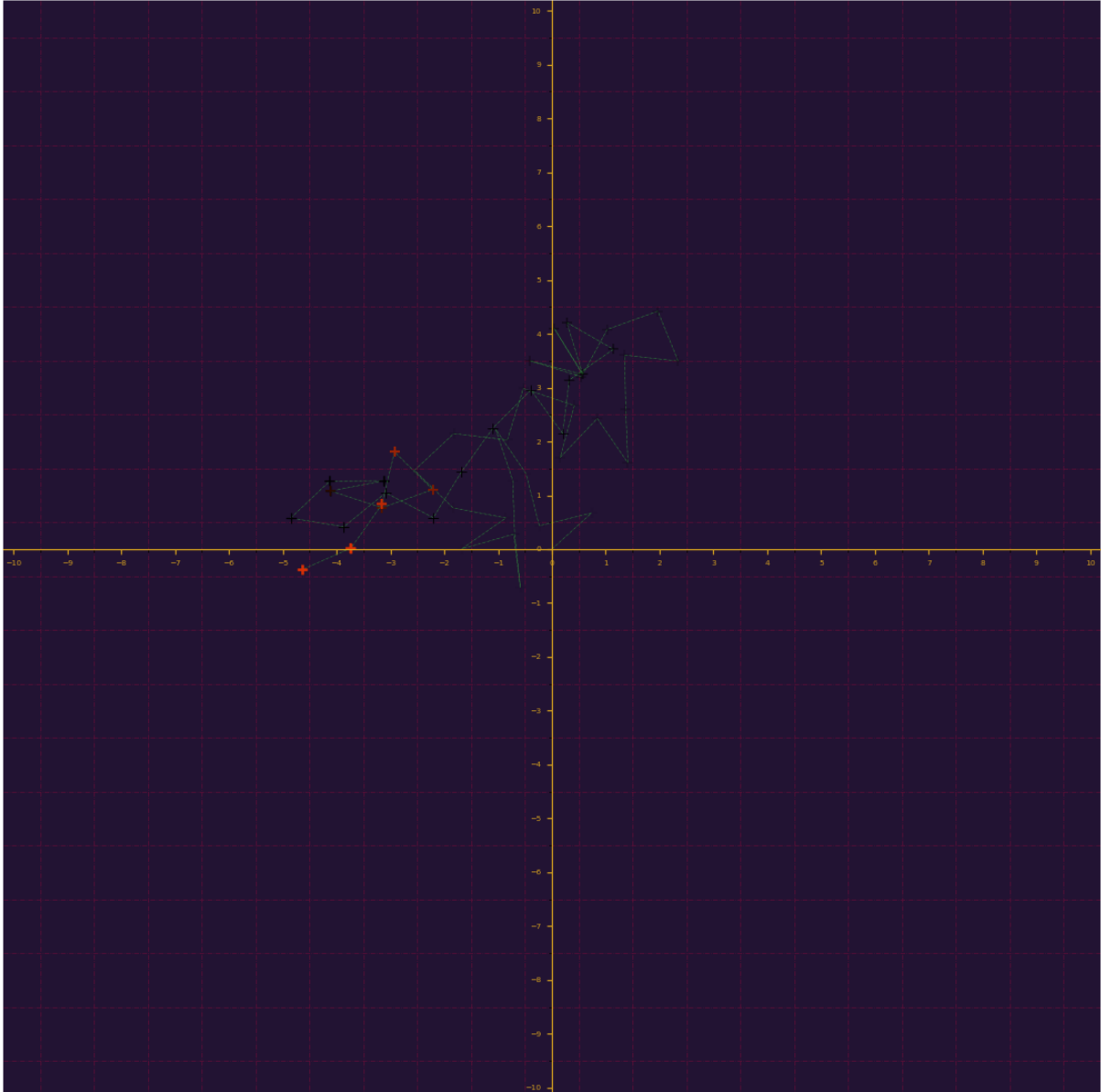
def update(frame):
    xdata.append(frame[0])
    ydata.append(frame[1])
    for idx in range(len(xdata), 1, -1):
        steps[idx].set_data(xdata[idx-2:idx], ydata[idx-2:idx])
        newcol = darken_color(steps[idx].get_markeredgcolor(), 1.1)
        steps[idx].set_markeredgcolor(newcol)

        newwidth = steps[idx].get_markeredgewidth()*0.9
        steps[idx].set_markeredgewidth(newwidth)
    return steps
```

