mechanika

Podstawy komputerowego modelowania układów biomechanicznych

Adam Ciszkiewicz

Kraków 2021

Cracow University of Technology

mechanika

Podstawy komputerowego modelowania układów biomechanicznych

Adam Ciszkiewicz

Kraków 2021

Tekst został opublikowany w ramach projektu „Programowanie doskonałości – PK XXI 2.0. Program rozwoju Politechniki Krakowskiej na lata 2018-22".
Dofinansowanie z Europejskiego Funduszu Społecznego: 18,048,774.96 PLN

Edycja online
eISBN 978-83-66531-56-7

5 ark. wyd.

Wydawnictwo PK, ul. Skarżyńskiego 1, 31-866 Kraków; 12 628 37 25, fax 12 628 37 60
wydawnictwo@pk.edu.pl
www.wydawnictwo.pk.edu.pl
Adres korespondencyjny: ul. Warszawska 24, 31-155 Kraków

SPIS TREŚCI

1. WSTĘP

Skrypt zawiera wprowadzenie do zagadnień związanych z modelowaniem wielo-członowych układów biomechanicznych. Stanowi on pomoc dydaktyczną do przedmiotu: *komputerowe modelowanie układów biomechanicznych* prowadzonego na Wydziale Mechanicznym Politechniki Krakowskiej.

Opracowanie przygotowane zostało przy założeniu, że nie jest wymagana znajomość programowania, zaawansowanych zagadnień mechaniki oraz metod numerycznych. Potrzebna wiedza teoretyczna przedstawiona została w każdym rozdziale. Czytelnik powinien znać jednak podstawy algebry liniowej i geometrii analitycznej w zakresie studiów I stopnia.

2. PODSTAWY PROGRAMOWANIA W JĘZYKU PYTHON

Python to język programowania ogólnego zastosowania. Można go wykorzystać do automatyzacji dowolnych czynności związanych z obsługą komputera. Z uwagi na dostępność darmowych bibliotek numerycznych, Python nadaje się do zastosowań naukowych. Jego największą zaletą jest także to, że można go używać bezpłatnie. Co więcej, dla Pythona dostępnych jest wiele bardzo rozbudowanych środowisk programistycznych. Jednym z nich jest pakiet *WinPython*. Poniżej zawarty został opis instalacji *WinPythona* oraz jego początkowej konfiguracji.

2.1. PRZYGOTOWANIE *WinPythona*

WinPython to darmowy pakiet dla systemu Windows, który zawiera sam język Python oraz zestaw potrzebnych oraz popularnych narzędzi i bibliotek. Dodatkowo, pakiet ten nie jest domyślnie instalowany. Działa od razu po rozpakowaniu. W tym podrozdziale krok po kroku zostanie opisana procedura przygotowania *WinPythona* do pracy.

W pierwszym etapie należy pobrać plik wykonywalny z *WinPythonem*. Wersja dla 64-bitowych systemów operacyjnych znajduje się pod poniższym adresem:

https://sourceforge.net/projects/winpython/files/WinPython_2.7/2.7.10.3/WinPython-64bit-2.7.10.3.exe/download

Warto nadmienić, że Python jest cały czas rozwijany i regularnie pojawiają się jego nowe wersje. W ostatnich latach coraz większą popularność zyskuje Python 3. Skrypty zawarte w tej książce zostały napisane i sprawdzone w Pythonie 2.7. Niemniej jednak Python 3 oraz Python 2.7 nie różnią się znacząco w podstawowej składni. Największa i kluczowa różnica, z punktu widzenia przygotowanych skryptów, to sposób wyświetlania tekstu w konsoli. Procedura ta zostanie dokładnie opisana w kolejnym rozdziale, ale w skróconej wersji wygląda to następująco. Aby wyświetlić w konsoli napis „lubie programowac", w Pythonie 2.7 należy wpisać:

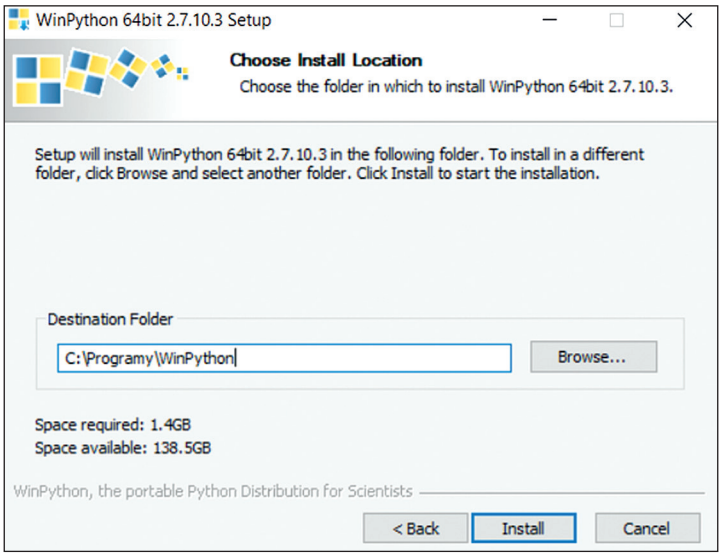
```
print "lubie programowac"
```

Natomiast w Pythonie 3 powyższa instrukcja przyjmie trochę inną formę:

```
print("lubie programowac")
```

Oczywiście różnice pomiędzy wersją 3 i 2.7 na tym się nie kończą. Niemniej jednak skrypty przedstawione w tej książce będą wymagały tylko takiej modyfikacji.

Proces przygotowania pakietu *WinPython* jest bardzo prosty. Po przeczytaniu i akceptacji licencji należy wybrać folder, do którego zostanie rozpakowany pakiet. Folder ten jest dowolny, ale trzeba go zapamiętać, bo później będzie przydatny. Okno wyboru folderu wygląda następująco. Wybranie opcji *install* rozpoczyna proces rozpakowywania pakietu do folderu, który może potrwać nawet kilka minut.

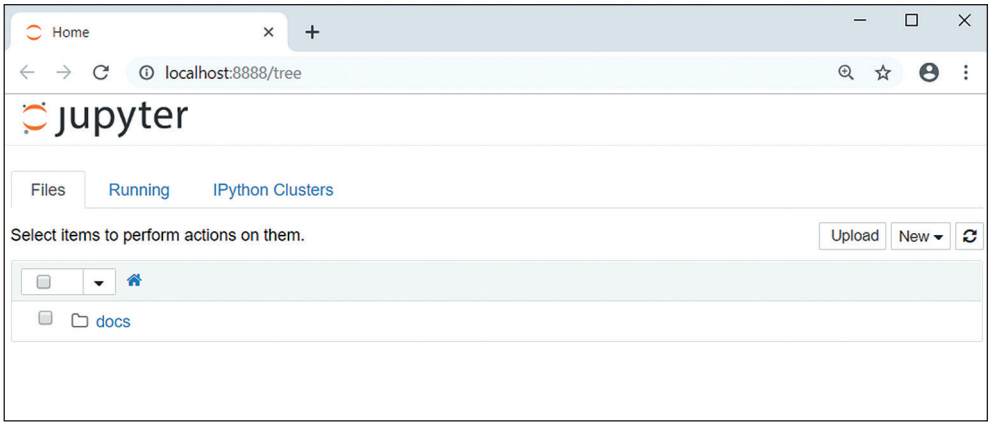


Rys. 2.1. Okno z wyborem folderu podczas instalacji pakietu *WinPython*

Dla Pythona dostępnych jest wiele środowisk programistycznych. Umożliwiają one łatwą edycję programów oraz ich uruchamianie i testowanie. Pakiet *WinPython* zawiera aż trzy takie środowiska. Dwa z nich zasługują na szczególną uwagę: *Spyder* oraz *Jupyter*. *Spyder* to zarówno edytor tekstu, jak i konsola do testowania przygotowanych skryptów. Interfejsem mocno przypomina oprogramowanie *Matlab* (oraz pokrewne: *Octave*, *Scilab*). Bardzo dobrze nadaje się do pisania dużych programów. Z drugiej strony *Jupyter* to nowoczesna forma środowiska programistycznego, która działa w przeglądarce (w trybie offline). Wyróżnia się tym, że pozwala zapisywać w jednym pliku zarówno kod, jak i jego opis, który nie jest wykonywany przez komputer podczas działania programu. Podstawowe skrypty Pythona też mają taką funkcjonalność, ponieważ za pomocą znaku # można w nich umieszczać komentarze. Niemniej jednak możliwości komentowania są ograniczone ze względu na brak

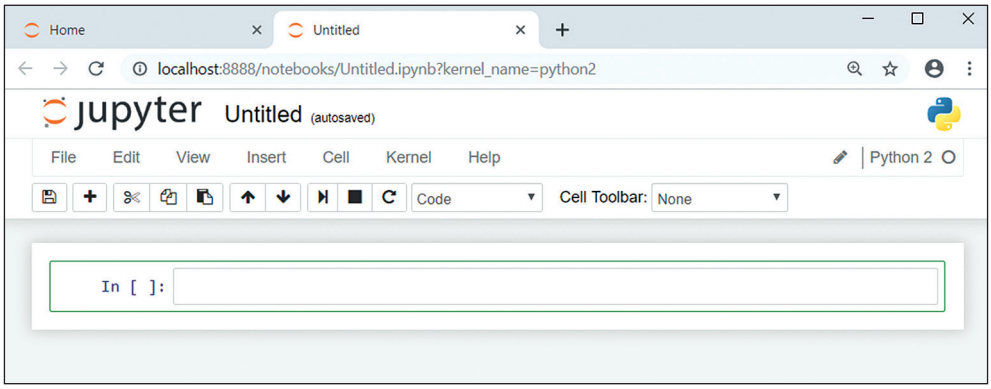
kontroli nad formatowaniem tekstu. Komentarze mają raczej pomóc w zrozumieniu kodu. Natomiast w *Jupyterze* można nawet napisać książkę z interaktywnym kodem. Ta łatwość łączenia kodu i sformatowanego tekstu spowodowała duże zainteresowanie tym środowiskiem. Przykładowo, w zakresie uczenia maszynowego jest to podstawowe środowisko do pisania programów. Doskonale nadaje się też do nauki programowania, i dlatego warto z niego korzystać podczas pracy z tą książką.

Jupyter dostępny jest w katalogu, który wybrano podczas instalacji *WinPythona*. Po jego uruchomieniu automatycznie otworzy się przeglądarka internetowa ze specjalną kartą zatytułowaną *Jupyter* – patrz: rys. 2.2. Ta karta służy tak naprawdę jako eksplorator programów. Zaraz po załadowaniu wyświetla zawartość katalogu *WinPython\notebooks*. Na tym etapie katalog ten nie zawiera żadnych plików, jest w nim tylko pod folder *docs*. Pierwszy „zeszyt" *Jupytera* (ang. *Jupyter notebook*) można utworzyć, klikając przycisk *new*, a następnie *Python 2* w kategorii *notebook* po prawej stronie karty.



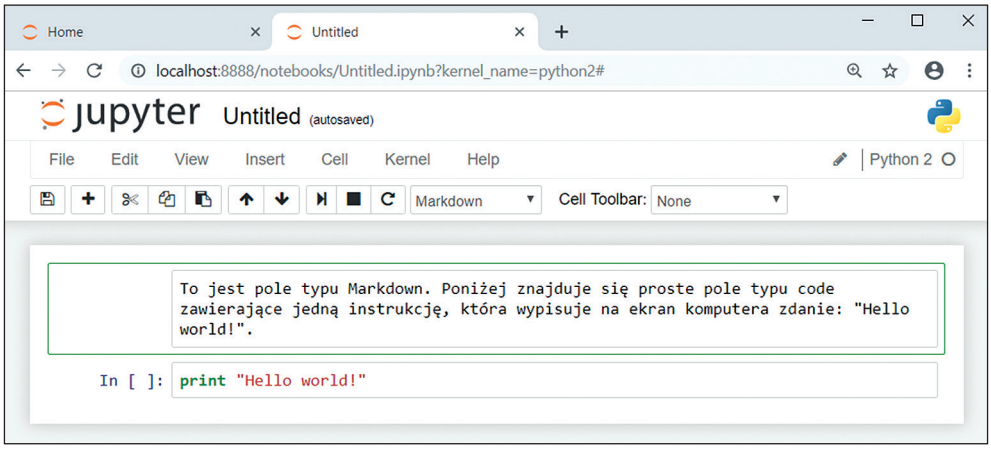
Rys. 2.2. Podstawowe okno *Jupytera*

Po utworzeniu zeszytu interfejs strony znacząco się zmienia. Przede wszystkim pojawiają się w nim pola tekstowe do edycji. Domyślnie zeszyt startuje z jednym polem (rys. 2.3), ale można je sukcesywnie dodawać. To właśnie te pola tekstowe są podstawą pracy z notatnikiem. W zasadzie cały program można od razu napisać i uruchomić, wykorzystując tylko jedno pole, ale to ogranicza czytelność kodu i możliwości jego rozbudowy. Nowe pole można dodać do zeszytu poprzez kliknięcie na przycisk w kształcie plusa w lewej górnej części karty, który opisany jest jako *insert cell below*. Każda z takich „komórek tekstowych" ma określony tryb pracy, który znajduje się w środkowej części ekranu. Domyślny tryb pracy to *code*, co oznacza, że w komórce znajduje się kod programu, który trzeba wykonać. Drugi ważny tryb pracy to *Markdown*, który w uproszczeniu oznacza zwykły tekst i można go wybrać z listy dostępnych trybów.



Rys. 2.3. Notatnik *Jupytera* tuż po utworzeniu

Niezależnie od rodzaju wszystkie pola można uruchomić, wykorzystując przycisk *run cell*, *select below* w środkowej części paska. Dostęp do tej funkcji jest też z poziomu menu *Cell*, polecenie *Run*. Istnieje także możliwość uruchomienia wszystkich pól po kolei za pomocą *Cell* > *Run all*. Pola typu *Code* różnią się od pól *Markdown* przede wszystkim tym, że są uruchomiane jako kod programu i zwracają wynik działania. Pola *Markdown* po uruchomieniu zyskują tylko ładne formatowanie. Przykładowy zeszyt z polami typu *Code* i *Markdown* przedstawiono na rysunku poniżej.



Rys. 2.4. Notatnik *Jupytera* z dwoma polami

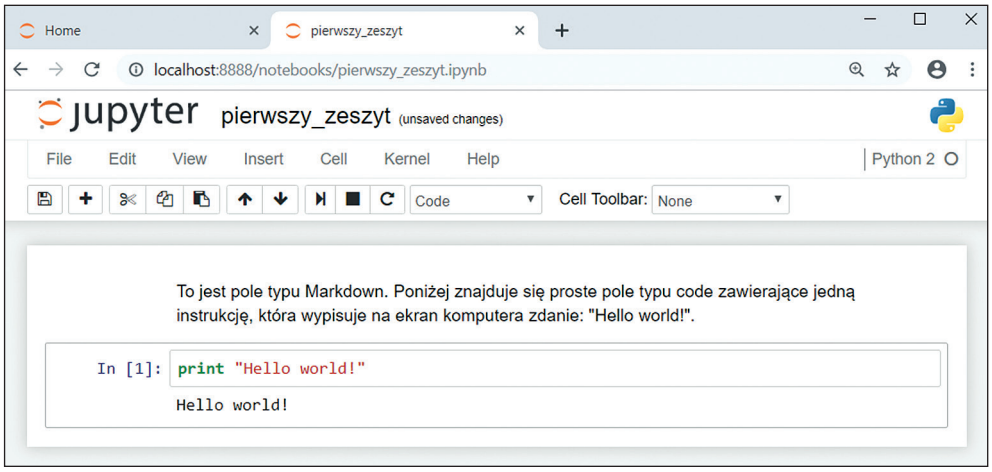
Po uruchomieniu wszystkich pól za pomocą *Cell* > *Run all* pole Markdown zmienia formatowanie, a pod polem *Code* zostaje wygenerowany wynik działania programu, którym w tym przypadku jest wyświetlenie tekstu *Hello world!* Warto zauważyć, że obok nazwy *Jupyter* pojawia się także napis *Untitled*. Jest to nazwa zeszytu, którą z tego poziomu można edytować. Wystarczy kliknąć na napis *Untitled* i wpisać nową nazwę, np. *pierwszy_zeszyt*. Zeszyt po zmianach przedstawiono na rysunku 2.5.

Ten skrypt został przygotowany z założeniem, że programy będą pisane w notatniku *Jupyter*. Najlepszą metodą na uruchamianie zawartych w nim skryptów jest utworzenie jednego notatnika na rozdział. Wtedy do kolejnych komórek można przepisać poszczególne fragmenty skryptów zawarte w polach oznaczonych następująco:

```
Pola o takim formatowaniu oznaczają kod gotowy do przepisania do
komórki w notatniku Jupitera.
```

Książka zawiera także wyniki działania tych skryptów. Są one oznaczone w następujący sposób:

```
Tak wyglądają wyniki działania skryptów.
```



Rys. 2.5. Notatnik *Jupytera* po uruchomieniu wszystkich pól

Dalsza część tego rozdziału przedstawia podstawowe zagadnienia z zakresu programowania w języku Python (z założeniem, że *Jupyter* został poprawnie uruchomiony). Tematy przedstawione w kolejnych podrozdziałach przedstawiają w uproszczony sposób podstawowe zagadnienia z programowania, które są potrzebne (i wystarczające) do przygotowania własnych skryptów obliczeniowych. Bardziej szczegółowy kurs języka Python znajduje się w [1].

2.2. ZMIENNE, CZYLI JAK PRZECHOWYWAĆ DANE W PAMIĘCI KOMPUTERA

W pewnym uproszczeniu każdy program komputerowy to seria komend, które modyfikują pewien zbiór danych. Ten zbiór może np. zawierać dane modelu biomechanicznego, zapis temperatury w pokoju, listę filmów i inne. Komendami mogą być różne operacje matematyczne (podstawowe: dodawanie, odejmowanie, sinus, tangens oraz bardziej złożone: mnożenie macierzy, iloczyn skalarny wektorów), a także bardziej utylitarne funkcje (np. wyświetlenie tekstu na ekranie – komenda *print*). Bardzo podstawowe komendy są wbudowane w język (*print*, +, –), bardziej złożone (mnożenie macierzy) można napisać samodzielnie, bazując na podstawowych albo zaimportować z istniejącej biblioteki.

Jak wspomniano wcześniej, program to seria komend, które modyfikują dane – najczęściej dostarczone przez użytkownika. Na początku warto więc zdefiniować, jak przechowywane są dane w pamięci komputera. W każdym języku programowania za przechowywanie danych odpowiadają zmienne (ang. *variables*). W uproszczeniu można o zmiennej myśleć jako o fizycznym zbiorniku. Zbiorniczek posiada naklejkę z nazwą i można w nim umieścić jakieś przedmioty (dane). W Pythonie takie zbiorniki definiuje się w następujący sposób:

```
zmienna1 = 3.2
```

Powyższy kod pozwolił na zdefiniowanie zmiennej (zbiornika), której nazwa to *zmienna1*. Przechowuje ona liczbę zmiennoprzecinkową równą 3.2. Zawartość tej zmiennej można w każdej chwili zmodyfikować w następujący sposób:

```
zmienna1 = 6.7
```

Teraz zdefiniowana wcześniej zmienna przechowuje wartość 6.7. Warto zaznaczyć, że nazwy zmiennych nie mogą się zaczynać od liczb oraz nie mogą zawierać polskich znaków i spacji. Oznacza to, że te nazwy bywają często trudne do zapamiętania. W takiej sytuacji dobrze do kodu programu dołączyć komentarze. Komentarze to fragmenty kodu, które nie są wykonywane przez komputer. Służą one do opisu programu i zawsze zaczynają się od znaku #.

```
zmienna1 = 6.7 # to jest komentarz
# komentarze stosuje sie do opisywania waznych fragmentow kodu
# sam kod nie powinien zawierac polskich znakow
```

Zmienne w języku Python mogą też przechowywać liczby całkowite, ciągi znaków oraz wartości logiczne:

```
zmienna_calkowita = 3
zmienna_tekstowa = 'to jest tekst w zmiennej tekstowej'
zmienna_logiczna = True
```

2.3. PODSTAWOWE OPERACJE NA ZMIENNYCH

Przechowywanie danych w pamięci komputera to już dużo, ale bardziej interesująca jest automatyzacja ich przetwarzania. Najprostsza operacja, którą można wykonywać na zmiennych, to *print*. Służy ona do wyświetlenia wartości zmiennej (lub wyrażenia) na ekranie komputera:

```
b = 5.3
print b
```

```
5.3
```

Można także konstruować bardziej złożone wyrażenia, łącząc ciągi znaków ze zmiennymi poprzez przecinki:

```
b = 5.3
print "To jest wartosc zmiennej b: ", b
```

```
To jest wartosc zmiennej b:  5.3
```

Dodawanie zmiennych odbywa się z wykorzystaniem operatora „+”. Przykładowo, zmienną *b* można dodać do nowo utworzonej zmiennej *c* i zapisać oraz wyświetlić wynik tego działania w nowej zmiennej *d* w następujący sposób:

```
c = 11.0
d = c + b
print "To jest wynik dodawania: ", d
```

```
To jest wynik dodawania:  16.3
```

Wartość zmiennej *b* można w każdej chwili zmienić i ponownie uruchomić cały skrypt. Wynik zostanie automatycznie zaktualizowany. To pokazuje największą zaletę programowania – program to tylko zapis operacji, które należy wykonać na danych. Dane, na których operacje zostaną wykonane, mogą być dowolne.

Kolejnymi ważnymi operacjami na zmiennych są: mnożenie i dzielenie.

```
a1 = 3.5
a2 = 3.7

print a1 * a2 # mnozenie - wyrazenia mozna wpisac
              # od razu po komendzie print,
              # nie trzeba zapisywac ich wczesniej do zmiennych
print a1 / a2 # dzielenie
```

```
12.95
0.945945945946
```

Mnożenie i dzielenie działają na liczbach zmiennoprzecinkowych jak należy. Warto sprawdzić, co dzieje się podczas dzielenia liczb całkowitych.

```
calkowita1 = 10
calkowita2 = 4

print calkowita1 / calkowita2
```

```
2
```

W tym przypadku Python w wersji 2.7 zwraca tylko część całkowitą bez reszty. Można jednak wymusić dzielenie zmiennoprzecinkowe w następujący sposób:

```
calkowita1 = 10
calkowita2 = 4
print calkowita1 / float(calkowita2)
```

```
2.5
```

Komenda *float* powoduje konwersję liczby całkowitej na zmiennoprzecinkowej. Jeżeli jedna ze zmiennych w dzieleniu jest zmiennoprzecinkowa, to Python wykonuje dzielenie zmiennoprzecinkowe.

Na zmiennych można także wykonywać operacje logiczne:

```
print a1 == a2
print a1 == a1
print calkowita1 > calkowita2
print calkowita1 <= a1
```

```
False
True
True
False
```

Wynikiem takich działań są wartości logiczne: prawda lub fałsz (*True* lub *False*). W podstawowych komendach Pythona nie ma jednak bardziej złożonych operacji matematycznych. Trzeba je zaimportować z odpowiedniej biblioteki. Biblioteka *math* zawiera podstawowe funkcje matematyczne.

```
import math         # import biblioteki math

sinus_ze_zmiennej_b = math.sin(b)
print sinus_ze_zmiennej_b
```

```
0.751573415352
```

Powyżej wykorzystana została funkcja *sin()* z biblioteki *math*. Nazwę biblioteki należy umieścić w wywołaniu funkcji: *math.sin()*. Jako argument funkcji podana została zmienna *b*, a wynik zapisany został w zmiennej *sinus_ze_zmiennej_b*. Wykorzystanie pojęcia „funkcja" w tym przypadku nie jest przypadkowe. Funkcje w programowaniu omówione zostaną w kolejnych rozdziałach.

