Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską

Politechnika Świętokrzyska
Wydział Mechatroniki i Budowy Maszyn

Kierunek studiów:
mechanika i budowa maszyn

Jarosław Gałkiewicz

Materiały dydaktyczne do przedmiotu

PROGRAMOWANIE W PRAKTYCE INŻYNIERSKIEJ

opracowane w ramach realizacji Projektu
„**Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki**”
FERS.01.05-IP.08-0234/23

Kielce, 2025

Spis treści

1. Wstęp 4
1.1. Środowisko pracy 4
1.2. Działania matematyczne 5
1.3. Łańcuchy tekstowe 7
1.4. Listy 11
1.5. Słowniki 13
2. Instrukcje warunkowe i pętle 15
2.1. Instrukcja if 15
2.2. Pętla for 17
2.3. Pętla while 19
2.4. Przerywanie działania pętli 20
3. Funkcje 21
4. Moduły 22
5. Obsługa błędów 24
6. Klasy 27
7. Odpluskwianie programu 30
8. Przebieg zmienności funkcji 32
8.1. Wykres 32
8.2. Modyfikacje wykresu 33
8.3. Wyznaczanie pochodnej 36
8.4. Dodawanie wykresu do już istniejącego 37
8.5. Wyznaczanie miejsc zerowych pochodnej 39
8.6. Tworzenie pliku WORD/PDF 41
8.7. Przedziały zmienności 43
9. Lot pocisku 50
9.1. Metoda Eulera 52
9.2. Metoda Runge-Kutty 54
10. Lot pocisku na wietrze 58
11. Układ planetarny 68
11.1. Tor lotu planety 68

12. Animacja ruchu planety 74
13. Karuzela 78
14. Literatura 83

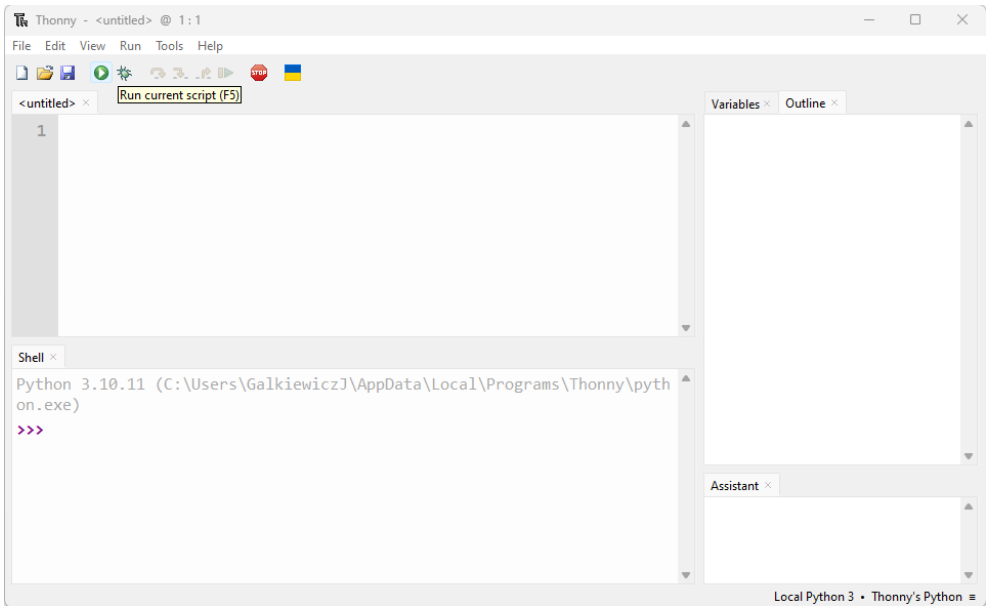
1. Wstęp

Język Python został stworzony na początku lat 90 ubiegłego wieku. Sama nazwa pochodzi od brytyjskiego serialu komediowego „Latający cyrk Monty Pythona”. Program zyskał sympatię programistów a funkcjonowanie jako projekt Open Source dodatkowo podbił ilość fanów. Efektem tego jest niezwykle rozbudowany zestaw bibliotek tworzonych przez użytkowników. W ramach kursu poznamy podstawy języka niezbędne do samodzielnej pracy oraz zaawansowane zagadnienia, które wykorzystywane są w pracy inżyniera.

1.1. Środowisko pracy

Istnieje wiele środowisk pracy od profesjonalnych, często płatnych, jak Anaconda (https://www.anaconda.com/) Pycharm (https://www.jetbrains.com/pycharm/) po darmowe, proste i mało obciążające system jak Mu (https://codewith.mu/) czy Jupiter (https://jupyter.org/) (Sweigart, 2020). W naszym kursie wykorzystywane będzie środowisko stworzone na Uniwersytecie w Tartu (Estonia) Thonny wykorzystujące Python w wersji 3.10. Plik instalacyjny można pobrać ze strony https://thonny.org/. Sama instalacja jest tak prosta, że nie wymaga żadnej pomocy. Po uruchomieniu otworzy się okienko pokazane na Rys. 1.

[Tekst alternatywny. Okno środowiska programistycznego Thonny. U góry widać menu i przyciski. Okno podzielone jest na cztery sekcje: program, powłoka, kontrola i asystent.]



Rys. 1. Środowisko Thonny.

Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki”
nr FERS.01.05-IP.08-0234/23

W lewej górnej części ekranu znajduje się miejsce na program. Pod nią umieszczono okienko powłoki (Shell) do bezpośredniej pracy z Pythonem. Wpisane tu polecenia są wykonywane od razu po wciśnięciu klawisza <ENTER>. Po prawej umieszczono okienka pozwalające śledzić wykonywanie programu. W prawy dolnym narożniku znajduje się znaczek umożliwiający konfiguracje środowiska tak, aby lepiej je dopasować do własnych potrzeba. Na obecnym etapie będziemy pracować z ustawieniami domyślnymi.

1.2. Działania matematyczne

Użytkowanie Python można zacząć od wykorzystania powłoki do prowadzenia zwykłych działań. Python może się zachowywać jak zwykły kalkulator wykonując podstawowe działania matematyczne z zachowaniem ich kolejności. Podstawowe działania wymieniono w Tabeli 1.

Tabela 1. Podstawowe działania matematyczne

Operator	Działanie	Przykład
+	Dodawanie	x + y
-	Odejmowanie	x - y
*	Mnożenie	x * y
/	Dzielenie	x / y
%	Reszta z dzielenia	x % y
**	Potęgowanie	x ** y
//	Dzielenie całkowite	x // y

Działania można wykonywać po prostu wprowadzając liczby i operatory w oknie powłoki, np.

```
>>> 5 % 3
2
```

Lub przypisując wartości zmiennym co ma tą zaletę, że Python będzie pamiętał wartość i dzięki temu będzie jej można użyć ponownie. W poniższym przykładzie najpierw zmiennym przypisano wartości, a następnie wyniki działań przypisano kolejnym zmiennym. Wyznaczono resztę z dzielenia oraz dokonano dzielenia całkowitego i porównano wyniki obu operacji.

```
>>> a=5
>>> b=3
```

```
>>> c=a%b
>>> c
2
>>> d=a//b
>>> d
1
>>> d==c
False
```

Python nie jest ograniczony do wymienionych operacji, ale wykonanie bardziej skomplikowanych obliczeń może wymagać zaimportowania dodatkowych programów definiujących te działania o czym później.
Dużą zaletą Pythona jest definiowanie typu zmiennych na podstawie przypisanej wartości. Zapis a=1 oznacza, że Python uzna „a” za zmienną typu liczba całkowita. Ma to swoje konsekwencje ponieważ nie każdą operację można wykonać da się wykonać na wybranym typie danych. Przyjmijmy, że „N” (wielkość liter ma znaczenie) jest łańcuchem tekstowym, natomiast „a” liczbą całkowitą. Dodanie do siebie „N” i „a” spowoduje błąd, ale już mnożenie jest możliwe.

```
>>> N='tekst'
>>> a=3
>>> b=a+N
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> b=a*N
>>> b
'teksttekstttekst'
```

Wykonajmy jeszcze małe doświadczenie. Zdefiniujemy zmienną „g” jako liczbę rzeczywistą i spróbujemy ją pomnożyć przez zmienną tekstową.

```
>>> g=2.0
>>> N*g
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
```

Jak widać taka operacja jest niemożliwa do wykonania. Gdyby kiedyś istniała konieczność sprawdzenia jakiego typu jest zmienna wystarczy użyć komendy „type”.

```
>>> type(N)
<class 'str'>
```

1.3. Łańcuchy tekstowe

Łańcuchy tekstowe są typami zmiennych, którym należy poświęcić trochę więcej czasu. W przykładzie widzieliśmy, że tworzy się je bardzo łatwo. Wystarczy przypisać zmiennej coś zamkniętego między dwoma apostrofami (’). Można też przy definicji zamiast apostrofów użyć cudzysłowów (”). W przypadku przypisywania więcej niż jednej linijki do zmiennej należy użyć trzech apostrofów.

```
>>> dluga_linia='''pierwsza linia
druga linia
trzecia linia'''
>>> dluga_linia
'pierwsza linia\ndruga linia\ntrzecia linia'
```

Jak widać wynik wyświetlony w powłoce trochę się różni od tego co wpisaliśmy. Pojawiły się tajemnicze symbole „\n” które oznaczają po prostu nową linię. Jeśli naszą zmienną spróbujemy wyświetlić w „profesjonalny” sposób za pomocą komendy „print” wówczas wyświetli się ona w prawidłowy sposób.

```
>>> print(dluga_linia)
pierwsza linia
druga linia
trzecia linia
```

Co zrobić, gdybyśmy chcieli użyć w tekście apostrofu? W języku polskim taka sytuacja nie zdarza się często, ale już w angielskim jest nagminna przez co w Python’ie zadbano o rozwiązanie problemu. Jeśli zastosujemy cudzysłowy zamiast apostrofów problem zniknie.

```
print(" Moduł Young'a jest bardzo ważny")
```

Podobny efekt uzyskamy stosując backslash (\) jak w przykładzie

```
print('Moduł Young\'a jest bardzo ważny')
```

Znak „\” służy ogólnie do wstawiania znaków, których nie da się wstawić innym sposobem. Podsumowuje to Tabela 2.

Tabela 2. Znaki specjalne w zmiennej tekstowej

Zestaw znaków	Wynik
\’	znak apostrofu
\”	znak cudzysłowu
\t	tabulator
\n	nowa linia
\\	znak backslash

Łańcuchy tekstowe wykorzystywane w programach często są wynikiem wcześniejszych operacji przez co ich zawartość jest dynamiczna. Możemy wstawić do jednego łańcucha inny taj jak w przykładzie:

```
>>> student='Jan Kowalski'
>>> komunikat='Student %s jest obecny'
>>> print(komunikat % student)
Student Jan Kowalski jest obecny
```

Zdefiniowaliśmy zmienną student, następnie ogólny komunikat, w którym symbolem „%s” oznaczamy miejsce wstawienia innego łańcucha tekstowego, w tym przypadku imienia i nazwiska studenta. Podobną technikę wykorzystamy do wstawienia w tekście liczby.

```
>>> liczba1=1.23456789
>>> liczba2=2.3456789
>>> print('Srednia %f i %f równa się %f' % (liczba1, liczba2, (liczba1+liczba2)/2))
Srednia 1.234568 i 2.345679 równa się 1.790123
```

Zdefiniowaliśmy dwie liczby, a następnie tekst w którym je wstawimy. Miejsca ich wstawienia oznaczone są „%f”, co oznacza wstawienie liczby zmiennoprzecinkowej. Spróbujemy jeszcze zmniejszyć wyświetlaną ilość miejsc po przecinku:

```
>>> print('Srednia %.2f i %.2f równa się %.2f' % (liczba1, liczba2,
(liczba1+liczba2)/2))
Srednia 1.23 i 2.35 równa się 1.79
```

Gdybyśmy mieli do czynienia z liczbami całkowitymi wówczas miejsce ich wstawienia oznaczamy symbolem „%d”:

```
>>> print('Srednia %d i %d równa się %d' % (liczba1, liczba2, (liczba1+liczba2)/2))
Srednia 1 i 2 równa się 1
```

Możemy zastosować inny zapis dający ten sam rezultat:

```
>>> print('Srednia {:.2f} i {:.2f} równa się {:.2f}'.format(liczba1, liczba2,
(liczba1+liczba2)/2))
Srednia 1.23 i 2.35 równa się 1.79
```

Nietypową cechą łańcuchów tekstowych jest ich niezmienność. Raz wprowadzony nie może zmienić swojej wartości.

```
>>> pom='testowa zmienna'
>>> pom[0]
't'
>>> pom[1]
'e'
>>> pom[1]='r'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

W pokazanym przykładzie zdefiniowaliśmy łańcuch tekstowy. Następnie odczytaliśmy jego wybrane znaki i taka operacja się udała, ale próba zmiany pojedynczego znaku w łańcuchu wywoła komunikat błędu.
Python ma rozbudowane możliwości wybierania fragmentów łańcucha. Możemy to wręcz nazwać szatkowaniem łańcucha.

```
>>> pom[0:5]
'testo'
>>> pom[:-5]
'testowa zm'
>>> pom[-5:]
'ienna'
>>> pom[5:-5]
'wa zm'
```

Jak pokazują przykłady można wybrać dowolny zakres znaków, pamiętając, że numerowanie zaczyna się od 0. Ujemny indeks oznacza liczenie od końca, a pusty indeks początek lub koniec łańcucha w zależności o miejsca jego użycia.
Praca z łańcuchami wiąże się z korzystaniem ze specyficznych funkcji. Poniżej przykład zamieniający wszystkie litery łańcucha tekstowego na wielkie litery:

```
>>> pom.upper()
'TESTOWA ZMIENNA'
```

W podobny sposób można wykorzystać funkcję lower(). Przy sprawdzaniu zawartości łańcucha tekstowego można skorzystać z funkcji zwracających wartość typu logicznego (True lub False), które wymieniono w Tabeli 3.

Tabela 3. Popularne funkcje sprawdzające wielkość liter w tekście

Funkcja	Znaczenie
islower()	sprawdza czy wszystkie litery są małe
isupper()	sprawdza czy wszystkie litery są wielkie
isdecimal()	sprawdza czy łańcuch zawiera tylko cyfry
isalpha()	sprawdza czy w łańcuchu znajdują się tylko litery
isalnum()	sprawdza czy w łańcuchu znajdują się tylko litery i cyfry

W przypadku każdej z tych funkcji sprawdzanie pustego ciągu oznacza wynik False. Mimo, że funkcji operujących na łańcuchach jest dużo więcej warto jeszcze zwrócić uwagę na funkcje strip() i jej dwie siostry lstrip() i strip(). Funkcje te powodują usuwanie białych znaczków (np. spacja) z początku (lstrip, strip) i końca łańcucha tekstowego (rstrip,strip) np.:

```
>>> pom='   123   '
>>> pom.strip()
'123'
>>> pom.lstrip()
'123   '
>>> pom.rstrip()
'   123'
```

1.4. Listy

Jeśli łańcuchy tekstowe traktować jako uporządkowany ciąg znaków to naturalnym rozwinięciem takiej zmienne będzie uporządkowany ciąg dowolnych elementów, nie tylko znaków. Python oferuje takie rozwinięcie. Zmienną taką nazywa się listą (Briggs, 2013). Przy jej wywołaniu korzysta się z kwadratowych nawiasów:

```
>>> lista=['Michal','Tomek','Ula','Agata','Rysiek']
>>> zestaw_liczb=[1,3,7,3,5,7]
```

Listy możemy „szatkować” tak samo jak łańcuchy:

```
>>> lista[2:4]
['Ula', 'Agata']
>>> zestaw_liczb[-3:]
[3, 5, 7]
```

W przeciwieństwie do łańcuchów możemy zmieniać wartość pojedynczych elementów listy:

```
>>> lista[2]='Kasia'
>>> lista
['Michal', 'Tomek', 'Kasia', 'Agata', 'Rysiek']
```

Możemy zamieniać je wzajemnie miejscami (warto zwrócić uwagę na dość ciekawy zapis operacji):

```
>>> lista[0],lista[1]=lista[1],lista[0]
>>> lista
['Tomek', 'Michal', 'Kasia', 'Agata', 'Rysiek']
```

Mimo, że wykorzystywane listy mają różnego typu obiekty możemy je dodać do siebie ponieważ dodawanie nie odbywa się według zasad matematycznych a logicznych. Jedyną konsekwencją jest pojawienie się zbioru w którego elementy są różnego typu:

```
>>> nowa_lista=lista+zestaw_liczb
>>> nowa_lista
```


Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki”
nr FERS.01.05-IP.08-0234/23

['Tomek', 'Michal', 'Kasia', 'Agata', 'Rysiek', 1, 3, 7, 3, 5, 7]

Do listy możemy dodawać elementy:

```
>>> lista.append('Hubert')
>>> lista+=['Olek']
>>> lista
['Tomek', 'Michal', 'Kasia', 'Agata', 'Rysiek', 'Hubert', 'Olek']
```

Zapis lista+=[‘Olek’] to skrócony zapis przypisania lista=lista+[‘Olek’].

Gdybyśmy jednak chcieli dodać element, ale w środku listy użyjemy innej komendy:

```
>>> lista.insert(1,'Zosia')
>>> lista
['Tomek', 'Zosia', 'Michal', 'Kasia', 'Agata', 'Rysiek', 'Hubert', 'Olek']
```

Możemy też usuwać:

```
>>> del lista[2]
>>> lista
['Tomek', 'Zosia', 'Kasia', 'Agata', 'Rysiek', 'Hubert', 'Olek']
```

Warto wspomnieć o kilku bardziej przydatnych funkcjach wykorzystywanych przy pracy z listami. Ilość elementów listy sprawdzimy funkcją len():

```
>>> len(lista)
7
```

Możemy sprawdzić czy element występuje na liście za pomocą operatora „in”:

```
>>> 'Tomek' in lista
True
```

Jeśli już wiemy że element występuję na liście możemy sprawdzić jego pozycję (trzeba pamiętać, że pierwszy element na numer 0):

```
>>> lista.index('Tomek')
0
```

Gdybyśmy chcieli elementy posortować użyjemy funkcji sort(), która domyślnie układa wyraz w porządku alfabetycznym, ale nie musi. Stosując parametr „reverse” elementy ułożą się w kierunku odwrotnym (sort(reverse=True)):

```
>>> lista.sort()
>>> lista
['Agata', 'Hubert', 'Kasia', 'Olek', 'Rysiek', 'Tomek', 'Zosia']
```

Możemy również odwrócić kolejność elementów:

```
>>> lista.reverse()
>>> lista
['Zosia', 'Tomek', 'Rysiek', 'Olek', 'Kasia', 'Hubert', 'Agata']
```

Jak widać biblioteka funkcji i metod działających na listach jest ciekawa i bardzo rozbudowana. Nie ma sensu ich dalej wymieniać poza jednym wyjątkiem. Zamienimy listę na łańcuch tekstowy:

```
>>> ','.join(lista)
'Zosia,Tomek,Rysiek,Olek,Kasia,Hubert,Agata'
```

W przykładzie jako znak rozdzielający, który został przy łączeniu wstawiony między każdy element listy uzyty został przecinek. Taki łańcuch można z powrotem zamienić na listę komendą split():

```
pom=','.join(lista)
>>> pom
'Zosia,Tomek,Rysiek,Olek,Kasia,Hubert,Agata'
>>> lista2=pom.split(',')
>>> lista2
['Zosia', 'Tomek', 'Rysiek', 'Olek', 'Kasia', 'Hubert', 'Agata']
```

1.5. Słowniki

Najdziwniejszą strukturę mają zmienne typu słownik (dictionary) (Briggs, 2013). Definicja wyróżnia się nawiasem sześciennym, a wewnątrz wpisane są klucze i odpowiadające im wartości:

kolory={'Monika':'czerwony','Stefan':'niebieski','Michal':'szary','Olga':'zolty'}

W przykładzie zdefiniowano osoby i ich ulubione kolory. Żeby sprawdzić ulubiony kolor wybranej osoby (klucza) piszemy:

```
>>> kolory['Monika']
'czerwony'
```

Usunięcie elementu odbywa się podobnie jak w listach przez podanie wartości klucza:

```
>>> del kolory['Stefan']
>>> kolory
{'Monika': 'czerwony', 'Michal': 'szary', 'Olga': 'zolty'}
```

W przykładzie usunęliśmy Stefana i jego kolor. możemy sprawdzić listę dostępnych kluczy:

```
>>> kolory.keys()
dict_keys(['Monika', 'Michal', 'Olga'])
```

lub wartości:

```
>>> kolory.values()
dict_values(['czerwony', 'szary', 'zolty'])
```

W przypadku rozbudowanego słownika sprawdzenie czy jakiś klucz występuje może być łatwiejsze przy użyciu operatora „in”:

```
>>> 'Monika' in kolory.keys()
True
```

Aby dopisać nową wartość po prostu nowy klucz i przypisujemy mu wartość

```
>>> kolory['Sefan']='niebieski'
>>> kolory
{'Monika': 'czerwony', 'Michal': 'szary', 'Olga': 'zolty', 'Sefan': 'niebieski'}
```

Stefan wrócił na listę, a dokładnie został dopisany do jej końca.

2. Instrukcje warunkowe i pętle

W każdym języku programowania przewidziano instrukcję warunkową, która porównuje dwie wielkości. Jedynym problemem jest więc zwykle składnia instrukcji i operatorów. Podstawowe operatory porównania i logiczne przedstawia Tabela 4.

Tabela 4. Podstawowe operatory logiczne

Operator	Znaczenie
==	równy
!=	różny
>	większy
<	mniejszy
>=	większy lub równy
<=	mniejszy lub równy
and	i
or	lub
not	nie

2.1. Instrukcja if

Składnia funkcji if wygląda następująco:

if warunek:
····komenda lub zestaw komend

Zaczyna się od słowa kluczowego if (jeśli), po której podany jest warunek. Warunek może mieć bardzo skomplikowaną postać, ale ostatecznie musi dać wynik w postaci jednej z dwóch wartości logicznych **True** (prawda) lub **False** (fałsz). Wielkość liter w nazwach wartości ma znaczenie. Po warunku pojawia się dwukropek, a po nim podawane są komendy do wykonywania, jeśli warunek zwrócił wartość **True**. Jeśli wykonana ma być pojedyncza komenda, zapis funkcji można skrócić i umieścić komendę wprost po dwukropku, ale często nawet pojedyncza komenda umieszczana jest w nowej linii. W razie konieczności łatwo dopisywać kolejne polecenia. Pojawia się pytanie skąd Python wie, gdzie się kończy zestaw komend wykonywanych po spełnieniu warunku? Decyduje o tym wcięcie. W przykładzie narysowane są 4 kropki, które oznaczają spacje. Każda kolejna linia z 4 spacjami na początku będzie zaliczana do bloku poleceń wykonywanych po spełnieniu warunku.

Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki”
nr FERS.01.05-IP.08-0234/23

```
>>> if 2<3:
····print('Wykonanie 1 komendy')
····print('Wykonanie 2 komendy')
····print('Wykonanie 3 komendy')
 Wykonanie 1 komendy
Wykonanie 2 komendy
Wykonanie 3 komendy
```

Od tej chwili pisząc kod musimy zwracać uwagę na ilość spacji na początku linii. Spróbujmy jeszcze raz uruchomić powyższy przykład, ale w linii z drugą komendą usuńmy jedną spację:

```
>>> if 2<3:
····print('Wykonanie 1 komendy')
···print('Wykonanie 2 komendy')
····print('Wykonanie 3 komendy')
  File "<stdin>", line 3
    print('Wykonanie 2 komendy')
                    ^
IndentationError: unindent does not match any outer indentation level
```

Jak widać program wykrył błąd nieprawidłowego wcięcia i nie wykonał polecenia. Instrukcja if może mieć dużo bardziej skomplikowaną strukturę. Oprócz bloku wykonywanego gdy warunek jest spełniony, możemy też określić co robić, gdy warunek okaże się nieprawdziwy:

```
>>> if 2>3:
····print('2 jest mniejsze od 3')
else:
····print('2 nie jest wieksze od 3')
2 nie jest wieksze od 3
```

Jak pokazuje przykład po zakończeniu bloku komend wykonywanych, gdy warunek jest prawdziwy, w nowej linii zmniejszamy wcięcie i wpisujemy słowo kluczowe **else** zakończone dwukropkiem. Dalej w nowej linii wpisujemy blok komend gdy warunek nie jest spełniony pamiętając o wcięciu. To jeszcze nie wszystko. Za pomocą instrukcji if możemy sprawdzić dowolną ilość warunków. Wykorzystamy do tego dwie zmienne a i b.

```
>>> a=1
>>> b=3
>>> if a==b:
    print('Liczby są równe')
elif a>b:
    print('a jest wieksze od b')
elif a<b:
    print('b jest większe od a')
b jest większe od a
```

Jak widać sprawdziliśmy 3 warunki, ale nie sprawdziliśmy co się stanie gdy więcej niż jeden warunek jest prawdziwy:

```
>>> if a<100:
    print('a jest mniejsze od 100')
elif a<10:
    print('a jest mniejsze od 10')
elif a<1:
    print('a jest mniejsze od 1')
else:
    print('a jest bardzo male')
a jest mniejsze od 100
```

Jak widać instrukcja wykonała pierwszy prawdziwy warunek i zakończyła działanie.

2.2. Pętla for

Jeśli chcielibyśmy wykonać jakąś czynność kilka razy wykorzystamy do tego pętlę. Najprostszą pętlą wykorzystywaną, kiedy wiemy ile razy czynność będzie powtórzona, jest pętla for (Briggs, 2013). Jej strukturę pokazuje przykład:

```
>>> for i in range(5):
    print(i)
0
1
2
3
4
```

Pętla zaczyna się od słowa kluczowego **for**. Następnie definiowana jest zmienna sterująca (w naszym przypadku i). Po zmiennej następuje słowo kluczowe **in** oraz definicja zbioru, którego elementy będzie przyjmowała zmienna sterująca.
W rzykładzie wykorzystano funkcję **range** w jej najprostszej wersji. Instrukcja ta generuje liczny z zakresu <0;5). Definicja zbioru, a nie zakres liczbowy jest dużą różnicą w porównaniu do pętli for spotykanej w innych językach programowania. dobrze obrazuje to poniższy przykład:

```
>>> lista=['Anna','Katarzyna','Robert','Stanislaw','Hubert','Michal']
>>> for imie in lista:
    print(imie)
Anna
Katarzyna
Robert
Stanislaw
Hubert
Michal
```

W przykładzie najpierw zdefiniowaliśmy listę imion, a następnie w pętli for przypisywaliśmy zmiennej imie imiona z listy i drukowaliśmy. W tym, momencie warto wspomnieć o instrukcji enumerate(). Poprawmy poprzedni przykład:

```
>>> for i,imie in enumerate(lista):
    print(str(i+1)+': '+imie)
1: Anna
2: Katarzyna
3: Robert
4: Stanislaw
5: Hubert
6: Michal
```

W pierwszej linii jako zmienne sterujące podaliśmy dwie zmienne: **i** oraz **imie**, a za słowem kluczowym **in** umieściliśmy instrukcje **enumerate**(lista). To spowodowało przypisywanie numeru elementu w zbiorze do zmiennej **i** oraz samej wartości elementu zbioru do zmienne **imie**. Obie zmienne drukujemy za pomocą komendy print ale, żeby numeracja zaczynała się od 1 do zmiennej **i** dodajemy 1 a wynik zamieniamy na łańcuch tekstowy funkcją **str**(). Dalsza część argumentu instrukcji print jest oczywista.

2.3. Pętla while

Gdy nie wiemy ile razy działanie naszej pętli będzie wykonane, wówczas korzystamy z pętli **while**, o której zakończeniu decyduje warunek logiczny. W przykładzie spróbujemy powtórzyć działanie pętli **for** z pierwszego przykładu:

```
>>> i =0
>>> while i<5:
    print(i)
    i+=1
0
1
2
3
4
```

W pierwszej linii zdefiniowaliśmy zmienną **i** nadając jej wartość początkową równą 0. Główna linia kodu uruchamia pętlę **while**. Zaczyna się od słowa kluczowego **while**, następnie definiujemy wyrażenie logiczne, które może przyjmować tylko dwie wartości **True** (prawda) i **False** (fałsz). Pętla **while** będzie działać dopóki wyrażenie będzie miało wartość **True**. Linijka zakończona jest dwukropkiem. Poniżej wpisaliśmy komendę drukującą aktualną wartość zmiennej **i**, a w ostatniej linijce kodu zwiększamy wartość **i** o 1. Ostatnia linijka jest bardzo ważna. Gdybyśmy jej nie wpisali zmienna i zawsze równa by była 0, a to oznacza wykonywanie pętli while w nieskończoność tak jak w przykładzie poniżej:

```
>>> while True:
    print('niekonczaca sie petla')
niekonczaca sie petla
niekonczaca sie petla
niekonczaca sie petla
niekonczaca sie petla
```

Napis ‘niekonczaca się petla’ będzie się wyświetlać w nieskończoność. Jedynym wyjściem w takim wypadku jest awaryjne zatrzymanie programu. Robimy to wciskając kombinację klawiszy <Enter> i <CTRL>. W środowisku Thonny można też skorzystać z czerwonego przycisku STOP w pasku narzędzi w górnej części aplikacji.

2.4. Przerywanie działania pętli

Python oferuje kilka możliwości przerwania lub „zakłócenia" działa pętli. Pierwszą jest komenda **break**, która powoduje bezwzględne wyjście z pętli. Wersja dla pętli **while** wygląda jak poniżej:

```
>>> i=0
>>> while i<5:
    print(i)
    i+=1
    if i==3:break
0
1
2
```

A tak wygląda użycie komendy break w pętli **for**.

```
>>> for i in range(5):
    print(i)
    if i==2:break
0
1
2
```

W przypadku, gdy w pętli chcemy pominąć wybrany krok możemy użyć komendy **continue**.

```
>>> for i in range(10):
>>> for i in range(10):
    if i%2==0:continue
    print(i)

1
3
5
7
9
```

W przykładzie drukujemy liczby nieparzyste w zakresie 0 do 10. W pierwszej linii wykonywanej w pętli sprawdzamy czy liczba jest parzysta i jeśli tak, jest to nie wykonujemy dalszych poleceń pętli w tym kroku.

3. Funkcje

Często zdarza się, że jakieś czynności są w trakcie zadania wykonywane wielokrotnie. Wygodnie w takim przypadku będzie je zgrupować i oznaczyć specjalną nazwą, którą będziemy wywoływać w razie potrzeby. Zbór poleceń wykonywanych pod jedną nazwą to po prostu funkcja. Formalna struktura funkcji nie jest skomplikowana. Zaczyna się od słowa kluczowego **def** po którym podajemy nazwę funkcji. Nazwa zakończona jest nawiasami wewnątrz których można zdefiniować parametry wejściowe. Jeśli funkcja ma za zadanie policzyć coś to musi zwrócić wartość, dlatego w wielu przypadkach funkcja zawiera słowo kluczowe **return**, po którym definiujemy zwracaną wartość. W przykładzie przeanalizujemy funkcję, która sprawdza czy wybrana liczba jest liczbą pierwszą. Tym razem dla wygody nie będziemy korzystać z okienka powłoki, ale z okienka do edycji programów.

```
from math import sqrt
def pierwsza(n):
    if n<2: return False
    for i in range(2,int(sqrt(n))):
        if n%i==0: return False
    return True
```

Zaczynamy od pobrania funkcji sqrt z biblioteki math, która wyznacza pierwiastek liczby. O wykorzystywaniu bibliotek jeszcze będziemy pisać, teraz skupimy się tylko na funkcji. Zaczyna się od słowa kluczowego **def**. Po nim następuje nazwa funkcji **pierwsza**. Argumentem funkcji jest liczba **n**, którą będziemy testować. W pierwszej linii kodu sprawdzamy czy liczba jest mniejsza od 2. Wówczas nie może być liczbą pierwszą i nasza funkcja musi zwrócić wartość False. Wykonanie instrukcji **return** powoduje zakończenie działania funkcji. Jeśli jednak liczba jest większa od 2, wówczas musimy ją dzielić przez wszystkie liczby mniejsze lub równe pierwiastkowi tej liczby. Jeśli jakaś podzieli liczbę będącą argumentem to oznacza, że mamy do czynienia z liczbą złożoną więc funkcja musi zwrócić wartość False. Jeśli po wykonaniu wszystkich wymaganych działań nie znaleźliśmy liczby dzielącej argument bez reszty to oznacza, że mamy do czynienia z liczbą pierwszą. To oznacza, że funkcja powinna zwrócić wartość True. Teraz wykorzystamy funkcję pierwsza do zliczenia wszystkich liczb mniejszych od 100. Dopisujemy poniższy kod:

Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki” nr FERS.01.05-IP.08-0234/23

```
ilosc=0
for i in range(100):
    if pierwsza(i)==True:ilosc+=1
print('Ilosc liczb pierwszych mniejszych niz 100 wynosi', ilosc)
```

W wyniku otrzymujemy komunikat:

Ilosc liczb pierwszych mniejszych niz 100 wynosi 25

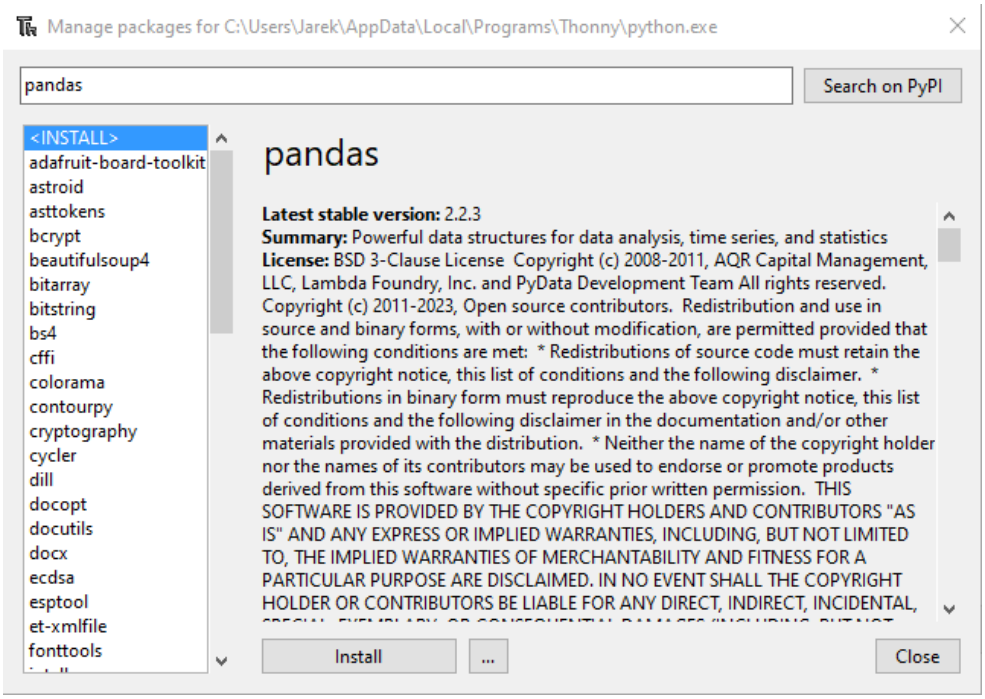
4. Moduły

Jedna z największych zalet Python'a jest olbrzymia ilość bibliotek grupujących tematycznie funkcje. Część z modułów jest dostarczana razem z dystrybucją, a inne trzeba zainstalować samodzielnie. Biblioteki nie są automatycznie ładowane w czasie uruchamiania programu, ponieważ nie są potrzebne a ich inicjowanie zajmowałaby dużo czasu. dołączamy je do programu, gdy uznamy, że są nam potrzebne. W ostatnim przykładzie skorzystaliśmy z biblioteki funkcji matematycznych importują funkcję **sqrt** wyznaczającą pierwiastek kwadratowy argumentu. Czasem przed użyciem biblioteki musimy ją wcześniej zainstalować. W środowisku Thonny jest to bardzo łatwe. Wystarczy wybrać Tools/Manage packages. Następnie w polu widocznym na Rys. 2 wpisujemy poszukiwany moduł i wybieramy „Search on PyPI". To spowoduje przeszukanie zasobów internetowych i stworzenie listy modułów o podobnej nazwie. Wybranie któregoś wyświetli jego opis, abyśmy mogli sprawdzić czy to właśnie poszukiwany moduł. Pozostaje wcisnąć przycisk „Install". W przypadku gdy wybrany moduł jest już zainstalowany zamiast przycisku „Install" pojawi się przycisk „Upgrade", który będzie aktywny jeśli będzie istniała nowsza wersja modułu.
Nie musimy się ograniczać do modułów pobieranych z Internetu. Może stworzyć własny moduł i zaimportować go do programu. Stwórzmy więc przykładowy moduł.

```
def silnia(n):
    f=1
    for i in range(1, n + 1):
        f *= i
    return f
def szescian(n):
    return n**3
if __name__ == '__main__':
    print('Silnia 3 =',silnia(3))
```

print('2 do potęgi 3 =',szescian(2))

[Tekst alternatywny. Okno zarządzania modułami dostępnymi w środowisku Thonny w górnej części posiada pole do wprowadzenia nazwy modułu a obok przycisk „Seach on PyPI". Poniżej po lewej znajduje się lista zainstalowanych modułów, za po prawej duże okno z opisem wybranego modułu. Po oknem z opisem znajdują się trzy przyciski: „Instal". „…" i „Close".]



Rys. 2. Okno zarządzania modułami w środowisku Thonny.

Nasz przykładowy moduł nazwiemy po prostu funkcje.py. Zawiera on dwie funkcje silnia i sześcian. Aby móc przetestować funkcje wewnątrz modułu wstawiliśmy instrukcję warunkową **if**, która gwarantuje wykonanie kodu tylko wtedy, kiedy uruchamiamy moduł z funkcjami. Kiedy uruchamiamy program specjalna zmienna __name__ przybiera wartość __main__. Jeśli wywołamy funkcję z modułu w innym programie zmienna __name__ będzie miała inną nazwę i końcówka programu nie wykona się. W kolejnym kroku stworzymy program wykorzystujący moduł i zapiszemy go w tym samym katalogu co moduł funkcje.py pod nazwą przykład.py:

import funkcje
print(funkcje.silnia(3))

Po wywołaniu programu zobaczymy wynik działania 3!. Zamiast importować cały moduł często wybieramy tylko potrzebne funkcje. Kolejny przykład obrazuje opisaną sytuację. Z modułu funkcje importujemy tylko funkcję sześcian.

```
from funkcje import szescian
print(szescian(3))
```

Jak łatwo się domyślić w wyniku uzyskamy wynik działania 3^3. Ponieważ nazwy funkcji często są dość rozbudowanie i ogólnie dość niewygodne w użyciu możemy importowanej funkcji nadać naszą własną nazwę:

```
from funkcje import szescian as s
print(s(4))
```

W przykładzie **funkcji** sześcian nadano nazwę **s**.

5. Obsługa błędów

Ważnym elementem każdego programu jest określenie sposobu w jaki sobie radzi z błędami np. nieprawidłowymi danymi wejściowymi. służy do tego blok try…except. Napiszmy najprostszy jedno-linijkowy program.

```
print(3/0)
```

W wyniku zobaczymy komunikat:

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ZeroDivisionError: division by zero
```

Niestety program wykrył dzielenie przez zero i dość nieelegancko powiedział nam o tym. Możemy coś z tym zrobić.

```
try:
    print(3/0)
except:
    print('Cos poszlo nie tak')
```

Tym razem w wyniku otrzymujemy komunikat:

```
>>> %Run -c $EDITOR_CONTENT
Cos poszlo nie tak
```

Podobny wynik otrzymamy jeśli użyjemy kodu:

```
try:
    print(3/0)
except ZeroDivisionError:
    print('Cos poszlo nie tak')
```

Spróbujmy czegoś bardziej rozbudowanego:

```
try:
    a=input()
    wynik=3/int(a)
except ZeroDivisionError:
    print('Nie mozna dzielic przez zero')
except ValueError:
    print('Nie mozna podzielić przez tekst')
else:
    print('Wszystko przebieglo prawidlowo, wyniki wynosi ',wynik)
finally:
    print('procedura zakonczona')
```

Widzimy pełną postać procedury obsługi błędów. Składa się ona z bloku **try**, w którym program próbuje coś wykonać, następnie mamy blok **except** obsługujący błędy, kolejny element to blok **else**, po którym następują komendy jeśli pogram zadziałał bez błędów i ostatecznie blok finalny, który wykonuje się zawsze (jeśli jest zdefiniowany). Spróbuj wykonać trzykrotnie program podając jako argument 5, 0 i „a”. Program za każdym razem zadziała inaczej. Najpopularniejsze błędy pokazano w Tabeli 5.

Tabela 5. Lista popularnych błędów

Błąd	Opis
ArithmeticError	Jest wywoływany w przypadku błędów w operacjach arytmetycznych.
ImportError	Błąd pojawia się, gdy próbujesz zaimportować moduł, który nie jest obecny. Ten rodzaj wyjątku ma miejsce, jeśli popełniłeś błąd

	w nazwie modułu lub w przypadku modułu, którego nie ma w standardowej ścieżce.
IndexError	Błąd zgłaszany podczas próby odniesienia się do elementu o indeksie, który jest poza zakresem.
ZeroDivisionError	Ten typ błędu pojawia się podczas dzielenia przez zero.
OSError	Błąd pojawia się, gdy operacja wejścia/wyjścia się nie powiedzie.
ValueError	błąd pojawia się w sytuacji gdy funkcja otrzymuje argument prawidłowego typu ale o nieprawidłowej wartości.
FloatingPointError	Błąd pojawia się gdy zmiennoprzecinkowa operacje nie powiedzie się.
NameError	Błąd pojawia się kiedy w programie odwołasz się do niezdefiniowanej zmiennej.