```
ani = FuncAnimation(fig, update, frames=partpos,
                    init_func=init, blit=True, repeat=False)
plt.show()
```

Listing 7: Generowanie animacji.

W trakcie wspomnianej iteracji każdy kolejny graf otrzymuje dane pary punktów, funkcją `darken_color()` pociemniany jest kolor a znacznik jest pomniejszany o 10%. Na koniec tworzony jest obiekt animacji, a ona sama wywołana zostaje poleceniem `plt.show()`.



Rysunek 1: Grafika przedstawiająca trasę pokonaną przez cząsteczkę w 50 krokach. Widoczne jest ciemnienie oraz pomniejszenie rozmiarów punktów chronologicznie wcześniejszych.

### 2.2.5 Agregacja statystyk

Celem jest wykonanie dwustu symulacji ruchów cząsteczki za pośrednictwem obiektu klasy `Particle`. Poniższy kod przedstawia ten proces. Wywołanie metody `.reset()` powoduje powrót parametrów aktualnego położenia i odległości od środka do stanu początkowego, umożliwiając wykonanie nowej symulacji bez definiowania większej ilości obiektów. Próba bezpośredniej konwersji powstałej w ten sposób listy dwuwymiarowej `temp` do obiektu `DataFrame` dałaby w rezultacie tabelę, gdzie to numer rzędu indeksowałby wyniki całej symulacji, podczas gdy zakładaliśmy dokonanie takowego indeksowania względem kolumn. Z uwagi na ten fakt, zmienna `temp` zmieniana jest w macierz, którą w tej samej chwili poddajemy transpozycji, zamieniając wiersze z kolumnami. Dopiero potem przystępujemy do stworzenia obiektu.

```
Particle_two = Particle((0,0), 1)
temp = []
for _ in range(100):
    temp.append([info[1] for info in Particle_two(100)])
```

```
Particle_two.reset()
dataframe = pd.DataFrame(np.matrix(temp).T)

dataframe['max'] = dataframe.max(axis=1)
dataframe['med'] = dataframe.median(axis=1)
dataframe['min'] = dataframe.min(axis=1)
dataframe['avg'] = dataframe.mean(axis=1)
dataframe['std'] = dataframe.std(axis=1)
```

Listing 8: Generowanie i obliczanie statystyk.

Dodajemy do niego kolumny przeznaczone na kolejno: wartość maksymalną, medianę, wartość minimalną, średnią oraz odchylenie standardowe, obliczone dla każdego z rzędów tabeli - czyli w danym kroku całej setki symulacji. Następnie sporządzone zostają cztery wykresy przedstawiające każdą z wartości. Średnia i mediana współdzielą podziałkę osi OY, przez co wykres tej drugiej zostaje wyskalowany względem pierwszej metodą `.sharey()`. Wartości minimum i maximum zostają narysowane na jednym grafie.

```
fig2, axs = plt.subplots(2, 2, figsize=(16,10), constrained_layout=True)

axs[1, 0].plot(dataframe['max'])
axs[1, 0].plot(dataframe['min'])
axs[1, 0].set_title('Wartość maksymalna i minimalna')
axs[0, 1].plot(dataframe['med'], 'tab:orange')
axs[0, 1].sharey(axs[0, 0])
axs[0, 1].set_title('Mediana')
axs[0, 0].plot(dataframe['avg'], 'tab:green')
axs[0, 0].set_title('Średnia')
axs[1, 1].plot(dataframe['std'], 'tab:red')
axs[1, 1].set_title('Odchylenie standardowe')

fig2.suptitle('Statystyki odległości od punktu startowego', fontsize=16)

for ax in axs.flat:
    ax.set_xlim(0, 100)
    ax.yaxis.grid(True, which='major')
    ax.xaxis.grid(True, which='major')
    ax.set(xlabel='X - Krok', ylabel='Y - Wartość')
```

Listing 9: Rysowanie statystyk.

Po dodaniu tytułu dla całej czwórki grafów dostosowywane są kolejno: wspólny dla wszystkich limit osi OX, widoczność kratownicy względem widocznej podziałki oraz podpisy osi. Dla kolejnych wykresów zmieniają się ilości wykonanych symulacji, obiekt `Particle_tree(1000)` zastępuje więc `Particle_two(100)` a limit `ax.set_xlim()` zostaje do tego dostosowany. Do narysowania błękitnych funkcji  $\sqrt{x}$  oraz  $x$  używam listy punktów  $X$  utworzonej przed rysowaniem osi. Odpowiada za to linijka

```
X = list(range(1001))
```

Rysowanie samych wykresów pomocniczych przebiega wewnątrz pętli `for ax in axs.flat`. Metoda `.flat` konwertuje macierz grafów do listy, umożliwiając iterowanie po nich.