Teraz, wykorzystując poznane podstawy Pythona, można napisać pierwszy użytkowy program, który dla dwóch podanych punktów (x_1, y_1 oraz x_2, y_2) wyznaczy długość odcinka *d* pomiędzy nimi:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2.1}$$

Funkcja, która pozwala wyznaczyć pierwiastek kwadratowy z liczby, znajduje się w bibliotece *math*; jej nazwa to: *sqrt*() (ang. *square root*).

```
# Import potrzebnych bibliotek
import math

# Wprowadzenie danych
x1 = 3.4
y1 = 6.0

x2 = 94.0
y2 = 12.2

# Obliczenia
d = math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))

# Wydruk wynikow
print 'Dane: punkt1 = (', x1, y1, '), punkt2 = (', x2, y2, ')'
print 'Odleglosc pomiedzy punktami d = ', d
```

```
Dane: punkt1 = ( 3.4 6.0 ), punkt2 = ( 94.0 12.2 )
Odleglosc pomiedzy punktami d =  90.8118934942
```

2.4. STRUKTURY DANYCH – LISTA

W poprzednim rozdziale przedstawione zostały zmienne i podstawowe operacje na nich. Zmienne, które przechowują pojedyncze liczby lub wartości, to bardzo użyteczne narzędzia. Niemniej jednak w praktycznych zagadnieniach najczęściej spotyka się duże zbiory liczb, np. układ mechaniczny o 40 parametrach. W tym przypadku definiowanie nowej zmiennej na każdy parametr układu jest żmudne i prowadzi do błędów, które są trudne do wykrycia. Do rozwiązania problemów tego typu potrzebny jest większy zbiornik, w którym można przechowywać jednocześnie wiele wartości, np. liczb. Takim zbiornikiem, w uproszczeniu, jest lista. Na potrzeby tego kursu lista będzie traktowana jako jednowymiarowa tablica liczb. Listę w języku Python definiuje się za pomocą nawiasów kwadratowych. Przykładowo, lista, która przechowa współrzędne punktu 1 i punktu 2 z poprzedniego programu, wygląda następująco:

```
punkt1 = [3.4, 6.0]
punkt2 = [94.0, 12.0]
```

Do obsługi list używa się trzech podstawowych komend: *print*, *len*() oraz odwołanie do konkretnego elementu listy. Poniżej znajdują się przykłady użycia tych komend.

```
print punkt1
dlugosc_listy = len(punkt1)
print dlugosc_listy
```

```
[3.4, 6.0]
2
```

Lista zawiera dwie współrzędne (dwa elementy), dlatego jej długość jest równa 2. Do jej poszczególnych elementów można się odwołać w następujący sposób:

```
pierwszy_element_listy = punkt1[0]
print 'Pierwszy element listy: ', pierwszy_element_listy

print 'Drugi element listy: ', punkt1[1]
```

```
Pierwszy element listy:  3.4
Drugi element listy:  6.0
```

Warto zauważyć, że pierwszy element listy ma indeks 0. Indeksowanie od zera obowiązuje w Pythonie i jego bibliotekach.

Można teraz przepisać program do wyznaczania długości odcinków w znacznie krótszej i bardziej przyjaznej formie.

```
# Import potrzebnych bibliotek
import math

# Wprowadzenie danych
punkt1 = [3.4, 6.0]
punkt2 = [94.0, 12.2]

# Obliczenia
d = math.sqrt((punkt1[0] - punkt2[0]) * (punkt1[0] - punkt2[0])
              + (punkt1[1] - punkt2[1]) * (punkt1[1] - punkt2[1]))

# Wydruk wynikow
print 'Dane: punkt1 = ', punkt1, ' punkt2 = ', punkt2
print 'Odleglosc pomiedzy punktami d = ', d
```

```
Dane: punkt1 =  [3.4, 6.0]  punkt2 =  [94.0, 12.2]
Odleglosc pomiedzy punktami d =  90.8118934942
```

W programowaniu zawsze dąży się do najbardziej ogólnego zapisu problemu. Zapis z listami jest intuicyjny oraz zwięzły – listy zachowują się jak wektory i przechowują współrzędne punktów. Niemniej jednak listy w języku Python nie obsługują wektorowych operacji matematycznych – do tego potrzebna jest specjalna biblioteka, która omówiona zostanie w kolejnych rozdziałach.

2.5. INSTRUKCJE WARUNKOWE *if/else*.

Program do wyznaczania długości odcinków jest już przejrzysty, ale nadal mało ogólny. Zakłada tylko przypadek dwuwymiarowy. Co się stanie, kiedy użytkownik poda tylko jedną współrzędną dla każdego z punktów (odcinek jednowymiarowy)?

Przetestuj, usuwając współrzędne y z punktów 1 i 2 w poprzednim programie

Podczas wykonywania programu wystąpi błąd. Python zwraca następującą uwagę: *list index out of range*. Oznacza to, że wystąpiło odwołanie do elementu listy, którego nie ma – w tym przypadku *punkt1[1]*. Jak ten problem rozwiązać? Dla przypadku jednowymiarowego można zastosować uproszczoną definicję długości:

$$d = |x_1 - x_2| \tag{2.2}$$

W Pythonie wartość bezwzględną zwraca funkcja *abs()*. Można z niej skorzystać w następujący sposób:

```
# Wprowadzenie danych
punkt1 = [3.4]
punkt2 = [94.0]

# Obliczenia
d = abs(punkt1[0] - punkt2[0])

# Wydruk wynikow
print 'Dane: punkt1 = ', punkt1, ' punkt2 = ', punkt2
print 'Odleglosc pomiedzy punktami, w przyp. jednowymiar., d = ', d
```

```
Dane: punkt1 =  [3.4]  punkt2 =  [94.0]
Odleglosc pomiedzy punktami, w przyp. jednowymiar., d =  90.6
```

Należy zwrócić uwagę, że nawet listę jednoelementową Python formalnie traktuje jako listę. To znaczy, że wartość tego elementu można uzyskać poprzez dodanie indeksu – *punkt1[0]*.

Rozwiązany został już przypadek dwuwymiarowy i jednowymiarowy. Kolejnym słusznym krokiem jest złączenie ich w jeden złożony program. W ten sposób zwiększona zostanie funkcjonalność skryptu. Na początku warto zauważyć, że do obydwu programów podajemy po dwie listy, które reprezentują dwa punkty (na osi – przypadek jednowymiarowy; na płaszczyźnie – przypadek dwuwymiarowy). Naturalnym rozwiązaniem jest więc sprawdzenie, ile elementów ma lista, i zastosowanie odpowiedniej formuły do wyliczenia długości. Słownie to zadanie można opisać w następujący sposób:

> *Jeżeli lista punkt1 ma jeden element, to zastosuj wzór z wartością bezwzględną. Jeżeli lista punkt1 ma dwa elementy, to zastosuj wzór z pierwiastkiem.*

W Pythonie konstrukcję *jeżeli, to* można zrealizować poprzez *if (warunek_logiczny)*. Ilustracja działania tej konstrukcji została przedstawiona na prostym przykładzie, w którym porównywane są dwie liczby zmiennoprzecinkowe:

```
a = 3.5
b = 7.4
```

```
if (a > b):
    print "Wartosc wiekszej liczby to: ", a
if (a < b):
    print "Wartosc wiekszej liczby to: ", b
```

```
Wartosc wiekszej liczby to:  7.4
```

W pierwszym etapie program sprawdza wartość wyrażenia logicznego $a > b$. Jeżeli jest ona równa *True* (prawda), to na ekranie konsoli drukowana jest wartość zmiennej *a*. W następnym przypadku sprawdzana jest wartość wyrażenia odwrotnego $a < b$ i jeżeli ma ono wartość *True*, to na ekranie drukowana jest wartość zmiennej *b*. Warto zauważyć, że program można by znacząco uprościć, dodając czynność do wykonania, jeżeli pierwsze wyrażenie ma wartość *False* (fałsz). W tym celu można wykorzystać komendę *else*. Oznacza ona dosłownie *w innym przypadku*.

```
if (a > b):
    print "Wartosc wiekszej liczby to: ", a
else:
    print "Wartosc wiekszej liczby to: ", b
```

```
Wartosc wiekszej liczby to:  7.4
```

Tabulator przed komendą *print* w tych programach nie jest przypadkowy. Tabulatory w Pythonie definiują bloki kodu. Co to znaczy? Często spotyka się sytuację, w której po pewnej komendzie (np. *if*) zachodzi potrzeba wykonania kilku czynności (np. dodanie dwóch liczb, a później wyświetlenie wyniku tego dodawania). W takiej sytuacji trzeba te czynności zgrupować w blok. W innych językach takie bloki często wyznaczają nawiasy. W Pythonie nawiasy zastępuje wcięcie. Bez wcięcia program nie będzie działał poprawnie. Konstrukcję *if* można teraz zastosować w programie do wyznaczania długości odcinka.

```
# Import potrzebnych bibliotek
import math

# Wprowadzenie danych
punkt1 = [3.4, 6.0]
punkt2 = [94.0, 12.2]

# Obliczenia
dlugosc_listy = len(punkt1)

## przypadek jednowymiarowy
if (dlugosc_listy == 1):
    d = abs(punkt1[0] - punkt2[0])
```

```
## przypadek dwuwymiarowy
if (dlugosc_listy == 2):
    d = math.sqrt((punkt1[0] - punkt2[0]) * (punkt1[0] - punkt2[0])
                + (punkt1[1] - punkt2[1]) * (punkt1[1] - punkt2[1]))

# Wydruk wynikow
print 'Dane: punkt1 = ', punkt1, ' punkt2 = ', punkt2
print 'Odleglosc pomiedzy punktami d = ', d
```

```
Dane: punkt1 =  [3.4, 6.0]  punkt2 =  [94.0, 12.2]
Odleglosc pomiedzy punktami d =  90.8118934942
```

Program działa teraz poprawnie dla dwóch przypadków – jednowymiarowego i dwuwymiarowego.

2.6. PĘTLA *for*

Dzięki instrukcji warunkowej *if* program do wyznaczania długości odcinka działa już zarówno dla przypadku jednowymiarowego, jak i dwuwymiarowego. W algebrze liniowej można jednak wyznaczyć długość odcinka pomiędzy dwoma *n*-wymiarowymi punktami. Jak to uwzględnić w programie? Pierwsza opcja to dodanie kolejnych instrukcji warunkowych dla kolejnych długości listy. Takie rozwiązanie jest jednak mało efektywne i bardzo ograniczone – trzeba uwzględnić wszystkie przypadki manualnie. Aby to zautomatyzować, należy przeanalizować sposób wyznaczania długości odcinka. Zakładając, że dane są dwa punkty $a = (a_1, a_2, ..., a_n)$ i $b = (b_1, b_2, ..., b_n)$, długość odcinka d pomiędzy a i b wyznacza się w następujący sposób:

$$d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}. \tag{2.3}$$

Wraz ze zmianą liczby wymiarów n we wzorze zmienia się liczba wyrażeń $(a_i - b_i)^2$ pod pierwiastkiem. Wzór na długość można w takim razie zapisać także za pomocą operatora sumowania:

$$d = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}. \tag{2.4}$$

Czyli, znając n, wiadomo dokładnie, ile takich kwadratów różnic współrzędnych należy do siebie dodać. Tę czynność można zautomatyzować za pomocą pętli. Pętla służy w programowaniu do wykonania tego samego kodu wiele razy, zazwyczaj na innych danych. W przypadku punktów wielowymiarowych należy n razy wyznaczyć wartość wyrażenia $(a_i - b_i)^2$ dla kolejnych współrzędnych.

Jeżeli z góry wiadomo, ile powtórzeń pętli jest potrzebnych, tak jak w tym przypadku, to można wykorzystać pętlę *for*. Jej działanie oraz składnia została przedstawiona na kilku prostych przykładach.

```
for i in range(4):
    print '!'
```

```
!
!
!
!
```

Powyższa pętla cztery razy wykonuje instrukcję *print '!'* Efektem działania są cztery wykrzykniki na ekranie konsoli. Definicję pętli zaczyna się od słowa *for*. Następnie definiuje się zmienną, która przechowuje numer powtórzenia (iteracji). Należy także napisać, w jakim zakresie zmienia się ta zmienna. W tym przypadku komenda *range(4)* oznacza zakres od 0 do 3, czyli cztery powtórzenia. Warto zauważyć, że podobnie jak w przypadku instrukcji warunkowej *if*, należy stosować wcięcia po definicji pętli. Można teraz wydrukować na ekran numer powtórzenia.

```
print 'Przed wejściem do petli for…'

for i in range(5):
    print 'To jest powtorzenie nr ', i

print 'Tu juz nie ma wcięcia; ta instrukcja wykona sie tylko 1 raz'
```

```
Przed wejściem do petli for...
To jest powtorzenie nr  0
To jest powtorzenie nr  1
To jest powtorzenie nr  2
To jest powtorzenie nr  3
To jest powtorzenie nr  4
Tu juz nie ma wcięcia; ta instrukcja wykona sie tylko 1 raz
```

Wewnątrz pętli *for* można wykorzystywać zmienną iterującą. Aby to zaprezentować, poniżej przedstawiono program, który drukuje na ekran konsoli ciąg kwadratów liczb od 1 do 6.

```
print 'Poczatek drukowania.'

# zmienna iterujaca moze miec dowolna nazwe - to zwykla zmienna
# tutaj: j
for j in range(6):
```

```
    akt_liczba = j + 1     # j zaczyna od 0
    akt_liczba_kw = aktualna_liczba * aktualna_liczba

    print 'Kwadrat liczby ', akt_liczba, ' to ', akt_liczba_kw

print 'Koniec drukowania.'
```

```
Poczatek drukowania.
Kwadrat liczby  1  to  1
Kwadrat liczby  2  to  4
Kwadrat liczby  3  to  9
Kwadrat liczby  4  to  16
Kwadrat liczby  5  to  25
Kwadrat liczby  6  to  36
Koniec drukowania.
```

Za pomocą zmiennej iterującej można w pętli uzyskać dostęp do kolejnych elementów listy:

```
punkt1 = [3.4, 6.0, 4.0, 4.2]   # punkt 4-wymiarowy

dlugosc_listy = len(punkt1)

# liczba powtorzen moze byc zalezna od innych zmiennych w programie
for j in range(dlugosc_listy):
    print ,Wspolrzedna ', j, ' punktu to ', punkt1[j]
```

```
Wspolrzedna  0  punktu to  3.4
Wspolrzedna  1  punktu to  6.0
Wspolrzedna  2  punktu to  4.0
Wspolrzedna  3  punktu to  4.2
```

Zaprezentowane przykłady pokazują, że wewnątrz pętli *for* można wykonywać złożone czynności. Można ją także wykorzystać do dalszej rozbudowy programu, czyli do wyznaczania długości odcinka *n*-wymiarowego.

```
# Import potrzebnych bibliotek
import math

# Wprowadzenie danych
p1 = [3.4, 6.0, 4.0, 4.2]      # punkty 4-wymiarowe
p2 = [94.0, 12.2, 5.0, 100.3]

# Obliczenia
dlugosc_listy = len(punkt1)
```

```
suma = 0.0

for i in range(dlugosc_listy):
    suma = suma + (p1[i] - p2[i]) * (p1[i] - p2[i])

# powyzsza petla wykonala się tyle razy, ile wymiarow ma odcinek –
# tutaj 4; w każdej iteracji do zmiennej suma dodawano
# kolejne kwadraty roznicy wspolrzednych

d = math.sqrt(suma)

# Wydruk wynikow
print 'Dane: p1 = ', p1, ' p2 = ', p2
print 'Odleglosc pomiedzy punktami d = ', d
```

```
Dane: p1 =  [3.4, 6.0, 4.0, 4.2]  p2 =  [94.0, 12.2, 5.0, 100.3]
Odleglosc pomiedzy punktami d =  132.223333796
```

Teraz program do wyznaczania długości odcinka zadziała dla list o dowolnej liczbie elementów. Co więcej, dzięki pętli *for* można było nie tylko zwiększyć funkcjonalność skryptu, ale też zmniejszyć jego długość. Krótki i czytelny kod ułatwia utrzymanie i rozbudowę programu.

2.7. FUNKCJE, CZYLI O ORGANIZACJI KODU

Do tej pory tło rozdziałów stanowiły podstawowe słowa kluczowe wykorzystywane w języku Python. Z tych konstrukcji można zbudować dowolnie złożony program. Niemniej jednak należy jeszcze poznać narzędzia, które ułatwiają pracę z kodem i jego późniejsze modyfikacje. Podstawowym narzędziem tego typu jest funkcja. Funkcja w programowaniu jest analogiem funkcji matematycznej. Posiada argumenty i zwraca pewne wartości, które powstają poprzez przetworzenie argumentów. W poprzednich rozdziałach przedstawione zostały już funkcje zdefiniowane w bibliotekach podstawowych Pythona, *sin()* i *abs()*. Sposób wywołania funkcji *sin()* był następujący:

```
sinus_ze_zmiennej_b = math.sin(b)
```

W bardziej ogólnej formie wygląda to tak:

wartosc_funkcji = funkcja(arg_funkcji_1, arg_funkcji_2)

Przed nazwą funkcji wpisywana jest zmienna, w której zapisywana jest wartość, jaką zwraca funkcja (np. wartość sinusa dla danego kąta). Później pojawia się znak równości, który pełni rolę operatora przypisania, czyli przypisuje wartość, jaką zwraca funkcja do zmiennej. Następnie podaje się nazwę funkcji oraz jej argumenty (np. inne zmienne albo wartości). Aby zdefiniować własną funkcję, należy zacząć od komendy *def*. Następnie podawana jest nazwa funkcji i nawias, w którym zawarte są jej argumenty. Pierwsza linia definicji kończy się dwukropkiem. Funkcja, tak samo jak instrukcja *if* oraz pętla *for*, otwiera nowy blok kodu, dlatego po pierwszej linii należy używać wcięć. Dla przykładu prosta funkcja, która dodaje dwie liczby i zwraca wynik tego dodawania, ma następującą postać:

```
def dodaj_dwie_liczby(liczba1, liczba2):
    wynik_dodawania = liczba1 + liczba2

    return wynik_dodawania
```

Na końcu powyższego kodu pojawiła się komenda *return*. To drugi po słowie *def* kluczowy składnik funkcji. Służy on do określenia zmiennej, której wartość zwraca funkcja – w tym przypadku jest to zmienna *wynik_dodawania*. Warto zauważyć, że zmienna *wynik_dodawania* została zdefiniowana wewnątrz funkcji. Takie zmienne nazywa się lokalnymi. Dostęp do niej możliwy jest tylko wewnątrz funkcji. Ogólny skrypt, taki jak wszystkie programy do tej pory, nie będzie miał dostępu do tej zmiennej. Jest ona wykorzystywana wyłącznie do uzyskania wyniku funkcji. Po wywołaniu funkcji (np. po policzeniu wartości sinusa dla danego kąta) jest ona usuwana z pamięci komputera. Komenda *return* nie jest niezbędna. Można zdefiniować funkcje, które nie zwracają żadnej wartości – mogą one np. wyświetlać wynik na ekranie:

```
def wyswietl_zmienna_na_ekranie(liczba):
    print 'Wartosc zmiennej to ', liczba
```

Przygotowane funkcje to dopiero połowa sukcesu. W kolejnym kroku trzeba je użyć (wywołać, z ang. *call*) w głównym programie. Przykład zastosowania zaprezentowanych funkcji przedstawiono poniżej.

```
def dodaj_dwie_liczby(liczba1, liczba2):
    wynik_dodawania = liczba1 + liczba2
    return wynik_dodawania

def wyswietl_zmienna_na_ekranie(liczba):
    print 'Wartosc zmiennej to ', liczba
```

```
a = 3.45
b = 7.89

wynik = dodaj_dwie_liczby(a, b)
wyswietl_zmienna_na_ekranie(wynik)
```

```
Wartosc zmiennej to  11.34
```

Funkcja to przepis (szereg operacji), co zrobić ze zmiennymi *liczba1* i *liczba2*. Zdefiniowaną funkcję można już wywołać na dowolnych argumentach. W tym przypadku Python wykonuje następującą operację podczas wywołania. Wartość zmiennej *a* jest kopiowana do zmiennej *liczba1*. Wartość zmiennej *b* jest kopiowana do zmiennej *liczba2*. Dalsze czynności wykonywane są już na zmiennych *liczba1* i *liczba2*. Zmienne *liczba1*, *liczba2* oraz *wynik_dodawania* są zmiennymi lokalnymi. To znaczy, że istnieją tylko w obrębie funkcji – po komendzie *return* są usuwane z pamięci. Zmienne *a* i *b*, które zdefiniowaliśmy poza funkcjami, to zmienne globalne. To znaczy, że dostęp do nich możliwy jest z poziomu całego programu oraz jego wszystkich funkcji. Poniższy program prezentuje różnicę w działaniu zmiennych lokalnych i globalnych.

```
a = 5.5 # zmienna globalna

# poznizej zdefiniowana została prosta funkcja,
# ktora nie przyjmuje zadnych argumentow;
# funkcja wyswietla tylko wartosc zmiennej globalnej a na ekranie

def funkcja_testowa():
    print a

# wywolanie funkcji i wydruk wyniku jej dzialania
print funkcja_testowa()
```

```
5.5
```

Przykład potwierdza, że można uzyskać dostęp do zmiennej globalnej *a* z poziomu funkcji. Co się stanie, jeżeli jednym z argumentów funkcji jest zmienna o takiej samej nazwie jak jedna ze zmiennych globalnych?

```
a = 2
b = 4

def funkcja_testowa(a):
    print a
```

```
print funkcja_testowa(b)
print funkcja_testowa(a)
```

```
4
2
```

Przy wywołaniu funkcji dla zmiennej *b* wydrukowana została wartość zmiennej *b*. Oznacza to, że zmienna *a* użyta w definicji funkcji jest zmienną lokalną. W tym samym czasie w programie występują dwie zmienne *a* – jedna o zasięgu globalnym i druga o lokalnym. Funkcja widzi tylko *a* o zasięgu lokalnym, ponieważ w przypadku podwójnego nazewnictwa priorytet ma zmienna lokalna funkcji. Ten przykład pokazuje, jak ważne jest staranne nazywanie zmiennych w programie. Wykorzystanie zmiennych globalnych powinno być zredukowane do minimum. Powodują one, że kod programu jest trudny w analizie i wymaga dodatkowych testów.