Zawsze może się jednak pojawić sytuacja, w której błąd wykracza poza wszystkie zdefiniowane rodzaje błędów. Wtedy można obsłużyć błąd samodzielnie. Najprostszym rozwiązaniem jest zastosowanie słowa kluczowego raise. Spróbujmy poniższego kodu:

```
a=int(input())
wynik=3/a
if a==5:
    raise TypeError("Wartość 5 jest niedozwolona")
```

Uruchomienie programu i podanie wartości 5 spowoduje awaryjne wyjście z programu:

```
>>> %Run -c $EDITOR_CONTENT
5
Traceback (most recent call last):
  File "<string>", line 4, in <module>
TypeError: Wartość 5 jest niedozwolona
```

Możemy też wpleść wygenerowany błąd do programu w taki sposób, aby obsługiwał go blok try..except.

```
try:
    a=int(input())
    wynik=3/a
    if a==5:
        raise TypeError('')
except ZeroDivisionError:
```

```
    print('Nie mozna dzielic przez zero')
except ValueError:
    print('Nie mozna podzielić przez tekst')
except:
    print('Wartość 5 jest niedozwolona')
else:
    print('Wszystko przebieglo prawidlowo, wyniki wynosi ',wynik)
finally:
    print('procedura zakonczona')
```

W tym wypadku po podaniu wartości 5 program zamknie się elegancko z odpowiednim komunikatem:

```
>>> %Run -c $EDITOR_CONTENT
5
Wartość 5 jest niedozwolona
procedura zakonczona
```

6. Klasy

Klasy definiują sposób w jaki tworzone są obiekty, określają ich cechy i specyficzne „zachowania" (Orbaiceta, 2021). Spróbujmy zdefiniować klasę obiektów typu punkt i wektor. Klasę zaczynamy definiować od słowa kluczowego **class**. Po niej następuje nazwa klasy. Żeby uniknąć pomyłek przyjmijmy zasadę, że nazwa klasy zaczyna się wielką literą. Pierwszą funkcją jaką definiujemy jest specjalna funkcja **__init__(self,x,y)**, która wykonuje się jednorazowo przy tworzeniu nowego obiektu tej klasy. W przypadku punktu funkcja **__init__()** definiuje położenie punktu, dlatego przy tworzeniu obiektu wymagane jest podanie dwóch współrzędnych. Przy definicji pojawia się jednak trzeci parametr self. Parametr ten mówi innym funkcjom w klasie, że wywołuje je inna funkcja z tej samej klasy. Jak zobaczymy później w trakcje wywoływania funkcji nie jest on formalnie wymagany. W ramach klasy Punkt wprowadziliśmy dwie funkcje. Pierwsza generuje nowy punkt na podstawie wybranego punktu i zdefiniowanego wektora przemieszczenia, a druga wyznacza odległość między wybranym punktem a innym podanym przywołaniu funkcji. W przypadku klasy Wektor widać, że obiekt typu wektor powstaje na podstawie dwóch przyrostów w kierunkach x i y. W przypadku klasy Wektor zdefiniowaliśmy dwie funkcje, które pozwalają na wyznaczanie długości wektora i jego cosinusów kierunkowych.

```
from math import sqrt
```

```
import matplotlib.pyplot as plt

class Punkt:
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def przesuwanie(self,wektor):
        return Punkt(self.x+wektor.dx,self.y+wektor.dy)
    def odleglosc(self,drugi):
        dx=self.x-drugi.y
        dy=self.y-drugi.y
        return sqrt(dx**2+dy**2)

class Wektor:
    def __init__(self,dx,dy):
        self.dx=dx
        self.dy=dy
    def dlugosc(self):
        return sqrt(self.dx**2+self.dy**2)
    def wersor(self):
        return (self.dx/self.dlugosc(),self.dy/self.dlugosc())
```

Spróbujmy skorzystać z opisanych klas tworząc prosty program. W pierwszej linii inicjujemy listę punktów. Do tej listy wstawiamy trzy punkt. Pierwszy ma współrzędne (0,0) a pozostałe dwa powstały przez przesunięcie pierwszego punktu o odpowiednie wektory. W kolejnych liniach zainicjowano listy wektorów oraz współrzędnych x i y w celu stworzenia wykresu. Kolejnym etapem jest generowanie wektorów odpowiadającym kolejnym bokom trójkąta tworzonego przez trzy punkty na liście lista i dopisywanie współrzędnych punktów z listy do zbiorów współrzędnych x i y. Po wyjściu z pętli dopisujemy dodatkowo ona końcu list współrzędne punktu początkowego aby na wykresie uzyskać zamknięty kształt. W dalszych liniach tworzymy wykres oraz wypisujemy na ekranie dane na temat naszego kształtu wykorzystując funkcje zdefiniowane dla wektorów. W końcowej części programu definiujemy wektor przemieszczenia i tworzymy na bazie pierwotnego kształtu nowy, przesunięty, który również umieszczamy na wykresie.

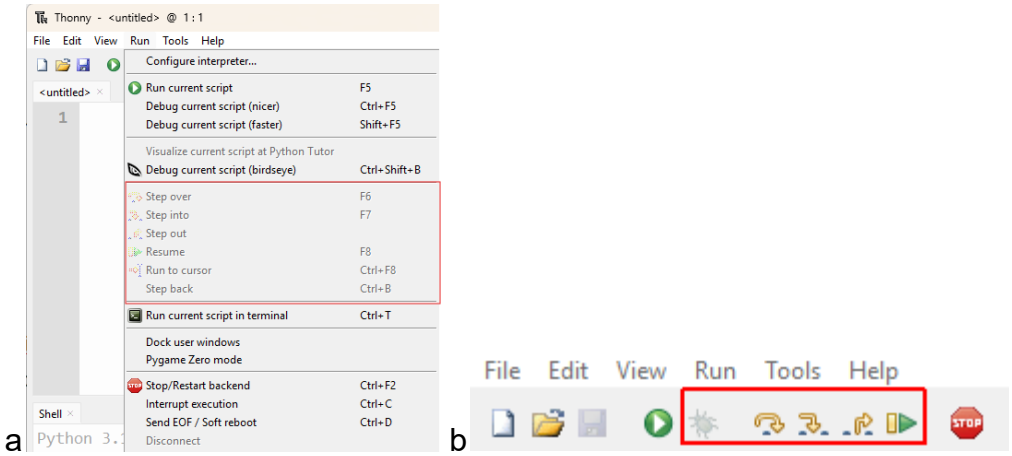
```
lista=[]
lista.append(Punkt(0,0))
lista.append(Punkt(0,0).przesuwanie(Wektor(1,0)))
lista.append(Punkt(0,0).przesuwanie(Wektor(0.5,sqrt(3)/2)))
boki=[]
```

```
x=[]
y=[]
for i,punkt in enumerate(lista):
    bok=Wektor(lista[(i+1)%2].x-punkt.x,lista[(i+1)%2].y-punkt.y)
    boki.append(bok)
    x.append(punkt.x)
    y.append(punkt.y)
x.append(x[0])
y.append(y[0])
plt.plot(x,y, marker='+', color='b', markersize=20)
plt.gca().set_aspect('equal')
print('Dlugosci bokow')
for i,bok in enumerate(boki):
    print('Bok nr {0:}: {1:.2f}'.format(i,bok.dlugosc()))
print('Cosinusy kierunkowe bokow')
for i,bok in enumerate(boki):
    print('Bok nr {0}: {1:.3f},{2:.3f}'.format(i,bok.wersor()[0],bok.wersor()[1]))
nowa_lista=[]
wektor_przmieszczenia=Wektor(5,3)
for i in lista:
    nowa_lista.append(i.przesuwanie(wektor_przmieszczenia))
nowe_boki=[]
nowe_x=[]
nowe_y=[]
for i,punkt in enumerate(nowa_lista):
    bok=Wektor(nowa_lista[(i+1)%2].x-punkt.x,nowa_lista[(i+1)%2].y-punkt.y)
    nowe_boki.append(bok)
    nowe_x.append(punkt.x)
    nowe_y.append(punkt.y)
nowe_x.append(nowe_x[0])
nowe_y.append(nowe_y[0])
print('Odleglosc miedzy odpowiednimi wierzcholkami trojkata:
',nowa_lista[0].odleglosc(lista[0]))
plt.plot(nowe_x,nowe_y, marker='+', color='r', markersize=20)
plt.show()
```

7. Odpluskwianie programu

Ciekawą zaletą pracy w środowisku jest możliwość skorzystania z wbudowanego debuggera. Jego komendy można znaleźć w menu Run (Rys.3a), lub wprost w oknie programu (Rys.3b).

[Tekst alternatywny. Na lewym rysunku widać rozwinięte menu ukazujące się po wciśnięciu napisu „Run”. Czerwona ramka otacza opcje odpluskwiacza. Na rysunku po prawej widać fragment paska narzędzi z przyciskami poleceń wykorzystywanymi podczas odpluskwiania otoczonymi czerwoną ramką.]



Rys. 3. Położenie komend do odpluskwiania programu w menu głównym (a) i na pasku przycisków (b).

Aby możliwe było odpluskwianie musimy dysponować zapisanym programem. Spróbujmy znaleźć błąd w poniższym kodzie:

```
i =0
while i<5:
    k=2*i+5
    print(k)
```

Przepisujemy go i zapisujemy pod dowolna nazwą. Jeśli go uruchomimy klawiszem F5 program zacznie drukować w nieskończoność wartość 5. Ewidentnie nie to mieliśmy na celu. Zatrzymujemy program przyciskiem STOP lub <CTRL>+F2 i uruchamiamy w trybie odpluskwiania. Możemy to zrobić wchodząc do menu Run i wybierając jedną z opcji Debug… lub wciskając kombinację klawiszy <CTRL>+F5 lub

<SHIFT>+F5 (możemy uruchomić moduł szybszy lub efektywniejszy w działaniu, ale na naszym poziomie to nie ma znaczenia). Ważne jest żeby przed uruchomieniem po prawej stronie okresu otworzyć blok ze zmiennymi. Robimy to wchodząc do menu View i zaznaczamy ptaszek dla pozycji Variables. Po wciśnięciu <CTRL>+F5 program przejdzie do pierwszej linii i poczeka na naszą decyzję. Wciśniecie F6 lub przycisku „Step over" spowoduje, że program spróbuje wykonać kolejny blok (czyli pętlę while) i zatrzymać się za nią. Zgodna na takie działanie spowoduje że program „wpadnie" ponownie w nieskończoną pętlę. Musimy program zatrzymać za pomocą komendy STOP i ponownie uruchomić odpluskwianie, ale tym razem dalsze wykonywanie będziemy wymuszać komendą „Step into" w menu lub za pomocą odpowiedniego przycisku, lub klawiszem F7. Wciskanie klawisza F7 powoduje że widzimy krok po kroku co robi program. Widać jak za zmienne podstawia wartości porównuje je w instrukcji warunkowej wyznacza wartość zmiennej i drukuje ją. Aktualne wartość wykorzystywanych zmiennych widać w oknie Variables (Rys. 4). Po pojawieniu się w oknie powłoki „5" program wraca do początku pętli while i jak widać zmienna i ponownie ma wartość 0.

[Tekst alternatywny. Rysunek prezentuje okno programu w trakcie odpluskwiania. W lewej części zamiast warunku widać małe pole z napisem „0<5". Po prawej pokazano listę zmiennych z aktualnymi wartościami.]

Rys. 4. Podgląd wartości zmiennych podczas odpluskwiania programu.

Już widać jaki błąd zrobiliśmy. Brakuje linii inkrementującej zmienną sterującą i. Dopisujemy ją i ponownie uruchamiany program:

```
i =0
while i<5:
    k=2*i+5
    print(k)
    i+=1
```

Teraz program działa prawidłowo.

8. Przebieg zmienności funkcji

Jedna z ważniejszych czynności inżynierskich jest analiza przebiegu zmienności funkcji. Podstawową czynnością w takim wypadku jest stworzenie wykresu aby widzieć obiekt zainteresowania.

8.1. Wykres

Do tworzenia wykresu możemy wykorzystać jedną z dwóch bibliotek: prostszą – pylab lub bardzo rozbudowaną – mathpoltlib. Ponieważ prostszy – pylab jest obecnie niezalecanym narzędziem jako przestarzałe skupimy się na bibliotece matplotlib, choć trzeba pamiętać, na naszym poziomie komendy tworzące wykres w obu przypadkach są takie same. Abyśmy mogli użyć wybranej biblioteki w programie musimy na początku tego programu wpisać komendę:

```
from matplotlib.pyplot import plot, show
```

która zaimportuje dwie funkcje plot i show, będące składnikami pakietu matplotlib. Pierwsza generuje wykres, a druga go wyświetla. Do narysowania funkcji potrzebujemy punktów o współrzędnych x i y. Do wyznaczania wartości funkcji wykorzystamy specjalnie zdefiniowaną przez nas funkcję:

```
def funkcja(x):
    return x**3-3*x**2-10*x+2
```

Struktura funkcji w tym wypadku jest bardzo prosta. Po słowie return wpisujemy wartość jaką funkcja ma zwracać. W naszym przypadku chodzi o funkcję (1):

$$y = x^3 - 3x^2 - 10x + 2 \tag{1}$$

Kolejnym krokiem niezbędnym do narysowania wykresu jest zdefiniowanie pustych zbiorów, w których będziemy przechowywać wartości „x” i „y” wyświetlanych punktów. Formalnie zbiory te będą listami.

```
x=[]
y=[]
```

Listy te uzupełnione zostaną wartościami przy wykorzystaniu pętli „for”.

```
for i in range(21):
    pom=(i-10)/2
    x.append(pom)
    y.append(funkcja(pom))
```

Wykres będzie zbudowany z 21 punktów dlatego zmienna sterująca w pętli „i” zmieniać będzie wartości od 0 do 20 (range(21)). W pierwszej linijce kodu odpowiadającego pętli for dokonamy prostego podstawienia (pom=(i-10)/2) dzięki któremu wartości zmienne „x” będą się zmieniać od -5 do 5. Tak przeliczone wartości dodajemy do listy wartości „x” za pomocą komendy append, a następnie dodajemy wartości „y” wyznaczone dzięki wcześniej zdefiniowanej funkcji.
Pozostaje stworzyć wykres. Najpierw wywołujemy funkcje plot której dwoma argumentami są dwa zbiory z wartościami „x” i „y”. Funkcja plot() zwraca obiekt, który zawiera wszystkie informacje o naszym wykresie. Żeby go wyświetlić wywołujemy funkcję show().

```
plot(x,y)
show()
```

Uruchomienie programu spowoduje pojawienie się na ekranie Rys. 5.
Pod rysunkiem pojawia się kilka przycisków. Ich znaczenie jest dość oczywiste, szczególnie po ich wypróbowaniu.

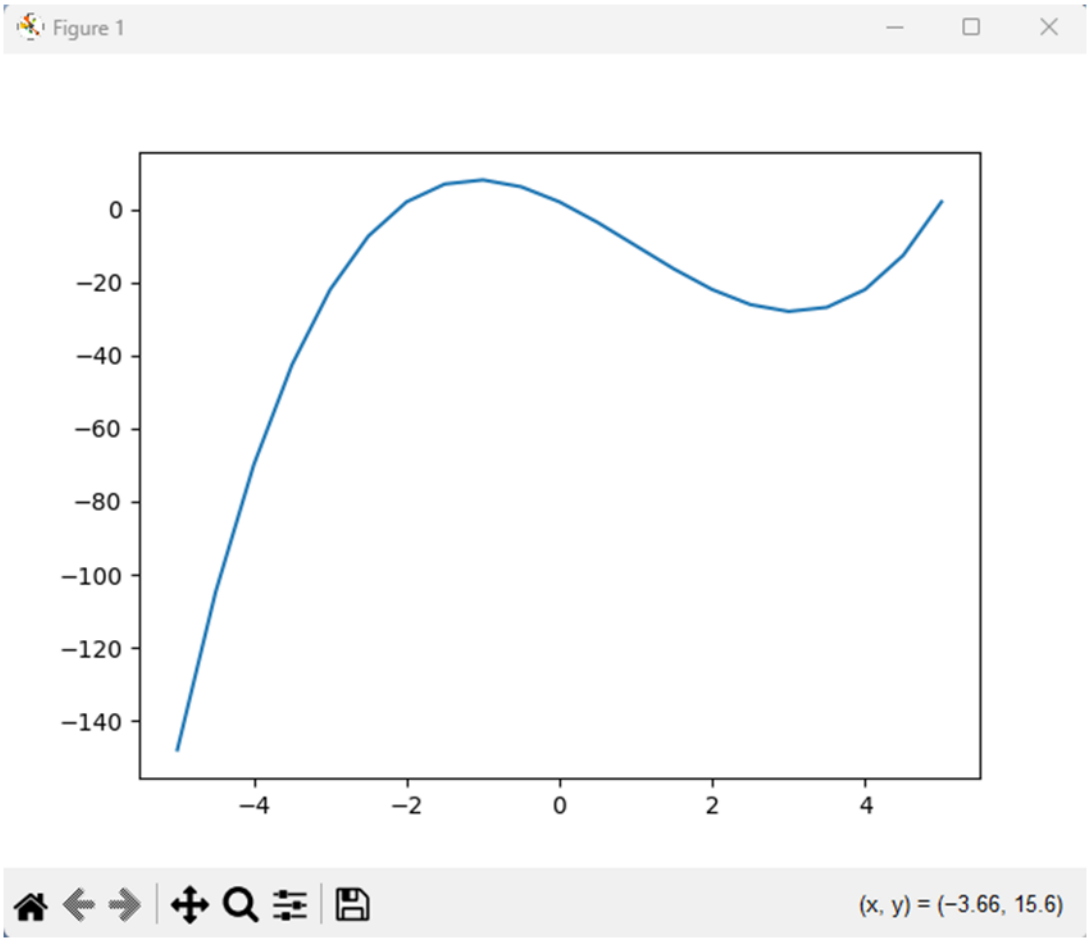
8.2. Modyfikacje wykresu

Taki wykres może być przydatny, gdy chcemy szybko podejrzeć uzyskane wyniki, ale do zaprezentowania do innym osobom wymaga wprowadzenia kilku poprawek. Pierwsza poprawką będzie wstawienie znaczków dla zdefiniowanych punktów. Robimy to poprawiając linijkę według wzoru poniżej:

```
plot(x,y, marker='o')
```

Opcja **marker** pozwala określić kształt stawianych znaczków. Za pomocą opcji **markersize** możemy ustalić wielkość znaczka. Opcja **color** pozwala określić kolor linii i znaczników, zaś opcja **linestyle** określa rodzaj linii. Wartości poszczególnych opcji zebrano w Tabeli 6.

[Tekst alternatywny. Rysunek prezentuje wykres funkcji narysowanym za pomocą niebieskiej, ciągłej linii. Osie wykresu nie mają opisu poza wartościami. Pod wykresem znajduje się 7 przycisków do manipulacji wyświetlanym zakresem funkcji i do zapisu.]



Rys 5. Okno wykresu.

Tabela 6. Opcje dla elementów graficznych wykresu

Kształt znacznika (marker)	Kolor (kolor)	Styl linii (linestyle)
'o' duży okrąg	'r' = czerwony	'-' linia ciągła
'*' gwiazdka	'g' = zielony	'--' linia kreskowa
'.' mały okrąg	'b' = niebieski	'-.' linia punktowa
',' punkt	'c' = cyjan	':' linia kropkowa
'x' X	'm' = różowy	
'X' pogrubiony X	'y' = żółty	
'+' plus	'k' = czarny	
'P' pogrubiony plus	'w' = biały	
's' wypełniony kwadrat		
'D' wypełniony romb		

'd' wypełniony cienki romb		
'p' pięciokąt		
'H', 'h' sześciokąt		
'v' trójkąt w dół		
'^' trójkąt w górę		
'<' trójkąt w lewo		
'>' trójkąt w prawo		
'1' gwiadka w dół		
'2' gwiadka w górę		
'3' gwiadka w lewo		
'4' gwiadka w prawo		
'|' pionowa linia		
'_' pozioma linia		

Przykładowa komenda

plot(x,y, marker='X', color='b', linestyle=':', markersize=12)

spowoduje narysowanie niebieskiej linii za pomocą kropek z punktami w kształcie pogrubionych **X** o wielkości 12 pkt.
Kolejnym krokiem jest dodanie tytułu wykresu oraz tytułów osi. Zadanie jest bardzo proste, ale przed jego wykonaniem musimy wzbogacić pierwszą linijkę programu o wywołanie niezbędnych funkcji. Poprawiamy pierwszą linijkę według przykładu.

from matplotlib.pyplot import plot, show, title, xlabel, ylabel

Następnie przed linijką z funkcją show() wprowadzamy trzy komendy definiujące kolejno opis osi x, opis osi y oraz tytuł wykresu.

xlabel('tytul osi x')
ylabel('tytul osi y')
title('probny wykres')

Wykres będziemy jeszcze modyfikować, ale przed tym wykonamy czynności pomocnicze.