```
for ax in axs.flat:
    ax.plot(X, np.sqrt(X), 'tab:cyan')
    ax.set_xlim(0, 1000)
    ax.yaxis.grid(True, which='major')
    ax.xaxis.grid(True, which='major')
    ax.set(xlabel='X - Krok', ylabel='Y - Wartość')
```

Listing 10: Pętla w rysowaniu drugiej czwórki wykresów.

Zebrane dane o 1000 kroków mają być potem podniesione do drugiej potęgi i ponownie narysowane wraz z wiodącą linią  $y = x$ . W trakcie modyfikacji danych usunięte zostają końcowe statystyki, które po operacji potęgowania obliczane są ponownie.

```
dataframe = dataframe.iloc[:, :-5]**2
dataframe['max'] = dataframe.max(axis=1)
dataframe['med'] = dataframe.median(axis=1)
dataframe['min'] = dataframe.min(axis=1)
dataframe['avg'] = dataframe.mean(axis=1)
dataframe['std'] = dataframe.std(axis=1)
```

Listing 11: Modyfikacja zebranych już danych.

```
for ax in axs.flat:
    ax.plot(X, X, 'tab:cyan')
    ax.set_xlim(0, 1000)
    ax.yaxis.grid(True, which='major')
```

```
ax.xaxis.grid(True, which='major')
ax.set(xlabel='X - Krok', ylabel='Y - Wartość')
```

Listing 12: Pętla w rysowaniu trzeciej czwórki wykresów.

Ponadto, pojawiające się w końcowej części wykresy funkcji  $F_n(\alpha)$  są generowane przez kod wypisany w listingu nr 13. Główna różnica tkwi w każdorazowym numerycznym przybliżaniu wartości na zadanym przedziale przy pomocy szacowania metodą trapezów `np.trapz()`.

```
fig5, axs = plt.subplots(2, 2, figsize=(16,10), constrained_layout=True)

def F(s,x):
    return np.sqrt(s**2 - 2*s*np.cos(x)+1)-s
X = np.linspace(0, 2*np.pi, 10000)

axs[0, 0].plot(X,F(1,X), 'tab:orange')
axs[0, 0].set_title(f's_n = 1. Całka to ok. {np.trapz(F(1,X), X,dx=0.001)}')
axs[0, 1].plot(X,F(5,X),'tab:green')
axs[0, 1].set_title(f's_n = 5. Całka to ok. {np.trapz(F(5,X),X,dx=0.001)}')
axs[1, 0].plot(X,F(0.7,X), 'tab:blue')
axs[1, 0].set_title(f's_n = 0.75. Całka to ok. {np.trapz(F(0.7,X), X,dx=0.001)}')
axs[1, 1].plot(X,F(10,X), 'tab:red')
axs[1, 1].set_title(f's_n = 10. Całka to ok. {np.trapz(F(10,X), X ,dx=0.001)}')