Funkcje można teraz wykorzystać do rozbudowy programu do wyznaczania długości odcinka.

```
# Import
import math

# Definicje funkcji
def wyznacz_dlugosc(p1, p2):
    dlugosc_listy = len(p1)

    suma = 0.0
    for i in range(dlugosc_listy):
        suma = suma + (p1[i] – p2[i]) * (p1[i] - p2[i])
    return math.sqrt(suma)

# Wprowadzenie danych
punkt1 = [3.4, 6.0, 4.0, 4.2]  #punkty 4-wymiarowe
punkt2 = [94.0, 12.2, 5.0, 100.3]

# Obliczenia
d = wyznacz_dlugosc(punkt1, punkt2)

# Wydruk wynikow
print 'Dane: p1 = ', punkt1, ' p2 = ', punkt2
```

```
print ,Odleglosc pomiedzy punktami d = ,, d
```

```
Dane: p1 =  [3.4, 6.0, 4.0, 4.2]  p2 =  [94.0, 12.2, 5.0, 100.3]
Odleglosc pomiedzy punktami d =  132.223333796
```

Warto zauważyć, że część obliczeniowa programu jest teraz bardzo przejrzysta i zwięzła – to jedna z zalet funkcji. Program można jednak jeszcze bardziej uprościć, przygotowując funkcję do wydruku wyników.

```
# Import
import math

# Definicje funkcji
def wyznacz_dlugosc(p1, p2):
    dlugosc_listy = len(p1)

    suma = 0.0
    for i in range(dlugosc_listy):
        suma = suma + (p1[i] – p2[i]) * (p1[i] - p2[i])
    return math.sqrt(suma)

def wydrukuj_wyniki(p1, p2, dd):
    print 'Dane: p1 = ', p1, ' p2 = ', p2
    print 'Odleglosc pomiedzy punktami d = ', dd

# Wprowadzenie danych
punkt1 = [3.4, 6.0, 4.0, 4.2]  # punkty 4-wymiarowe
punkt2 = [94.0, 12.2, 5.0, 100.3]

# Obliczenia
d = wyznacz_dlugosc(punkt1, punkt2)

# Wydruk wynikow
wydrukuj_wyniki(punkt1, punkt2, d)
```

```
Dane: p1 =  [3.4, 6.0, 4.0, 4.2]  p2 =  [94.0, 12.2, 5.0, 100.3]
Odleglosc pomiedzy punktami d =  132.223333796
```

Drugą i najważniejszą zaletą stosowania funkcji jest to, że przygotowane funkcje można wielokrotnie wywołać w kodzie programu na różnych danych wejściowych. W przypadku wyznaczania długości odcinka może to być kilkanaście par punktów, co często ma miejsce w rozwiązywaniu modeli biomechanicznych, gdzie długości te to parametry członów zastępujących np. więzadła w układach szkieletowo-stawowych człowieka. Dwukrotne wywołanie przygotowanej funkcji na różnych zestawach danych wejściowych zaprezentowano na poniższym przykładzie.

```
# Import
import math

# Definicje funkcji
```

```
def wyznacz_dlugosc(p1, p2):
    dlugosc_listy = len(p1)

    suma = 0.0
    for i in range(dlugosc_listy):
        suma = suma + (p1[i] – p2[i]) * (p1[i] - p2[i])
    return math.sqrt(suma)

def wydrukuj_wyniki(p1, p2, dd):
    print 'Dane: p1 = ', p1, ' p2 = ', p2
    print 'Odleglosc pomiedzy punktami d = ', dd

# Dane1
punkt3 = [3.4, 4.0]
punkt4 = [2.0, 7.2]

# Obliczenia i wydruk wynikow
d34 = wyznacz_dlugosc(punkt3, punkt4)
wydrukuj_wyniki(punkt3, punkt4, d34)

# Dane2
punkt5 = [1.4, 4.0, 20.0, 40.0, 230.0]
punkt6 = [2.0, 7.2, 15.9, 2.0, 21.5]

# Obliczenia i wydruk wynikow
d56 = wyznacz_dlugosc(punkt5, punkt6)
wydrukuj_wyniki(punkt5, punkt6, d56)
```

```
Dane: p1 =  [3.4, 4.0]  p2 =  [2.0, 7.2]
Odleglosc pomiedzy punktami d =  3.49284983931
Dane: p1 =  [1.4, 4.0, 20.0, 40.0, 230.0]
      p2 =  [2.0, 7.2, 15.9, 2.0, 21.5]
Odleglosc pomiedzy punktami d =  211.999198112
```

W ten sposób podstawowy program do wyznaczania długości odcinka na podstawie dwóch wektorów został rozbudowany do wersji, która jest czytelna, łatwa do rozbudowy i działa na dowolnych wektorach. W ramach tego rozdziału przedstawione zostały podstawowe komendy i konstrukcje Pythona, które pozwalają tworzyć złożone programy obliczeniowe. Kolejnym etapem będzie wprowadzenie do specjalizowanych bibliotek numerycznych Pythona: *Numpy* i *Scipy* oraz biblioteki do wizualizacji graficznej *Matplotlib*.

Warto dodać, że definicje funkcji nie muszą być zawarte w głównym skrypcie. Python umożliwia definiowanie funkcji w innych plikach. Takie funkcje można potem zaimportować w głównym kodzie. Dla niedużych programów nie jest to jednak wymagane.

2.8. DODATKOWE BIBLIOTEKI Pythona

Python jako język programowania oferuje podstawowe struktury danych, takie jak lista, którą przedstawiono wcześniej. Niemniej jednak te struktury dostosowane są do typowych problemów programistycznych i biznesowych, a nie naukowych. Przykładowo, jeżeli w programie zdefiniowane zostały dwie listy, które reprezentują dwa wektory, to przydatną operacją byłoby dodanie tych list do wektorów, czyli element po elemencie. Niestety zachowanie Pythona w tym przypadku znacznie odbiega od oczekiwań, co przedstawiono na przykładzie poniżej.

```
wektor_1 = [1.0, 2.0, 4.0]
wektor_2 = [-1.0, -2.0, -4.0]

wynik_dodawania = wektor_1 + wektor_2

print wynik_dodawania
```

```
[1.0, 2.0, 4.0, -1.0, -2.0, -4.0]
```

Matematyczny wynik to w tym przypadku wektor, w którym wszystkie współrzędne są równe 0. Wynik otrzymany z działania skryptu to dwa razy dłuższa lista, która zawiera w sobie *wektor_1* i *wektor_2*. Rezultat ten potwierdza, że bazowe struktury zaimplementowane w języku Python nie pozwalają na wygodne przeprowadzanie obliczeń. Z pomocą przychodzi tutaj zewnętrzna biblioteka do obsługi macierzy – *Numpy* [2]. Biblioteka ta jest już zainstalowana w pakiecie *WinPython*, dlatego można ją od razu zaimportować.

```
import numpy as np
```

Warto zauważyć, że funkcja *import* pozwala nie tylko importować, ale i nadawać określone nazwy importowanym bibliotekom. W tym przypadku biblioteka *Numpy* będzie w skrypcie figurowała jako *np*. Ten skrót nie jest przypadkowy, wykorzystuje go większość dostępnych przykładów w internecie.

Po zaimportowaniu biblioteki można utworzyć tablice *Numpy*, które w programie będą reprezentowały pełnoprawne wektory. Tablice te tworzy się na podstawie list, co zaprezentowano poniżej.

```
wektor_1np = np.array(wektor_1)
wektor_2np = np.array(wektor_2)

print wektor_1np, wektor_2np
```

```
wynik_dodawania_np = wektor_1np + wektor_2np

print wynik_dodawania_np
```

```
[1. 2. 4.] [-1. -2. -4.]
[0. 0. 0.]
```

W tym przypadku wynik dodawania jest już poprawny. Operacja ta została przeprowadzona w sposób matematyczny.

Do elementów tablicy *Numpy* można odwoływać się podobnie jak dla list.

```
print wektor_1np[1]
```

```
2.0
```

Tablice można też tworzyć od razu za pomocą jednej linijki kodu.

```
wektor_3 = np.array([1., 6., 8.])

print wektor_3
```

```
[1. 6. 8.]
```

Array w bibliotece *Numpy* to uniwersalna struktura, która może reprezentować także wielowymiarowe tablice. Takie tablice inicjalizuje się, umieszczając listy w listach, jak w przykładzie poniżej.

```
macierz = np.array([[1., 2.], [4., 5.]])

print macierz
```

```
[[1. 2.]
 [4. 5.]]
```

Powyższy przykład prezentuje proces tworzenia macierzy 2×2 za pomocą biblioteki *Numpy*. W analogiczny sposób można tworzyć macierze o innych wymiarach, a także trójwymiarowe tensory. Dostęp do elementów macierzy 2×2 jest bardzo podobny do metody dla wektorów. Różni się tylko tym, że należy podać dwa indeksy, które odpowiadają wierszowi o kolumnie, przykładowo:

```
print macierz[1, 0]
print macierz[0, 1]
```

```
4.0
2.0
```

Biblioteka *Numpy* oferuje bardzo wiele przydatnych operacji na tablicach *Array*. Wybrane z nich przedstawiono w skrypcie poniżej.

```
wektor4 = np.array([4., 5.])
macierz2 = np.array([[-1., 0.], [0., -1.]])
macierz3 = np.array([[0., 1.], [2., 0.]])

# mnozenie macierzy
wynik1 = np.dot(macierz2, wektor4)
print wynik1
print "------"

# mnozenie macierzy typu element-wise
# tzw. tablicowe
wynik2 = macierz2 * macierz3
print wynik2
wynik3 = wektor4 * wektor4
print wynik3
print "------"

# operacja sinus/cosinus typu element-wise
# //wykonywana na każdym elemencie macierzy; bardzo szybka
print np.sin(wektor4)
print "------"

print np.sin(macierz2)
```

```
[-4. -5.]
------
[[-0.  0.]
 [ 0. -0.]]
------
[16. 25.]
------
[-0.7568025  -0.95892427]
------
[[-0.84147098  0.        ]
 [ 0.         -0.84147098]]
```

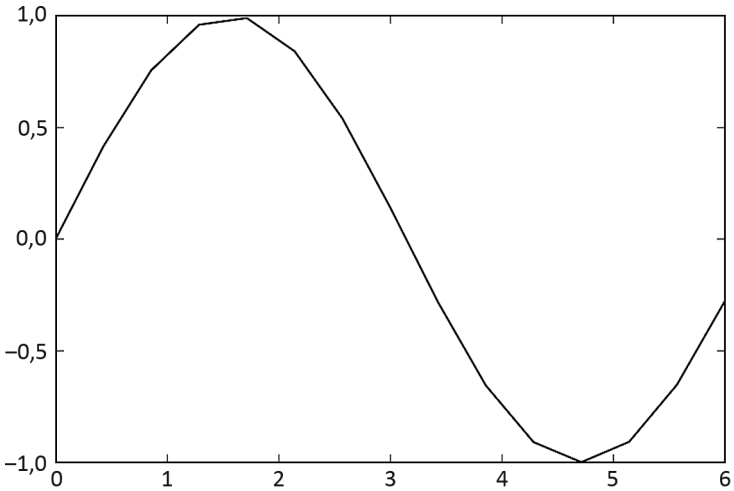
Proces tworzenia typowych macierzy można także zautomatyzować za pomocą funkcji wbudowanych w bibliotekę *Numpy*.

```
x = np.linspace(0.0, 5.0, num = 10)
print x
```

```
[0.         0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
3.33333333 3.88888889 4.44444444 5.        ]
```

Drugą bardzo przydatną biblioteką dostępną w pakiecie *WinPython* jest biblioteka *Matplotlib*. Umożliwia ona łatwe generowanie wykresów, do których dane możemy dostarczyć za pomocą tablicy *Numpy*. Poniżej przedstawiono prosty przykład, w którym rysowany jest wykres funkcji sinus (rys. 2.6).

```
# import modułu pyplot z biblioteki matplotlib
import matplotlib.pyplot as plt

# ponizsza linijka kodu konfiguruje notatnik Jupyter tak,
# aby wyswietlal wykresy od razu pod komorkami;
# bez niej wyswietli je w osobnych oknach
%matplotlib inline

# argumenty funkcji sinus:
argumenty = np.linspace(0.0, 6.0, num = 15)

# wartosci funkcji sinus:
wartosci = np.sin(argumenty)

# rysowanie wykresu:
plt.plot(argumenty, wartosci)
plt.show()
```



Rys. 2.6. Funkcja sinus narysowana z wykorzystaniem biblioteki *Matplotlib*

Biblioteka *Matplotlib* [3] pozwala także opisywać wykresy, modyfikować ich kolory, eksportować je w wysokiej rozdzielczości i inne. Niektóre z tych funkcji zostaną wprowadzone w kolejnych rozdziałach. Niemniej jednak szczegółowy poradnik użytkownika dla tej biblioteki można znaleźć pod adresem [4].

3. DYNAMIKA UKŁADÓW JEDNOWYMIAROWYCH

W tym rozdziale wprowadzone zostaną podstawowe metody rozwiązywania zwyczajnych równań różniczkowych drugiego rzędu w zastosowaniu do układów mechanicznych i biomechanicznych rozważanych w jednym wymiarze.

Zasadą, która opisuje dynamiczne zachowanie ciał w przestrzeni, jest druga zasada dynamiki Newtona. Według niej przyspieszenie, z którym porusza się ciało, jest wprost proporcjonalne do działającej na nie wypadkowej siły. Współczynnikiem proporcjonalności jest masa ciała:

$$m\boldsymbol{a} = \boldsymbol{F}. \tag{3.1}$$

gdzie:

- m – masa ciała,
- $\boldsymbol{a}$ – wektor przyspieszenia, z jakim porusza się ciało,
- $\boldsymbol{F}$ – wektorowa suma wszystkich sił działających na działo, która działa na ciało.

Przyspieszenie definiuje się jako zmianę prędkości w czasie, czyli:

$$\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt}. \tag{3.2}$$

Podstawiając (3.2) do (3.1), uzyskuje się:

$$m\frac{d\boldsymbol{v}}{dt} = \boldsymbol{F}. \tag{3.3}$$

Powyższe równanie jest równaniem różniczkowym zwyczajnym pierwszego rzędu. Przyjmując, że prędkość to zmiana położenia w czasie, czyli:

$$\boldsymbol{v} = \frac{d\boldsymbol{x}}{dt}. \tag{3.4}$$

Wykorzystując (3.4), równanie (3.3) można zapisać także w następującej formie:

$$\frac{d^2\boldsymbol{x}}{dt^2} = \frac{\boldsymbol{F}}{m}. \tag{3.5}$$

Równanie (3.5) to równanie różniczkowe zwyczajne drugiego rzędu. Używane jest najczęściej podczas poszukiwania rozwiązań analitycznych do zadań z zakresu

mechaniki ogólnej. Warto zauważyć, że zamiast podstawienia, równania (3.3) i (3.4) można zapisać razem, jako układ równań różniczkowych pierwszego rzędu:

$$\begin{cases} \dfrac{d\boldsymbol{v}}{dt} = \dfrac{\boldsymbol{F}}{m} \\ \dfrac{d\boldsymbol{x}}{dt} = \boldsymbol{v} \end{cases}. \tag{3.6}$$

Układ ten jest szczególnie przydatny podczas numerycznego rozwiązywania zadań mechanicznych – większość dostępnych metod numerycznych dostosowana jest do rozwiązywania równań pierwszego rzędu.

Aby wyprowadzić te metody, należy najpierw zrozumieć, czym jest iloraz funkcji:

$$\frac{\Delta f}{\Delta x} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}. \tag{3.7}$$

Alternatywna postać wzoru (3.7) to:

$$\frac{\Delta f}{\Delta x} = \frac{f(x_1 + \Delta x) - f(x_1)}{\Delta x}. \tag{3.8}$$

Jeżeli zmiana argumentów Δx (lub $x_2 - x_1$) dąży do zera, to powyższy iloraz przedstawia pochodną funkcji $\dfrac{df}{dx}$. Przyjęcie dowolnej, skończonej zmiany argumentów spowoduje, że iloraz stanie się przybliżeniem pochodnej. Jednakże dla bardzo małej różnicy argumentów będzie to użyteczne przybliżenie. Wartość tego przyrostu dobierana jest najczęściej przez użytkownika podczas symulacji.

Tak wyprowadzone przybliżenie pochodnej można teraz podstawić do wzoru (3.3). Dodatkowo, równanie można też rozpisać już na współrzędnych, w tym przypadku – jednej współrzędnej x. W ten sposób otrzymuje się algebraiczną formę równania różniczkowego opisującego prędkość ciała:

$$\frac{v(t_2) - v(t_1)}{t_2 - t_1} = \frac{F}{m}. \tag{3.9}$$

Równanie w takiej formie to po prostu „przepis", który pozwala oszacować prędkość $v(t + dt)$, z którą ciało będzie się poruszało w chwili $t + dt$, jeżeli znana jest prędkość $v(t)$ w chwili t. To oszacowanie będzie tym dokładniejsze, im mniejszy przyrost czasu dt. Zapis ten można uprościć i przygotować do implementacji w programie komputerowym poprzez wprowadzenie następujących oznaczeń: prędkość w chwili t, v_{cur} (ang. *current* – aktualna), a prędkość w chwili $t + dt$, v_{next} (ang. *next* – następna). Równanie z nowymi oznaczeniami wygląda następująco:

$$\frac{v_{\text{next}} - v_{\text{cur}}}{\Delta t} = \frac{F}{m}. \tag{3.10}$$

Warto zauważyć, że jeżeli znana jest wartość prędkości dla jakiegoś czasu, np. prędkość początkowa $v(0{,}0\ \mathrm{s})$, to wykorzystując powyższe równanie, można wyznaczyć kolejną wartość prędkości po upływie czasu Δt. Następnie ta nowo wyznaczona wartość może posłużyć do obliczenia prędkości po upływie $2 \cdot \Delta t$. Cały proces można powtarzać tak długo, jak wymaga tego symulacja. Otrzymuje się w ten sposób przybliżoną funkcję prędkości w czasie. Z uwagi na wymaganą „pierwszą" wartość prędkości jest to przykład problemu z warunkami początkowymi (ang. *initial conditions*). Równanie (3.10) reprezentuje rodzinę funkcji. Dopiero po określeniu warunków początkowych, czyli w tym przypadku, prędkości dla danej wartości czasu, można wyznaczyć konkretną funkcję dla danego przypadku. Oznacza to, że dla różnych warunków początkowych (np. $v(0{,}0\ \mathrm{s}) = 4{,}0\ \mathrm{m/s}$ i $v(0{,}0\ \mathrm{s}) = -2{,}0\ \mathrm{m/s}$) otrzymane funkcje będą różne.

Aby bardziej sformalizować zapis, należy przekształcić równanie tak, aby po lewej stronie pozostały wszystkie niewiadome, czyli w tym przypadku tylko prędkość v_{next}. Do równania należy też dołączyć warunek początkowy (*IC*), ponieważ jest niezbędny do uzyskania rozwiązania:

$$\begin{aligned} \boldsymbol{IC}&: v_{\mathrm{cur}} = 0\ \mathrm{m/s} \quad \text{dla} \quad t = 0\ \mathrm{s} \\ & v_{\mathrm{next}} = \frac{F}{m}\Delta t + v_{\mathrm{cur}} \end{aligned} \quad . \tag{3.11}$$

Równanie w tej formie to można już łatwo rozwiązać z wykorzystaniem komputera. W analogiczny sposób można rozpisać wzór na położenie (3.4):

$$\begin{aligned} \boldsymbol{IC}&: x_{\mathrm{cur}} = 0\ \mathrm{m} \quad \text{dla} \quad t = 0\ \mathrm{s} \\ & x_{\mathrm{next}} = v_{(?)}\Delta t + x_{\mathrm{cur}} \end{aligned} \quad . \tag{3.12}$$

W tym przypadku również podawany jest warunek początkowy, który określa położenie ciała na początku analizy. Po rozpisaniu pochodnej w sposób przybliżony pojawia się jednak pewien problem – co podstawić za prędkość? Najłatwiej wszystkie wielkości w równaniach (v, F i inne) wyznaczyć dla aktualnego stanu układu (ang. *current*; indeks: *cur*). Zgodnie z powyższym, za v podstawione zostanie v_{cur}, a siłę wyznaczy się, wykorzystując t_{cur}, x_{cur} i v_{cur}: $F_{\mathrm{cur}} = F(t_{\mathrm{cur}}, x_{\mathrm{cur}}, v_{\mathrm{cur}})$. Po uwzględnieniu powyższych rozważań równania można zapisać w formie następującego układu:

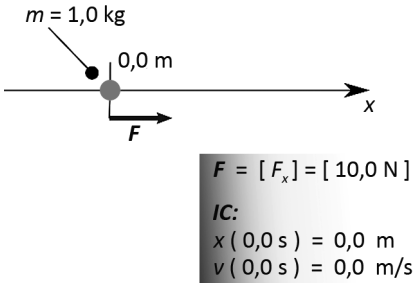
$$\begin{aligned} \boldsymbol{IC1}&: v_{\mathrm{cur}} = 0\ \mathrm{m/s} \quad \text{dla} \quad t = 0\ \mathrm{s} \\ \boldsymbol{IC2}&: x_{\mathrm{cur}} = 0\ \mathrm{m} \quad \text{dla} \quad t = 0\ \mathrm{s} \\ & v_{\mathrm{next}} = \frac{F}{m}\Delta t + v_{\mathrm{cur}} \\ & x_{\mathrm{next}} = v_{\mathrm{cur}}\Delta t + x_{\mathrm{cur}} \end{aligned} \quad . \tag{3.13}$$

W ten sposób z układu równań różniczkowych pierwszego rzędu uzyskany został układ liniowych równań algebraicznych, które opisują zachowanie ciała w przypad-

ku jednowymiarowym. Powyższa metoda ma w matematyce nazwę metody Eulera i należy do grupy metod pierwszego rzędu [5]. Otrzymane równania można teraz zaimplementować w programie komputerowym.

3.1. METODA EULERA W PROSTYCH SYMULACJACH DYNAMICZNYCH

Symulacje treningowe warto rozpocząć od prostych przypadków, dla których znane są rozwiązania analityczne. Takim zadaniem jest ruch punktu materialnego pod wpływem siły zewnętrznej o stałej wartości. Niech punkt ma masę m = 1,0 kg, a działająca na niego siła zewnętrzna ma wartość 10,0 N oraz zwrot i kierunek zgodny ze zwrotem i kierunkiem osi x. Dodatkowo niech ciało w chwili t = 0,0 s będzie w spoczynku (v (0,0 s) = 0,0 m/s) oraz w położeniu x (0,0 s) = 0,0 m. Zadanie przedstawia rys. 3.1.



Rys. 3.1. Wizualizacja zadania – siła o stałej wartości działa na punkt materialny

Warto zauważyć, że w równaniach dynamiki (3.6) występuje tylko jedna siła. Jest to wypadkowa siła, która na działa na ciało. Powstaje ona poprzez wektorowe zsumowanie wszystkich sił działających na ciało podczas ruchu. Z tego powodu rozwiązanie każdego zadania poprzedza wykonanie tak zwanego diagramu ciała swobodnego. Powstaje on poprzez oddzielenie od badanego ciała elementów, które na niego oddziałują (liny, cięgna, sprężyny, wymuszenia kinematyczne i inne) i zastąpienie ich oddziaływania siłami. W przypadku ruchu pod wpływem stałej siły zewnętrznej, otrzymujemy bardzo prosty diagram, patrz: rys. 3.2. Siła w równaniach (3.6) to po prostu siła zewnętrzna. Diagramy będą jednak konsekwentnie stosowane we wszystkich przedstawionych przykładach.

Rys. 3.2. Wizualizacja zadania – diagram ciała swobodnego

Po określeniu sił działających na ciało można już przygotować program do numerycznego rozwiązania zadania. Dobrze go zacząć od wprowadzenia danych układu:

```
# Dane ukladu
m = 1.0 #[kg]
F = 1.0 #[N]

# IC - initial conditions
x0 = 0.0 #[m]
v0 = 0.0 #[m/s]

# parametry rozwiazania
delta_t = 0.01 #[s]  -> krok czasowy
steps = 10 #         -> liczba krokow
```

Teraz można już zaimplementować wprowadzoną wcześniej metodę Eulera. Wzór (3.13) zawiera „przepis", który pozwala wyznaczyć kolejną prędkość i kolejne położenie punktu na podstawie prędkości i położenia poprzedniego. W pierwszym kroku do wyznaczenia kolejnych wartości służą warunki początkowe – początkową prędkość i położenie.

```
# pierwszy krok - metoda Eulera

# a) podstawiamy warunki poczatkowe do zmiennych
# okreslajacych aktualna predkosc i polozenie;
# inicjalizujemy zmienna przechowujaca aktualny czas
v_cur = v0  #[m/s]
x_cur = x0  #[m]
t = 0.0     #[s]

# wyznaczenie kolejnych wartosci predkosci
# i polozenia oraz aktualnego czasu
v_next = F / m * delta_t + v_cur
x_next = v_cur * delta_t + x_cur
t = t + delta_t

# wydruk wynikow
print "Akt. czas: ", t, "s | akt. pol.: ", x_next, "m | akt. pred.:
", v_next, "m/s"
```

```
Akt. czas:  0.01 s | akt. pol.:  0.0 m | akt. pred.:  0.01 m/s
```

Powyższy skrypt wyznacza kolejną prędkość i położenie ciała dla czasu $t + dt$ według metody Eulera. To na razie tylko jedno, kolejne położenie i prędkość. Program

należy teraz rozbudować tak, żeby wyznaczał cały szereg kolejnych położeń i prędkości.