8.3. Wyznaczanie pochodnej

Znajomość pochodnej funkcji jest niezbędna do oceny przebiegu zmienności funkcji. Do wyznaczenia pochodnej naszej funkcji wykorzystamy Python. Postać pochodnej wyznaczymy w nowym programie za pomocą obliczeń symbolicznych.
Pierwszym krokiem jest wywołanie niezbędnych funkcji z potężnej biblioteki sympy.

```
from sympy import Symbol, Derivative
```

Funkcja Symbol() pozwoli na zdefiniowanie zmiennej względem której wyznaczona zostanie pochodna. Druga funkcja wyznaczy pochodną tzn. na podstawie zdefiniowanej funkcji wyznaczy jej pochodną.
Definiujemy w programie zmienną „x" którą we wpisanym wzorze będzie, jakżeby inaczej, litera „x".

```
x = Symbol('x')
```

W kolejnym kroku definiujemy postać funkcji dla której wyznaczymy pochodną (to ta sama funkcja, którą rysowaliśmy w poprzednim etapie).

```
f = x**3-3*x**2-10*x+2
```

Dalej przypisujemy zmiennej „d" wartość pochodnej.

```
d = Derivative(f, x)
```

Na koniec wyświetlamy ją (na tym etapie użyjemy pochodnej w najprostszy możliwy sposób po prostu przepisując ją, ale należy pamiętać, że możliwości w tym zakresie są dużo większe).

```
print(d.doit())
```

W wyniku uzyskamy pochodną w postaci:

```
3*x**2 - 6*x – 10
```

Pełna wersja programu wygląda następująco:

```
from sympy import Symbol, Derivative
x = Symbol('x')
f = x**3-3*x**2-10*x+2
d = Derivative(f, x)
print(d.doit())
```

8.4. Dodawanie wykresu do już istniejącego

Wracamy do głównego programu, w którym dorysujemy pochodną.
Pod definicją funkcji „funkcja” dopisujemy nową funkcję wyznaczającą wartość pochodnej:

```
def pochodna(x):
    return 3*x**2-6*x-10
```

Druga modyfikacja polega na uzupełnieniu kodu. Po linijkach definiujących puste listy dla współrzędnych „x” i „y” wstawiamy definicję pustej listy dla wartości pochodnej.

```
yp=[]
```

Na końcu pętli for dopisujemy linijkę dodającą do nowo zadeklarowanej listy wartości pochodnej.

```
yp.append(pochodna(pom))
```

Pod komendą plot wstawiamy następną komendę, która dorysuje wykres pochodnej.

```
plot(x,yp, marker='x', color='r', linestyle='-', markersize=2)
```

Po narysowaniu tak zdefiniowanego wykresu ujrzymy drugą linię w kolorze czerwonym, która prezentuje pochodną (dla naszej funkcji pochodna jest parabolą). W naszym zadaniu bardzo przydałoby się widzieć gdzie pochodna przecina oś X. W tym celu wykorzystamy funkcję axhline. Musimy dokonać modyfikacji pierwszej linii aby zaimportować potrzebną nam funkcję.

```
from matplotlib.pyplot import plot, show, title, xlabel, ylabel, axhline
```

Teraz przed funkcją show() dopisujemy linijkę:

axhline(y=0, color='k')

dzięki, której na wykresie pojawi się pozioma linia w kolorze czarnym o współrzędnej y=0. Pozostał jeszcze jeden mały problem. Aby rozróżniać linie wstawimy legendę. W pierwszej linii wymuszamy więc import funkcji legend.

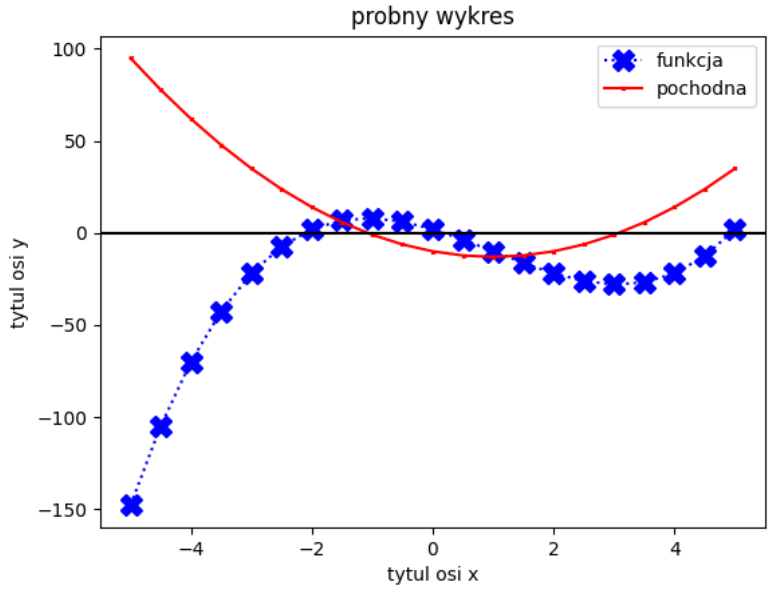
from matplotlib.pyplot import plot, show, title, xlabel, ylabel, axhline, legend

Teraz przed komendą show() dopisujemy komendę.

legend(['funkcja','pochodna'])

Po uruchomieniu programu otrzymamy wykres widoczny na Rys. 6.

[Tekst alternatywny. Rysunek prezentuje wykres funkcji narysowanym za pomocą niebieskiej, punktowej linii łączącej punkty w postaci grubych iksów. Druga funkcja to pochodna narysowana za pomocą czerwonej, ciągłej linii. W prawym, górnym rogu znajduje się legenda. Oś pozioma na tytuł: „tytul os x", a oś pionowa ma tytuł: „tytul osi y".]



Rys. 6. Wykres funkcji i jej pochodnej.

Napisany przez nas program ma poniższą postać.

```
from matplotlib.pyplot import plot, show, title, xlabel, ylabel, axhline, legend
def funkcja(x):
    return x**3-3*x**2-10*x+2
def pochodna(x):
    return 3*x**2-6*x-10
x=[]
y=[]
yp=[]
for i in range(21):
    pom=(i-10)/2
    x.append(pom)
    y.append(funkcja(pom))
    yp.append(pochodna(pom))
plot(x,y, marker='X', color='b', linestyle=':', markersize=12)
plot(x,yp, marker='x', color='r', linestyle='-', markersize=2)
xlabel('tytul osi x')
ylabel('tytul osi y')
title('probny wykres')
axhline(y=0, color='k')
legend(['funkcja','pochodna'])
show()
```

8.5. Wyznaczanie miejsc zerowych pochodnej

Najbardziej interesujące z naszego punktu widzenia są miejsca zerowe pochodnej. Potencjalnie tam możemy znaleźć ekstremum analizowanej funkcji. Stworzymy program, który w szybki sposób wyznaczy interesujące nas wartości. W pierwszej linii importujemy niezbędne funkcje.

```
from sympy import Symbol, Derivative, solve
```

W kolejnych liniach definiujemy zmienną niezależną oraz naszą funkcję.

```
x = Symbol('x')
f = x**3-3*x**2-10*x+2
```

Teraz nakażemy Python'owi wyznaczyć symbolicznie postać pochodnej i przypiszemy ją do zmiennej „d". Aby zobaczyć rozwiązanie wydrukujemy je na ekranie.

```
d=Derivative(f,x).doit()
print("y'=",d)
```

Kolejna niepozorna linia stanowi sedno pisanego programu, dzięki niej postanie lista punktów w których pochodna ma wartość 0 i zostanie przypisana do zmiennej „miejsca_zerowe”. Wyniki pozostaje w pamięci komputera, aby je wyświetlić musimy do tego program zmusić komendą print.

```
miejsca_zerowe=solve(d)
print('Miejsca zerowe:')
print(miejsca_zerowe)
```

W wyniku uzyskamy wartości: [1 - sqrt(39)/3, 1 + sqrt(39)/3]
Wykorzystamy je do wyznaczenia wartości funkcji w tych miejscach, jako że z wykresu wynika, że pojawiają się tam lokalne ekstrema funkcji. Żeby to zrobić za pomocą funkcji subs() podstawiamy za „x” w funkcji wartości, w których pierwsza pochodna się zeruje i funkcją evalf() wymuszamy obliczenie wyniku.

```
print('Wartosci funkcji dla miejsc zerowych pochodnej:')
print(f.subs({x:miejsca_zerowe[0]}).evalf())
print(f.subs({x:miejsca_zerowe[1]}).evalf())
```

Efektem działania programu jest poniższy wydruk:

```
y'= 3*x**2 - 6*x - 10
Miejsca zerowe:
[1 - sqrt(39)/3, 1 + sqrt(39)/3]
Wartosci funkcji dla miejsc zerowych pochodnej:
8.04110532870648
-28.0411053287065
```

Pełna wersja programu wygląda następująco.

```
from sympy import Symbol, Derivative, solve
x = Symbol('x')
f = x**3-3*x**2-10*x+2
d=Derivative(f,x).doit()
print("y'=",d)
miejsca_zerowe=solve(d)
```

```
print('Miejsca zerowe:')
print(miejsca_zerowe)
print('Wartosci funkcji dla miejsc zerowych pochodnej:')
print(f.subs({x:miejsca_zerowe[0]}).evalf())
print(f.subs({x:miejsca_zerowe[1]}).evalf())
```

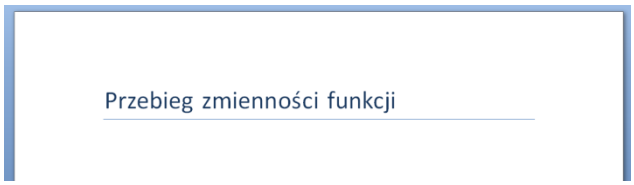
8.6. Tworzenie pliku WORD/PDF

Spróbujmy teraz zapisać wyniki naszych analiz do pliku PDF. Zrobimy to w sposób pośredni, ponieważ moduł obsługujący pliki w formacie PDF nie jest w stanie pracować z tekstem, może jedynie kopiować zawartość całej strony (Sweigart, 2020). Rozwiązaniem jest stworzenie pliku w formacie DOCX i zapisanie go jako PDF. Dostęp do dokumentu w formacie MS Word zapewni nam biblioteka docx (Sweigart, 2020). Na początek zapiszemy dokument z nagłówkiem „Przebieg zmienności funkcji" w pliku o nazwie „test.docx":

```
import docx
doc=docx.Document()
doc.add_heading('Przebieg zmienności funkcji',0)
doc.save('test.docx')
```

W katalogu, w którym uruchomiliśmy program, pojawił się plik test.docx. Po jego otwarciu widać w nim tylko jedną linijkę tekstu tak jak na Rys. 7:

[Tekst alternatywny. Na rysunku znajduje się fragment strony z napisem „Przebieg zmienności funkcji" a pod nim niebieska linia. Strona otoczona jest niebieskim brzegiem.]



Rys. 7. Przykładowa strona z nagłówkiem.

Funkcja add_heading wstawiła odpowiednio ozdobiony tekst. O tym jak nagłówek ma być wyróżniony decyduje parametr podany po przecinku określający jego poziom. W ramach testów można uruchomić program z innymi wartościami w zakresie 0-4.
W drugiej linii wpiszemy treść zadania, tak aby wiedzieć co się znajduje w dokumencie. Będzie to już zwykły paragraf a nie nagłówek, więc polecenie wygląda trochę inaczej:

doc.add_paragraph('Dokonaj analizy przebiegu zmienności funkcji opisanej wzorem y=x^3-3x^2-10x+2')

W kolejnej linii wyznaczymy pochodną wykorzystując wyjaśniany już kod. W pierwszej kolejności dołączamy odpowiednie funkcje z biblioteki SymPy, a następnie na końcu programu dopisujemy definiujemy naszą funkcję i wyznaczamy jej pochodną (Saha, 2015).

```
from sympy import Symbol, Derivative
…
x = Symbol('x')
f = x**3-3*x**2-10*x+2
d = Derivative(f, x).doit()
```

Teraz wynik wstawiamy do plik MS Word. W celu zamiany funkcji na łańcuch tekstowy wykorzystamy funkcję str():

```
…
doc.add_paragraph('Pochodna funkcji y='+str(f)+' równa jest y\'='+str(d))
…
```

W dalszej części wstawimy do naszego raportu wykres. Wykorzystamy znany już kod z małymi usprawnieniami. Rozpoczniemy od nagłówka, a dalej wywołamy bibliotekę matplotlib. Następnie za pomocą znanego już kodu wygenerujemy dane, stworzymy wykres, zapiszemy go do pliku a stworzony plik wstawimy do dokumentu MS Word:

```
…
import matplotlib.pyplot as plt
…
doc.add_heading('Wykres funkcji y='+str(f)+' i jej pochodnej',2)
wsp_x=[]
wsp_y=[]
wsp_yp=[]
for i in range(21):
    wsp_x.append((i-10)/2)
    wsp_y.append(f.subs({x:(i-10)/2}))
    wsp_yp.append(d.subs({x:(i-10)/2}))
plt.plot(wsp_x,wsp_y, marker='X', color='b', linestyle=':', markersize=12)
plt.plot(wsp_x,wsp_yp, marker='x', color='r', linestyle='-', markersize=2)
```

```
plt.xlabel('wartości x')
plt.ylabel('wartości y')
plt.title('wykres funkcji')
plt.axhline(y=0, color='k')
plt.legend(['funkcja','pochodna'])
plt.savefig('funkcja.png')
doc.add_picture('funkcja.png')
```

Na uwagę zasługuje sposób wyznaczania wartości funkcji i pochodnej. W pętli for wykorzystano do tego metodę subs() w której wstawiono zmienną typu słownik (dictionary), opisującą jakie wartości podstawiać pod zmienne.

8.7. Przedziały zmienności

Kolejnym krokiem jest określenie przedziałów zmienności funkcji. Wyznaczamy je rozwiązując równanie pochodnej, po uprzednim importowaniu z biblioteki SymPy funkcji solve. Wyniki przypisujemy do zmiennej miejsca_zerowe i wstawiamy wynik w estetycznej formie do pliku. Ponieważ wiem jak wygląda przebieg pochodnej możemy podsumować nasze badania zdaniem: „W przedziale od –oo do x_0 funkcja jest rosnąca”, „W przedziale od x_0 do x_1 funkcja jest malejąca”, „W przedziale od x_1 do +oo jest rosnąca”.

```
miejsca_zerowe=solve(d)
doc.add_heading('Przedziały zmienności',2)
doc.add_paragraph('Pochodna funkcji y='+str(f)+' ma miejsca zerowe w punktach:')
doc.add_paragraph('x1:'+str(miejsca_zerowe[0])+', x2='+str(miejsca_zerowe[1]))
pom=doc.add_paragraph('W przedziale od ')
pom.add_run('–oo')
pom.add_run(' do ')
pom.add_run(str(miejsca_zerowe[0]))
pom.add_run(' funkcja jest ')
pom.add_run('rosnąca')
pom.add_run('.')
pom.runs[1].bold=True
pom.runs[3].bold=True
pom.runs[5].font.color.rgb = RGBColor(0x00, 0xff, 0x00)
pom=doc.add_paragraph('W przedziale od ')
pom.add_run(str(miejsca_zerowe[0]))
pom.add_run(' do ')
pom.add_run(str(miejsca_zerowe[1]))
pom.add_run(' funkcja jest ')
```

```
pom.add_run('malejąca')
pom.runs[5].font.color.rgb = RGBColor(0xff, 0x00, 0x00)
pom.add_run('.')
pom.runs[1].bold=True
pom.runs[3].bold=True
pom=doc.add_paragraph('W przedziale od ')
pom.add_run(str(miejsca_zerowe[1]))
pom.add_run(' do ')
pom.add_run('+oo')
pom.add_run(' funkcja jest ')
pom.add_run('rosnąca')
pom.add_run('.')
pom.runs[1].bold=True
pom.runs[3].bold=True
pom.runs[5].font.color.rgb = RGBColor(0x00, 0xff, 0x00)
```

W naszym kodzie pojawiło się coś dziwnego. Zamiast wpisać te zdania jako paragrafy, podzielone one zostały na kawałki zwane run. Każdy paragraf składa się z takich kawałków tekstu, które mają odmienne właściwości (np. inny kolor, pogrubienie itp.). Dzięki takiemu podziałowi będziemy mogli wyróżnić wartości ograniczające przedziały. Jak widać wartości naszych miejsc zerowych zostaną pogrubione, a słowa „rosnąca” i „malejąca” będą zapisane odpowiednio kolorem zielonym i czerwonym. Kolor zapisano za pomocą modelu RGB podając intensywność barw składowych: czerwonego, zielonego i niebieskiego. Intensywność można było zapisać za pomocą liczb całkowitych z zakresu 0-255, ale w naszym przypadku pojawił się nietypowy zapis 0xff. Jest to nic innego jak liczba 255, ale w zapisana w system szesnastkowym (o zapisie w tym systemie informuje prefiks „0x”). Żeby wykorzystać funkcję RGBColor musimy jeszcze pobrać jej definicję, dlatego na początku programu dopisujemy fragment:

```
from docx.shared import RGBColor
```

W ostatnim kroku wypiszemy jeszcze miejsca zerowe i lokalne wartości ekstremów. Zaczynamy od odpowiedniego nagłówka. Wyznaczanie miejsc zerowych funkcji, czy poszukiwanie ekstremów, czyli miejsc w których pierwsza pochodnia jest równa zero nie stanowi już dla nas problemu, ale w wyniku naszych działań możemy dostać liczby których format nam nie będzie pasował i musimy nad nim zapanować.

```
doc.add_heading('Miejsca zerowe funkcji',2)
miejsca_zerowe=solve(f)
doc.add_paragraph('Miejsca zerowe funkcji znajdują sie w punktach: ')
```

```
for i,m in enumerate(miejsca_zerowe):
    doc.add_paragraph(str(i+1)+': '+str(m)+' czyli '+str(m.evalf()))
miejsca_zerowe=solve(d)
doc.add_paragraph('Lokalne maksimum znajduje się w
x='+f"{miejsca_zerowe[0].evalf():.3f}"+' i wynosi y='+
                f"{f.subs({x:miejsca_zerowe[0]}).evalf():.3f}")
doc.add_paragraph('Lokalne minimum znajduje się w
x='+"{:.3f}".format(miejsca_zerowe[1].evalf())+' i wynosi y='+
                "{:.3f}".format(f.subs({x:miejsca_zerowe[1]}).evalf()))
```

W przypadku miejsc zerowych „surowy wynik" jest przedstawiany w postaci formuły, żeby dostać wartość wykorzystujemy funkcję evalf(). Dzięki temu w raporcie uzyskamy i formułę i wartość. Jest jednak niewielki problem z wyświetlanymi liczby mają zbyt dużą dokładność. Aby ograniczyć ilość cyfr zastosujemy formatowanie w postaci: f"{miejsca_zerowe[1].evalf():.3f}" lub +"{:.3f}".format(miejsca_zerowe[1].evalf()). Zapis ten powoduje, że po przecinku pojawią się tylko 3 liczby.