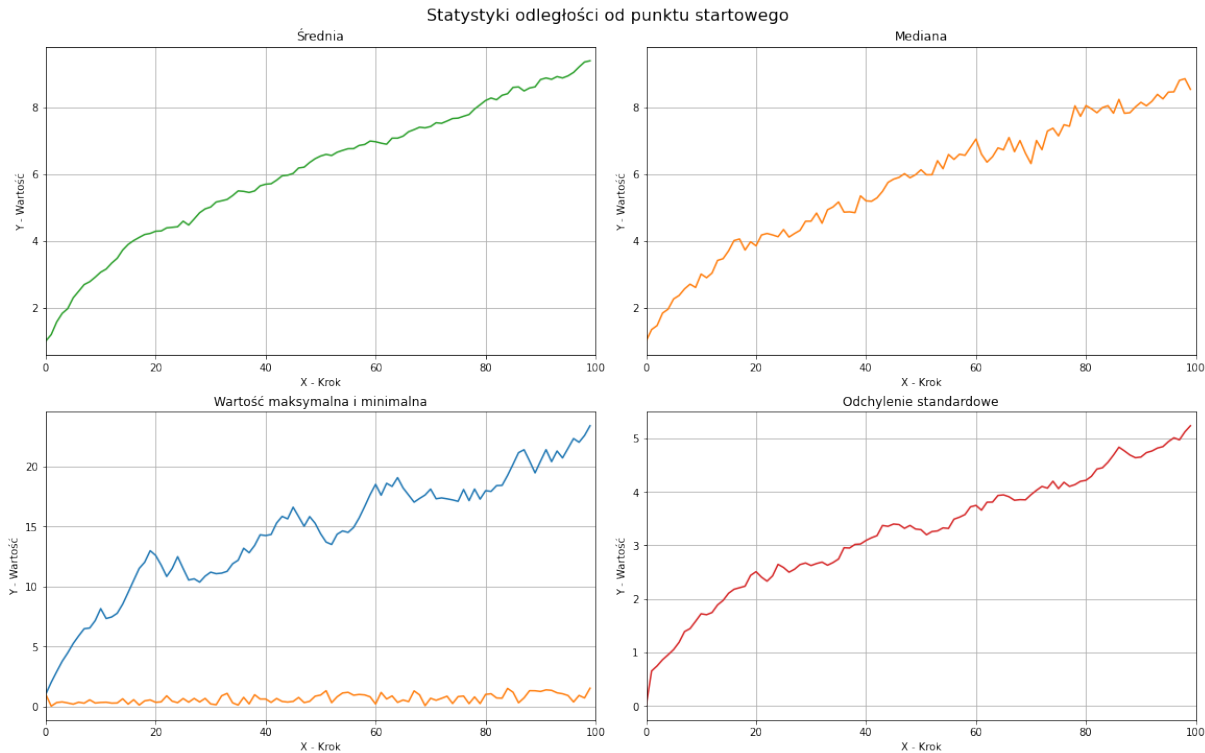
fig5.suptitle('Wykres funkcji F_n(alpha)', fontsize=16)