Jak wspomniano wcześniej, wyznaczone wartości prędkości i położenia można wykorzystać jako nowe warunki początkowe w układzie równań (3.13) i na ich podstawie wyznaczyć kolejne (dla czasu t + 2.0 * *delta_t*). Proces ten można powtarzać tak długo, aż czas symulacji osiągnie założoną wartość. Do zautomatyzowania tego procesu najbardziej nadaje się pętla *for*:

```
# a) warunki poczatkowe
v_cur = v0  #[m/s]
x_cur = x0  #[m]
t = 0.0     #[s]

############################################################
# b) petla for, w ktorej rozwiazywany jest uklad (3.13)
for i in range(steps):
    # wyznaczenie kolejnych wartosci predkosci i polozenia
    # oraz aktualnego czasu
    v_next = F / m * delta_t + v_cur
    x_next = v_cur * delta_t + x_cur

    # --- ! --- ważne --- ! ---
    # podstawiamy wyliczona predkosc i polozenie
    # do zmiennych przechowujacych aktualne polozenie/predkosc
    v_cur = v_next
    x_cur = x_next

    # aktualizacja czasu
    t = t + delta_t

    # wydruk wynikow z iteracji
    print "#", i, ", czas:", t, ", pol.:", x_next, ", pred.:", v_next
```

```
# 0 , czas: 0.01 , pol.: 0.0 , pred.: 0.01
# 1 , czas: 0.02 , pol.: 0.0001 , pred.: 0.02
# 2 , czas: 0.03 , pol.: 0.0003 , pred.: 0.03
# 3 , czas: 0.04 , pol.: 0.0006 , pred.: 0.04
# 4 , czas: 0.05 , pol.: 0.001 , pred.: 0.05
# 5 , czas: 0.06 , pol.: 0.0015 , pred.: 0.06
# 6 , czas: 0.07 , pol.: 0.0021 , pred.: 0.07
# 7 , czas: 0.08 , pol.: 0.0028 , pred.: 0.08
# 8 , czas: 0.09 , pol.: 0.0036 , pred.: 0.09
# 9 , czas: 0.1 , pol.: 0.0045 , pred.: 0.1
```

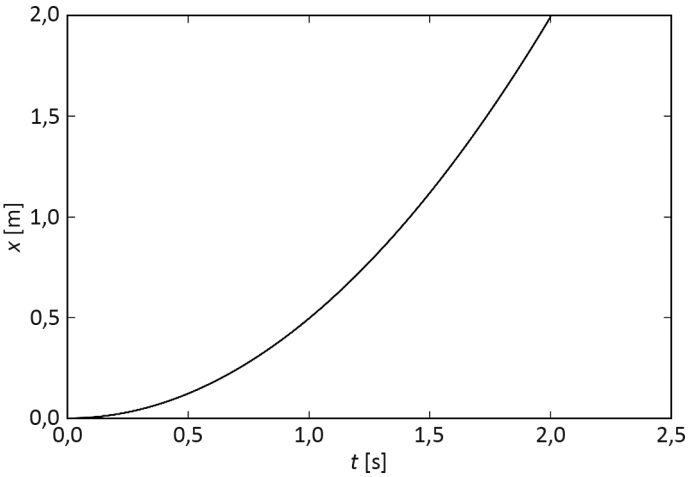
Można teraz rozbudować program, dodając do niego graficzną wizualizację położenia ciała w czasie. Do rozwiązania tego problemu wykorzystane zostaną wprowadzone wcześniej biblioteki *Numpy* oraz *Matplotlib*:

```
import numpy as np
import matplotlib.pyplot as plt
```

W obecnej wersji skrypt drukuje wyniki po każdej iteracji pętli. Poza wydrukiem na ekranie w pamięci komputera nie ma historii zmian położenia i prędkości ciała, ponieważ zmienne, które przechowują te wartości, są modyfikowane w każdej iteracji. Aby temu zaradzić, do skryptu należy dodać dwie macierze *Numpy*, które będą przechowywały aktualne położenie oraz czas w każdej iteracji – prędkość w tym przypadku nie będzie rejestrowana.

```
# a) utworzenie pustych (na razie) macierzy dla czasu i polozenia;
# ich rozmiar okreslany jest na podstawie zalozonej wczesniej
# liczby powtorzen w petli for
t_matrix = np.zeros(steps)
x_matrix = np.zeros(steps)

# b) warunki poczatkowe
v_cur = v0  #[m/s]
x_cur = x0  #[m]
t = 0.0     #[s]

# c) petla for, w ktorej rozwiazywany jest uklad (3.13)
for i in range(steps):
    # wyznaczenie kolejnych wartosci predkosci i polozenia
    # oraz aktualnego czasu
    v_next = F / m * delta_t + v_cur
    x_next = v_cur * delta_t + x_cur

    # --- ! --- wazne --- ! ---
    # podstawiamy wyliczona predkosc i polozenie
    # do zmiennych przechowujacych aktualne polozenie/predkosc
    v_cur = v_next
    x_cur = x_next

    #aktualizacja czasu
    t = t + delta_t

    #zapisanie wynikow do macierzy
    t_matrix[i] = t
```

```
    x_matrix[i] = x_cur

# d) wydruk wyników w postaci wykresu polozenia od czasu
plt.plot(t_matrix, x_matrix)

# podpisy pod osiami
plt.xlabel('t [s]')
plt.ylabel('x [m]')

plt.show()
```



Rys. 3.3. Wynik działania skryptu – wykres położenia punktu od czasu

Wynikiem działania skryptu jest teraz wykres. Warto zauważyć, że otrzymany wykres przypomina charakterem funkcję kwadratową.

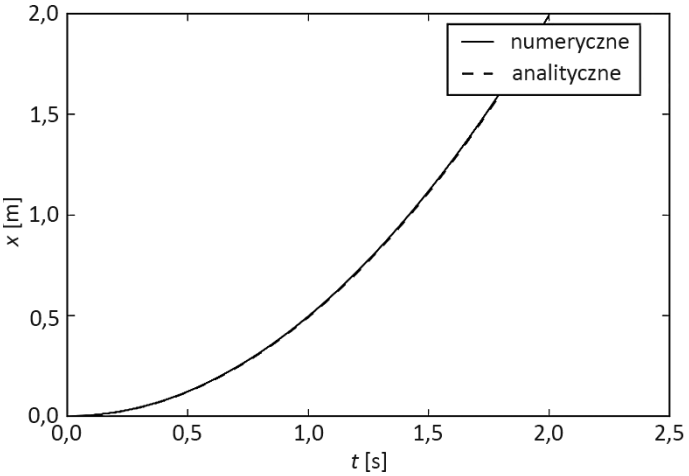
W kolejnym kroku należy porównać otrzymane rozwiązanie numeryczne z analitycznym. Rozwiązaniem analitycznym rozważanego równania różniczkowego jest wspomniana funkcja kwadratowa w następującej postaci:

$$x(t) = \frac{F}{m}\frac{t^2}{2}. \tag{3.14}$$

Implementacja z wizualizacją tego rozwiązania w programie wygląda następująco:

```
# a) rozwiazanie analityczne
x_a_matrix = np.zeros(steps)

for i in range(steps):
```

```
    x_cur = F / m * (delta_t * i) * (delta_t * i) / 2.0
    x_a_matrix[i] = x_cur

plt.plot(t_matrix, x_matrix)
plt.plot(t_matrix, x_a_matrix, “r”)

# podpisy pod osiami
plt.xlabel('t [s]')
plt.ylabel('x [m]')

plt.show()
```



Rys. 3.4. Wynik działania skryptu – wykres położenia punktu od czasu dla rozwiązania numerycznego oraz analitycznego dla 200 kroków

Na podstawie powyższego wykresu można zauważyć, że metoda Eulera w tym bardzo prostym przypadku mechanicznym zwróciła sensowne wyniki. Niemniej jednak, żeby je uzyskać, potrzebny był bardzo mały krok obliczeniowy. Następny przykład pozwoli sprawdzić bardziej złożone zagadnienia, kiedy siły nie są stałe w czasie i zależą od położenia ciała.

Przykładem bardziej złożonego zagadnienia jest przypadek, w którym na ciało działa dodatkowo liniowa sprężyna o współczynniku sztywności k = 100,0 N/m oraz o długości swobodnej *l_swob* = 1,0 m. Problem przedstawiono na rys. 3.5. Drugi koniec sprężyny doczepiony jest do nieruchomej podstawy w położeniu b = –1,0 m. Reszta parametrów układu nie ulega zmianie.

Problem modelowania elementów podatnych jest szczególnie ważny z punktu widzenia biomechaniki i modelowania układów biomechanicznych. Jednowymiarowa sprężyna o liniowej charakterstyce, a w szczegól-

ności jednowymiarowa sprężyna o jednokierunkowym działaniu, to jeden z podstawowych modeli więzadeł.

Modele elementów podatnych przygotowuje się ogólnie za pomocą rachunku wektorowego. Niemniej jednak dla układów jednowymiarowych, opis ten można znacząco uprościć. W pierwszym etapie warto przyjąć zależność, która pozwoli wyznaczyć wartość siły od sprężyny. W przypadku liniowym jest to iloczyn wydłużenia sprężyny, czyli różnicy pomiędzy jej aktualną i swobodną długością, oraz współczynnika sztywności:

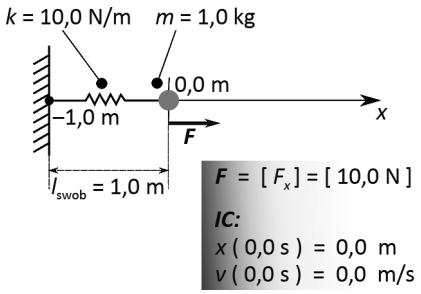
$$F_s = k\Delta l, \tag{3.15}$$

gdzie:

k – współczynnik sztywności sprężyny,

$\Delta l = l_{\text{cur}} - l_{\text{swob}}$ – wydłużenie sprężyny.

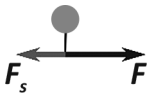
Warto zauważyć, że długość swobodna jest stała i znana. Natomiast długość aktualną l_{cur} trzeba wyznaczyć na podstawie położenia ciała. Przy założeniu, że przyczep ruchomy sprężyny ***a*** porusza się tak samo jak ciało, jego położenie w każdym momencie zgodne jest z położeniem ciała. W takim przypadku długość aktualna sprężyny to po prostu $l_{cur} = |x_{\text{cur}} - b|$. Ten wzór jest słuszny tylko dla liniowej sprężyny w przypadku jednowymiarowym. W dalszych rozdziałach zostanie uogólniony z wykorzystaniem rachunku wektorowego.



Rys. 3.5. Wizualizacja zadania – siła o stałej wartości działa na punkt materialny, do którego doczepiona jest także sprężyna o liniowej charakterystyce

Można teraz przystąpić do implementacji zadania (rys. 3.5). Jego rozwiązanie należy rozpocząć od przygotowania diagramu ciała swobodnego. Jak wspomniano wcześniej, przygotowanie takiego diagramu polega na odłączeniu od ciała badanego wszystkich innych ciał (np. sprężyny, cięgna, tłumiki, ciała sztywne itd.) i zastąpieniu ich siłami oddziaływań, zgodnie z trzecią zasadą dynamiki Newtona. W tym przypadku na ciało ruchome działa siła zewnętrzna oraz sprężyna. Po odłączeniu sprężyny do ciała należy przyłożyć siłę generowaną przez sprężynę. Wynikowy diagram przedstawiono na rysunku poniżej. Aby lepiej uwidocznić zjawiska za-

chodzące w układzie, liczba kroków obliczeniowych w skrypcie zostanie zwiększona z 10 do 500.



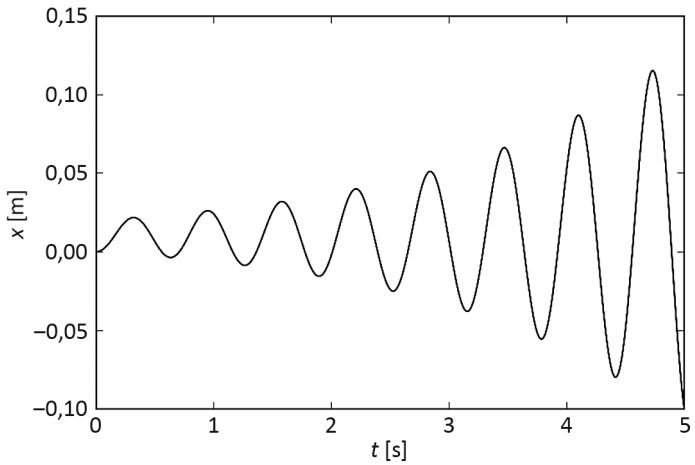
Rys. 3.6. Wizualizacja zadania – diagram ciała swobodnego

Korzystając z rysunku 3.6, można rozpisać równania dla tego zadania:

$$\begin{aligned} \textbf{\textit{IC}1}&: v_{\text{cur}} = 0{,}0\ \text{m/s} \quad \text{dla} \quad t = 0{,}0\ \text{s} \\ \textbf{\textit{IC}2}&: x_{\text{cur}} = 0{,}0\ \text{m} \quad \text{dla} \quad t = 0{,}0\ \text{s} \\ v_{\text{next}} &= \frac{F_{\text{next}} - F_s}{m}\Delta t + v_{\text{cur}} \\ x_{\text{next}} &= v_{\text{cur}}\Delta t + x_{\text{cur}} \end{aligned} \quad . \tag{3.16}$$

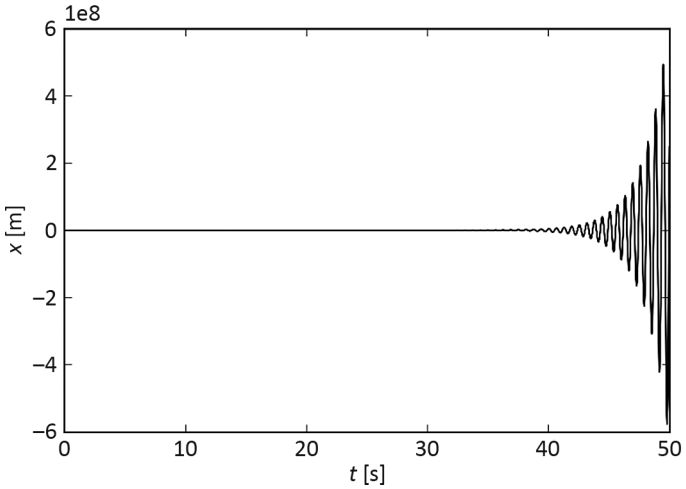
```
steps = 500   # zwiekszamy liczbe krokow obliczeniowych
k = 100.0     #[N/m] - wspolczynnik sztywnosci sprezyny
l_swob = 1.0  #[m]   - dlugosc swobodna sprezyny
b = -1.0      #[m]   - polozenie nieruchomego przyczepu sprezyny

# a) utworzenie pustych (na razie) macierzy dla czasu i polozenia;
# rozmiar okreslany jest na podstawie parametru steps
t_matrix = np.zeros(steps)
x_matrix = np.zeros(steps)

# b) warunki poczatkowe
v_cur = v0    #[m/s]
x_cur = x0    #[m]
t = 0.0       #[s]

# c) petla for, w ktorej rozwiazywany jest uklad (3.13)
for i in range(steps):
    # wyznaczenie wypadkowej sily dzialajacej na cialo
    # (sila zewnetrzna + sila od sprezyny)
    l_cur = x_cur - b         # aktualna dlugosc sprezyny
    delta_l = l_cur - l_swob  # wydluzenie sprezyny
    F_spr = k * delta_l       # sila od sprezyny

    F_wyp = F - F_spr         # sila wypadkowa

    # wyznaczenie kolejnych wartosci predkosci i polozenia
    # oraz aktualnego czasu
```

```
    v_next = F_wyp / m * delta_t + v_cur
    x_next = v_cur * delta_t + x_cur

    # --- ! --- wazne --- ! ---
    # podstawiamy wyliczona predkosc i polozenie
    # do zmiennych przechowujacych aktualne polozenie/predkosc
    v_cur = v_next
    x_cur = x_next

    #aktualizacja czasu
    t = t + delta_t

    #zapisanie wynikow do macierzy
    t_matrix[i] = t
    x_matrix[i] = x_cur

# d) wydruk wyników w postaci wykresu polozenia od czasu
plt.plot(t_matrix, x_matrix)
plt.xlabel('t [s]')
plt.ylabel('x [m]')
plt.show()
```



Rys. 3.7. Wynik działania skryptu – wykres położenia punktu od czasu dla układu ze sprężyną

Analizując powyższe rozwiązanie, można stwierdzić, że energia symulowanego układu wzrasta w czasie, ponieważ stale rośnie amplituda drgań. Jest to oczywiście niepoprawne. Amplituda w tym przypadku, przy braku tłumienia, powinna być stała. Błędy symulacyjne są jeszcze bardziej widoczne dla większej liczby kroków – 5000 (rys. 3.8).



Rys. 3.8. Wynik działania skryptu – wykres położenia punktu od czasu dla układu ze sprężyną – duża liczba iteracji

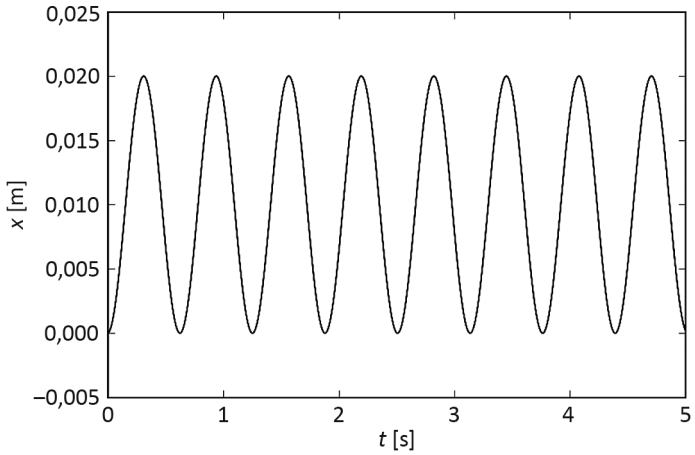
W tej symulacji amplituda szybko wzrasta do nieskończoności. Już po 50 sekundach jej wartość wynosi 493 222 031,17926294 m. Bardzo wyraźnie widać, że metoda Eulera nie zwraca dobrych wyników w zastosowaniu do symulacji mechanicznych. W literaturze można spotkać bardzo wiele metod do rozwiązywania zwyczajnych równań różniczkowych pierwszego rzędu. Jedną z bardziej popularnych jest tzw. pół-niejawna metoda Eulera (ang. *semi-implicit Euler method*), nazywana także metodą Eulera-Cromera [6]. Jej zalety to przede wszystkim bardzo mała złożoność numeryczna oraz bardzo łatwa implementacja. Warto też wspomnieć, że w ostatnich latach metoda ta używana jest przy symulacjach ciał odkształcalnych [7]. Do implementacji metody pół-niejawnej można wykorzystać przygotowany wcześniej skrypt z metodą Eulera. Jedyna modyfikacja, którą należy wykonać, to zamiana kodu:

```
x_next = v_cur * delta_t + x_cur
```

na:

```
x_next = v_next * delta_t + x_cur
```

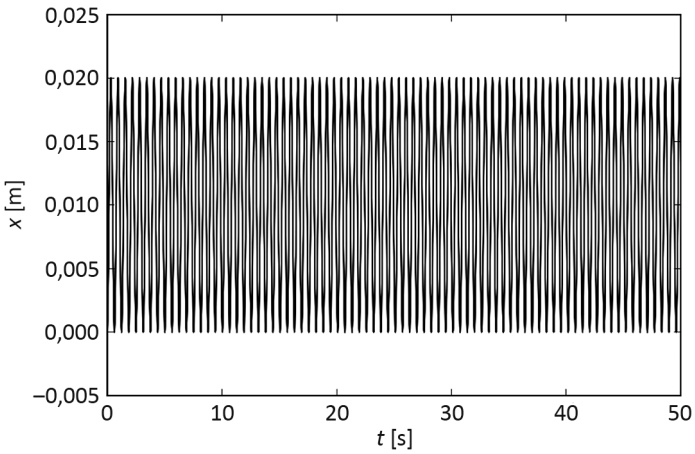
W tym przypadku do wyznaczenia kolejnego położenia nie jest wykorzystywana aktualna prędkość *v_cur*, a prędkość następna *v_next*, wyznaczona w tym kroku obliczeniowym. Wyniki otrzymane zmodyfikowaną metodą dla przykładu ze sprężyną oraz kod potrzebny do uzyskania wyników są następujące.



Rys. 3.9. Wynik działania skryptu – wykres położenia punktu od czasu dla układu ze sprężyną – metoda pół-niejawna – 500 iteracji

```
steps = 500

# a) utworzenie pustych (na razie) macierzy dla czasu i polozenia;
# rozmiar okreslany jest na podstawie zalozonej
# wczesniej liczby powtorzen w petli for
t_matrix = np.zeros(steps)
x_matrix = np.zeros(steps)

# dodatkowe parametry ukladu - sprezyna
k = 100.0     #[N/m] - wspolczynnik sztywnosci sprezyny
l_swob = 1.0  #[m]   - dlugosc swobodna sprezyny
b = -1.0      #[m]   - polozenie nieruchomego przyczepu sprezyny

# b) warunki poczatkowe
v_cur = v0    #[m/s]
x_cur = x0    #[m]
t = 0.0       #[s]

# c) petla for, w ktorej rozwiazywany jest uklad (4)
for i in range(steps):
    # wyznaczenie wypadkowej sily dzialajacej na cialo
    # (sila zewnetrzna + sila od sprezyny)
    l_cur = x_cur - b         # aktualna dlugosc sprezyny
    delta_l = l_cur - l_swob  # wydluzenie sprezyny
    F_spr = -k * delta_l      # sila od sprezyny

    F_wyp = F + F_spr         # sila wypadkowa
```

```
    # wyznaczenie kolejnych wartosci predkosci i polozenia
    # oraz aktualnego czasu
    v_next = F_wyp / m * delta_t + v_cur
    x_next = v_next * delta_t + x_cur  ##### modyfikacja #####

    # --- ! --- wazne --- ! ---
    # podstawiamy wyliczona predkosc i polozenie
    # do zmiennych przechowujacych aktualne polozenie/predkosc
    v_cur = v_next
    x_cur = x_next

    # aktualizacja czasu
    t = t + delta_t

    # zapisanie wynikow do macierzy
    t_matrix[i] = t
    x_matrix[i] = x_cur

# d) wydruk wyników w postaci wykresu polozenia od czasu
plt.plot(t_matrix, x_matrix)
# podpisy pod osiami
plt.xlabel('t [s]')
plt.ylabel('x [m]')

plt.show()
```



Rys. 3.10. Wynik działania skryptu – wykres położenia punktu od czasu dla układu ze sprężyną – metoda pół-niejawna – 5000 iteracji

Niewielka modyfikacja programu ma ogromny wpływ na jakość otrzymywanych wyników. Amplituda jest w tym przypadku względnie stała w czasie całej sy-

mulacji. Warto też przeanalizować rozwiązanie dla większej liczby kroków obliczeniowych.