Ostatnim krokiem jest zapisanie pliku w obu formatach tj. MS Word i PDF. Zrealizuje to poniższy kod:

```
sciezka_do_pliku=os.getcwd()
doc.save(sciezka_do_pliku+'/test.docx')
#zapis do pdf
word = win32com.client.Dispatch('Word.Application')
doc = word.Documents.Open(sciezka_do_pliku+'/test.docx')
doc.SaveAs(sciezka_do_pliku+'/test.pdf', FileFormat=17)
doc.Close()
word.Quit()
```

Niestety potrzebujemy dodatkowego modułu, który odczyta nazwę katalogu w kutym pracujemy. Na początku programu dopisujemy linie importujące potrzebne biblioteki:

```
import win32com.client
import os
```

Moduł od zawiera funkcję getcwd(), która odczytuje ścieżkę do aktualnego katalogu. Natomiast moduł win32com.clinet z pakietu **Pywin32** pozwoli nam zapisać plik MS Word w formacie PDF. Pozostaje jedynie po wywołaniu funkcji SaveAs określić

format pliku na numer 17. Oczywiście można nasz plik zapisać w innym formacie wykorzystując poniższą Tabelę 7:

Tabela 7. Formaty plików

Nazwa	Wartość	Opis
wdFormatDocument	0	Microsoft Office Word 97
wdFormatTemplate	1	Word template
wdFormatText	2	Microsoft Windows tekst
wdFormatTextLineBreaks	3	Windows tekst z łamanymi liniami
wdFormatDOSText	4	Microsoft DOS tekst.
wdFormatDOSTextLineBreaks	5	Microsoft DOS tekst z łamanymi liniami
wdFormatRTF	6	Rich tekst format (RTF)
wdFormatUnicodeText	7	Unicode tekst
wdFormatHTML	8	Standard HTML
wdFormatWebArchive	9	Web archive
wdFormatFilteredHTML	10	Filtered HTML
wdFormatXML	11	Extensible Markup Language (XML)
wdFormatXMLDocument	12	XML dokument
wdFormatXMLDocumentMacroEnabled	13	XML dokument z makrami
wdFormatXMLTemplate	14	XML template
wdFormatXMLTemplateMacroEnabled	15	XML template z makrami
wdFormatDocumentDefault	16	domyślny dokument Word (DOCX)
wdFormatPDF	17	PDF
wdFormatXPS	18	XPS
wdFormatFlatXML	19	Open XML
wdFormatFlatXMLMacroEnabled	20	Open XML z makrami
wdFormatFlatXMLTemplate	21	Open XML template
wdFormatFlatXMLTemplateMacroEnabled	22	Open XML z makrami
wdFormatOpenDocumentText	23	OpenDocument

wdFormatStrictOpenXMLDocument	24	Strict Open XML

Pełny tekst naszego skryptu generującego plik wygląda następująco:

```
import docx
from docx.shared import RGBColor
from sympy import Symbol, Derivative, solve
import matplotlib.pyplot as plt
import win32com.client
import os

doc=docx.Document()
doc.add_heading('Przebieg zmienności funkcji',0)
doc.add_paragraph('Dokonaj analizy przebiegu zmienności funkcji opisanej wzorem
y=x^3-3x^2-10x+2')
#wyznaczanie pochodnej
x = Symbol('x')
f = x**3-3*x**2-10*x+2
d = Derivative(f, x).doit()
doc.add_paragraph('Pochodna funkcji y='+str(f)+' równa jest y\'='+str(d))
#wstawianie wykresu
doc.add_heading('Wykres funkcji y='+str(f)+' i jej pochodnej',2)
wsp_x=[]
wsp_y=[]
wsp_yp=[]
for i in range(21):
    wsp_x.append((i-10)/2)
    wsp_y.append(f.subs({x:(i-10)/2}))
    wsp_yp.append(d.subs({x:(i-10)/2}))
plt.plot(wsp_x,wsp_y, marker='X', color='b', linestyle=':', markersize=12)
plt.plot(wsp_x,wsp_yp, marker='x', color='r', linestyle='-', markersize=2)
plt.xlabel('wartości x')
plt.ylabel('wartości y')
plt.title('wykres funkcji')
plt.axhline(y=0, color='k')
plt.legend(['funkcja','pochodna'])
plt.savefig('funkcja.png')
doc.add_picture('funkcja.png')
#przedzialy zmiennosci
miejsca_zerowe=solve(d)
doc.add_heading('Przedziały zmienności',2)
```

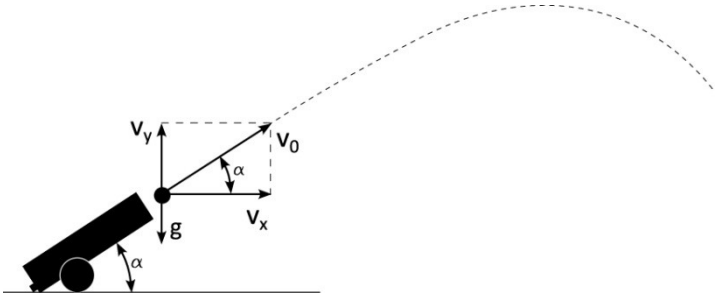
```
doc.add_paragraph('Pochodna funkcji y='+str(f)+' ma miejsca zerowe w punktach:')
doc.add_paragraph('x1:'+str(miejsca_zerowe[0])+', x2='+str(miejsca_zerowe[1]))
pom=doc.add_paragraph('W przedziale od ')
pom.add_run('–oo')
pom.add_run(' do ')
pom.add_run(str(miejsca_zerowe[0]))
pom.add_run(' funkcja jest ')
pom.add_run('rosnąca')
pom.add_run('.')
pom.runs[1].bold=True
pom.runs[3].bold=True
pom.runs[5].font.color.rgb = RGBColor(0x00, 0xff, 0x00)
pom=doc.add_paragraph('W przedziale od ')
pom.add_run(str(miejsca_zerowe[0]))
pom.add_run(' do ')
pom.add_run(str(miejsca_zerowe[1]))
pom.add_run(' funkcja jest ')
pom.add_run('malejąca')
pom.runs[5].font.color.rgb = RGBColor(0xff, 0x00, 0x00)
pom.add_run('.')
pom.runs[1].bold=True
pom.runs[3].bold=True
pom=doc.add_paragraph('W przedziale od ')
pom.add_run(str(miejsca_zerowe[1]))
pom.add_run(' do ')
pom.add_run('+oo')
pom.add_run(' funkcja jest ')
pom.add_run('rosnąca')
pom.add_run('.')
pom.runs[1].bold=True
pom.runs[3].bold=True
pom.runs[5].font.color.rgb = RGBColor(0x00, 0xff, 0x00)
#wyznaczenie miejsc zerowych i lokalnych ekstremow
doc.add_heading('Miejsca zerowe funkcji',2)
miejsca_zerowe=solve(f)
doc.add_paragraph('Miejsca zerowe funkcji znajdują sie w punktach: ')
for i,m in enumerate(miejsca_zerowe):
    doc.add_paragraph(str(i+1)+': '+str(m)+' czyli '+str(m.evalf()))
miejsca_zerowe=solve(d)
doc.add_paragraph('Lokalne maksimum znajduje się w
x='+f"{miejsca_zerowe[0].evalf():.3f}"+' i wynosi y='+
```

```
            f"{f.subs({x:miejsca_zerowe[0]}).evalf():.3f}")
doc.add_paragraph('Lokalne minimum znajduje się w
x='+"{:.3f}".format(miejsca_zerowe[1].evalf())+' i wynosi y='+
            "{:.3f}".format(f.subs({x:miejsca_zerowe[1]}).evalf()))
#zapis do pliku
sciezka_do_pliku=os.getcwd()
doc.save(sciezka_do_pliku+'/test.docx')
#zapis do pdf
word = win32com.client.Dispatch('Word.Application')
doc = word.Documents.Open(sciezka_do_pliku+'/test.docx')
doc.SaveAs(sciezka_do_pliku+'/test.pdf', FileFormat=17)
doc.Close()
word.Quit()
```

9. Lot pocisku

Wykorzystując lot pocisku wystrzelonego z działa dokonamy prostej symulacji jego ruchu i porównamy z rozwiązaniem teoretycznym. W momencie wylotu z lufy, tak jak pokazano na Rys. 8, pocisk wystrzelony pod kątem $\alpha=30^0$ i o masie m=1kg ma prędkość początkową V_o=100m/s.

[Tekst alternatywny. Na rysunku pokazano pocisk w postaci kuli wylatujący z armaty. Do pocisku doczepione są wektory obrazujące prędkość wypadkową i jej składowe oraz siła grawitacji skierowana w dół. Zaznaczono też kąt lufy literą alfa oraz punktową linia tor lotu pocisku.]



Rys. 8. Pocisk wylatujący z lufy.

Prędkość początkową musimy rozdzielić na składową poziomą V_x i pionową V_y. Prędkości te w momencie początkowym mają wartości (2):

$$V_{0x} = V_0 \cos\alpha = 100\cos 30^0 = 86,6m/s$$
$$V_{0y} = V_0 \sin\alpha = 100\sin 30^0 = 50m/s \quad (2)$$

W trakcie lotu składowa pozioma nie ulega zmianie, ale składowa pionowa w wyniku działania siły grawitacji będzie się zmieniała. Wartość siły grawitacji jest równa (3):

$$G=mg=9,81N \quad (3)$$

Prędkość w kierunku pionowym będzie się więc zmieniać według wzoru (4):

$$V_y = \int_t -Gdt = -Gt + V_{0y} \quad (4)$$

Z powyższych równań wynika, że potrafimy wyznaczyć położenie pocisku dla dowolnej chwili czasu liczonego od momentu wylotu z lufy. Wystarczy scałkować równania opisujące składowe prędkości aby dostać współrzędne x i y.

$$X = \int_t V_{0x} dt = V_{0x} t$$
$$Y = \int_t \left(V_{0y} - Gt \right) dt = V_{0y} t - 0,5 G t^2 \quad (5)$$

Spróbujemy teraz narysować wynik za pomocą kodu napisanego w Python. Importujemy niezbędne biblioteki służące do rysowania wykresów oraz wyznaczania wartości potrzebnych nam funkcji trygonometrycznych. W dalszej części inicjujemy opisane wcześniej zmienne. Ponieważ wyniki obliczeń będą zapisywane na dwóch listach, inicjujemy je (xt i yt) oraz zapisujemy w nich współrzędne początkowe (0,0).

```
import matplotlib.pyplot as plt
from math import sin,cos,pi
t=0
dt=0.1
t_max=100
m=1
g=9.81
G=g*m
pos_xt=0
pos_yt=0
predkosc=100
kat=30
predkosc_x=predkosc_xp=predkosc*cos(kat*pi/180)
predkosc_y=predkosc_yp=predkosc*sin(kat*pi/180)
xt=[]
yt=[]
xt.append(0)
yt.append(0)
```

Po części wstępnej w pętli while wyznaczymy kolejne położenia pocisku kończąc w momencie, kiedy współrzędna y osiągnie wartość mniejszą niż 0.

```
while pos_yt>=0:
```

```
pos_xt=t*predkosc_xp
pos_yt=t*predkosc_yp-0.5*g*t**2
xt.append(pos_xt)
yt.append(pos_yt)
t+=dt
```

Pozostaje tylko wygenerować i wyświetlić wykres. Przed wyświetleniem wydajemy dodatkową komendę, która zapewnia proporcjonalność skali na obu osiach.

```
plt.plot(xt,yt, color='r', linestyle='-')
plt.gca().set_aspect('equal')
plt.show()
```

9.1. Metoda Eulera

Spróbujemy teraz rozwinąć naszą analizę i wyznaczyć położenia pocisku poprzez całkowanie numeryczne. Najprostszą metodą jest metoda Eulera. W metodzie tej kolejne położenia będą liczone według równań (6):

$$\begin{aligned} x &= x_0 + hf_x(t) \\ y &= y_0 + hf_y(t) \end{aligned} \tag{6}$$

W naszym przypadku h równe jest przyrostowi czasu dt, zaś funkcje *f(t)* to wyrażenia podcałkowe. Na początku do części wstępnej dopiszemy zmienne do przechowywania aktualnego położenia pocisku i jego prędkości oraz dwie listy przechowujące położenia pocisku wyznaczone przez całkowanie metodą Eulera:

```
…
pos_xe=pos_xt
pos_ye=pos_yt
predkosc_xe=predkosc_x
predkosc_ye=predkosc_y
xe=[]
ye=[]
xe.append(0)
ye.append(0)
while pos_yt>=0: …
```


Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki” nr FERS.01.05-IP.08-0234/23

W pętli musimy wstawić równania realizujące całkowanie metodą Eulera. Odbędzie się to etapami. Najpierw wyznaczymy aktualną prędkość (wystarczy wyznaczyć składową pionową ponieważ składowa pozioma jest stała), a następnie używając nowych wartości wyznaczymy aktualne położenie:

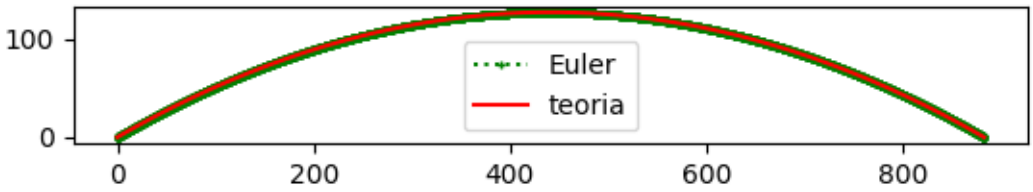
```
…
 yt.append(pos_yt)
predkosc_ye-=g*dt
pos_xe+=dt*predkosc_xe
pos_ye+=dt*predkosc_ye
xe.append(pos_xe)
ye.append(pos_ye)
t+=dt
```

W bloku komend tworzących wykres dodajemy linię generującą nowy wykres, a że mamy już dwa wykresy, legenda staje się potrzebna, więc również ją wymuszamy:

```
plt.plot(xe,ye, marker='+', color='g', linestyle=':', markersize=3)
plt.plot(xt,yt, color='r', linestyle='-')
plt.legend(['Euler','teoria'])
```

Efektem działania program jest wyświetlenie wykresu pokazanego na Rys. 9.

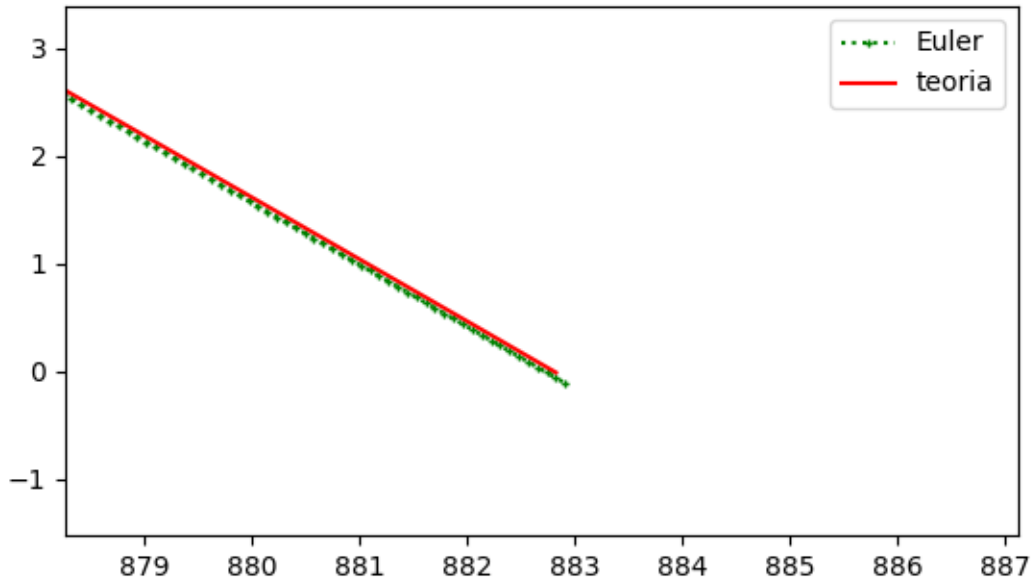
[Tekst alternatywny. Na rysunku widać dwie parabole pokazujące tor pocisku wyznaczony za pomocą równań teoretycznych (linia czerwona) i metodą Eulera (linia zielona). Oś X ma zakres od 0 do 800, a oś Y ma zakres od 0 do 100.]



Rys. 9. Porównanie torów lotu pocisku uzyskanych za pomocą równań i metodą Eulera.

Na pierwszy rzut oka między teoretycznym przebiegiem i symulacją nie widać różnicy, ale zbliżenie końca wykresu, gdzie możemy się spodziewać największych błędów pokazuje, że wykresy jednak nie są identyczne, jak widać na Rys. 10.

[Tekst alternatywny. Na rysunku widać końcowy przebieg lotu pocisku. Oś C ma zakres od 879 do 887. Oś Y ma zakres od -1 do 3. Teoretyczny tor pocisku (linia czerwona) różni się od otrzymanego metodą Eulera (linia zielona).]



Rys. 10. Końcowy etap lotu pocisku.

Różnice te nie są duże, ale i nasza symulacja nietrwała długo. Niestety zła wiadomość jest taka, że błąd ten będzie z czasem narastał. Zwiększenie dokładności obliczeń osiągniemy przez zmniejszanie kroku całkowania, czyli naszego dt. W tym miejscu w ramach samodzielnej pracy możemy porównać wyniki dla różnych kroków całkowania.

9.2. Metoda Runge-Kutty

Metoda Eulera jest metodą pierwszego rzędu co oznacza, że różnica między wartości wyliczoną a dokładną jest proporcjonalna do pierwszej potęgi kroku (w naszym przypadku po czasie czyli dt). Dużo dokładniejszą, a dodatkowo najczęściej wykorzystywaną jest metoda Runge-Kutty czwartego rzędu co oznacza, że błąd (różnica) rośnie w czwartej potędze założonego przyrostu całkowanej wielkości.
W przypadku metody Runge-Kutty wyznaczenie wartości całkowanej w następnym kroku odbywa się według wzorów:

$$k_1 = f\left(x_{n,}y_n\right)$$ $$k_2 = f\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1\right)$$ $$k_3 = f\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2\right)$$ $$k_4 = f\left(x_n + h, y_n + hk_3\right)$$ $$y_{n+1} = y_n + h\left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\right)$$	(7)

We wstępnej części programu dopisujemy nowe zmienne, które wykorzystamy przy obliczeniach nową metodą:

```
pos_xr=pos_xt
pos_yr=pos_yt
predkosc_xr=predkosc_x
predkosc_yr=predkosc_y
xr=[]
yr=[]
xr.append(0)
yr.append(0)
W petli musimy zaaplikować algorytm metody Runge-Kutty:
    k1y=-dt*g
    k2y=dt*(-g+k1y/2)
    k3y=dt*(-g+k2y/2)
    k4y=dt*(-g+k3y)
    predkosc_yr+=(k1y+2*k2y+2*k3y+k4y)/6
    s1x=dt*predkosc_x
    s1y=dt*predkosc_yr
    s2x=dt*(predkosc_x+s1x/2)
    s2y=dt*(predkosc_yr+s1y/2)
    s3x=dt*(predkosc_x+s2x/2)
    s3y=dt*(predkosc_yr+s2y/2)
    s4x=dt*(predkosc_x+s3x)
    s4y=dt*(predkosc_yr+s3y)
    pos_xr+=(s1x+2*s2x+2*s3x+s4x)/6
    pos_yr+=(s1y+2*s2y+2*s3y+s4y)/6
    xr.append(pos_xr)
```

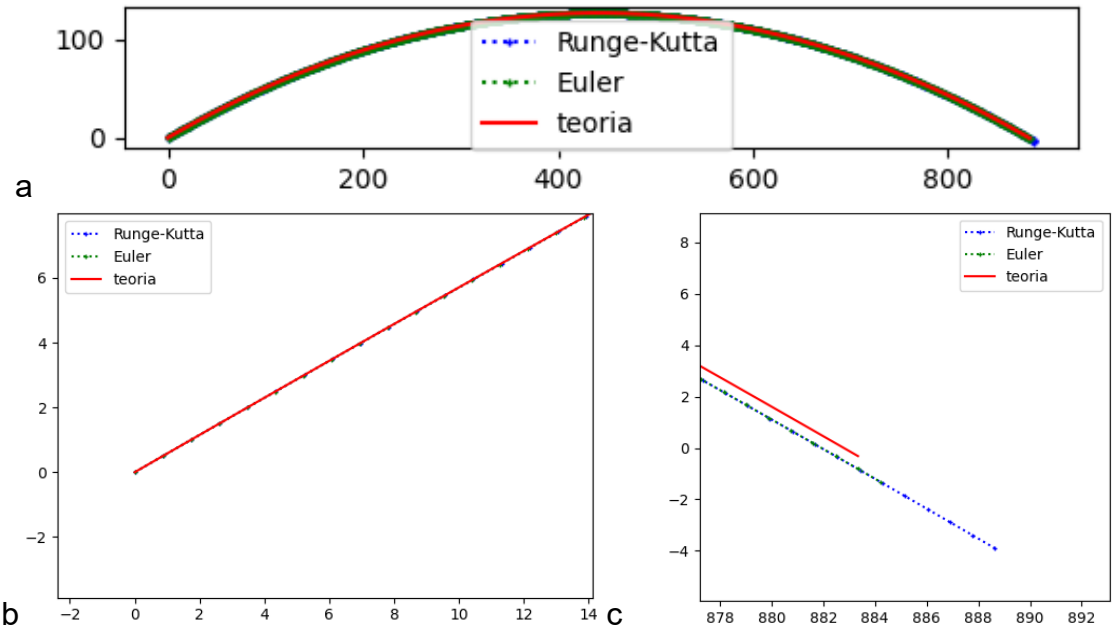
```
yr.append(pos_yr)
t+=dt
```

Pozostaje w części końcowej dodać rysunek nowych danych i dopisanie odpowiedniej nazwy w legendzie:

```
plt.plot(xr,yr, marker='+', color='b', linestyle=':', markersize=3)
…
plt.legend(['Runge-Kutta','Euler','teoria'])
```

Po uruchomieniu powinniśmy uzyskać Rys. 11.