for ax in axs.flat:
    ax.plot(X,[0 for _ in range(10000)], 'k-')
    ax.set_xlim(0, 2*np.pi)
    ax.yaxis.grid(True, which='major')
    ax.xaxis.grid(True, which='major')
    ax.set(xlabel='alpha', ylabel='Zyskana odległość')
```

Listing 13: Generowanie wykresów oraz przybliżenie wartości całki dla funkcji  $F_n(\alpha)$ .

### 3 Analiza danych

Po zgromadzeniu danych z setki stukrokowych symulacji otrzymujemy poniższe wykresy.



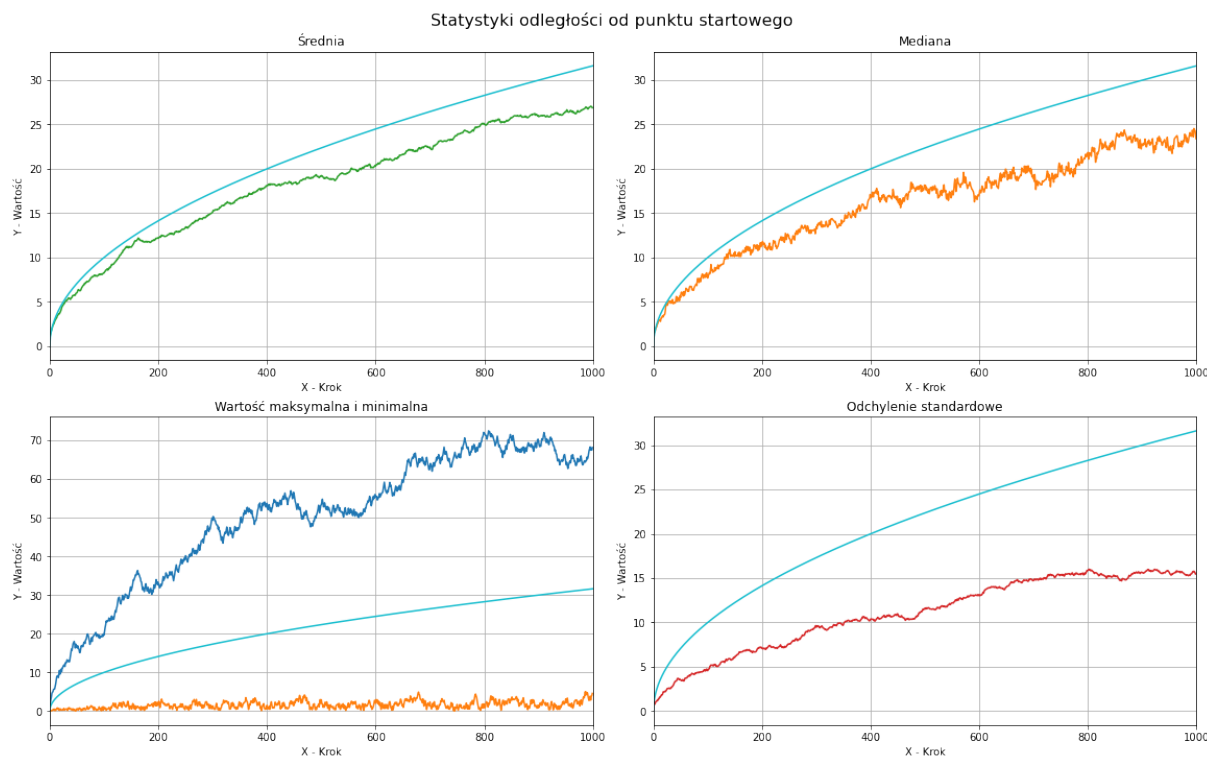
Rysunek 2: Grafika przedstawiająca wykresy wspomnianych wartości wyliczonych dla stulementowej próbki odległości w  $n$ -tym kroku  $\{s_1(n), s_2(n), \dots, s_{100}(n)\}$ .

Zebrane dane i wyprowadzone statystyki wskazują jednoznacznie na wzrostową tendencję mierzonej odległości. Ponadto proces ten wydaje się być dosyć niestabilny, o czym mówi rosnące wraz z ilością kroków odchylenie standardowe, mające swoje odzwierciedlenie w rosnącej różnicy między wartościami minimalnymi a maksymalnymi. Ciekawym i wartym odnotowania jest fakt,



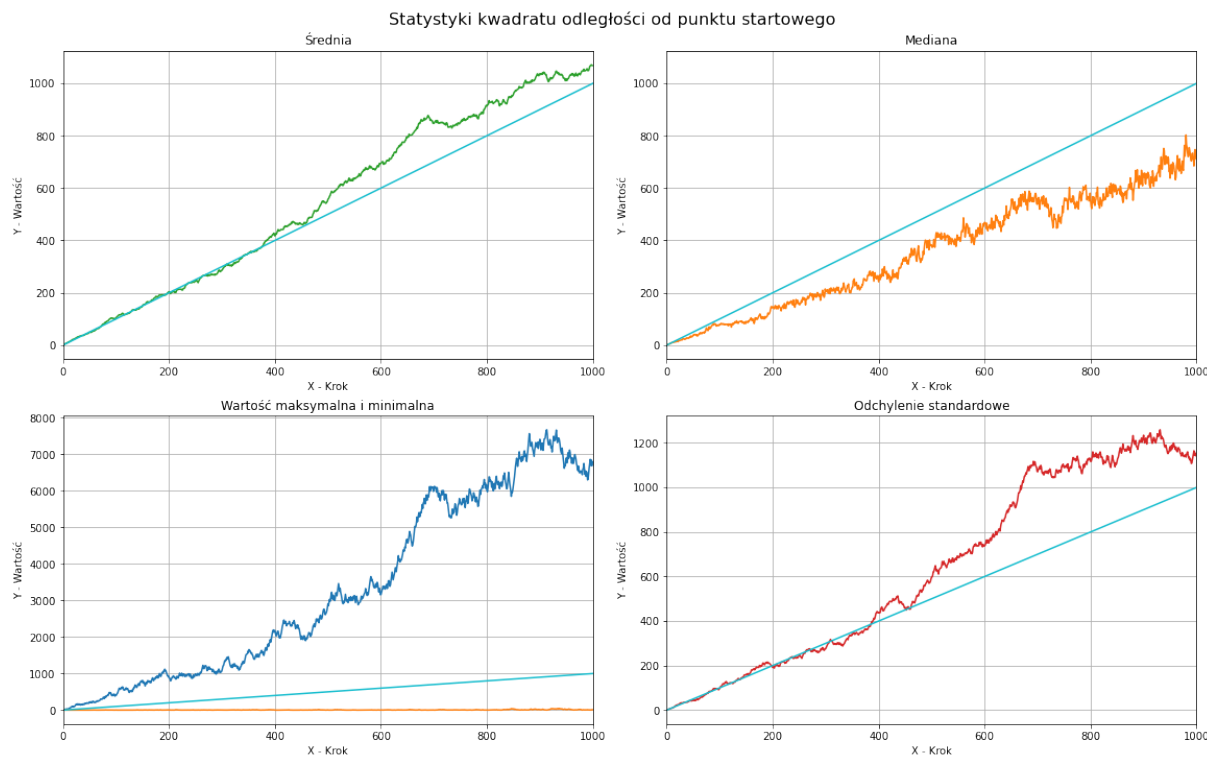
że wartość minimalna jako jedyna nie wykazuje tendencji wzrostowej wraz ze wzrostem liczby kroków.

Dla uzyskania lepszego pojęcia o naszych danych, rozszerzamy ilość kroków do tysiąca. Uwagę przykuwa w tej chwili kształt otrzymywanych wykresów, które wydają przypominać wykres funkcji  $\sqrt{x}$ .



Rysunek 3: Wykresy sporządzone dla setki symulacji 1000 kroków wraz z nałożoną wzdłuż nich błękitną linią  $y = \sqrt{x}$ .

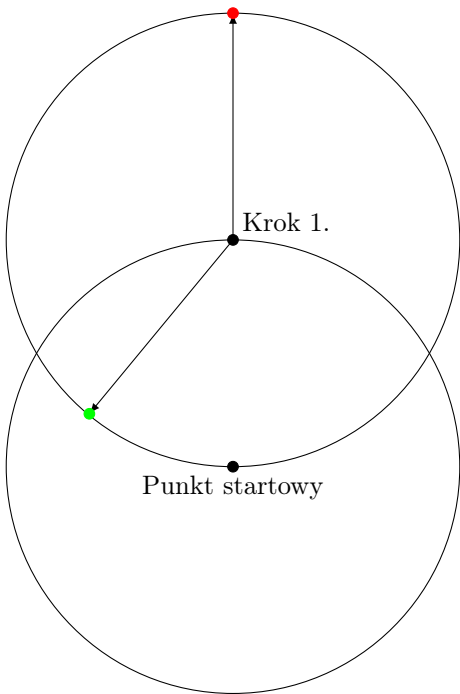