Zarówno metoda Eulera, jak i pół-niejawna metoda Eulera to tylko dwie z bardzo dużej liczby dostępnych metod. Zainteresowani czytelnicy mogą jednak poszerzyć ten temat, wykorzystując między innymi procedury dostępne w bibliotece *Scipy* [8], a także książkę [5].

4. METODY ROZWIĄZYWANIA ZADAŃ STATYKI DLA UKŁADÓW JEDNOWYMIAROWYCH

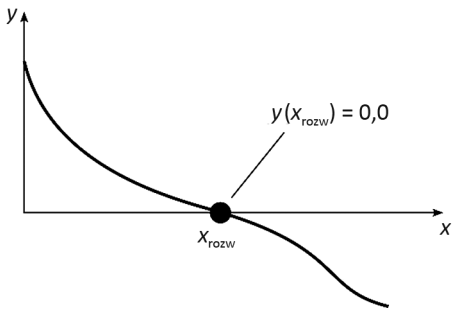
W poprzednim rozdziale omówione zostały metody numerycznego rozwiązywania zadań dynamiki dla jednowymiarowych układów mechanicznych. Można teraz przystąpić do problemów statycznych. W problemach statyki równaniami, które opisują stan układu, są równania równowagi [9]:

$$\boldsymbol{F} = 0, \tag{4.1}$$

gdzie:

$\boldsymbol{F}$ – to wektorowa suma sił działających na badane ciało.

Równania równowagi, w odróżnieniu od równań dynamiki, są równaniami algebraicznymi, nie występuje w nich już przyspieszenie, czyli druga pochodna po przemieszczeniu. W ogólnym przypadku są to jednak równania nieliniowe, dla których nie można uzyskać rozwiązań metodami analitycznymi (patrz: funkcja nieliniowa $y(x)$ na rys. 4.1). W związku z tym do ich rozwiązania wykorzystuje się metody numeryczne. Prawdopodobnie najbardziej popularną metodą w tej dziedzinie jest metoda Newtona-Raphsona, którą w różnych wariantach przedstawiono między innymi w [5]. Polega ona na wielokrotnym, lokalnym przybliżaniu równań nieliniowych za pomocą liniowych. Rozwiązanie takiego przybliżonego równania liniowego rzadko odpowiada rozwiązaniu równania nieliniowego, ale każde kolejne przybliżenie jest najczęściej lepsze. Proces ten ilustruje rys. 4.1.



Rys. 4.1. Równanie nieliniowe

Cała procedura zaczyna się od podania punktu startowego. Punkt startowy to po prostu początkowe oszacowanie rozwiązania badanych równań, które podaje użytkownik. Dobrze jest, gdy znajduje się on w pobliżu właściwego rozwiązania. Następnie, na podstawie rozwinięcia w równań w szereg Taylora, wyznaczane jest liniowe przybliżenie $y(x)$. W kolejnym kroku przybliżenie to jest rozwiązywane, a rozwiązanie przyjmuje się za następny punkt startowy, i cały proces jest powtarzany. Proces powtarzany jest tak długo, aż otrzymane rozwiązanie wyzeruje $y(x)$ z założoną tolerancją, zazwyczaj oznaczaną *eps*.

Metoda Newtona-Raphsona polega na lokalnym przybliżaniu równań nieliniowych liniowymi. Należy w takim razie zastanowić się nad sposobem, który umożliwi linearyzowanie funkcji nieliniowych. Wyprowadzenia zostaną ograniczone do skalarnej funkcji jednej zmiennej. Do zlinearyzowania funkcji można wykorzystać wzór Taylora [10], którego postać jest następująca:

$$f(x) = f(x_0) + \frac{f'(x_0)(x-x_0)}{1!} + \frac{f''(x_0)(x-x_0)^2}{2!} + \dots . \tag{4.2}$$

Zgodnie z tym wzorem wartość dowolnej funkcji f dla argumentu x można wyrazić poprzez jej wartość dla innego argumentu x_0 i sumę kolejnych pochodnych tej funkcji w punkcie x_0 przemnożonych m.in. przez różnicę argumentów.

Jeżeli suma pochodnych ograniczona zostanie do pochodnej pierwszego rzędu, to wzór będzie reprezentował właśnie liniowe przybliżenie badanej funkcji.

$$f(x) = f(x_0) + f'(x_0)(x-x_0). \tag{4.3}$$

Celem w każdej iteracji jest rozwiązanie otrzymanego równania liniowego, dlatego (4.3) warto przyrównać do zera:

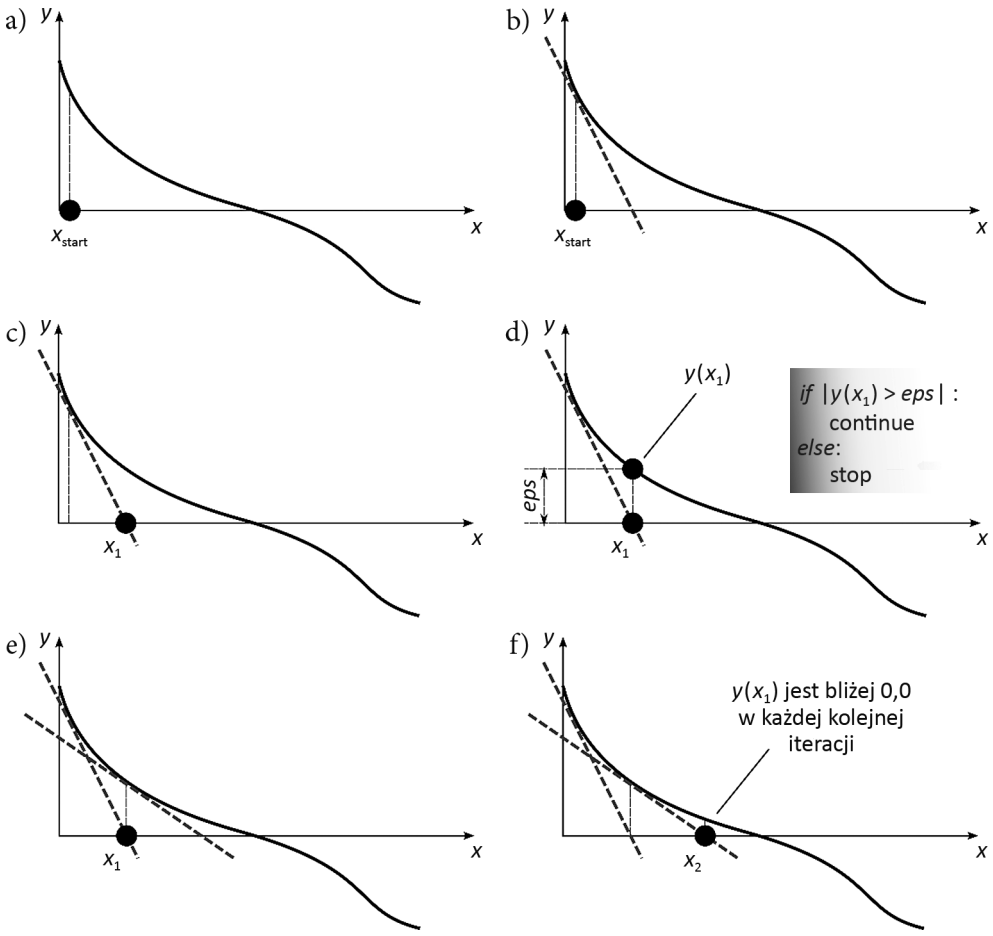
$$f(x_0) + f'(x_0)(x-x_0) = 0. \tag{4.4}$$

Zakładając, że pochodna $f'(x_0)$ jest znana (ten problem zostanie poruszony w dalszej części skryptu), w powyższym równaniu występuje tylko jedna niewiadoma – x. Jest to równanie liniowe, które przybliża początkowe równanie nieliniowe w rejonie argumentu x_0. Po prostych przekształceniach otrzymuje się:

$$x = \frac{f'(x_0)x_0 - f(x_0)}{f'(x_0)}. \tag{4.5}$$

Powyższy wzór przedstawia przybliżone rozwiązanie nieliniowego równania $y(x) = 0$. Do wyznaczenia pozostaje tylko pochodna funkcji f dla argumentu x_0. Ten problem najłatwiej rozwiązać numerycznie. Rozważania należy rozpocząć od definicji pochodnej funkcji f dla argumentu x_0. Pomijając założenia, jakie musi spełnić funkcja f, definicja pochodnej jest następująca [10]:

$$f'(x_0) = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}. \tag{4.6}$$



Rys. 4.2. Kolejne etapy metody Newtona-Raphsona

W programach komputerowych nie można zastosować nieskończenie małego przyrostu argumentów Δx. Przyrost ten zawsze jest skończony. Przy dokładnych obliczeniach może być bardzo mały, ale nigdy nie będzie nieskończenie mały. Dlatego, dla zastosowań numerycznych, powyższą definicję można sprowadzić do ilorazu różnicowego poprzez usunięcie granicy [5]:

$$f'(x_0) = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}. \tag{4.7}$$

Jest to tylko przybliżenie pochodnej, ale najczęściej wystarcza ono do obliczeń numerycznych. Warto zauważyć, że im mniejszy przyrost argumentów, tym bardziej dokładna wartość przybliżenia.

Przedstawiony sposób szacowania pochodnej ma jednak duże ograniczenia. Pochodna dla argumentu x_0 wyznaczana jest tylko na podstawie dodatkowego argu-

mentu $(x_0 + \Delta x)$ tylko z jednej strony. Dużo stabilniejszym podejściem jest wykorzystanie argumentów $x_0 + \Delta x$ i $x_0 - \Delta x$ [5]:

$$f'(x_0) = \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x}. \tag{4.8}$$

Tak wyznaczoną numeryczną pochodną nazywa się pochodną centralną.

Można teraz zaimplementować zaprezentowane podejścia w Pythonie. Przygotowany program przetestowany zostanie dla funkcji kwadratowej:

$$f(x) = x^2. \tag{4.9}$$

```
#Skrypt wyznacza numerycznie wartosc pochodnej funkcji kwadratowej
#w punkcie x0 dla zadanego przyrostu

#Dane:
x0 = 2.0        #punkt, w ktorym chcemy wyznaczyc pochodna
delta_x = 0.1   #zalozony przyrost

#Pochodna jednostronna:
f_prim_jedn = ((x0 + delta_x)**2 - (x0)**2) / delta_x

#Pochodna centralna:
f_prim_cent = ((x0 + delta_x)**2 - (x0 - delta_x)**2) / (2.0 *
delta_x)

#Wydruk wyników
print "Pochodna jednostronna: ", f_prim_jedn
print "Pochodna centralna:", f_prim_cent
```

```
Pochodna jednostronna:  4.1
Pochodna centralna: 4.0
```

Dokładną wartość pochodnej funkcji kwadratowej dla argumentu $x_0 = 2{,}0$ można wyznaczyć według wzoru:

$$f'(x) = 2x. \tag{4.10}$$

```
#Pochodna dokladna:
f_prim = 2.0 * x0
print "Pochodna dokladna: ", f_prim
```

```
Pochodna dokladna:  4.0
```

Na podstawie powyższego przykładu warto zauważyć, że pochodna centralna zwraca dokładniejsze wyniki niż pochodna jednostronna. W tym przypadku, z uwa-

gi na prostotę badanej funkcji, pochodna centralna ma dokładnie taką samą wartość jak analityczna – ogólnie tak nie jest. Przygotowany kod warto teraz uogólnić poprzez zastosowanie funkcji:

```
def f(x):
    #funkcja zwraca wartosc f(x) dla podanego argumentu x

    return x * x

def wyznacz_pochodna_jednostronna(funkcja, x, delta_x = 0.1):
    #funkcja zwraca wartosc pochodnej funkcji
    #dla podanego argumentu x dla podanego przyrostu delta_x
    #wykorzystywany jest wzor na pochodna jednostronna

    return (funkcja(x0 + delta_x) - funkcja(x0)) / delta_x

def wyznacz_pochodna_centralna(funkcja, x, delta_x = 0.1):
    #funkcja zwraca wartosc pochodnej funkcji
    #dla podanego argumentu x dla podanego przyrostu delta_x
    #wykorzystywany jest wzor na pochodna centralna

    licznik = (funkcja(x0 + delta_x) - funkcja(x0 - delta_x))
    mianownik = (2.0 * delta_x)

    return  licznik / mianownik
```

Wprowadzenie funkcji sprawi, że kod programu będzie bardziej przejrzysty i czytelny. Łatwiej będzie także wykonać w nim zmiany. Przykładowo, teraz zmiana funkcji, dla której wyznaczamy pochodną, będzie wymagała tylko modyfikacji pythonowej funkcji $f(x)$. Wcześniej trzeba było niezależnie zmienić wyrażenie na pochodną jednostronną i centralną. Warto zauważyć, że w tym przypadku *Python* pozwala na przekazywanie funkcji (zdefinowiowanych przez *def*) jako argumentów do innych funkcji. Takie funkcje można później wykorzystywać wewnątrz funkcji, do której zostały przekazane. Odbywa się to w taki sam sposób jak w skrypcie głównym. Warto zaznaczyć, że wprowadzenie funkcji to duża zmiana w kodzie programu – skrypt trzeba ponownie przetestować, najlepiej na poprzednim przykładzie z funkcją kwadratową.

```
#Skrypt wyznacza numerycznie wartosc pochodnej funkcji kwadratowej
#w punkcie x0 dla zadanego przyrostu

#Dane:
x0 = 2.0        #punkt, w ktorym chcemy wyznaczyc pochodna
delta_x = 0.1   #zalozony przyrost
```

```
#Pochodna jednostronna:
f_prim_jedn = wyznacz_pochodna_jednostronna(f, x0, delta_x)

#Pochodna centralna:
f_prim_cent = wyznacz_pochodna_centralna(f, x0, delta_x)

#Pochodna dokladna:
f_prim = 2.0 * x0

#Wydruk wyników
print "Pochodna jednostronna: ", f_prim_jedn
print "Pochodna centralna:", f_prim_cent
print "Pochodna dokladna: ", f_prim
```

```
Pochodna jednostronna:  4.1
Pochodna centralna: 4.0
Pochodna dokladna:  4.0
```

Uzyskane wyniki są w zgodzie z poprzednią wersją programu – skrypt po zmianach działa poprawnie. Można go teraz przetestować na bardziej złożonej funkcji – $sin(x)$. Najpierw należy ponownie zdefiniować w Pythonie funkcję $f(x)$:

```
def f(x):
    #funkcja zwraca wartosc f(x) dla podanego argumentu x

    return sin(x)
```

Ponownie zdefiniowana funkcja $f(x)$ nadpisze poprzednią implementację dla funkcji kwadratowej. Można teraz wywołać obliczenia pochodnych.

```
from math import sin, cos   #import funkcji sinus oraz cosinus

#Dane:
x0 = 1.2                #punkt, w ktorym wyznaczana jest pochodna
delta_x = 0.1           #zalozony przyrost

#Pochodna jednostronna:
f_prim_jedn = wyznacz_pochodna_jednostronna(f, x0, delta_x)

#Pochodna centralna:
f_prim_cent = wyznacz_pochodna_centralna(f, x0, delta_x)

#Pochodna dokladna:
f_prim = cos(x0)
```

```
#Wydruk wyników
print "Pochodna jednostronna: ", f_prim_jedn
print "Pochodna centralna:", f_prim_cent
print "Pochodna dokladna: ", f_prim
```

```
Pochodna jednostronna:  0.3151909945
Pochodna centralna: 0.361754126779
Pochodna dokladna:  0.362357754477
```

Na tym przykładzie wyraźnie widać, że pochodna jednostronna zwraca mniej dokładne wyniki niż centralna. Jednakże nawet pochodna centralna jest tylko przybliżeniem i różni się od wartości analitycznej. Na tę różnicę duży wpływ ma założony przyrost argumentów. Aby zbadać jego wpływ, można przygotować niewielki program, który wydrukuje wartość pochodnej centralnej dla różnych wartości przyrostu. Zostanie w tym celu wykorzystana pętla *for*. Poszczególne wyniki będą zapisywane w tablicy *Numpy*.

```
import numpy as np

liczba_powtorzen = 4

#przygotowanie miejsca na wyniki w pamieci komputera
delta_x_macierz = np.zeros(liczba_powtorzen)
f_prim_macierz = np.zeros(liczba_powtorzen)

for i in range(liczba_powtorzen):
    delta_x_macierz[i] = 10**(i + 1) * 0.0001  #aktualne delta_x

    f_prim_c = wyznacz_pochodna_centralna(f, x0, delta_x_macierz[i])
    f_prim_macierz[i] = f_prim_c

print "Macierz delta_x: ", delta_x_macierz
print "Macierz f_prim : ", f_prim_macierz
print "Pochodna dokladna: ", f_prim
```

```
Macierz delta_x:  [0.001 0.01  0.1   1.   ]
Macierz f_prim :  [0.36235769 0.36235172 0.36175413 0.30491354]
Pochodna dokladna:  0.362357754477
```

Powyższy przykład potwierdza wspomnianą wcześniej zależność: im mniejszy przyrost, tym dokładniejsza wartość pochodnej. Już przy przyroście *delta_x* = 0,001 otrzymywana jest zgodność do piątego miejsca po przecinku pomiędzy pochodną centralną a dokładną.

Zaimplementowane zostały już funkcje, które pozwalają na numeryczne wyznaczenie pochodnej dowolnej funkcji. W programie przygotowana jest także funk-

cja $f(x)$, która w wygodny sposób przechowuje badaną funkcję. Można teraz zaprogramować metodę Newtona-Raphsona. Przykładem testowym do sprawdzenia procedury będzie równanie kwadratowe:

$$x^2 + 3,0x - 4,0 = 0,0. \tag{4.11}$$

W tym przypadku delta równania wynosi:

$$\Delta = b^2 - 4,0ac = 3,0^2 - 4,0 \cdot 1,0 * (-4,0) = 25,0. \tag{4.12}$$

Delta jest dodatnia, co oznacza, że równanie posiada dwa rozwiązania:

$$\begin{aligned} x_1 &= -4,0 \\ x_2 &= 1,0 \end{aligned}. \tag{4.13}$$

Teraz można przejść do implementacji rozwiązania numerycznego. Rdzeniem programu będzie wyprowadzony wcześniej wzór na rozwiązanie zlinearyzowanego równania (4.5). Wzór ten umieszczony zostanie w pętli *while*, ponieważ przybliżanie równania nieliniowego wykonywane będzie tak długo, aż różnica pomiędzy otrzymanym rozwiązaniem przybliżonym i dokładnym będzie większa od parametru *eps*. Pętla *while* różni się od pętli *for* tylko tym, że liczba jej powtórzeń nie jest z góry określona. Kod w niej zawarty wykonywany jest tak długo, aż wyrażenie logiczne zawarte po słowie *while* ma wartość *True*. W programie użyta będzie też przygotowana wcześniej funkcja, która wyznacza wartość pochodnej centralnej dowolnej funkcji.

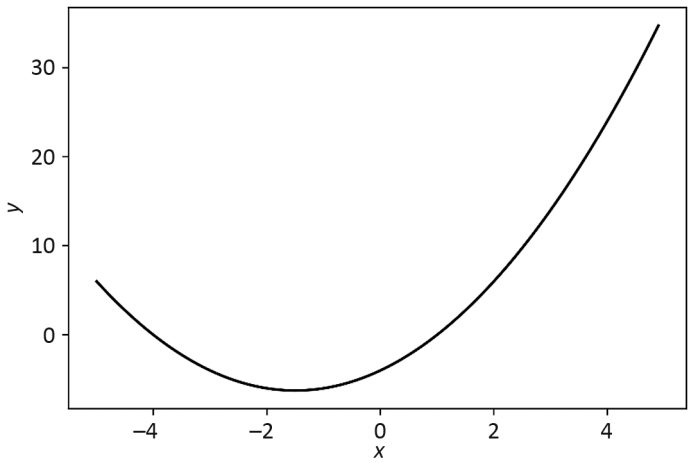
Na początku należy zaktualizować w programie funkcję $f(x)$:

```
def f(x):
    #funkcja zwraca wartosc funkcji kwadratowej
    #x^2 + 3.0 * x - 4.0 dla podanego argumentu x

    return x**2 + 3.0 * x - 4.0
```

Dla lepszej wizualizacji zadania można także wygenerować wykres badanej funkcji – rys. 4.3. Kod do generowania wykresu znajduje się poniżej.

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

x_min = -5.0
x_max = 5.0

delta_x = 0.1
x_range = x_max - x_min
liczba_krokow = int(x_range / delta_x)
```

```
x_cur = x_min
f_macierz = np.zeros(liczba_krokow)
x_macierz = np.zeros(liczba_krokow)

for i in range(liczba_krokow):
    f_macierz[i] = f(x_cur)
    x_macierz[i] = x_cur

    x_cur = x_cur + delta_x

plt.plot(x_macierz, f_macierz)

# podpisy pod osiami
plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



Rys. 4.3. Badana funkcja

Jako rozwiązanie początkowe do metody przyjęte zostanie $x_0 = 0{,}0$. W tym przypadku procedura numeryczna została zaimplementowana jak poniżej.

```
#Skrypt rozwiazuje numerycznie rownanie nieliniowe
x0 = 0.0          #rozwiazanie poczatkowe [podaje uzytkownik]
eps = 0.001       #dokladnosc rozwiazania
delta_x = 0.001   #przyrost

x_cur = x0        #aktualne rozwiazanie
```

```
# [jego wartosc zmienia sie w petli; na początku jest rowne x0]

i = 0             #licznik iteracji

#Petla, w ktorej linearyzowanie jest rownanie f(x) = 0.0
while (abs(f(x_cur)) > eps):
    # a) wyznaczenie pochodnej funkcji f(x) dla argumentu x_cur:
    #    [f(x) zawiera juz badana funkcje]
    f_prim_cur = wyznacz_pochodna_centralna(f, x_cur, delta_x)

    # b) wyznaczenie rozwiazania zlinearyzowanego
    # rownania nieliniowego:

    x_new = (f_prim_cur * x_cur - f(x_cur)) / f_prim_cur

    # c) aktualizacja wartosci x_cur:
    x_cur = x_new

    # d) wydruk wynikow posrednich:
    print "[", i, "] ", "rozw. aktualne: ", x_cur
    print "       f(x_cur)  = ", f(x_cur)

    # e) aktualizacja wartosci licznika iteracji:
    i = i + 1

print "Rozwiazanie koncowe to: ", x_cur
```

```
[ 0 ]  rozw. aktualne:  1.33333333333
       f(x_cur)  =  1.77777777778
[ 1 ]  rozw. aktualne:  0.740740740741
       f(x_cur)  =  -1.22908093278
[ 2 ]  rozw. aktualne:  1.150434385
       f(x_cur)  =  0.774802429203
[ 3 ]  rozw. aktualne:  0.892166908601
       f(x_cur)  =  -0.527537481392
...
[ 17 ]  rozw. aktualne:  0.999643972771
       f(x_cur)  =  -0.0017800093897
[ 18 ]  rozw. aktualne:  1.00023730923
       f(x_cur)  =  0.00118660248675
[ 19 ]  rozw. aktualne:  0.999841775072
       f(x_cur)  =  -0.000791099605048
-------------------------------
Rozwiazanie koncowe to:  0.999841775072
```

Rozwiązanie, z założoną dokładnością, otrzymane zostało już po 19 iteracjach. Jest ono zgodne z rozwiązaniem analitycznym do piątego miejsca po przecinku. Można oczywiście otrzymać jeszcze dokładniejsze rozwiązanie, modyfikując parametr *eps* w programie. Warto dodać teraz do programu wizualizację *f*(*x_cur*) w kolejnych iteracjach. Można to zrobić, wykorzystując funkcję *plot* z biblioteki *Matplotlib*.