[Tekst alternatywny. Na górnym rysunku widać trzy parabole pokazujące tor pocisku wyznaczony za pomocą równań teoretycznych (linia czerwona), metodą Eulera (linia zielona) i metodą Runge-Kutty (linia niebieska). Oś X ma zakres od 0 do 800, a oś Y ma zakres od 0 do 100. W lewy dolnym rogu pokazano początki tych samych parabol. Oś X ma zakres od -2 do 14, a oś Y ma zakres od -3 do 6. W prawym dolnym rogu pokazano koniec tych samych parabol. Oś X ma zakres od 878 do 892, a oś Y ma zakres od -4 do 8.]



Rys. 11. Porównanie torów lotu pocisku uzyskanych za pomocą równań, metodą Eulera i metodą Runge–Kutty (a), początek lotu (b), koniec lotu (c)

Jak widać początek wykresów pokrywa się dokładnie, jednak koniec pokazuje, że rozwiązania metod przybliżonych lekko się różnią dokładnego. Pozostaje przetestować jak zachowają się rozwiązania przybliżone przy wykorzystaniu kroku całkowania o różnej wielkości.
Pełny listing programu zamieszczono poniżej.

```
import matplotlib.pyplot as plt
from math import sin,cos,pi
t=0
dt=0.0001
t_max=100
m=1
g=9.81
G=g*m
pos_xt=0
pos_yt=0
predkosc=100
kat=30
predkosc_x=predkosc_xp=predkosc*cos(kat*pi/180)
predkosc_y=predkosc_yp=predkosc*sin(kat*pi/180)
xt=[]
yt=[]
xt.append(0)
yt.append(0)
pos_xe=pos_xt
pos_ye=pos_yt
predkosc_xe=predkosc_x
predkosc_ye=predkosc_y
xe=[]
ye=[]
xe.append(0)
ye.append(0)
pos_xr=pos_xt
pos_yr=pos_yt
predkosc_xr=predkosc_x
predkosc_yr=predkosc_y
xr=[]
yr=[]
xr.append(0)
yr.append(0)
while pos_yt>=0:
```

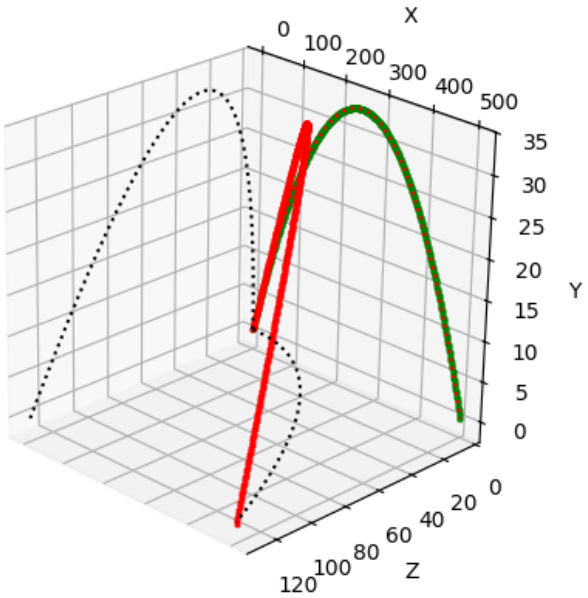
```
    pos_xt=t*predkosc_xp
    pos_yt=t*predkosc_yp-0.5*g*t**2
    xt.append(pos_xt)
    yt.append(pos_yt)
    #Euler
    predkosc_ye-=g*dt
    pos_xe+=dt*predkosc_xe
    pos_ye+=dt*predkosc_ye
    xe.append(pos_xe)
    ye.append(pos_ye)
    #RK
    k1y=-dt*g
    k2y=dt*(-g+k1y/2)
    k3y=dt*(-g+k2y/2)
    k4y=dt*(-g+k3y)
    predkosc_yr+=(k1y+2*k2y+2*k3y+k4y)/6
    s1x=dt*predkosc_x
    s1y=dt*predkosc_yr
    s2x=dt*(predkosc_x+s1x/2)
    s2y=dt*(predkosc_yr+s1y/2)
    s3x=dt*(predkosc_x+s2x/2)
    s3y=dt*(predkosc_yr+s2y/2)
    s4x=dt*(predkosc_x+s3x)
    s4y=dt*(predkosc_yr+s3y)
    pos_xr+=(s1x+2*s2x+2*s3x+s4x)/6
    pos_yr+=(s1y+2*s2y+2*s3y+s4y)/6
    xr.append(pos_xr)
    yr.append(pos_yr)
    t+=dt
plt.plot(xr,yr, marker='+', color='b', linestyle=':', markersize=3)
plt.plot(xe,ye, marker='+', color='g', linestyle=':', markersize=3)
plt.plot(xt,yt, color='r', linestyle='-')
plt.legend(['Runge-Kutta','Euler','teoria'])
plt.gca().set_aspect('equal')
plt.show()
```

10. Lot pocisku na wietrze

Można się zastanawiać po co stosować przybliżone metody całkowania skoro znamy dokładne rozwiązanie. Niestety w przypadku pocisku równanie ruchu jest bardzo

proste (stąd trudno zauważyć różnicę między metodą Eulera i Runge-Kutty), a rozwiązanie znamy. W życiu niestety bywa, że znamy wyrażenie podcałkowe, a rozwiązanie analityczne nie istnieje. Mogą się też pojawić zakłócenia. Wyobraźmy sobie, że na nasz pocisk w trakcie lotu oddziałuje wiatr z niewielką poprzeczną siłą. Będzie on spychał pocisk toru lotu. Jak widać na Rys. 12 pocisk, który bez zakłóceń dotarł do punku oddalonego o 500 jednostek wzdłuż osi X (linia zielona) po dodaniu efektu wiatru zboczył z kursu jak pokazuje czerwona linia. Zmieniony tor pocisku można obserwować na poszczególnych płaszczyznach, gdzie widoczny jest w postaci kresowych czarnych linii.

[Tekst alternatywny. Na rysunku widać 4 parabole. Zielona leży w płaszczyźnie X-Y i pokazuje jak poruszałby się pocisk, kiedy nie wieje wiatr. Czerwona parabola leży w przestrzeni 3d i pokazuje jak wiatr zmienia tor lotu pocisku. Dwie czarne kropkowe linie leżą w płaszczyznach X-Z i Y-Z. Pokazują rzuty rzeczywistego toru pocisku na wymienione płaszczyzny. Oś X ma zakres od 0 do 500, oś Y ma zakres od 0 do 35, a oś Z ma zakres od 0 do 120.]

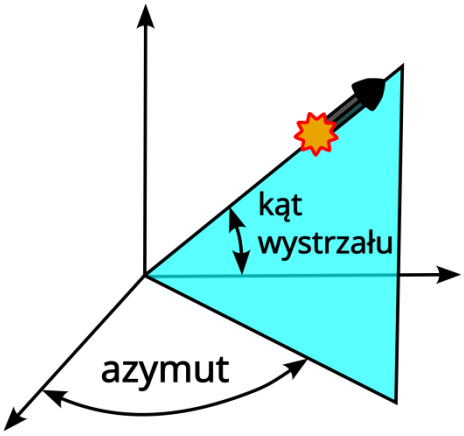


Rys. 12. Wpływ wiatru na tor lotu pocisku.

Rozwiązaniem jest odpowiednia korekta kąta wystrzału i azymutu (Rys. 13) tak, aby trafić w poprzednio ustalony punkt. Spróbujemy jednak nie ustalać wartości kątów samodzielnie, a zmusić do tego program.

[Tekst alternatywny. Na rysunku pokazano trójwymiarowy układ współrzędnych. W tym układzie przemieszczony jest pocisk. Płaszczyzna lotu pocisku jest zdefiniowana za pomocą turkusowego trójkąta. Oznaczony jest kąt wystrzału między torem

pocisku, a osią poziomą oraz azymut między rzutem toru lotu na płaszczyznę poziomą, a drugą osią poziomą.]



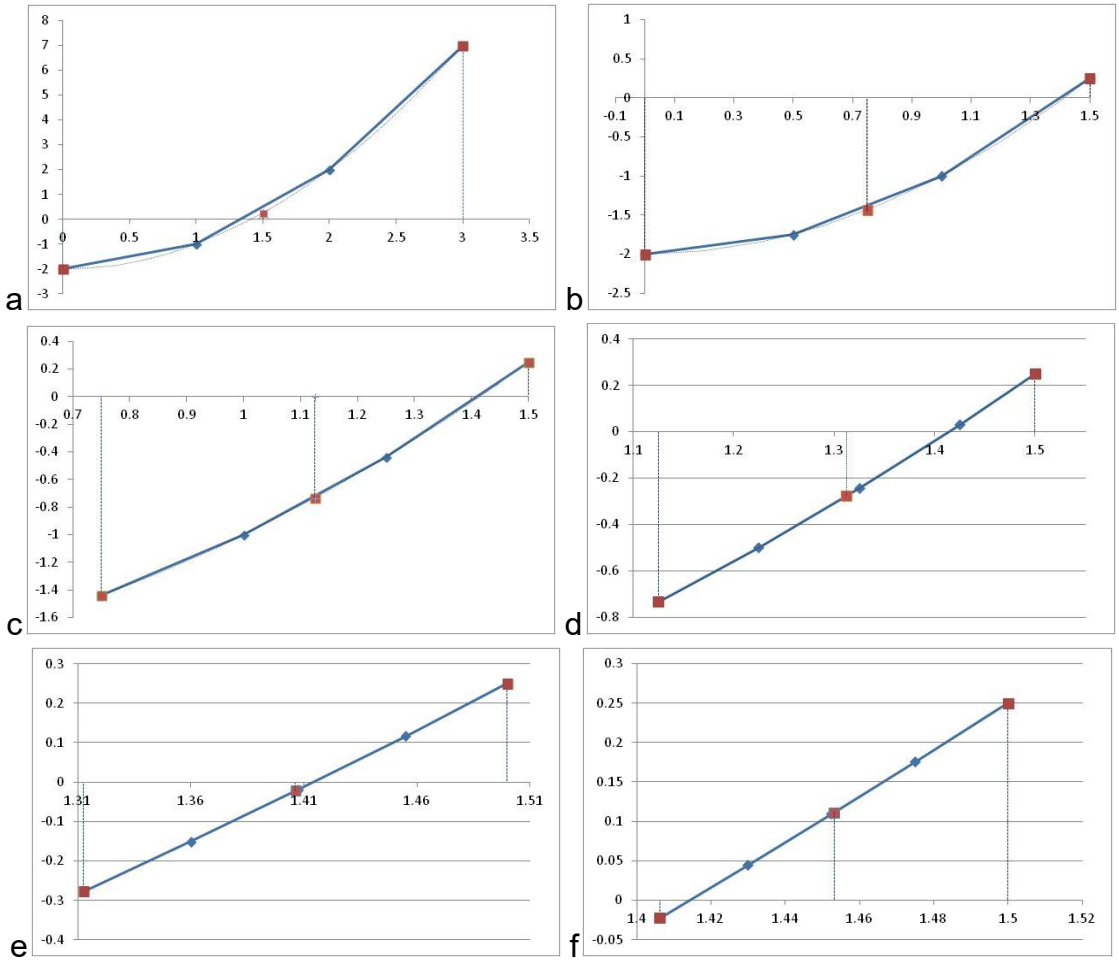
Rys. 13. Kąty określające kierunek lotu.

Do rozwiązania równań ruchu wykorzystamy metodę Eulera. Wiemy już że jest mniej dokładna, ale w naszym zadaniu nie chodzi przede wszystkim o dokładność, ale o automatyczne znalezienie parametrów strzału do czego wykorzystamy metodę bisekcji. Zasadę działania metody bisekcji prześledzimy próbując znaleźć rozwiązanie równania (8) (Stewart, 2023):

$$y(x) = x^2 - 2 \tag{8}$$

Pierwszym krokiem tej metody jest wybranie dwóch punktów dla których funkcja ma przeciwne znaki. To oznacza, że punkt w którym funkcja ma wartość 0 leży między nimi. W naszym przykładzie wybieramy punkty X=0 i X=3. Jak widać na wykresie (Rys. 14a) funkcja ma wartości o przeciwnych znakach. Wyznaczamy wartość funkcji dla punktu, który leży w środku przedziału X=(0+3)/2=1,5. Dla tego punktu funkcja ma wartość dodatnią co oznacza, że miejsce zerowe znajdują się między lewym i środkowym punktem. Zawężamy więc przedział analizy do zakresu 0-1,5 (Rys. 14b). W tym zakresie dla środka przedziału (X=0,75) wartość funkcji jest ujemna. To oznacza, że miejsce zerowe leży między punktem środkowym a prawym więc ponownie zawężamy przedział do zakresu 0,75-1,5 (Rys. 14c). W tym zakresie funkcja dla punktu środkowego ma wartość ujemną co oznacza, że miejsce zerowe leży w zakresie 1,125-1,5 (Rys. 14d). W tym zakresie wartość funkcji dla punktu środkowego jest ujemna więc nowym lewym zakresem jest punkt środkowy czyli X=1,3125.

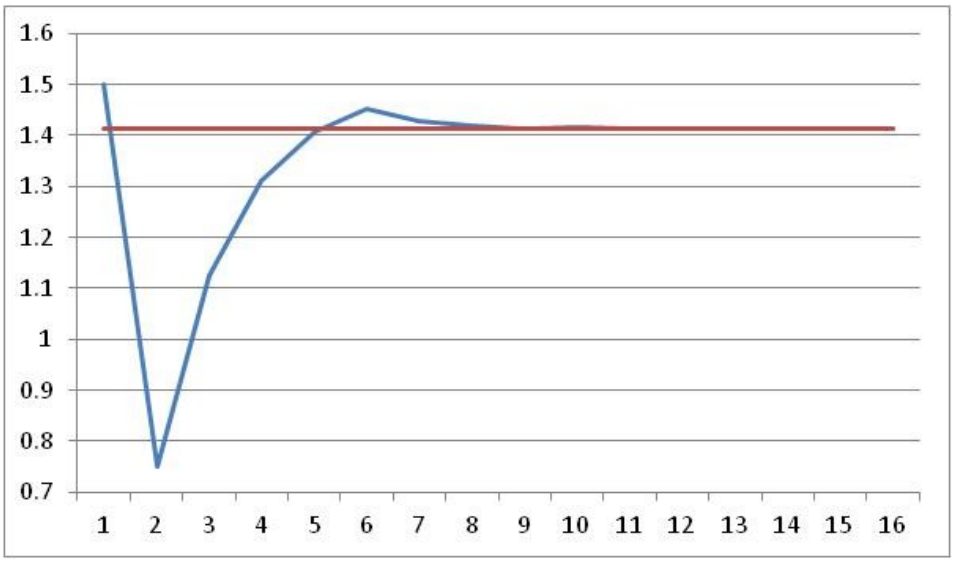
[Tekst alternatywny. Rysunek składa się z 6 części. Na każdej zobrazowano kolejny etap metody bisekcji. Na każdym rysunku widać liniowy wykres funkcji stworzony z 4 punktów oraz trzy charakterystyczne punkty metody bisekcji zaznaczone czerwonymi kropkami. Na kolejnych rysunkach zakresy osi X i Y zmniejszają się.]



Rys. 14. Etapy znajdowania rozwiązania metodą bisekcji.

W kolejnym kroku punkt środkowy ma wartość ujemną (Rys. 14e) więc ponownie przesuwamy lewą granicę przedziału do punktu środkowego X= 1.40625 (Rys. 14f). Jak widać wartość w punkcie środkowym jest dodatnia co oznacza, że tym razem to prawy przedział przesuniemy do punktu środkowego. Po pięciu krokach zawęziliśmy przedział w którym występuje miejsce zerowe do zakresu 1.40625-1.453125. Dokładna wartość to 1.41421. Już po 15 iteracjach osiągniemy wartość 1.414169312, co w przybliżeniu daje nam wartość dokładną do 4. miejsca po przecinku (Rys. 15).

[Tekst alternatywny. Na rysunku pokazano dwie linie. Czerwona, pozioma linia leży na poziomie odpowiadającym dokładnemu rozwiązaniu, zaś niebieska pokazuje wartość funkcji w środku przedziału w kolejnych krokach. Po początkowym nieregularnym przebiegu linia niebieska asymptotycznie zbliża się do linii czerwonej.]



Rys. 15. Wartość funkcji dla punktu środkowego w kolejnych krokach metody bisekcji.

Spróbujemy zaaplikować teraz algorytm do programu. Importujemy niezbędne funkcje zewnętrzne, w tym Axes3D niezbędną do narysowania wykresu 3D. Następnie ponieważ będziemy dokonywać całkowania metodą Eulera wielokrotnie zdefiniujemy funkcję, która będzie dokonywać tych obliczeń. Jako argument funkcja przyjmuje prędkość, kąt, azymut i siłę wiatru, a wynikiem są 3 zbiory ze współrzędnymi x,y,z.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from math import sin,cos,pi,sqrt
def strzal(predkosc,kat,azymut,sila_wiatru):
    t=0
    dt=0.01
    g=9.81
    predkosc_xe=predkosc*cos(kat*pi/180)*cos(azymut*pi/180)
    predkosc_ye=predkosc*sin(kat*pi/180)
    predkosc_ze=predkosc*cos(kat*pi/180)*sin(azymut*pi/180)
    pos_xe=0
    pos_ye=0
```

```
pos_ze=0
xe=[]
ye=[]
ze=[]
xe.append(0)
ye.append(0)
ze.append(0)
while pos_ye>=0:
    predkosc_ye-=g*dt
    predkosc_ze+=sila_wiatru*dt
    pos_xe+=predkosc_xe*dt
    pos_ye+=predkosc_ye*dt
    pos_ze+=predkosc_ze*dt
    xe.append(pos_xe)
    ye.append(pos_ye)
    ze.append(pos_ze)
    t+=dt
return(xe,ye,ze)
```

W części wstępnej definiujemy niezbędne zmienne:

```
odleglosc_do_celu=500
dokladnosc=1
predkosc=100
sila_wiatru=15
blad_x=2*dokladnosc
blad_z=2*dokladnosc
lewy_azymut=-90
prawy_azymut=0
lewy_kat=90
prawy_kat=0
i=0
```

Pojawia się kilka nowych zmiennych. Zmienna dokladnosc określa maksymalną odległość między celem a pociskiem mierzoną oddzielnie w obu kierunkach, która musi być mniejsza aby uznać, że cel został trafiony. Zmienne blad_x i blad_z określają odległości miedzy celem a pociskiem. Wielkości te muszą być mniejsze niż zmienna dokładność, aby zakończyć obliczenia. Ich początkowe wartość ustalone są na podwójną wartość błędu, aby możliwe było uruchomienie pętli while. Zarówno kąt wystrzału jak i azymut będą ustalane osobno. W głównej pętli while, która sprawdza czy warunek dokładności w obu kierunkach jest spełniony wstawione są dwie

podpętle while, które będą ustalać optymalny kąt wystrzału, a następnie dla tej wartości kąt wyznaczany będzie azymut, który zapewni ze pocisk upadnie na linii łączącej punkt wystrzału z celem. To oczywiście spowoduje zmianę odległości lotu, którą trzeba będzie poprawić poprzez dobranie nowego kąta wystrzału i tak w kółko, aż do osiągnięcia wymaganej dokładności lub do osiągnięcia maksymalnej liczby iteracji, która zliczana jest za pomocą zmiennej i. Osobne zmienne zliczające iteracje wykorzystano w podpętlach. Inaczej program mógłby działać w nieskończoność.