Spostrzeżenie to staje się tym bardziej uzasadnione, że wykresy naszych danych podniesionych do kwadratu zaczynają w przybliżeniu przebiegać jak prosta.



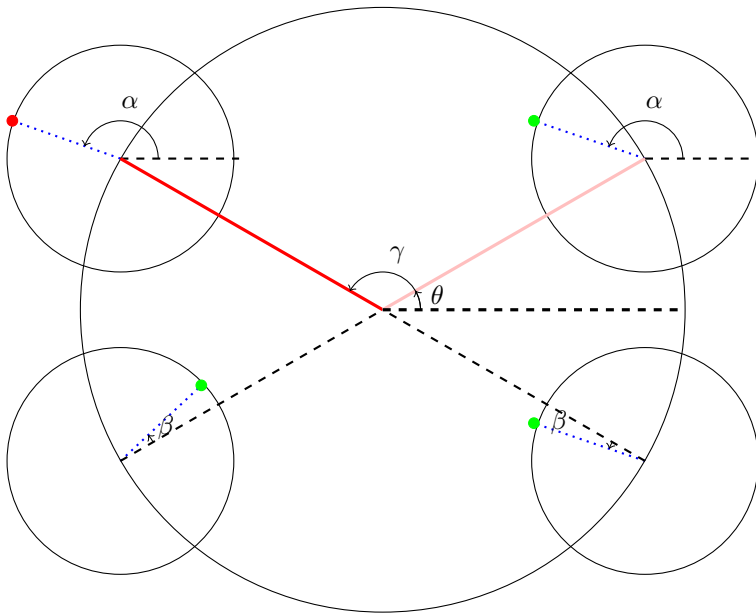
Rysunek 4: Wykresy poprzednich danych podniesionych do kwadratu wraz z nałożoną wzdłuż nich błękitną linią  $y = x$ .

3.1 Analiza geometryczno-probabilistyczna

Istnieje pewna teoretyczna intuicja wyjaśniająca zaobserwowaną "tendencję do ucieczki" cząsteczki od punktu startowego. Jeżeli przyjrzeć się możliwym pozycjom na kolejny skok cząsteczki między  $n$ -tym a  $n + 1$ -szym krokiem, składają się one na okrąg o promieniu 1. To, gdzie na tym okręgu wyląduje cząsteczka w kolejnym ruchu mówi nam, czy ten krok zbliżył nas, czy oddalił od punktu początkowego. Wystarczy wyobrazić sobie przypadek, gdzie cząsteczka dokonuje drugiego skoku, tj. znajduje się już w odległości 1 od punktu startowego, powiedzmy, na północ od niego. Kolejny krok bezpośrednio na północ zwiększy jej odległość od centrum o 1, podczas gdy krok w przeciwną stronę zmniejszy ją do zera. Pytanie brzmi, jaką wartość odległość zyskuje (lub traci) na pozycjach pośrednich?



Jako, że punkt na obwodzie losujemy zgodnie z rozkładem jednostajnym, możemy polegać na intuicji wynikającej z prawdopodobieństwa geometrycznego. Na skutek tego widzimy, że "sprzyjający nam" wycinek obwodu jest krótszy od tego, gdzie postawienie kolejnego kroku zwiększy naszą odległość od pozycji początkowej. Funkcja odległości  $n + 1$ -ego kroku zależy będzie więc tylko od aktualnej odległości od punktu startowego  $s(n)$  oraz losowanego kąta  $\phi$  z dokładnością do przesunięcia przedziału wartości tego argumentu - to, w