Liczba iteracji potrzebnych do rozwiązania zadania nie jest z góry określona, dlatego też poszczególne wyniki wygodnie jest zapisać początkowo w liście.

```
#Skrypt rozwiazuje numerycznie rownanie nieliniowe
#wykorzystujac metode Newtona-Raphsona
import numpy as np
import matplotlib.pyplot as plt

#Dane:
x0 = 0.0           #rozwiazanie poczatkowe [podaje uzytkownik]
eps = 0.00001      #dokladnosc rozwiazania
delta_x = 0.001    #przyrost

x_cur = x0         #aktualne rozwiazanie

i = 0              #licznik iteracji

lista_f_x_cur = [] #lista, w ktorej beda umieszczane
                   #wyniki poszczegolnych iteracji
lista_f_x_cur.append(abs(f(x_cur)))

#Petla, w ktorej linearyzowanie jest rownanie f(x) = 0.0
while (abs(f(x_cur)) > eps):
    # a) wyznaczenie pochodnej funkcji f(x) dla argumentu x_cur:
    #    [f(x) zawiera juz badana funkcje]
    f_prim_cur = wyznacz_pochodna_centralna(f, x_cur, delta_x)

    # b) wyznaczenie rozwiazania zlinearyzowanego
    # rownania nieliniowego:
    x_new = (f_prim_cur * x_cur - f(x_cur)) / f_prim_cur

    # c) aktualizacja wartosci x_cur:
    x_cur = x_new

    # d) aktualizacja wartosci licznika iteracji:
    i = i + 1

    # e) zapis wyniku iteracji:
    lista_f_x_cur.append(abs(f(x_cur)))
```

```
#Wydruk koncowy
print "[", i, "] ", "rozw. koncowe: ", x_cur, "
print "f(x_cur)  = ", f(x_cur)

#Wydruk wykresu
plt.plot(range(i + 1), lista_f_x_cur)
plt.xlabel("iteracja")
plt.ylabel("f(x_cur)")
plt.show()
```

```
[ 31 ]  rozw. koncowe:  1.00000182915
f(x_cur)  =  9.14577184119e-06
```

Po zmianie parametru *eps* na 0,00001 program potrzebował 31 iteracji, żeby odszukać rozwiązanie z założoną dokładnością. Dodatkowo, przygotowany wykres pokazuje, że zmiany wartości funkcji dla aktualnego rozwiązania są silnie nieliniowe – największe poprawy rozwiązania otrzymywane są w początkowych iteracjach.

Można teraz rozbudować skrypt, przygotowując złożoną funkcję *rozwiaz_rownanie*, ktora będzie zawierała kod przygotowany w poprzednim skrypcie. Argumentami *rozwiaz_rownanie* będą:

- równanie do rozwiązania (tzn. funkcja $f(x)$),
- rozwiązanie początkowe x0 (podawane przez użytkownika),
- założona dokładność rozwiązania *eps* (dokładność będzie parametrem z zadaną wartością domyślną 0,0001),
- założony przyrost do numerycznego wyznaczania pochodnych *delta_x* (przyrost będzie parametrem z zadaną wartością domyślną 0,0001),
- zmienna logiczna *drukuj* z wartością domyślną *False*; jeżeli *drukuj* będzie miało wartość *True*, to program wydrukuje wszystkie wyniki pośrednie,
- zmienna *max_iter*, która określa maksymalną liczbę iteracji w pętli *while*.

Zmienna *max_iter* oraz dodatkowy warunek w pętli *while* są niezbędne. Pomimo tego, że procedura na wcześniejszych przykładach działała bardzo dobrze, istnieją problemy, których metoda Newtona-Raphsona w tej wersji nie rozwiąże. Zmienna *max_iter* to zabezpiecznie, które nie pozwoli, żeby program wpadł w pętlę nieskończoną.

Funkcja będzie zwracała przybliżone rozwiązanie równania oraz, opcjonalnie, drukowała wykres wyników z poszczególnych iteracji.

```
def rozwiaz_rownanie(f, x0, eps = 0.0001,
                delta_x = 0.0001,
                max_iter = 1000,
                drukuj = False):

    x_cur = x0         #aktualne rozwiazanie
```

```
    i = 0              #licznik iteracji

    #Petla, w ktorej linearyzowanie jest rownanie f(x) = 0.0
    while (abs(f(x_cur)) > eps and i < max_iter):
        # a) wydruk wynikow posrednich:
        if drukuj:
            print "[", i, "] ", „rozw. aktualne: ", x_cur
            print "f(x_cur)  = ", f(x_cur)

        # b) wyznaczenie pochodnej funkcji f(x) dla argumentu x_cur:
        #    [f(x) zawiera badana funkcje]
        f_prim_cur = wyznacz_pochodna_centralna(f, x_cur, delta_x)

        # c) wyznaczenie rozwiazania zlinearyzowanego
        # rownania nieliniowego:
        x_new = (f_prim_cur * x_cur - f(x_cur)) / f_prim_cur

        # d) aktualizacja wartosci x_cur:
        x_cur = x_new

        # e) aktualizacja wartosci licznika iteracji:
        i = i + 1

    # Wydruk koncowy
    if drukuj:
        print "--------------------------------"
        print "Rozwiazanie koncowe to: ", x_cur
        print "otrzymane zostalo w iteracji ", i

    return x_cur
```

Można teraz przetestować przygotowaną funkcję na rozwiązanym wcześniej przykładzie.

```
#Dane:
x0 = 0.0           #rozwiazanie poczatkowe [podaje uzytkownik]
eps = 0.00001      #dokladnosc rozwiazania
delta_x = 0.001    #przyrost [uzywany do numerycznego wyznaczania
pochodnej funkcji]
max_iter = 100     #maksymalna liczba iteracji

x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter)

print "Rozwiazanie koncowe to: ", x
```

Rozwiazanie koncowe to: 1.00000182915

Otrzymane rozwiązanie jest poprawne. Warto zauważyć, że do tej pory program zwracał tylko drugie z możliwych rozwiązań $x_2 = -1{,}0$. Nie jest to błąd, tylko ograniczenie procedury. Procedura zawsze zwraca jedno rozwiązanie równania. Jest nim to, do którego najłatwiej dotrzeć z rozwiązania startowego x_0. Warto sprawdzić, co stanie się po zmianie rozwiązania początkowego na –10,0.

```
#Dane:
x0 = -10.0          #rozwiazanie poczatkowe [podaje uzytkownik]
eps = 0.00001       #dokladnosc rozwiazania
delta_x = 0.001     #przyrost
max_iter = 1000

x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter, False)

print "Wartosc funkcji f(x) dla argumentu x0: ", f(x0)
print "Rozwiazanie koncowe to: ", x,
print "Wartosc funkcji f(x) wynosi: ", f(x)
```

```
Wartosc funkcji f(x) dla argumentu x0:  66.0
Rozwiazanie koncowe to:  -4.00000160471
Wartosc funkcji f(x) wynosi:  8.02356062124e-06
```

Zmiana rozwiązania początkowego x_0 z 0,0 na –10,0 spowodowała, że program odszukał pierwsze rozwiązanie x_1 badanego równania.

Do tej pory program został przetestowany tylko dla rozwiązań początkowych bliskich rzeczywistym rozwiązaniom równania. Warto zbadać, co się stanie, jeżeli za rozwiązanie początkowe będzie bliskie wierzchołkowi paraboli $x_0 = -2{,}3$.

```
#Dane:
x0 = -2.3           #rozwiazanie poczatkowe [podaje uzytkownik]
eps = 0.00001       #dokladnosc rozwiazania
delta_x = 0.001     #przyrost
max_iter = 100

x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter, True)
```

```
[ 0 ]  rozw. aktualne:  -2.3
       f(x_cur)  =  -5.61
[ 1 ]  rozw. aktualne:  -5.80625
       f(x_cur)  =  12.2937890625
[ 2 ]  rozw. aktualne:  1.87736816406
       f(x_cur)  =  5.1566157156
[ 3 ]  rozw. aktualne:  5.10025298631
       f(x_cur)  =  8.3284494649e+20
...
```

```
[ 8 ]  rozw. aktualne:  5.20528091585e+20
       f(x_cur)  =  2.70949494129e+41
[ 9 ]  rozw. aktualne:  1.69343433831e+41
       f(x_cur)  =  2.86771985816e+82
[ 10 ]  rozw. aktualne:  1.79232491135e+82
       f(x_cur)  =  3.21242858784e+164
---------------------------------------------------------------
OverflowError
Traceback (most recent call last)
<ipython-input-23-cae6a0d662cb> in <module>()
      7 max_iter = 100
      8
----> 9 x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter, True)
     10
     11 print "Wartosc funkcji f(x) dla argumentu x0: ", f(x0)

<ipython-input-22-49ed4ebbd172> in rozwiaz_rownanie(f, x0, eps,
delta_x, max_iter, drukuj)
      4
      5   #Petla, w ktorej linearyzowanie jest rownanie f(x) = 0.0
----> 6    while (abs(f(x_cur)) > eps and i < max_iter):
      7        # a) wydruk wynikow posrednich:
      8        if drukuj:

<ipython-input-10-a6fee71e2e1c> in f(x)
      2    #funkcja zwraca wartosc funkcji kwadratowej x^2 + 3.0 *
x - 4.0 dla podanego argumentu x
      3
----> 4    return x**2 + 3.0 * x - 4.0

OverflowError: (34, 'Result too large')
```

Program nie jest w stanie znaleźć rozwiązania. Procedura działa poprawnie, jednakże pochodna w rejonie wierzchołka paraboli ma bardzo małą wartość i rozwiązanie aktualne jest dalekie od początkowego. To sprawia, że numeryczne rozwiązanie zaczyna oscylować wokół rozwiązania dokładnego z coraz większą amplitudą. Wartości funkcji *f*(*x_cur*) szybko zmierzają do nieskończoności i następuje przepełnienie zmiennej, która tę wartość przechowuje. Python zwraca wtedy błąd przepełnienia: *Result too large*.

Procedura w obecnej wersji działa dobrze tylko wtedy, gdy podane rozwiązanie początkowe jest blisko rozwiązania dokładnego. Zbieżność procedury jest lokalna. W literaturze przedstawiono bardzo wiele modyfikacji podstawowej metody Newtona-Raphsona, między innymi w zakresie zbieżności globalnej. Jedną z nich jest dodatkowy warunek narzucony na kolejne rozwiązanie przybliżonego równania

zlinearyzowanego [11]. W tym przypadku rozwiązanie to przyjmuje się tylko, gdy nie jest ono gorsze od poprzedniego, czyli wartość bezwzględna z wartości funkcji dla poprzedniego rozwiązania jest większa od wartości bezwzględnej z wartości funkcji dla aktualnego rozwiązania. W przeciwnym przypadku w taki sam sposób sprawdzane jest rozwiązanie dokładnie pomiędzy aktualnym a poprzednim. Ten proces powtarza się tak długo, aż aktualne rozwiązanie spełni warunek o niepogarszaniu wartości funkcji. Tę modyfikację można łatwo zrealizować, wykorzystując zagnieżdżoną pętlę *while*. Poniżej znajduje się zmodyfikowana wersja funkcji *rozwiaz_rownanie* oraz jej wywołanie.

```
def rozwiaz_rownanie(f, x0, eps = 0.0001,
               delta_x = 0.0001,
               max_iter = 1000,
               drukuj = False):

    x_cur = x0         #aktualne rozwiazanie
    i = 0              #licznik iteracji

    #Petla, w ktorej linearyzowanie jest rownanie f(x) = 0.0
    while (abs(f(x_cur)) > eps and i < max_iter):
        # a) wydruk wynikow posrednich:
        if drukuj:
            print "[", i, "] ", „rozw. aktualne: ", x_cur
            print "f(x_cur)  = ", f(x_cur)

        # b) wyznaczenie pochodnej funkcji f(x) dla argumentu x_cur:
        #    [f(x) zawiera juz nasza nowa funkcje]
        f_prim_cur = wyznacz_pochodna_centralna(f, x_cur, delta_x)

        # c) wyznaczenie rozwiazania zlinearyzowanego
        # rownania nieliniowego:
        x_new = (f_prim_cur * x_cur - f(x_cur)) / f_prim_cur

        # -- modyfikacja --
        # d) aktualizacja wartosci x_cur:
        while (abs(f(x_cur)) < abs(f(x_new))):
            x_new = (x_cur + x_new) / 2.0

        x_cur = x_new
        # e) aktualizacja wartosci licznika iteracji:
        i = i + 1

    # Wydruk koncowy
    if drukuj:
```

```
        print "-------------------------------"
        print "Rozwiazanie koncowe to: ", x_cur
        print "otrzymane zostalo w iteracji ", i

    return x_cur
```

```
Dane:
x0 = -2.3           #rozwiazanie poczatkowe [podaje uzytkownik]
eps = 0.00001       #dokladnosc rozwiazania
delta_x = 0.001     #przyrost
max_iter = 100

x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter, True)

print "Wartosc funkcji f(x) dla argumentu x0: ", f(x0)
print "Rozwiazanie koncowe to: ", x,
print "Wartosc funkcji f(x) wynosi: ", f(x)
```

```
[ 0 ]  rozw. aktualne:  -2.3
       f(x_cur)  =  -5.61
[ 1 ]  rozw. aktualne:  -4.053125
       f(x_cur)  =  0.268447265624
[ 2 ]  rozw. aktualne:  -3.96923522949
       f(x_cur)  =  -0.152877381434
[ 3 ]  rozw. aktualne:  -4.01700941119
       f(x_cur)  =  0.0853363760211
...
[ 15 ]  rozw. aktualne:  -4.00001716859
       f(x_cur)  =  8.58432267332e-05
[ 16 ]  rozw. aktualne:  -3.99999034258
       f(x_cur)  =  -4.82870165328e-05
[ 17 ]  rozw. aktualne:  -4.00000543227
       f(x_cur)  =  2.71613830396e-05
[ 18 ]  rozw. aktualne:  -3.99999694434
       f(x_cur)  =  -1.52782981324e-05
-------------------------------
Rozwiazanie koncowe to:  -4.00000171881
Otrzymane zostalo w iteracji  19
Wartosc funkcji f(x) dla argumentu x0:  -5.61
Rozwiazanie koncowe to:  -4.00000171881
Wartosc funkcji f(x) wynosi:  8.59403631637e-06
```

Wydrukowane na ekran komputera wyniki potwierdzają, że zmodyfikowana procedura bez trudu radzi sobie z testowym przykładem. Pozostało już tylko zoptymali-

zować kod. Warto zauważyć, że wewnątrz programu wielokrotnie wyznaczamy wartość funkcji *f*(*x_cur*) – to duża strata zasobów komputera. Wielokrotnie wyznaczaną wartość lepiej wyznaczyć raz i zapisać w zmiennej.

```
def rozwiaz_rownanie(f, x0, eps = 0.0001,
                delta_x = 0.0001,
                max_iter = 1000,
                drukuj = False):

    x_cur = x0          #aktualne rozwiazanie
    i = 0               #licznik iteracji

    f_x_cur_abs = abs(f(x_cur))

    #Petla, w ktorej linearyzowanie jest rownanie f(x) = 0.0
    while (f_x_cur_abs > eps and i < max_iter):
        # a) wydruk wynikow posrednich:
        if drukuj:
            print "[", i, "] ", "rozw. aktualne: ", x_cur
      print "f(x_cur)  = ", f(x_cur)

       # b) wyznaczenie pochodnej funkcji f(x) dla argumentu x_cur:
       #    [f(x) zawiera juz nowa funkcje]
       f_prim_cur = wyznacz_pochodna_centralna(f, x_cur, delta_x)

       # -- modyfikacja --
       if f_prim_cur == 0:
           f_prim_cur = 0.00001

       # c) wyznaczenie rozwiazania zlinearyzowanego
       #rownania nieliniowego:
       x_new = (f_prim_cur * x_cur - f(x_cur)) / f_prim_cur

       # -- modyfikacja --
       # d) aktualizacja wartosci x_cur:
       while (f_x_cur_abs < abs(f(x_new))):
           x_new = (x_cur + x_new) / 2.0

       x_cur = x_new
       f_x_cur_abs = abs(f(x_cur))

       # e) aktualizacja wartosci licznika iteracji:
       i = i + 1
```

```
    # Wydruk koncowy
    if drukuj:
        print "--------------------------------"
        print "Rozwiazanie koncowe to: ", x_cur
        print "otrzymane zostalo w iteracji ", i

    return x_cur
```

Teraz można wykonać końcowy test procedury. Testy modyfikowanych programów na każdym etapie rozbudowy są bardzo ważne. W przypadku dużych systemów, przygotowywanych przez wielu programistów, stosuje się dużo bardziej złożone metodologie testowania. Zainteresowani czytelnicy więcej informacji mogą znaleźć, poszukując materiałów dotyczących testów jednostkowych.

```
#Dane:
x0 = -2.3          #rozwiazanie poczatkowe [podaje uzytkownik]
eps = 0.00001      #dokladnosc rozwiazania
delta_x = 0.001    #przyrost
max_iter = 100

x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter, True)

print "Wartosc funkcji f(x) dla argumentu x0: ", f(x0)
print "Rozwiazanie koncowe to: ", x, ",
print "Wartosc funkcji f(x) wynosi: ", f(x)
```

```
[ 0 ]  rozw. aktualne:  -2.3
       f(x_cur)  =  -5.61
[ 1 ]  rozw. aktualne:  -4.053125
       f(x_cur)  =  0.268447265624
[ 2 ]  rozw. aktualne:  -3.96923522949
       f(x_cur)  =  -0.152877381434
[ 3 ]  rozw. aktualne:  -4.01700941119
       f(x_cur)  =  0.0853363760211
...
[ 15 ]  rozw. aktualne:  -4.00001716859
       f(x_cur)  =  8.58432267332e-05
[ 16 ]  rozw. aktualne:  -3.99999034258
       f(x_cur)  =  -4.82870165328e-05
[ 17 ]  rozw. aktualne:  -4.00000543227
       f(x_cur)  =  2.71613830396e-05
[ 18 ]  rozw. aktualne:  -3.99999694434
       f(x_cur)  =  -1.52782981324e-05
--------------------------------
Rozwiazanie koncowe to:  -4.00000171881
```

```
Otrzymane zostalo w iteracji  19
Wartosc funkcji f(x) dla argumentu x0:  -5.61
Rozwiazanie koncowe to:  -4.00000171881
Wartosc funkcji f(x) wynosi:  8.59403631637e-06
```

Procedura działa poprawnie.

W poprzednich skryptach zaimplementowana została metoda Newtona-Raphsona. Przygotowany program przestestowano na kilku ogólnych przykładach obliczeniowych. Celem skryptu jest jednak przedstawienie zagadnień związanych z modelowaniem układów biomechanicznych. Dlatego na tym etapie warto zastosować przygotowane programy do statyki układów mechanicznych. Pierwsze próby zostaną wykonane na prostym układzie – ciało, do którego dołączona jest sprężyna liniowa.

Ten przykład rozwiązany został już w zakresie dynamiki w poprzednim rozdziale – wzory matematyczne, pozwalające wyznaczyć siłę od sprężyny oraz sumę sił, pozostają bez zmian. Niemniej jednak w tej wersji program będzie przygotowany z wykorzystaniem funkcji, które ułatwią dalszą jego rozbudowę i utrzymanie.

W pierwszym kroku należy przygotować funkcję, która dla zadanych parametrów sprężyny – współrzędne przyczepów, współczynnik sztywności oraz długość swobodna – wyznaczy siłę, z jaką sprężyna działa na ciało dołączone do jej ruchomego końca.

```
def wyznacz_sile_od_spr(a, b, k, lswob):
    l_cur = a - b             #aktualna dlugosc sprezyny
    delta_l = l_cur – lswob   #wydluzenie sprezyny
    F_spr = -k * delta_l      #sila od sprezyny

    return F_spr
```

Warto zauważyć, że w przypadku statycznym równanie równowagi do rozwiązania określa po prostu sumę sił. Jeżeli suma sił działających na ciało jest równa 0, to ciało jest w stanie równowagi. Zmienną w tym zadaniu jest położenie ciała. Rozwiązaniem będzie położenie, w którym suma sił jest równa 0. Mając to na uwadze, można zaimplementować nową funkcję $f(x)$, tym razem nie jako funkcję kwadratową, a sumę sił:

```
def f(x):
    #funkcja, ktora, dla zadanego polozenia ciala x wyznacza,
    #sume sil dzialajacych na cialo w tym polozeniu

    #uwaga: obciazenie zewnetrzne Fext i parametry sprezyny
    #       podawane sa jako zmienne o zasiegu globalnym
```

```
    silaOdSpr = wyznacz_sile_od_spr(x, b, k, lswob)
    sumaSil = Fextx + silaOdSpr

    return sumaSil
```

Program główny warto rozpocząć od wprowadzenia danych układu.

```
#Dane ukladu:
x0 = 10.0     #[m] - poczatkowe polozenie ciala
              #      i ruchomego przyczepu sprezyny
              #      //rozwiazanie podane przez uzytkownika//

b = -1.0      #[m] - polozenie nieruchomego przyczepu sprezyny
k = 10.0      #[N/m] - wspolczynnik sztywnosci sprezyny
l_swob = 2.0  #[m] - dlugosc swobodna sprezyny
Fextx = 10.0  #[N] - sila zewnetrzna, ktora dziala na cialo
```

Teraz można wykorzystać przygotowaną wcześniej funkcję do rozwiązywania numerycznego rozwiązywania równań algebraicznych.

```
#Ustawienia procedury numerycznej:
eps = 0.00001       #dokladnosc rozwiazania
delta_x = 0.001     #przyrost
max_iter = 100

print "Wartosc funkcji f(x) dla argumentu x0: ", f(x0)
x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter, True)
```

```
Wartosc funkcji f(x) dla argumentu x0:  -80.0
[ 0 ]  rozw. aktualne:  10.0  | |f(x_cur)|  =  80.0
--------------------------------
Rozwiazanie koncowe to:  1.99999999999
Otrzymane zostalo w iteracji  1
```

Program w tym przypadku zwrócił poprawne rozwiązanie $x = 1{,}0$ już po pierwszej iteracji. Wynika to z przyjętego sposobu wyznaczania siły od sprężyny. Sprężyna jest liniowa. Powstałe równanie na sumę sił także jest liniowe. Już pierwsze przybliżenie tego równania prowadzi do dokładnego wyniku. W rzeczywistości problemy statyczne w zakresie modelowania stawów człowieka są dużo bardziej złożone – więzadła mają nieliniowe charakterystyki siła–wydłużenie. Co więcej, więzadła zachowują się jak cięgna, to znaczy, że generują siłę tylko wtedy, gdy są rozciągane, a przy ściskaniu pozostają swobodne.