```
while (abs(blad_x)>dokladnosc or abs(blad_z)>dokladnosc) and i<100:
    i+=1
    azymut=(lewy_azymut+prawy_azymut)/2
    prawy_kat=45
    xe,ye,ze=strzal(predkosc,prawy_kat,azymut,sila_wiatru)
    prawy_blad_x=odleglosc_do_celu-xe[-1]
    lewy_kat=0
    xe,ye,ze=strzal(predkosc,lewy_kat,azymut,sila_wiatru)
    lewy_blad_x=odleglosc_do_celu-xe[-1]
    i_kat=0
    while abs(blad_x)>dokladnosc and i_kat<100:
        i_kat+=1
        kat=(lewy_kat+prawy_kat)/2
        xe,ye,ze=strzal(predkosc,kat,azymut,sila_wiatru)
        blad_x=odleglosc_do_celu-xe[-1]
        blad_z=ze[-1]
        if lewy_blad_x*blad_x>0:
            lewy_kat=kat
        else:
            prawy_kat=kat
    kat=(lewy_kat+prawy_kat)/2
    prawy_azymut=0
    xe,ye,ze=strzal(predkosc,kat,prawy_azymut,sila_wiatru)
    prawy_blad_z=ze[-1]
    lewy_azymut=-90
    xe,ye,ze=strzal(predkosc,kat,lewy_azymut,sila_wiatru)
    lewy_blad_z=ze[-1]
    i_azymut=0
    while abs(blad_z)>dokladnosc and i_azymut<100:
        i_azymut+=1
        azymut=(lewy_azymut+prawy_azymut)/2
        xe,ye,ze=strzal(predkosc,kat,azymut,sila_wiatru)
        blad_x=odleglosc_do_celu-xe[-1]
```

```
        blad_z=ze[-1]
        if lewy_blad_z*blad_z>0:
            lewy_azymut=azymut
        else:
            prawy_azymut=azymut
    xe,ye,ze=strzal(predkosc,kat,azymut,sila_wiatru)
    blad_x=odleglosc_do_celu-xe[-1]
    blad_z=ze[-1]
```

W końcowej części programu wydrukujemy na ekranie wartość kąta wystrzału i azymut oraz błędy. Jeśli osiągnęliśmy wymaganą dokładność będą to wartości prowadzące do prawidłowego lotu pocisku. W przypadku osiągnięcia limitu iteracji będą to ostatnie wartości kąta i azymutu.

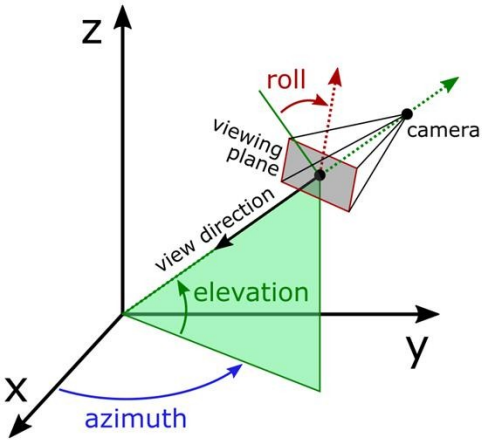
```
if i==100:
    print('Nie osiagnieto zadowalajacego wyniku',i )
    print('Parametry ostaniej analizy: ', kat, ' i azymutu: ',azymut)
    print('Bledy w kierunkach x i z wynosza odpowienio: ',odleglosc_do_celu-xe[-
1],ze[-1])
else:
    print('Cel osiagnieto przy ustawieniu kata wystrzału: ', kat, ' i azymutu: ',azymut)
    print('Bledy w kierunkach x i z wynosza odpowienio: ',odleglosc_do_celu-xe[-
1],ze[-1])
```

Ostatnim etapem jest stworzenie wykresu 3D. Dodatkowo narysujemy też rzuty toru pocisku na poszczególne płaszczyzny, wyświetlane jako czarne kreskowe linie.

```
fig = plt.figure()
ax = fig.add_subplot(projection = '3d')
plt.plot(xe,ye,ze, marker='X', color='g', linestyle=':', markersize=2)
plt.plot(xe,ye,0, color='r', linestyle=':')
plt.plot(xe,0,ze, color='k', linestyle=':')
plt.plot(0,ye,ze, color='k', linestyle=':')
plt.xlabel('X')
plt.ylabel('Y')
ax.set(xlabel=('X'), ylabel=('Y'), zlabel=('Z'))
ax.view_init(elev=55, azim=8, roll=92)
ax.set_box_aspect([1.0, 1.0, 1.0])
plt.show()
```

Ciekawym elementem kodu jest sposób wyświetlenia wykresu. Decyduje o tym komenda view_init, której parametry wyjaśnia Rys. 16.

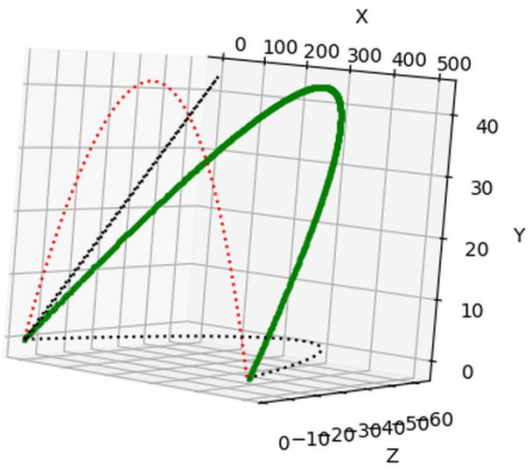
[Tekst alternatywny. Na rysunku pokazano trójwymiarowy układ współrzędnych. W tym układzie przemieszczona jest kamera. Za pomocą kątów oznaczono trzy parametry ustawienia kamery: roll, elevation i azimuth.]



Rys. 16. Parametry ustawienia kamery (źródło: https://matplotlib.org/stable/api/toolkits/mplot3d/view_angles.html).

Wywołajmy program z danymi podanymi w trakcie wyjaśniania kodu. W wyniku otrzymamy przebieg pokazany na Rys. 17. Wykresem można manipulować za pomocą myszki. Przytrzymując lewy klawisz myszy możemy go obracać, zaś przytrzymując prawy przycisk możemy go przesuwać.
Spróbujmy teraz zmienić prędkość wiatru na 20. Niestety przy tak silnym wietrze pocisk nie będzie w stanie osiągnąć celu z wymaganą dokładnością. Co ciekawe, w tym wypadku kąt 45^0, który w locie bez zakłóceń pozwoliłby osiągnąć maksymalną odległość, w tym przypadku wcale nie gwarantuje tego.

[Tekst alternatywny. Na rysunku widać 4 funkcje. Zielona parabola leży w przestrzeni 3-d i pokazuje jak wygląda tor lotu pocisku trafiającego w cel po ustaleniu parametrów uwzględniających wiatr. Czerwona parabola to rzut toru pocisku na płaszczyznę X-Y, dwie czarne kropkowe linie pokazują rzuty toru pocisku w płaszczyznach X-Z i Y-Z. Oś X ma zakres od 0 do 500, oś Y ma zakres od 0 do 40, a oś Z ma zakres od 0 do 60.]

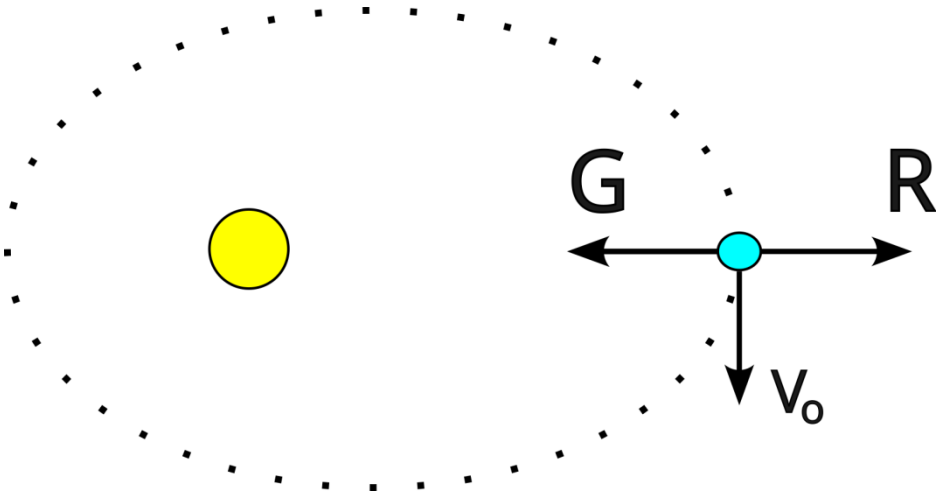


Rys. 17. Tor lotu pocisku po uwzględnieniu wiatru.

11. Układ planetarny

Planeta krążąca wokół gwiazdy poddana jest dwóch przeciwstawnym siłom: sile grawitacji i sile odśrodkowej (Rys. 18). Spróbujemy zasymulować wzajemne oddziaływania tych siły wykorzystując do tego najprostszą metodę całkowania tzw. metodę Eulera.

[Tekst alternatywny. Rysunek prezentuje żółty okrąg w środku oraz turkusowy okrąg po jego prawej. Turkusowy okrąg leży na elipsie narysowanej kropkową linia. Doczepione są do niego trzy wektory: G, R i V_o.]



Rys. 18. Schemat układu planetarnego.

11.1. Tor lotu planety

W naszej symulacji planeta wiruje z prędkością V_0. W stronę gwiazdy przyciąga ją siła grawitacji G, ale w przeciwnym kierunku działa siła odśrodkowa R. Na wstępie zaimportujemy potrzebne nam biblioteki. Pierwsza posłuży do stworzenia wykresu, a z drugiej wybieramy potrzebne funkcje trygonometryczne i pierwiastek kwadratowy wykorzystywane w obliczeniach. W dalszej części zainicjujemy niezbędne zmienne: aktualny czas symulacji zapisywać będziemy pod zmienną t, przyrosty czasu oznaczamy dt, zaś t_max oznacza czas, po którym symulacja powinna się zatrzymać. Zmienne pos_x i pos_y wyznaczają początkowe a później aktualne położenie planety. Stałą grawitacji przypiszemy zmiennej grawitacja. Do obliczenia siły grawitacji niezbędne są też masy planety i gwiazdy. W kolejnych linijkach

definiujemy składowe prędkości początkowej planety oraz wyznaczamy wypadkową wartość prędkości ze wzoru (9):

$$v = \sqrt{v_x^2 + v_y^2} \tag{9}$$

Listę inicjowanych zmiennych zamykają listy, w których zapisywać będziemy położenia planety dla wybranych chwili czasu oraz listy pamiętające współrzędne ostatnich trzech kroków niezbędnych do liczenia krzywizny lotu. Początkowa krzywizna toru jest niezbędna do obliczania siły odśrodkowej. Ponieważ w pierwszej chwili nie jesteśmy w stanie jej wyliczyć w żaden sposób, więc przyjmiemy, że promień krzywizny jest równy odległości planety od gwiazdy. Gdyby planeta poruszała się po kołowej orbicie to założenie byłoby prawdziwe. Dodatkową zmienną jest zmienna krok, którą wykorzystamy aby zmniejszyć ilość danych rysowanych na wykresie co wyjaśnimy w dalszej części sekcji. W ostatnich linijkach części wstępnej wyznaczamy wartości sin i cos oraz inicjujemy wartości dla trzech punktów w listach wykorzystywanych do wyznaczania krzywizny.

```
import matplotlib.pyplot as plt
from math import sqrt,sin,cos,pi,atan

t=0
dt=0.01
t_max=5000
pos_x=0
pos_y=2000
grawitacja=200
masa_planety=0.001
masa_gwiazdy=10000000
predkosc_x=40
predkosc_y=0
predkosc=sqrt(predkosc_x**2+predkosc_y**2)
x=[]
y=[]
xr=[]
yr=[]
x.append(pos_x)
y.append(pos_y)
krzywizna=sqrt(pos_x**2+pos_y**2)
krok=0
cos_kierunkowy_r=predkosc_y/predkosc
```

```
sin_kierunkowy_r= predkosc_x/predkosc
for i in range(3):
    xr.append(0)
    yr.append(0)
```

Po części wstępnej czas na symulację ruchu. Wykonamy ją wykorzystując metodę Eulera w pętli while. Aby wzbogacić program o nowy element zastosujemy najprostszą obsługę błędów za pomocą bloku **try-except**. Robimy tak, ponieważ symulacja potrafi być kapryśna i czasem w wyniku błędu lub nieprawidłowych danych wejściowych zacina się. Aby zobaczyć co się działo przed zatrzymaniem programu, dzięki komendzie **break** w bloku **except** program opuści pętlę i mimo błędu narysuje tor poruszania się planety z danych zebranych do momentu błędu.
Na początku pętli obliczamy wartość siły odśrodkowej i siły grawitacji według wzorów (10):

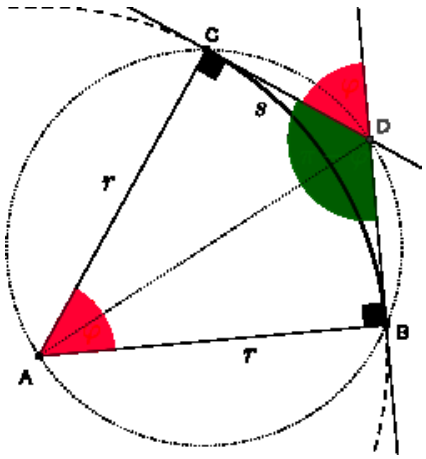
$$R = \frac{mv^2}{r\_krzywizny}$$
$$G = \frac{gm_s m_p}{odl^2} \qquad (10)$$

W początkowej fazie zakładamy, że siły te są skierowane przeciwnie, ale leżą na jednej linii. W miarę jak symulacja zacznie się rozwijać, kąty działania tych sił nie będą zgodne. Podczas, gdy grawitacja zawsze będzie działa wzdłuż prostej łączącej oba obiekty i będzie zależna od kwadratu odległości między planetą i gwiazdą, to siła odśrodkowa będzie zależała od promienia krzywizny toru planety oraz kąta który tworzy prosta łącząca planetę ze środkiem obrotu.
Po dwóch początkowych krokach będziemy w stanie liczyć promień krzywizny (Rys. 19), dlatego blok liczący promień krzywizny i kąt zaczyna się od instrukcji if len(x)>2:. Mając trzy punkty B, C, D możemy policzyć kąt zawarty między dwiema prostymi łączącymi początkowy punkt B i końcowy punkt C ze środkiem obrotu A. W przybliżeniu jest on równy zmianie kąta między odcinkami BD i CD. Ponieważ tor poruszania się planety może być pionowy, zabezpieczamy się na taką okoliczność przyjmując bardzo mała zmianę współrzędnej x. Moglibyśmy przyjąć w takiej sytuacji, że kąt jest równy jednej z wartości +/-pi/2, ale wtedy tracimy informację o znaku, która jest ważna ze względu na kierunek działania siły odśrodkowej.
W przybliżony sposób możemy też policzyć promień krzywizny. Pamiętając, że jeśli kąt wyrażamy w radianach to długość łuku równa jest (11):

$$l = \phi r \qquad (11)$$

łatwo wyznaczać promień jeśli mamy długość łuku l. W naszym przypadku będzie to suma długości odcinków BD i CD. Podobnie jak przy wyznaczaniu kąta, jeśli tor planety tworzy dwa współliniowe segmenty, doszłoby do dzielenia przez zero. W takim wypadku zakładamy bardzo duży promień krzywizny, co zredukuje siłę odśrodkową do zera.
W trakcie obliczeń generujemy dużą liczbę danych. Dla przykładu przy t_max=20000 i dt=0.001 dostaniemy 40 milionów liczb tworzących pary (x,y) dla położenia planety. Do stworzenia wykresu nie potrzebujemy aż tylu punktów, dlatego punkty do wykresu będziemy zapisywać wykorzystując instrukcję warunkową if krok%10==0. W przytoczonym zapisie na wykres trafi co dziesiąty punkt, ale możemy zdecydować o dużo większej redukcji liczby punktów.

[Tekst alternatywny. Rysunek pokazuje fragment łuku o środku w punkcie A. Z jego punktów końcowych C i B narysowane są styczne, które przecinają się w punkcie D. Kąt miedzy BD i CD oznaczono jako π-φ. Kąt między AC i AB oznaczono jako φ. Odcinki AC i AB mają długość r.]



Rys. 19. Schemat wyznaczania promienia krzywizny. (źródło: https://pl.wikipedia.org/wiki/Krzywizna_krzywej).

```
while t<t_max:
    try:
        krok+=1
        odleglosc=sqrt(pos_x**2+pos_y**2)
        sila_odsrodkowa=masa_planety*predkosc**2/krzywizna
        sila_grawitacji=masa_planety*masa_gwiazdy*grawitacja/odleglosc**2
        sin_kierunkowy_g=pos_y/odleglosc
        cos_kierunkowy_g=pos_x/odleglosc
```

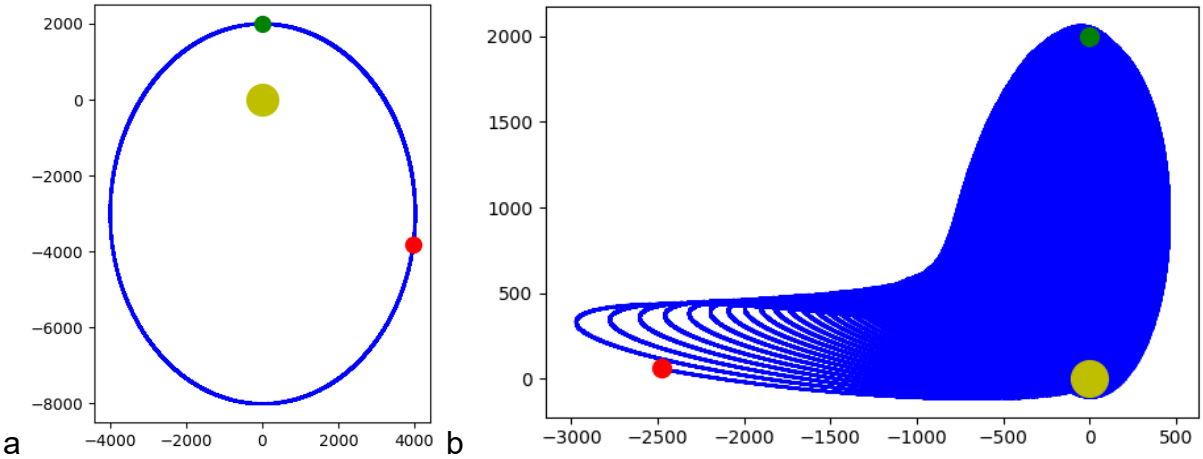
```
        przyspieszenie_x=sila_odsrodkowa*cos_kierunkowy_r-
sila_grawitacji*cos_kierunkowy_g
        przyspieszenie_y=sila_odsrodkowa*sin_kierunkowy_r-
sila_grawitacji*sin_kierunkowy_g
        predkosc_x+=przyspieszenie_x*dt
        predkosc_y+=przyspieszenie_y*dt
        predkosc=sqrt(predkosc_x**2+predkosc_y**2)
        pos_x+=predkosc_x*dt
        pos_y+=predkosc_y*dt
        xr[-3]=xr[-2]
        xr[-2]=xr[-1]
        xr[-1]=pos_x
        yr[-3]=yr[-2]
        yr[-2]=yr[-1]
        yr[-1]=pos_y
        if krok%10==0:
            x.append(pos_x)
            y.append(pos_y)
        if len(x)>2:
            if (xr[-1]-xr[-2])!=0:
                kat1=atan((yr[-1]-yr[-2])/(xr[-1]-xr[-2]))
            else:
                kat1=atan((yr[-1]-yr[-2])/(0.0000001))
            if (xr[-2]-xr[-3])!=0:
                kat2=atan((yr[-2]-yr[-3])/(xr[-2]-xr[-3]))
            else:
                kat2=atan((yr[-2]-yr[-3])/(0.0000001))
            d_kat=kat2-kat1
            kat=atan(-(xr[-1]-xr[-3])/(yr[-1]-yr[-3]))
            cos_kierunkowy_r=cos(kat)
            sin_kierunkowy_r=sin(kat)
            if d_kat!=0:
              krzywizna=(((yr[-2]-yr[-3])**2+(xr[-2]-xr[-3])**2)**0.5+((yr[-1]-yr[-2])**2+(xr[-1]-xr[-
2])**2)**0.5)/d_kat
            else:
                krzywizna=1000000
        t+=dt
    except:
        break
```

Końcowa część służy wizualizacji wyników. Na wykresie umieścimy tak naprawdę kilka wykresów. Pierwszy to niebieska linia kropkowa pokazująca tor poruszania się planety. Drugi wykres to zaledwie jeden punkt. Za pomocą żółtego okręgu oznaczymy na nim położenie gwiazdy. Kolejne dwie kropki o mniejszych rozmiarach niż gwiazda to zielona i czerwona oznaczające odpowiednio początkowe i końcowe położenie planety. Ustalamy też proporcjonalną długość osi, w przeciwnym wypadku każda elipsa mogłaby wyglądać jak okrąg. Ostatnia komenda wyświetla wykres.