jaki sposób zmieni się wartość funkcji dla konkretnego kąta w roli argumentu zależy też od położenia aktualnego punktu względem punktu startowego, jednakże możemy uniezależnić się od tego czynnika odpowiednio dostosowując oś od której liczymy kąt. Tłumaczy to poniższa grafika.



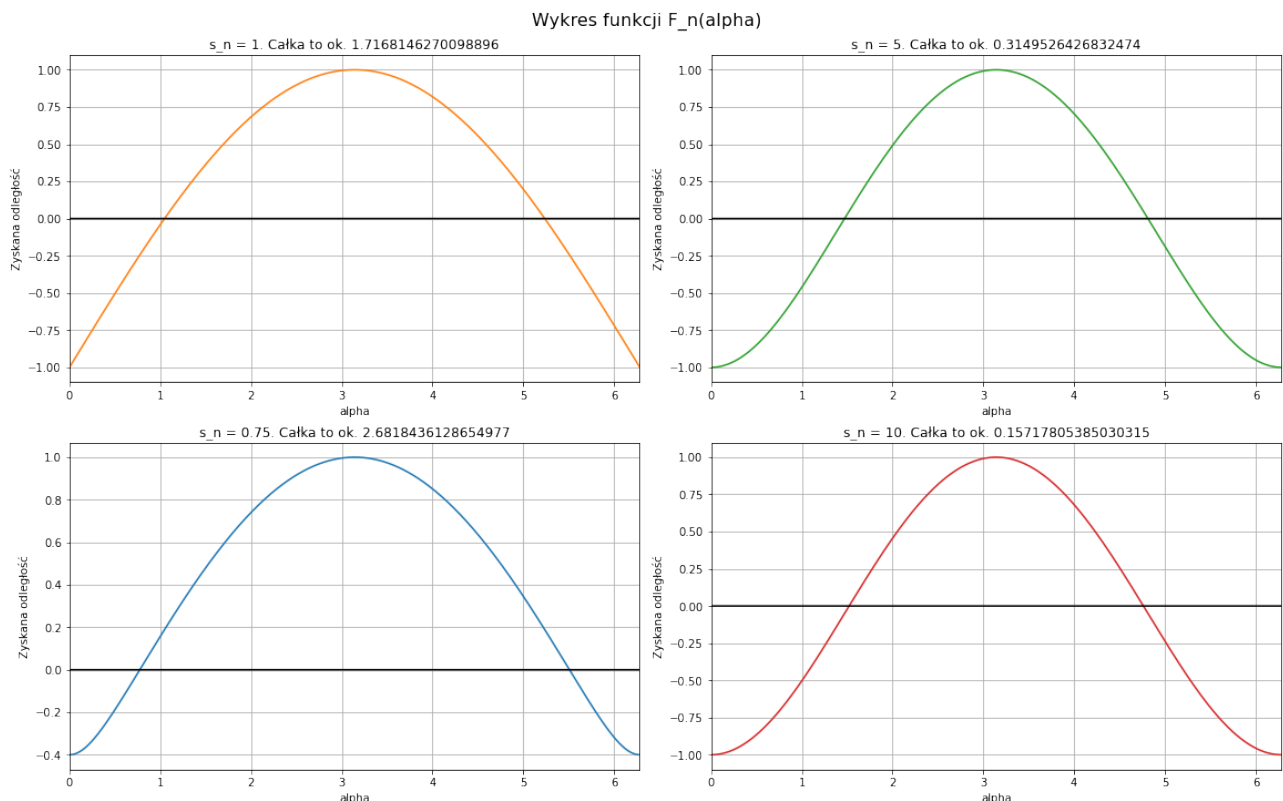
Promienie przerywane wskazują osie względem których program rozumie kąt. Okręgi u góry obrazują, że pomimo zachowania kąta nowego punktu  $\alpha$  jego odległość od centrum nadal jest zależna od kąta, jaki nasz aktualny punkt tworzy z centrum -  $\theta$  oraz  $\gamma$ . Translacja wartości

kąta nowego punktu względem nowej osi - wyprowadzonej w kierunku od aktualnego punktu do punktu początkowego - pozwala na zachowanie wartości funkcji, co ukazują dwa niżej umiejscowione kółka. Innymi słowy  $\beta(\alpha, \theta) = (\pi + \theta) - \alpha \bmod 2\pi$  dla  $\theta$  równego kątowi między idącą w prawo półprostą o początku w punkcie startowym a prostą łączącą ów punkt z punktem aktualnym. Chociaż policzenie jej całki explicite jest wyjątkowo trudne, tak możemy przeanalizować jej graf.

Badając trójkąt rozpinany przez punkt początkowy, punkt aktualny i punkt potencjalny możemy wyprowadzić wzór na zmianę odległości metodami znanymi z geometrii. Tym sposobem nasza funkcja ma postać:

$$\begin{aligned} F_n(\beta) &= \sqrt{(s_n - \cos \beta)^2 + \sin^2 \beta} - s_n \\ &= \sqrt{s_n^2 - 2s_n \cos \beta + 1} - s_n \end{aligned}$$

Gdzie  $s_n$  to aktualna odległość od centrum, a  $\beta$  jest kątem takim jak w na szkicu powyżej.