Na podstawie tego uproszczonego skryptu można teraz zaimplementować układ z bardziej złożonym modelem więzadła. W rzeczywistości charakterystyka więza-

dła, czyli siła, jaką generuje do wydłużenia, jest silnie nieliniowa [12–14], a model liniowy znacznie upraszcza jego zachowanie. Dlatego też w kolejnym przykładzie do uwzględnienia nieliniowości wykorzystana zostanie funkcja kwadratowa. Jest to jedno z najczęściej stosowanych podejść w modelowaniu więzadeł i stawów człowieka. Podstawowa zmiana nastąpi w równaniu do wyznaczenia wartości siły [14]:

$$F_s = k(\Delta l)^2. \tag{4.14}$$

Dodatkowo, w nowej funkcji uwzględniony zostanie fakt, że więzadło generuje siłę tylko wtedy, gdy jest rozciągane, czyli działa jak cięgno. Ten warunek zaimplementowany będzie z wykorzystaniem instrukcji warunkowej *if/else*.

```
def wyznacz_sile_od_wiez(a, b, k, l_swob):
    l_cur = a - b            #aktualna dlugosc wiezadla
    delta_l = l_cur – l_swob  #wydluzenie wiezadla

    #sila od wiezadla
    if delta_l > 0.0:
        F_wiez = -k * delta_l**2
    else:
        F_wiez = 0.0

    return F_wiez
```

Zmianę modelu elementu podatnego należy uwzględnić także w funkcji $f(x)$, która zwraca sumę sił działających na ciało.

```
def f(x):
    #funkcja, ktora, dla zadanego polozenia ciala x,
    #wyznacza sume sil dzialajacych na cialo w tym polozeniu

    #uwaga: obciazenie zewnetrzne Fext i parametry sprezyny
    #       podawane sa jako zmienne o zasiegu globalnym

    silaOdSpr = wyznacz_sile_od_wiez(x, b, k, l_swob)
    sumaSil = Fextx + silaOdSpr

    return sumaSil
```

Dane pozostaną na razie bez zmian, ale należy pamiętać o tym, że współczynnik sztywności k dla nieliniowej sprężyny wyrażony jest w [N/m^2], a nie [N/m] – jest to inna wielkość fizyczna.

Rozwiązanie zadania statyki dla układu z nieliniowym modelem więzadła zostało przedstawione poniżej.

```
x0 = 7.1      #[m] - poczatkowe polozenie ciala

print "Wartosc funkcji f(x) dla argumentu x0: ", f(x0)
x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter, True)
```

```
Wartosc funkcji f(x) dla argumentu x0:  -362.1
[ 0 ]  rozw. aktualne:  7.1
       f(x_cur)  =  -362.1
[ 1 ]  rozw. aktualne:  4.13196721311
       f(x_cur)  =  -88.0921862401
[ 2 ]  rozw. aktualne:  3.40990011278
       f(x_cur)  =  -48.0761855359
...
[ 30 ]  rozw. aktualne:  2.00494616121
       f(x_cur)  =  -0.0991678693901
[ 31 ]  rozw. aktualne:  2.00413330983
       f(x_cur)  =  -0.0828370390148
[ 32 ]  rozw. aktualne:  2.0034543177
       f(x_cur)  =  -0.00160539275995
...
[ 65 ]  rozw. aktualne:  2.00000936267
       f(x_cur)  =  -0.000187254206192
[ 66 ]  rozw. aktualne:  2.0000078278
       f(x_cur)  =  -0.000156556531488
...
[ 79 ]  rozw. aktualne:  2.00000076342
       f(x_cur)  =  -1.52683552574e-05
[ 80 ]  rozw. aktualne:  2.00000063827
       f(x_cur)  =  -1.27653444419e-05
[ 81 ]  rozw. aktualne:  2.00000053363
       f(x_cur)  =  -1.06726637981e-05
------------------------------
Rozwiazanie koncowe to:  2.00000044615
Otrzymane zostalo w iteracji  82
```

Warto sprawdzić, co się stanie, jeżeli rozwiązanie początkowe zostanie dobrane tak, że układ będzie w nim nienapięty – wydłużenie więzadła poniżej zera. Takim położeniem jest na przykład $x_0 = -3{,}0$.

```
x0 = -3.0      #[m] //wiezadlo w tym polozeniu nie generuje sily//

print "Wartosc funkcji f(x) dla argumentu x0: ", f(x0)
x = rozwiaz_rownanie(f, x0, eps, delta_x, max_iter, True)
```

```
Wartosc funkcji f(x) dla argumentu x0:  10.0
[ 0 ]  rozw. aktualne:  -3.0
```

```
       f(x_cur)  =  10.0
[ 1 ]  rozw. aktualne:  -1000003.0
       f(x_cur)  =  10.0
[ 2 ]  rozw. aktualne:  -2000003.0
       f(x_cur)  =  10.0
[ 3 ]  rozw. aktualne:  -3000003.0
       f(x_cur)  =  10.0
...
[ 97 ]  rozw. aktualne:  -97000003.0
       f(x_cur)  =  10.0
[ 98 ]  rozw. aktualne:  -98000003.0
       f(x_cur)  =  10.0
[ 99 ]  rozw. aktualne:  -99000003.0
       f(x_cur)  =  10.0
------------------------------
Rozwiazanie koncowe to:  -100000003.0
Otrzymane zostalo w iteracji  100
```

Wyniki są niespodziewane. Wiadomo już, że przyjęte zadanie ma rozwiązanie, ponieważ rozwiązanie to zostało uzyskane w poprzedniej próbie. Jednakże dla $x_0 = -3{,}0$ program nie wykazuje zbieżności. Problem można przeanalizować dokładniej, drukując wartość sumy sił – $f(x)$ – dla wielu położeń ciała x. Kod do generowania wykresu funkcji przedstawiono już wcześniej – wystarczy go nieznacznie zmodyfikować.

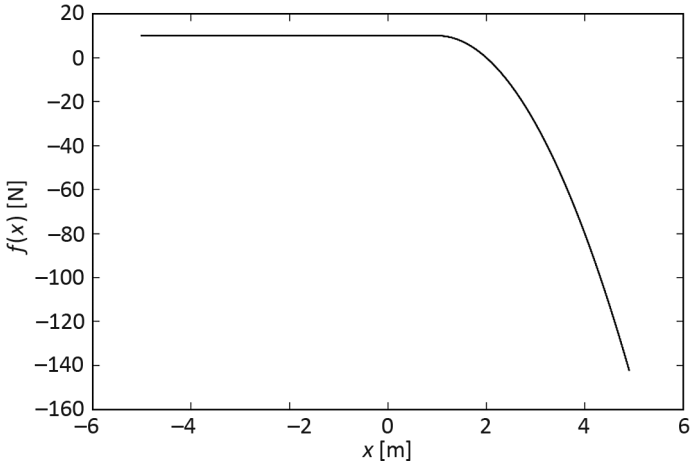
Analiza uzyskanego wykresu (rys. 4.4) w pełni tłumaczy zaobserwowane zjawisko. Z uwagi na to, że więzadło działa jak cięgno, w przedziale $x \in (-\infty;\ 1{,}5\ \text{m})$ suma sił – $f(x)$ – jest funkcją stałą. Jej pochodna ma wtedy wartość zero. Jeżeli procedura zostanie wystartowana dla dowolnego x_0 z tego zakresu, to nigdy nie trafi na lepsze rozwiązanie, co wynika bezpośredniego z przepisu na kolejne rozwiązanie:

$$x_{\text{new}} = \frac{f'(x_{\text{cur}})x_{\text{cur}} - f(x_{\text{cur}})}{f'(x_{\text{cur}})}. \tag{4.15}$$

W tym przypadku iloczyn $f'(x_{\text{cur}})x_{\text{cur}}$ ma bardzo małą wartość. Składnikiem, który determinuje nowe rozwiązanie, jest $f(x_{\text{cur}})$. Z uwagi na minus, nowe rozwiązanie będzie zawsze bliżej $-\infty$ niż poprzednie, czyli program nigdy nie dojdzie do rozwiązań w zakresie $(1{,}5;\ \infty)$.

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

x_min = -5.0
x_max = 5.0
```

```
delta_x = 0.1
x_range = x_max - x_min
liczba_krokow = int(x_range / delta_x)
x_cur = x_min
f_macierz = np.zeros(liczba_krokow)
x_macierz = np.zeros(liczba_krokow)

for i in range(liczba_krokow):
    f_macierz[i] = f(x_cur)
    x_macierz[i] = x_cur

    x_cur = x_cur + delta_x

plt.plot(x_macierz, f_macierz)
plt.xlabel("x [m]")
plt.ylabel("f(x) [N]")

plt.show()
```



Rys. 4.4. Wartość sumy sił – $f(x)$ – dla wielu położeń ciała x

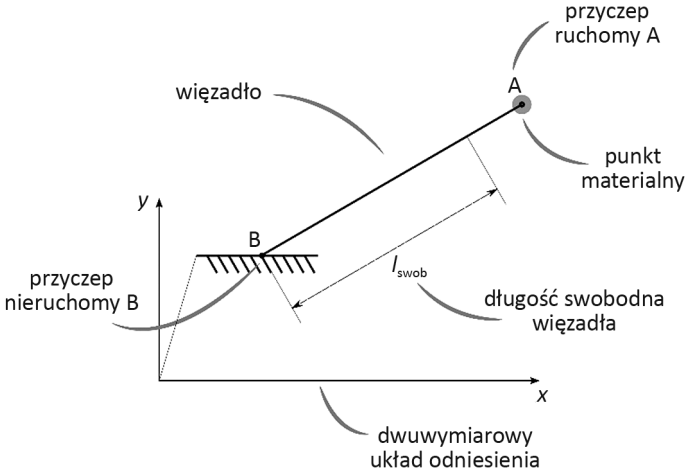
Warto zaznaczyć, że nie jest to błąd wynikający z nieprawidłowej implementacji procedury. Niektóre problemy mechaniczne w zakresie statyki są bardzo wrażliwe na wybrane rozwiązanie początkowe. Sytuacje, w których program nie zwraca rozwiązania problemu, są bardzo częste. W literaturze najczęściej spotykanym sposobem radzenia sobie z tym utrudnieniem jest wielokrotny restart procedury numerycznej dla różnych rozwiązań początkowych [9].

5. DWUWYMIAROWE MODELOWANIE WIĘZADEŁ STAWÓW CZŁOWIEKA

W poprzednich rozdziałach zostały wprowadzone zostały jednowymiarowe modele więzadeł – liniowe oraz nieliniowe cięgno. Jednowymiarowe elementy podatne reprezentują jednak przypadek szczególny, niezbyt użyteczny w zaawansowanych problemach biomechaniki. Dlatego też rozdział ten poświęcony będzie rozbudowie przedstawionych modeli do przypadku dwuwymiarowego.

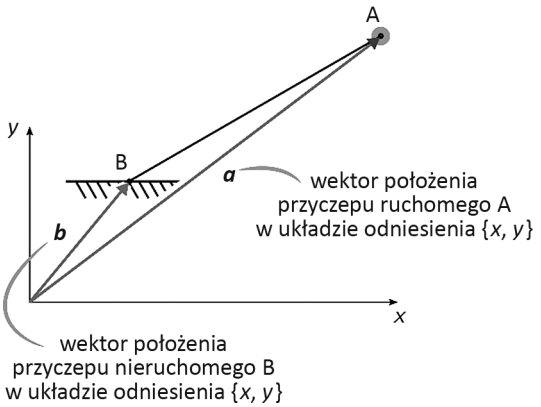
Podstawowym zadaniem podczas rozwiązywania modeli stawów jest wyznaczenie siły, z jaką więzadło oddziałuje na kości, które łączy. Pewnym uproszczeniem tego zadania jest przypadek, w którym dwuwymiarowe więzadło działa na punkt materialny (rys. 5.1 oraz 5.2).

Położenie przyczepu B więzadła jest zdeterminowane przez wektor położenia $\boldsymbol{b}$ wyrażony w układzie $\{x, y\}$ – patrz: rys. 5.2. Niezależnie od parametrów symulacji będzie on stały, ponieważ przyczep B nigdy nie zmieni swojego położenia względem układu odniesienia $\{x, y\}$. Przyczep ruchomy A doczepiony jest do punktu materialnego, który może zmieniać swoje położenie względem $\{x, y\}$. Warto zauważyć, że z uwagi na możliwość zmiany położenia punktu materialnego wektor położenia $\boldsymbol{a}$ jest zmienny. Od tego wektora zależy, z jaką siłą więzadło działa na



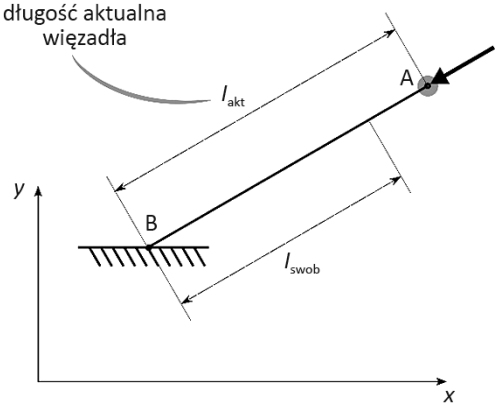
Rys. 5.1. Dwuwymiarowe więzadło doczepione do punktu materialnego

punkt. Rozważane więzadło opisuje także współczynnik sztywności k oraz długość swobodna l_{swob}.



Rys. 5.2. Dwuwymiarowe więzadło doczepione do punktu materialnego

Dla wektora ***a***, przyjętego jak na rys. 5.1 oraz 5.2, długość swobodna więzadła jest mniejsza od jego długości aktualnej. Oznacza to, że w takiej konfiguracji więzadło jest napięte i działa siłą na punkt (rys. 5.3).



Rys. 5.3. Dwuwymiarowe więzadło oraz siła, z jaką działa na punkt materialny

Więzadła oraz inne elementy podatne w modelach wieloczłonowych przyjmuje się najczęściej za bezmasowe. Przy takim założeniu kierunek siły generowanej przez więzadło jest zgodny z kierunkiem więzadła wyznaczonym przez wektor ***b*** – ***a***. Co więcej, napięte więzadło zawsze dąży do powrotu do swojej nienapiętej, swobodnej konfiguracji. Oznacza to, że zwrot siły jest również zgodny ze zwrotem wektora ***b*** – ***a*** (patrz: rys. 5.3). Na tej podstawie można otrzymać wersor siły więzadła, dzieląc wektor ***b*** – ***a*** przez jego długość:

$$\hat{\boldsymbol{F}} = \frac{\boldsymbol{b}-\boldsymbol{a}}{\left|\boldsymbol{b}-\boldsymbol{a}\right|}. \tag{5.1}$$

Ogólnie wartość siły generowanej przez więzadło zależy od jego sztywności oraz przyrostu długości. Przyrost długości Δl oblicza się jako różnicę pomiędzy długością aktualną l_{akt} i swobodną l_{swob}:

$$\Delta l = l_{\text{akt}} - l_{\text{swob}}. \tag{5.2}$$

Warto zauważyć, że tak wyznaczony przyrost długości więzadła może mieć wartość dodatnią albo ujemną. Dodatnia oznacza, że więzadło jest rozciągane. Natomiast wartość ujemna sygnalizuje, że więzadło jest nienapięte (swobodne). W tym przypadku nie generuje ono żadnej siły.

Znając przyrost długości Δl, można wyznaczyć wartość siły. Najprostszy model, przedstawiony już wcześniej dla więzadeł jednowymiarowych, jest modelem liniowym. W tym przypadku wystarczy pomnożyć przyrost długości przez współczynnik sztywności:

$$F = k\Delta l, \tag{5.3}$$

gdzie:

k – współczynnik sztywności modelu liniowego wyrażony w N/m.

Warto jednak zaznaczyć, że w ostatnich latach modele liniowe tracą popularność na rzecz modeli nieliniowych, które lepiej odwzorowują rzeczywiste zachowanie więzadeł. Jak wspomniano wcześniej, obecnie najczęściej wykorzystuje się modele kwadratowe i eksponencjalne. W modelu kwadratowym wartość siły wyznacza się analogicznie do modelu liniowego. Jedyną różnicą jest to, że przyrost długości podnoszony jest do kwadratu:

$$F = k(\Delta l)^2. \tag{5.4}$$

W powyższym wzorze to k współczynnik sztywności modelu kwadratowego wyrażony w N/m^2 i nie może być bezpośrednio porównywany ze współczynnikiem z poprzedniego wzoru. Warto zauważyć, że przedstawione wzory na wartość siły nie różnią się od wzorów dla układów jednowymiarowych.

Teraz można wyznaczyć siłę jako wektor poprzez:

$$\boldsymbol{F} = \hat{\boldsymbol{F}}F. \tag{5.5}$$

Należy tylko pamiętać o tym, że więzadło generuje siłę tylko wtedy, gdy jest rozciągane. Oznacza to, że dla przyrostu długości więzadła Δl mniejszego od 0,0, siła jest równa 0,0. Ostatecznie, uwzględniając powyższy warunek, wzór na siłę od więzadła można zapisać w następujący sposób:

$$\boldsymbol{F} = \begin{cases} \hat{\boldsymbol{F}}F & \text{dla} \quad \Delta l > 0 \\ \boldsymbol{0} & \text{dla} \quad \Delta l \leq 0 \end{cases}. \tag{5.6}$$

Powyższy wzór wektorowy jest słuszny niezależnie od przyjętego modelu więzadła.

Przygotowane wzory można zaimplementować w Pythonie. Celem będzie zaprogramowanie prostego skryptu, który dla zadanych parametrów więzadła – $\boldsymbol{a}$, $\boldsymbol{b}$, k i l_{swob} – wyznaczy wektor siły, którą więzadło działa na punkt związany z jego ruchomym przyczepem A. Z uwagi na to, że program działa na wektorach, a nie wielkościach skalarnych (przypadek jednowymiarowy), do obliczeń wygodnie będzie wykorzystać wprowadzoną wcześniej bibliotekę *Numpy*.

```
import numpy as np
```

W kolejnym etapie warto zdefiniować dane wejściowe do programu.

```
#Dane:
# [m] - dwuwymiarowy wektor polozenia przyczepu ruchomego A
a = np.array([2.0, 3.0])

# [m] - dwuwymiarowy wektor polozenia przyczepu nieruchomego B
b = np.array([-1.0, 0.2])

# [N/m^2] - wspolczynnik sztywnosci wiezadla w modelu kwadratowym
k = 11.0

# [m] - dlugosc swobodna wiezadla
l_swob = 2.1
```

Teraz można już zaimplementować część obliczeniową programu. Warunek związany z przyrostem długości więzadła zostanie ponownie zaprogramowany z wykorzystaniem instrukcji warunkowej *if/else* (patrz: poprzedni rozdział). Z uwagi na złożoność obliczeń i wykorzystanie biblioteki *Numpy* pierwsza wersja kodu nie będzie zapisana w formie funkcji. Ułatwi to podstawową weryfikację działania skryptu.

```
import matplotlib.pyplot as plt
%matplotlib inline

#Wyznaczenie sily, ktora wiezadlo dziala na punkt
# 1) wyznaczenie aktualnej dlugosci wiezadla
```

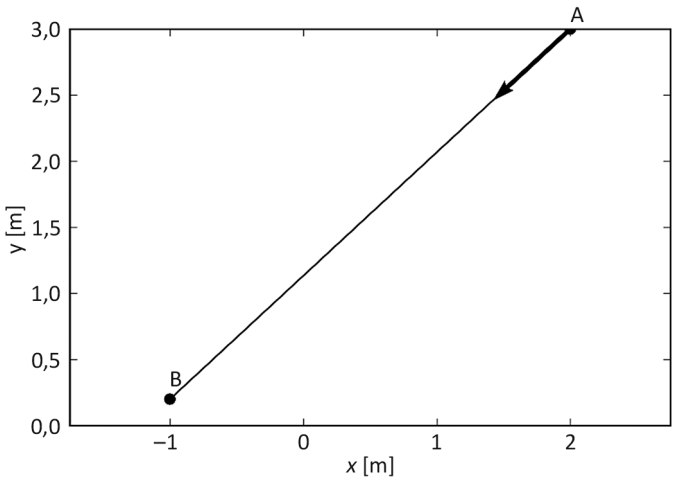
```
wek_kier = b - a
# //dzieki bibliotece numpy powyzsza operacja zostanie
# //poprawnie wykonana jako odejmowanie wektorow

l_akt = np.linalg.norm(wek_kier)
# //do wyznaczenia dlugosci wektora wykorzystywana jest
# //funkcja norm z biblioteki numpy, ktora
# //znajduje sie w module linalg

# 2) wyznaczenie przyrostu dlugosci wiezadla
delta_l = l_akt - l_swob
# przyrost dlugosci

# 3) warunek if/else
if delta_l > 0.0:
    F_wart = k * delta_l**2
    # wartosc sily

    F_wer = wek_kier / l_akt
    # wersor sily – wykorzystywane sa policzone wczesniej:
    # wek_kier oraz l_akt; to przyspieszy obliczenia

    F = F_wer * F_wart
    # wektor sily [N]

else:
# -- jezeli przyrost dlugosci jest mniejszy od zera --
    F = np.array([0.0, 0.0])
    # wektor sily [N] - wektor zerowy

# 4) wydruk wynikow
print "Sila dla zadanych parametrow wiezadla wynosi: ", F, "N"
```

```
Sila dla zadanych parametrow wiezadla wynosi:
[-32.28416895 -30.13189102] N
```

Warto także przygotować wizualizację zadanego problemu. Pomoże ona sprawdzić otrzymane wyniki. Wizualizację można przygotować, wykorzystując bibliotekę *Matplotlib*.