```
plt.plot(x,y, marker='X', color='b', linestyle=':', markersize=1)
plt.plot(0,0,marker='o', color='y',markersize=20)
plt.plot(x[0],y[0],marker='o', color='g',markersize=10)
plt.plot(x[-1],y[-1],marker='o', color='r',markersize=10)
#plt.axis(ymin=-limit,ymax=limit,xmin=-limit,xmax=limit)
plt.gca().set_aspect('equal')
plt.show()
```

W wyniku uruchomienia skrypty z danymi jak na początku rozdziału uzyskamy obraz jak na Rys. 20a zmieniając prędkość początkową na 10, a maksymalny czas na 30000, na wynik trzeba będzie poczekać trochę dłużej, ale za to wykres będzie miał dość zaskakujący przebieg (Rys. 20b). Nasza planeta zmieni oś orbity.

[Tekst alternatywny. Rysunek po lewej pokazuje żółty okrąg wewnątrz niebieskiej elipsy. Na elipsie oznaczono dwa punkty: zielony i czerwony. Rysunek po prawej pokazuje żółty okrąg na tle wielu nakładających się elips w kolorze niebieskim. Na elipsach oznaczono dwa punkty: zielony i czerwony.]



a b

Rys. 20. Przykładowe orbity planety dla prędkości początkowej 40 (a) oraz 10 (b). Punkt zielony punkt początkowy, punkt czerwony punkt końcowy.

12. Animacja ruchu planety

Prezentacja toru planety jest najważniejsza z punktu widzenia rozwiązania równań ruchu, ale jeśli otrzyma się wykres podobny do Rys. 20b ciężko się zorientować jak planeta się poruszała, dlatego spróbujemy zmusić planetę do ruchu na wykresie. Wykorzystamy do tego program z poprzedniego rozdziału. Pierwszym krokiem jest import odpowiednich bibliotek. W porównaniu do poprzedniego programu dołożyliśmy nową linię, która umożliwi nam wykonanie animacji.

```
import matplotlib.pyplot as plt
from math import sqrt,sin,cos,pi,atan
from matplotlib import animation
```

Kolejnym krokiem jest zdefiniowanie funkcji, która wykona obliczenia i stworzy listy ze współrzędnymi planety. Nietypowe linie definiują zmienne pos_x i pos_y jako zmienne globalne. Zmienne te są wykorzystywane w kilku funkcjach. Aby program pamiętał ich wartości muszą one być oznaczone jako zmienne globalne.

```
def przedzialy():
    global pos_x
    global pos_y
    t=0
    dt=0.01
    t_max=5000
    pos_x=0
    pos_y=2000
    grawitacja=200
    masa_planety=0.001
    masa_gwiazdy=10000000
    predkosc_x=40
    predkosc_y=0
    predkosc=sqrt(predkosc_x**2+predkosc_y**2)
    x=[]
    y=[]
    xr=[]
    yr=[]
    x.append(pos_x)
    y.append(pos_y)
    krzywizna=sqrt(pos_x**2+pos_y**2)
    krok=0
```

```
    cos_kierunkowy_r=predkosc_y/predkosc
    sin_kierunkowy_r= predkosc_x/predkosc
    for i in range(3):
        xr.append(0)
        yr.append(0)
    while t<t_max:
        try:
            krok+=1
            odleglosc=sqrt(pos_x**2+pos_y**2)
            sila_odsrodkowa=masa_planety*predkosc**2/krzywizna
            sila_grawitacji=masa_planety*masa_gwiazdy*grawitacja/odleglosc**2
            sin_kierunkowy_g=pos_y/odleglosc
            cos_kierunkowy_g=pos_x/odleglosc
            przyspieszenie_x=sila_odsrodkowa*cos_kierunkowy_r-
sila_grawitacji*cos_kierunkowy_g
            przyspieszenie_y=sila_odsrodkowa*sin_kierunkowy_r-
sila_grawitacji*sin_kierunkowy_g
            predkosc_x+=przyspieszenie_x*dt
            predkosc_y+=przyspieszenie_y*dt
            predkosc=sqrt(predkosc_x**2+predkosc_y**2)
            pos_x+=predkosc_x*dt
            pos_y+=predkosc_y*dt
            xr[-3]=xr[-2]
            xr[-2]=xr[-1]
            xr[-1]=pos_x
            yr[-3]=yr[-2]
            yr[-2]=yr[-1]
            yr[-1]=pos_y
            if krok%500==0:
                x.append(pos_x)
                y.append(pos_y)
            if len(x)>2:
                if (xr[-1]-xr[-2])!=0:
                    kat1=atan((yr[-1]-yr[-2])/(xr[-1]-xr[-2]))
                else:
                    kat1=atan((yr[-1]-yr[-2])/(0.0000001))
                if (xr[-2]-xr[-3])!=0:
                    kat2=atan((yr[-2]-yr[-3])/(xr[-2]-xr[-3]))
                else:
                    kat2=atan((yr[-2]-yr[-3])/(0.0000001))
                d_kat=kat2-kat1
```

```
            kat=atan(-(xr[-1]-xr[-3])/(yr[-1]-yr[-3]))
            cos_kierunkowy_r=cos(kat)
            sin_kierunkowy_r=sin(kat)
            if d_kat!=0:
                krzywizna=(((yr[-2]-yr[-3])**2+(xr[-2]-xr[-3])**2)**0.5+((yr[-1]-yr[-
2])**2+(xr[-1]-xr[-2])**2)**0.5)/d_kat
            else:
                krzywizna=1000000
        t+=dt
    except:
        break
  return x,y
```

Funkcja ta zawiera dokładnie te same elementy co wstęp i główna część programu opisywanego w poprzedniej sekcji. Kolejna funkcja wykorzystując zwrócone za pomocą funkcji przedziały() listy wybiera współrzędne dla i-tego kroku. Przypisuje te współrzędne środkowi obiektu circle i zwraca ten obiekt oraz wybrane współrzędne.

```
def nowe_polozenie(i,planeta,x,y):
  pos_x=x[i]
  pos_y=y[i]
  planeta.center=pos_x,pos_y
  return planeta, pos_x,pos_y
```

Ostatnia funkcja wyświetla animację. W pierwszej linii wywołujemy funkcję przedziały(), aby wygenerować listy ze współrzędnymi. Kolejne polecenie tworzy wykres. Na podstawie wyników z poprzedniego rozdziału ustalamy granice osi. Na stworzonym wykresie wstawiamy gwiazdę jako okrąg o promieniu 200, w kolorze żółtym, w punkcie (0,0). Dalej wstawiany na wykres planetę. Tym razem okrąg ma kolor niebieski jest 4x mniejszy od gwiazdy i pojawia się w zdefiniowanym przez nas punkcie początkowym. Punkt ten jest teraz zdefiniowany w funkcji przedziały(). Najważniejsza komenda z punktu widzenia animacji to komenda animation.FuncAnimation()(Saha, 2015). Pierwszym argumentem jest obiekt, który będziemy zmieniali. W naszym przypadku to wykres fig. Drugim argumentem jest funkcja wywoływana co klatka filmu. W naszym przypadku jest to funkcja wybierająca z listy kolejne położenia planety, czyli nowe_polozenie(). Kolejny, tym razem opcjonalny argument, wskazuje jakie argumenty będą przekazywane do wywoływanej funkcji. Kolejne argumenty to ilość klatek, która w naszym przypadku odpowiada ilości zapisanych pozycji planety oraz czas jaki minie między kolejnymi klatkami w milisekundach (interval).

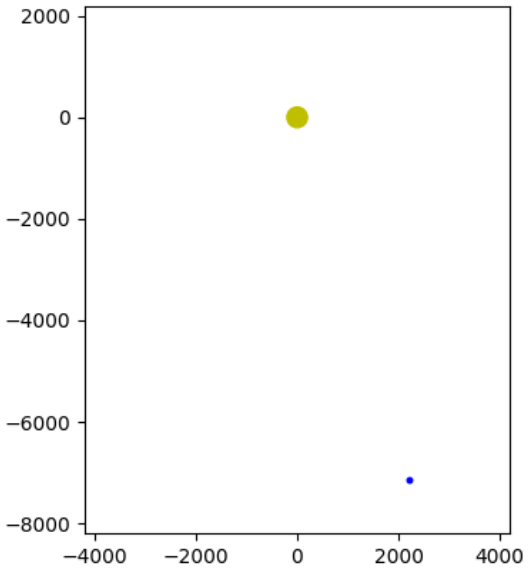
```
def animacja():
    x,y=przedzialy()
    fig=plt.gcf()
    ax=plt.axes(xlim=(-4200,4200),ylim=(-8200,2200))
    slonce=plt.Circle((0,0),200,color='y')
    ax.add_patch(slonce)
    planeta=plt.Circle((pos_x,pos_y),50,color='b')
    ax.add_patch(planeta)
    plt.gca().set_aspect('equal')
    #linia=plt.plot(x,y,linestyle=':')
    anim=animation.FuncAnimation(fig, nowe_polozenie,
                                 fargs=(planeta,x,y),frames=len(x),interval=1)
    plt.show()
```

W listingu widać linia oznaczoną znakiem komentarza. Jeśli usuniemy znak „#” wówczas trajektoria naszej planety będzie widoczna na ekranie. Pozostaje dodać część główną programu, którą jest po prostu wywołanie funkcji animacja():

```
animacja()
```

Na Rys. 21 pokazano wybraną klatkę z filmu.

[Tekst alternatywny. Rysunek przedstawia prostokąt wewnątrz którego znajduje żółty okrąg w górnej części po środku, natomiast w prawym dolnym rogu umieszczony jest turkusowy okrąg. Lewą i dolną krawędź prostokąta stanowią osi układu współrzędnych]

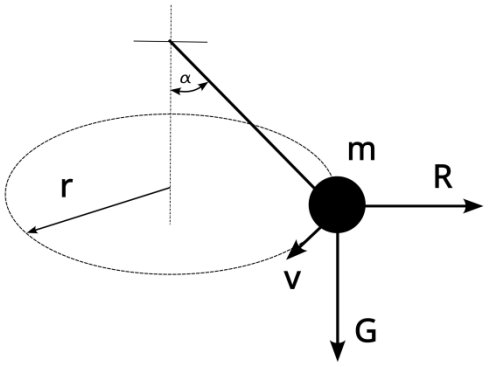


Rys. 21. Klatka z animacji układu planetarnego.

13. Karuzela

Krzesełko na karuzeli porusza się po okręgu. To oznacza że działają na nie 3 siły równoważące się siły: siła w lince, siła grawitacji i siła odśrodkowa (Rys. 22). Siła w lince jest siłą pasywną równą wypadkowej siły grawitacji i odśrodkowej o przeciwnym zwrocie, aby je równoważyć. Jedyną siłą, na którą mamy wpływ, jest siła odśrodkowa, którą kontrolujemy poprzez prędkość obrotową karuzeli.

[Tekst alternatywny. Rysunek prezentuje czarny okrąg umieszczony na elipsie narysowanej linia punktową. Okrąg połączony jest linią umieszczoną pod kątem alfa. Do kuli dołączone są trzy wektory: v (styczny do elipsy), R (skierowany na zewnątrz elipsy) i G (skierowany pionowo w dół).]



Rys. 22. Siły działające na obracającą się masę zawieszoną na cięgnie.

Siła odśrodkowa i grawitacji liczone są według wzorów (12):

$$R = \frac{mv^2}{r} \qquad (12)$$

G=mg

gdzie m to masa ciała, r to promień obrotu, v to prędkość, g to przyspieszenie ziemskie. Siły R i G oraz kąt α jaki cięgno na którym zawieszona jest masa tworzy w linią pionową są ze sobą powiązane poprzez funkcję tangens (13).

$$\frac{R}{G} = tg\alpha \qquad (13)$$

Za pomocą programu spróbujemy wyznaczyć tor poruszającej się masy, która początkowo obraca się wokół pionowej osi z prędkością v_1, a po określonym czasie przyspiesza do prędkości v_2. W części wstępnej programu importujemy niezbędne biblioteki pozwalające stworzyć wykresy, w tym 3d oraz funkcje trygonometryczne. W dalszej kolejności ustalamy czas początkowy i końcowy oraz wielkość skoku w czasie. W kolejnych liniach ustalamy początkowe położenie. Jak widać problem będzie rozwiązywany w przestrzeni trójwymiarowej. Spowodowane jest to działaniem wiatru w kierunku poprzecznym do toru lotu przez co pocisk porusza się w płaszczyźnie nachylonej do ziemi. Kolejna część zawiera kilka wyników, które ułatwią i zorganizują dalsze analizy. Na podstawie położenia punktu w płaszczyźnie XY wyznaczony został promień okręgu po którym wiruje masa oraz początkowy kąt jaki na okręgu przyjęła masa. Kąt zero jest zgodny z dodatnią częścią osi X. Dalej definiujemy wymiar r_max, który oznacza punkt w którym zaczepione jest nasze cięgno podtrzymujące masę. Lokalizując go w przestrzeni XYZ powinniśmy określić wszystkie współrzędne punktu, w którym zaczepione jest cięgno. Punkt ten ma współrzędną 0,0,r_max. Na szczęście potrzebujemy tylko współrzędnej na osi Z. Znając ją możemy na podstawie początkowego położenia masy wyznaczyć długość cięgna. Zdefiniowanie masy ciała pozwala na obliczenie siły grawitacji i odśrodkowej, a to z kolei pozwoli dopasować prędkość poruszającej się masy do jej położenia. Po rozpoczęciu analizy to prędkość będzie główną zmienną i od niej będą zależeć pozostałe wielkości. Zależność między prędkością, a położeniem i wartości sił będą wyznaczane dynamicznie, ale dla momentu startowego musimy „zgrać” te wielkość ze sobą. Ta część kodu kończy się zainicjowaniem trzech list ze współrzędnymi oraz wpisaniu współrzędnych początkowego punktu.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from math import sqrt,atan,cos,sin
t=0
dt=0.01
t_max=50
pos_x=1.25
pos_y=0
pos_z=0
r=sqrt(pos_x**2+pos_y**2)
alfa=atan(pos_y/pos_x)
z_max=5
dlugosc_ciegna=sqrt(z_max**2+r**2)
masa_kuli=1
sila_grawitacji=9.81*masa_kuli
sila_odsrodkowa=r/z_max*sila_grawitacji
```

```
predkosc=sqrt(sila_odsrodkowa*r/masa_kuli)
predkosc_katowa=predkosc/r
x=[]
y=[]
z=[]
x.append(pos_x)
y.append(pos_y)
z.append(pos_z)
```

Główna cześć programu wyznaczy położenia masy w każdym kroku symulacji, ale nie będziemy korzystać z metod całkowania (z jednym wyjątkiem) a z rozwiązań analitycznych explicite. W pętli while na podstawie kroku w czasie metodą Eulera wyznaczamy aktualny kąt (to ten wyjątek w obliczeniach). Dzięki kątowi wyznaczamy współrzędne x i y. Nowe położenie zapisujemy na odpowiednie listy. Dalej korzystając z instrukcji warunkowej między 10-tą a 20-tą sekundą zwiększamy prędkość kątową w każdym kroku o 0.001 radian/sekundę. Na podstawie położenia masy i prędkości kątowej wyznaczamy prędkość liniową i siłę odśrodkową. Jeśli prędkość się nie zmienia, a dzieje się tak przed 10 i po 20 sekundzie symulacji te wielkości będą stałe. Znając wartość siły odśrodkowej można wyznaczyć kąt cięgna α a na jego podstawie współrzędną z masy oraz nowy promień okręgu po którym porusza się masa. Obie wielkości będą się zmieniać tylko wtedy, gdy zmieni się prędkość.

```
while t<t_max:
    alfa+=predkosc_katowa*dt
    pos_x=r*cos(alfa)
    pos_y=r*sin(alfa)
    x.append(pos_x)
    y.append(pos_y)
    z.append(pos_z)
    if t>10 and t<20:
        predkosc_katowa+=0.001
    predkosc=predkosc_katowa*r
    sila_odsrodkowa=masa_kuli*predkosc**2/r
    kat=atan(sila_odsrodkowa/sila_grawitacji)
    pos_z=z_max-dlugosc_ciegna*cos(kat)
    r=dlugosc_ciegna*sin(kat)
    t+=dt
```

Czas na prezentację wyników. Będzie ona o tyle nietypowe, że pokażemy jednocześnie trzy wykresy. Tworzymy nowy obiekt typu wykres. Zmienna limit

pozwoli nam decydować o zakresie osi za pomocą jednej cyfry. Za pomocą funkcji add_subplot() definiujemy podwykres, który powstanie w przestrzeni 3D. Argumenty funkcji 1,2,1 oznaczają, że przestrzeń wykresu podzielimy na 1 wiersz i dwie kolumny, a aktualnie tworzony wykres powstanie w pierwszej (czyli lewej) części. Na wykresie za pomocą niebieskiej linii rysujemy trajektorię masy, ale dodatkowo za pomocą zielonego i czerwonego okręgu zaznaczamy pozycję początkową i końcową. Ostatni element to czarny „x" którym zaznaczamy położenie punkt zaczepienia cięgna. Ostatnie dwa kroki to przypisanie zakresów osiom wykorzystując tylko jedną wartość przypisaną zmiennej limit oraz podpisanie wykresu.

```
fig = plt.figure()
limit=5
#wykres trojwymiarowy
ax = fig.add_subplot(1,2,1,projection = '3d')
plt.plot(x,y,z, marker='X', color='b', linestyle=':', markersize=1)
plt.plot(x[0],y[0],z[0],marker='o', color='g',markersize=10)
plt.plot(x[-1],y[-1],z[-1],marker='o', color='r',markersize=10)
plt.plot(0,0,5,marker='x', color='k',markersize=10)
plt.axis(zmin=-1,zmax=limit,ymin=-limit,ymax=limit,xmin=-limit,xmax=limit)
plt.title('Widok 3d')
```

Kolejny wykres to widok dwuwymiarowy trajektorii na płaszczyźnie XY (widok z góry). Tym razem w funkcji subpolt() dzielimy wykres na dwa wiersze i dwie kolumny, a do rysowania wykorzystujemy 4-te pole czyli prawy dolny narożnik.

```
#wykres dwuwymiarowy xy
plt.subplot(2, 2, 4)
plt.plot(y,x, marker='X', color='b', linestyle=':', markersize=1)
plt.plot(y[0],x[0],marker='o', color='g',markersize=10)
plt.plot(y[-1],x[-1],marker='o', color='r',markersize=10)
plt.plot(0,0,marker='x', color='k',markersize=10)
plt.axis(ymin=-limit,ymax=limit,xmin=-limit,xmax=limit)
plt.title('Widok z gory')
```

Ostatni wykres to widok dwuwymiarowy trajektorii na płaszczyźnie YZ (widok z boku). Podobnie jak poprzednio w funkcji subpolt() dzielimy wykres na dwa wiersze i dwie kolumny, ale do rysowania wykorzystujemy 2-te pole czyli prawy górny narożnik.

```
#wykres dwuwymiarowy yz
plt.subplot(2, 2, 2)
```

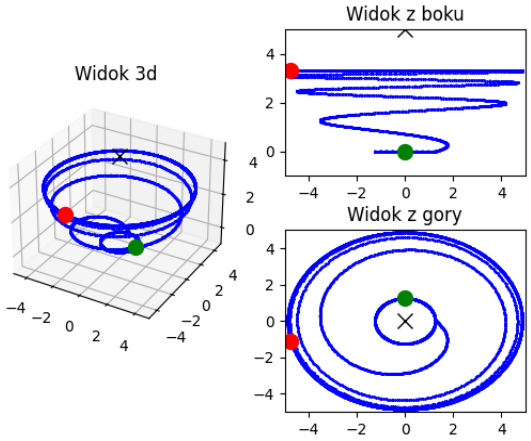
```
plt.plot(y,z, marker='X', color='b', linestyle=':', markersize=1)
plt.plot(y[0],z[0],marker='o', color='g',markersize=10)
plt.plot(y[-1],z[-1],marker='o', color='r',markersize=10)
plt.plot(0,z_max,marker='x', color='k',markersize=10)
plt.axis(ymin=-1,ymax=limit,xmin=-limit,xmax=limit)
plt.title('Widok z boku')
```

Pozostaje ustawić proporcjonalność we wszystkich trzech wymiarach i wyświetlić efekt.

```
#ogolne
plt.gca().set_aspect('equal')
plt.show()
```

Po wykonaniu programu na ekranie zobaczymy obraz jak na Rys. 23.

[Tekst alternatywny. Rysunek pokazuje trzy wykresy – trajektorię prezentowaną w różnych płaszczyznach. Trajektoria narysowana jest niebieską linią. Punkt początkowy oznaczony jest zieloną kropką, a punkt końcowy czerwoną kropką. Rysunek po lewej przedstawia trajektorię w rzucie aksonometrycznym. Górny rysunek po prawej pokazuje tą samą trajektorię z boku, a dolny prawy rysunek prezentuje trajektorię widzianą z góry.]



Rys. 23. Wynik działania programu.

14. Literatura

- Briggs J.R. (2013). Python for kids. No Starch Press, Inc. San Francisco.
- Saha A. (2015). Doing math with Python. No Starch Press, Inc. San Francisco.
- Orbaiceta A.S. (2021). Hardcore programming for mechanical engineers. No Starch Press, Inc. San Francisco.
- Stewart J.M., Mommert M. (2023). Python for scientists, third edition. Cambridge University Press & Assessment. Croydon.
- Sweigart A. (2020). Automate the boring stuff with python, 2nd edition. No Starch Press, Inc. San Francisco.