Rysunek 5: Wykresy funkcji  $F_n(\alpha)$  dla 4 różnych wartości  $s_n$  na przedziale  $[0, 2\pi]$

Widać, że jej wartości w miarę zwiększania się wartości  $s_n$  będą dążyć kształtem do sinusoïdy. W związku z tym można przypuszczać, że wartości całki będą zbiegać do zera od góry. Wartość takiej całki odzwierciedla, w sensie probabilistycznym, wartość oczekiwaną zyskanego/skróconego w danym kroku dystansu, a fakt, że jest ona dodatnia, udowadnia, że punkt będzie mieć "tendencję do ucieczki" od centrum. Dodatkowo można odnotować, iż translacja kąta nie ma wpływu na wynik całki, gdyż przedział całkowania dokładnie tej samej długości, równej okresowi samej funkcji  $F_n(\alpha)$ .

Słonność punktu do oddalania się ukazuje również porównanie dwóch poprzednich szkiców. Można zaobserwować, jak zwiększający się promień centralnego okręgu powoduje objęcie większej części obwodu mniejszego kółka, a co za tym idzie, zwiększa szansę na zmniejszenie dystansu.

### 3.2 Wnioski

Skonstruowana zgodnie z początkowymi założeniami symulacja Monte Carlo przedstawia ruch cząsteczki o tendencji pokrywającej się z uzasadnianą poprzez analizę geometryczno-probabilistyczną intuicją, według której cząsteczka jest o wiele bardziej skłonna do oddalania się od punktu początkowego, niż do pozostawania w jego okolicy.

Warto dostrzec, że nie jest to obserwacja wyjątkowa wyłącznie dla punktu początkowego, gdyż tak zaprogramowana cząsteczka pozostaje jednakowo niechętna do pozostawania w okolicy dowolnego punktu płaszczyzny - wszystkie przeprowadzone przez nas analizy pozostaną w sile dla odległości  $s$  mierzonej od dowolnego punktu.