```
#rysowanie przyczepow
plt.plot(a[0], a[1], 'bo')   #rysowanie przyczepu ruchomego
plt.plot(b[0], b[1], 'bo')   #rysowanie przyczepu nieruchomego
```

```
#rysowanie podpisow przyczepów
# //rysowanie podpisu przyczepu ruchomego
plt.text(a[0], a[1] + 0.1, 'A')

# //rysowanie podpisu przyczepu nieruchomego
plt.text(b[0], b[1] + 0.1, 'B')

#rysowanie wiezadla - linii pomiedzy A i B
plt.plot([a[0], b[0]], [a[1], b[1]])

#rysowanie sily – wykorzystywana jest funkcja quiver
# //sila dziala na cialo w punkcie A
# //a jej wspolrzedne przechowuje zmienna F
# //wektor sily F jest recznie przeskalowany
plt.quiver(a[0], a[1], F[0], F[1], scale = 250.0)

#ustawienie zakresow osi
plt.axis([-2, 3, -2, 3])

#ustawienie skalowania osi - jednakowe dla x i y
plt.axis('equal')

#podpisy osi
plt.xlabel('x [m]')
plt.ylabel('y [m]')

#wyswietlanie wykresu
plt.show()
```



Rys. 5.4. Wizualizacja więzadła oraz siły w programie

Powyższa wizualizacja potwierdza, że zarówno zwrot, jak i kierunek siły zostały poprawnie wyznaczone.

Można teraz przygotować oraz przetestować funkcję *wyznacz_sile_od_wiez*, która przyjmuje parametry więzadła i zwraca wektor siły. Pierwszy test wykonamy na danych przyjętych w powyższym przykładzie. W drugim teście *l_swob* zostanie zwiększone do do 10,0 m. Więzadło będzie wtedy swobodne i nie będzie generowało siły – funkcja powinna wtedy zwrócić wektor zerowy.

Na tym etapie warto też spojrzeć krytycznie na rozpatrywany model. Podstawowym utrudnieniem jest w nim dobór wartości wejściowych. Model wymaga czterech parametrów do wyznaczenia siły: wektora położenia przyczepu nieruchomego $\boldsymbol{b}$, wektora położenia przyczepu ruchomego $\boldsymbol{a}$, długości swobodnej l_{swob} oraz współczynnika sztywności k. Te wielkości bardzo trudno wyznaczyć na podstawie badań eksperymentalnych więzadeł. W literaturze przedstawiono wiele takich badań, ale należy pamiętać, że nawet wyznaczenie długości swobodnej stanowi problem przy nieregularnej geometrii więzadeł. Dlatego też modele stawów poddaje się procedurze poszukiwania optymalnych parametrów (ang. *parameters estimation*). W tym zagadnieniu specjalna procedura numeryczna dostraja wartości przyjętych parametrów tak, aby charakterystyki wyjściowe układu (np. zależność pomiędzy przemieszczeniem kątowym a momentem zginającym w modelu triady kręgosłupa) były zbliżone do charakterystyk otrzymanych w sposób eksperymentalny. To zagadnienie wykracza poza zakres tej książki. Zainteresowani więcej informacji znajdą w następujących pracach: [15, 16].

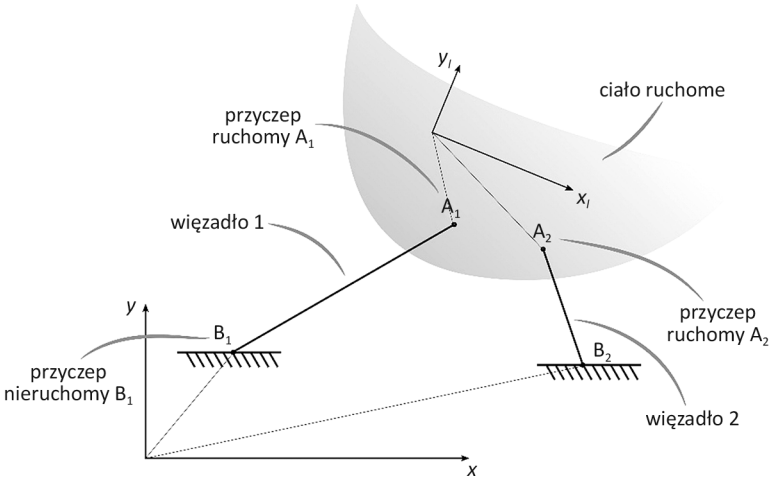
```
def wyznacz_sile_od_wiez(a, b, k, l_swob):
    #Wyznaczenie sily, ktora wiezadlo dziala na punkt
    # 1) wyznaczenie aktualnej dlugosci wiezadla
    wek_kier = b – a

    # //dzieki bibliotece numpy ta operacja zostanie
    # //poprawnie wykonana jako odejmowanie wektorow

    l_akt = np.linalg.norm(wek_kier)

    # //do wyznaczenia dlugosci wektora wykorzystywana jest
    # //funkcja norm z bilbioteki numpy, ktora
    # //znajduje sie w module linalg

    # 2) wyznaczenie przyrostu dlugosci wiezadla
    delta_l = l_akt - l_swob          # przyrost dlugosci

    # 3) warunek if/else
    if delta_l > 0.0:
        F_wart = k * delta_l**2       # wartosc sily
```

```
        F_wer = wek_kier / l_akt      # wersor sily
        # wykorzystujemy policzone wczesniej:
        # wek_kier oraz l_akt; to przyspieszy obliczenia

        F = F_wer * F_wart            # wektor sily [N]

    else: # -- jezeli przyrost dlugosci jest mniejszy od zera --
        F = np.array([0.0, 0.0])      # wektor zerowy

    return F

print "--- TEST 1 ---"
print "Sila dla zadanych parametrow wiezadla wynosi: "
print wyznacz_sile_od_wiez(a, b, k, l_swob), "N"

print "--- TEST 2 ---"
l_swob = 10.0
print "Sila dla zadanych parametrow wiezadla wynosi: "
print wyznacz_sile_od_wiez(a, b, k, l_swob), "N"
```

```
--- TEST 1 ---
Sila dla zadanych parametrow wiezadla wynosi:
[-32.28416895 -30.13189102] N

--- TEST 2 ---
Sila dla zadanych parametrow wiezadla wynosi:
[ 0.  0.] N
```

6. WSTĘP DO MODELOWANIA UKŁADÓW WIELOCZŁONOWYCH W DWÓCH WYMIARACH

Wprowadzone wcześniej metody dotyczyły przypadków, w których do punktu materialnego dołączone było jedno- lub dwuwymiarowe więzadło. Niemniej jednak punkt materialny nie wystarczy do zamodelowania elementów stawów człowieka. W rzeczywistości w skład typowego stawu wchodzą przynajmniej dwie kości, które mogą się przemieszczać względem siebie. Kości te połączone są poprzez złożony układ więzadeł oraz mogą wchodzić w pośredni kontakt, który umożliwia chrząstka. Na początku warto zaznaczyć, że wszystkie elementy stawów są odkształcalne. Jednakże niektóre z nich, kości, odkształcają się tylko w niewielkim stopniu. Dlatego też w typowych modelach wieloczłonowych zakłada się, że są one ciałami sztywnymi (rys. 6.1). Z tego względu w tej części skryptu przedstawione zostaną podstawy opisu układów złożonych z ciał sztywnych i elementów podatnych.



Rys. 6.1. Ruchome ciało sztywne oraz dwa, dołączone do niego, więzadła

Mechanika ciała sztywnego jest rozszerzeniem mechaniki punktu materialnego. Kluczowa różnica w tych dwóch zagadnieniach polega na tym, że w mechanice ciała sztywnego uwzględnia się jego obroty – przemieszczenia kątowe. Przemieszczenia kątowe powstają na skutek działania momentów sił. W ogólnym przypadku

moment siły to iloczyn wektorowy ramienia, na którym działa siła, oraz samej siły. Ramię oczywiście należy wyznaczyć względem pewnego bieguna. W przypadku zadań statycznych wybór bieguna nie ma wpływu na wyniki. Należy tylko pamiętać o tym, żeby wszystkie momenty wyznaczać względem tego samego bieguna. Takim biegunem może być przykładowo środek nieruchomego układu współrzędnych. Zadania dynamiczne wymagają już bardziej ścisłej definicji bieguna. Najprostszą formę równań otrzymuje się jednak, gdy za biegun przyjęty zostanie środek ciężkości ciała ruchomego. Z uwagi na to, że ciało ruchome ma w tym przypadku określoną geometrię – nie jest punktem – siły można do niego przyłożyć w dowolnych miejscach. Analogicznie, więzadła i inne elementy podatne też można dołączyć dowolnie – patrz: rys. 6.1.

Miejsce przyczepu więzadła determinuje jego odkształcenia oraz to, jaki moment wytworzy siła, którą to więzadło wygeneruje. Metoda wyznaczania siły, którą działa więzadło, pozostaje w tym przypadku bez zmian względem wcześniejszego rozdziału. Niemniej jednak wyznaczenie punktu przyczepu więzadła do ciała ruchomego – przykładowo A_1 z rys. 6.1 – jest już bardziej skomplikowane. Wymaga to podania współrzędnych tego punktu – A_1 – w układzie lokalnym $\{x_l\, y_l\}$ – np. poprzez wektor przyczepu $\boldsymbol{a}_1^l$, a także aktualnego położenia liniowego i kątowego ciała ruchomego. Po podaniu tych parametrów, wektor przyczepu w układzie globalnym $\boldsymbol{a}_1$ można wyznaczyć za pomocą następującej transformacji wektorowej:

$$\boldsymbol{a}_1 = \mathbf{R}\boldsymbol{a}_1^l + \boldsymbol{p}, \tag{6.1}$$

gdzie:

- $\mathbf{R}$ – macierz obrotu zależna od położenia kątowego ciała ruchomego,
- $\boldsymbol{p}$ – wektor przemieszczenia środków układu lokalnego i globalnego, zależny od położenia liniowego ciała ruchomego.

Na dalszym etapie rozważań opisane w ten sposób siły i momenty można już uwzględnić w równaniach równowagi albo dynamiki. Z uwagi na uwzględnienie przemieszczeń kątowych ciała sztywnego równania te będą miały formę dwóch równań wektorowych albo trzech równań po rozpisie na współrzędne. Drugie równanie wektorowe reprezentuje ruch obrotowy i związane jest z momentami generowanymi przez siły działające na układ. Przykładowo, w przypadku statyki, równania te mają następującą postać:

$$\begin{cases} \boldsymbol{F} = \mathbf{0} \\ \boldsymbol{M} = \mathbf{0} \end{cases}. \tag{6.2}$$

gdzie:

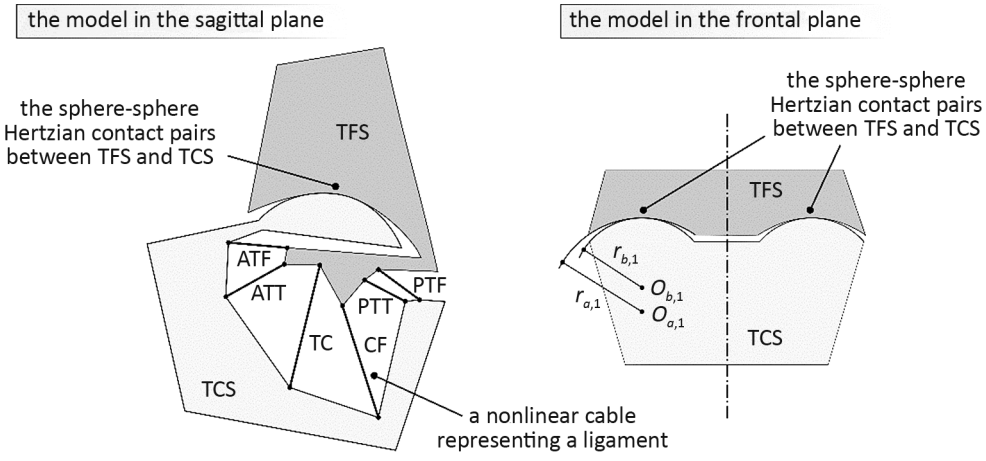
- $\boldsymbol{F}$ – wektorowa suma sił działających na ciało ruchome,
- $\boldsymbol{M}$ – wektorowa suma momentów działających na ciało ruchome.

Warto zauważyć, że wersja wektorowa przedstawionych równań jest słuszna także w układach trójwymiarowych.

7. PRZEGLĄD WYBRANYCH MODELI STAWÓW CZŁOWIEKA ORAZ SZCZEGÓŁOWA ANALIZA WIELOCZŁONOWEGO MODELU STAWU SKOKOWEGO GÓRNEGO

Na przestrzeni lat przedstawionych zostało bardzo wiele różnorodnych modeli stawów człowieka. Modele te można podzielić na dwie podstawowe grupy. Pierwsza z nich to modele przygotowane metodą elementów skończonych (MES) [17–21]. Druga to systemy opisane metodą układów wieloczłonowych. Modele MES służą głównie do analizy wytężenia elementów stawu podczas różnych aktywności fizycznych (najczęściej w zakresie statycznym), a także do badania wpływu układów ortotycznych na otaczające tkanki. Pomimo dużej liczby zalet modele MES są bardzo złożone numerycznie. Co więcej, podczas ich rozwiązywania można napotkać problemy ze zbieżnością numeryczną.

Alternatywę dla modeli MES stanowią układy wieloczłonowe, omówione w tym skrypcie. Modelowanie wieloczłonowe wyróżnia się tym, że w rzeczywistych układach wyszczególnia się najważniejsze elementy i zastępuje się je uproszczonymi odpowiednikami mechanicznymi. W literaturze przedstawiono wiele takich modeli, między innymi dla stawu kolanowego [14, 22, 23], stawu skokowego [15, 24, 25]

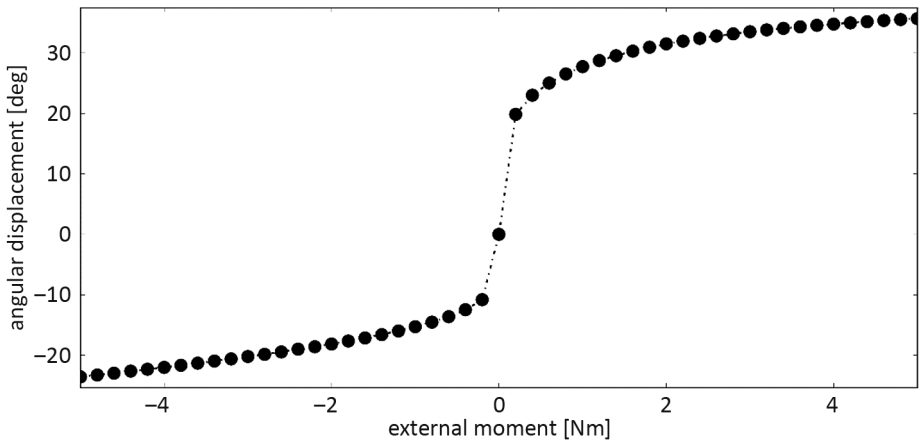


Rys. 7.1. Płaski model górnego stawu skokowego (rysunek skopiowany z [15] za pozwoleniem)

oraz triady kręgosłupa [9]. Przykład takiego postępowania zostanie zaprezentowany na podstawie modelu stawu skokowego [15] – rys. 7.1.

W rzeczywistości staw skokowy składa się z wielu mniejszych stawów, które zapewniają wszystkim kościom stopy dużą ruchomość [26]. Analiza tak złożonego układu może przysporzyć wiele problemów. Jednym ze sposobów uproszczenia tego zadania jest wyizolowanie ważniejszych połączeń w stawie i przesztywnienie pozostałych, takie podejście zastosowano w [15], gdzie rozważany jest staw skokowy górny, a według anglojęzycznej literatury, prawdziwy staw skokowy (ang. *true ankle joint*). Ta część stawu skokowego umożliwia ruchy zginania i prostowania kości skokowej względem kości piszczelowej i strzałkowej. Ruchy te stabilizowane są przez złożony układ sześciu przestrzennych więzadeł oraz pośredni kontakt kości realizowany poprzez tkankę miękką – chrząstkę.

W przedstawionym modelu układ więzadeł zastąpiono sześcioma dwuwymiarowymi cięgnami o eksponencjalnej charakterystyce. Pośredni kontakt kości opisany został za pomocą dwóch odkształcalnych par typu sfera-sfera. Warto zauważyć, że z uwagi na przestrzenny charakter układ więzadeł stawu przyjęte cięgna wymagały dostrojenia parametrów. Poszukiwanie optymalnych wartości parametrów cięgien zrealizowano w modelu za pomocą procedury optymalizacyjnej, która iteracyjnie przeszukiwała różne kombinacje parametrów materiałowych tak, żeby zminimalizować różnice odpowiedzi zastępujących dwuwymiarowych cięgien oraz rzeczywistych trójwymiarowych więzadeł. Najważniejsze wyniki otrzymane z modelu przedstawiono na rysunku 7.2 w formie funkcji przemieszczenia kątowego od momentu powodującego zginanie/prostowanie.



Rys. 7.2. Wykres zależności pomiędzy przemieszczeniem kątowym a momentem zginającym/prostującym w modelu stawu skokowego (rysunek skopiowany z [15] za pozwoleniem)

Warto zauważyć, że model wykazuje charakterystyczny oraz stosunkowo duży zakres przemieszczeń o bardzo małej sztywności kątowej. Co więcej, po tym zakre-

sie sztywność układu znacząco wzrasta – dalszy przyrost przemieszczenia kątowego wymaga bardzo dużego zwiększenia momentu zginającego. Zarówno zakres przemieszczeń kątowych, jak i szczegółowo przeanalizowane w artykule odkształcenia i siły generowane przez cięgna, pozwalają stwierdzić, że wyniki otrzymane z modelu w dużym stopniu pokrywają się z dostępnymi wynikami badań eksperymentalnych. Potwierdza to skuteczność metody układów wieloczłonowych w zastosowaniu do modelowania stawów człowieka. Modele te posiadają także znacznie mniejszą złożoność numeryczną od modeli MES. Sprawia to, że można je stosować do bardzo zróżnicowanych zadań, w tym zadań, które wymagają wielokrotnych rozwiązań, np. szacowanie wartości sił mięśniowych albo optymalizacja parametrów modelu. Pomimo niewątpliwych zalet z układami wieloczłonowymi wiążą się też pewne trudności. Przede wszystkim samo określenie struktury modelu jest zazwyczaj zadaniem trudnym. Proste odpowiedniki mechaniczne nie zawsze odpowiadają złożonym strukturom występującym w stawach człowieka. Dużym problemem może być także dobór parametrów modelu, co widoczne było już w przytoczonym modelu stawu skokowego. O ile w metodzie MES geometrię modelu można uzyskać prawie bezpośrednio ze zdjęć z rezonansu magnetycznego lub tomografii komputerowej, o tyle w modelach wieloczłonowych taką geometrię trzeba odpowiednio przetworzyć i uprościć, co często jest zadaniem trudnym i ma duży wpływ na wyniki otrzymywane.

Warto także wspomnieć, że niektóre grupy badawcze podchodzą do problemu modelowania stawów w sposób hybrydowy [27]. W tym podejściu część złożonych elementów, takich jak tkanki miękkie pośredniczące w kontakcie kości, modelowana jest za pomocą MES-u. Elementy spełniające mniej skomplikowane funkcje, np. więzadła, które pracują głównie na rozciąganie, zastępowane są przez proste odpowiedniki mechaniczne, np. nieliniowe cięgna.

Powyższy rozdział kończy już niniejszy skrypt. Czytelnik, który dotarł do tego etapu, może już samodzielnie studiować literaturę naukową z zakresu modelowania systemów biomechanicznych metodą układów wieloczłonowych.

LITERATURA

[1] *Python 2.7 – tutorial*, https://docs.python.org/2/tutorial/ (dostęp: 18.12.2019).

[2] S. van der Walt, S.C. Colbert, G. Varoquaux, *The NumPy Array: A Structure for Efficient Numerical Computation*, Comput. Sci. Eng. 13 (2011) 22–30. DOI: 10.1109/MCSE.2011.37.

[3] J.D. Hunter, *Matplotlib: A 2D graphics environment*, Comput. Sci. Eng. 9 (2007) 99–104. DOI:10.1109/MCSE.2007.55.

[4] *Matplotlib – user guide*, https://matplotlib.org/3.1.1/users/index.html (accessed December 18, 2019).

[5] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Second Edi, Cambridge University Press, Cambridge 1992.

[6] A. Cromer, *Stable solutions using the Euler approximation*, Am. J. Phys. 49 (1981) 455–459. DOI: 10.1119/1.12478.

[7] F. Goulette, Z.-W. Chen, *Fast computation of soft tissue deformations in real--time simulation with Hyper-Elastic Mass Links*, Comput. Methods Appl. Mech. Eng. 295 (2015) 18–38. DOI: 10.1016/j.cma.2015.06.015.

[8] *Scipy – user guide*, https://docs.scipy.org/doc/scipy/reference/integrate.html (accessed December 18, 2019).

[9] M.R. Gudavalli, J.J. Triano, *An analytical model of lumbar motion segment in flexion*, J. Manipulative Physiol. Ther. 22 (1999) 201–208. DOI: 10.1016/S0161-4754(99)70045-X.

[10] G.M. Fichtenholz, *Rachunek różniczkowy i całkowy*, PWN, Warszawa 1966.

[11] C.T. Kelley, *Iterative Methods for Optimization*, 1999. DOI: 10.1137/1.9781611970920.

[12] J.R. Funk, G.W. Hall, J.R. Crandall, W.D. Pilkey, *Linear and Quasi-Linear Viscoelastic Characterization of Ankle Ligaments*, J. Biomech. Eng. 122 (2002) 15. DOI:10.1115/1.429623.

[13] A. Sensini, L. Cristofolini, *Biofabrication of Electrospun Scaffolds for the Regeneration of Tendons and Ligaments*, Materials (Basel). 11 (2018) 1963. DOI: 10.3390/ma11101963.

[14] M. Machado, P. Flores, J.C.P. Claro, J. Ambrósio, M. Silva, A. Completo, H.M. Lankarani, *Development of a planar multibody model of the human knee joint*, Nonlinear Dyn. 60 (2009) 459–478. DOI: 10.1007/s11071-009-9608-7.

[15] A. Borucka, A. Ciszkiewicz, *A Planar Model of an Ankle Joint with Optimized Material Parameters and Hertzian Contact Pairs*, Materials (Basel). 12 (2019) 2621.

[16] N. Sancisi, V. Parenti-Castelli, *A 1-Dof parallel spherical wrist for the modelling of the knee passive motion*, Mech. Mach. Theory. 45 (2010) 658–665. DOI: 10.1016/j.mechmachtheory.2009.11.009.

[17] J. Clin, C.É. Aubin, S. Parent, H. Labelle, *Biomechanical modeling of brace treatment of scoliosis: Effects of gravitational loads*, Med. Biol. Eng. Comput. 49 (2011) 743–753. DOI:10.1007/s11517-011-0737-z.

[18] C.G. Fontanella, F. Nalesso, E.L. Carniel, A.N. Natali, *Biomechanical behavior of plantar fat pad in healthy and degenerative foot conditions*, Med. Biol. Eng. Comput. 54 (2016) 653–661. DOI:10.1007/s11517-015-1356-x.

[19] R.E. Tannous, F.A. Bandak, T.G. Toridis, R.H. Eppinger, *A Three-Dimensional Finite Element Model of the Human Ankle: Development and Preliminary Application to Axial Impulsive Loading*, in: SAE 962427, Proc. 40th Stapp Car Crash Conf., 1996, 219–238.

[20] K. Szkoda-Poliszuk, M. Żak, C. Pezowicz, *Finite element analysis of the influence of three-joint spinal complex on the change of the intervertebral disc bulge and height*, Int. j. Numer. Method. Biomed. Eng. 34 (2018). DOI: 10.1002/cnm.3107.

[21] P.C. Liacouras, J.S. Wayne, *Computational Modeling to Predict Mechanical Function of Joints: Application to the Lower Leg With Simulation of Two Cadaver Studies*, J. Biomech. Eng. 129 (2007) 811. DOI: 10.1115/1.2800763.

[22] N. Sancisi, V. Parenti-Castelli, *A sequentially-defined stiffness model of the knee*, Mech. Mach. Theory. 46 (2011) 1920–1928. DOI: 10.1016/j.mechmachtheory.2011.07.006.

[23] A. Ciszkiewicz, J. Knapczyk, *Load analysis of a patellofemoral joint by a quadriceps muscle*, Acta Bioeng. Biomech. 18 (2016). DOI: 10.5277/ABB-00344-2015-03.

[24] R. Gregorio, V. Parenti-Castelli, J.J. O'Connor, A. Leardini, *Mathematical models of passive motion at the human ankle joint by equivalent spatial parallel mechanisms*, Med. Biol. Eng. Comput. 45 (2007) 305–313. DOI: 10.1007/s11517-007-0160-7.

[25] P.K. Jamwal, S. Hussain, Y.H. Tsoi, M.H. Ghayesh, S.Q. Xie, *Musculoskeletal modelling of human ankle complex: Estimation of ankle joint moments*, Clin. Biomech. 44 (2017) 75–82. DOI: 10.1016/j.clinbiomech.2017.03.010.

[26] G. Wu, E. Al., *ISB recommendation on definitions of joint coordinate system of various joints for the reporting of human joint motion—part I: ankle, hip, and spine*, J. Biomchanics. 35 (2002) 543–548. DOI: 10.3138/jsp.34.1.59.

[27] B. Weisse, A.K. Aiyangar, C. Affolter, R. Gander, G.P. Terrasi, H. Ploeg, *Determination of the translational and rotational stiffnesses of an L4-L5 functional spinal unit using a specimen-specific finite element model*, J. Mech. Behav. Biomed. Mater. 13 (2012) 45–61. DOI: 10.1016/j.jmbbm.2012.04.002.

eISBN 978-83-66531-56-7

Cracow University
of Technology