Algoritmy Obliczeniowe w Elektronice i Telekomunikacji

Wykład 12 – Powtórzenie materiału

Politechnika Poznańska – Instytut Radiokomunikacji
pawel.sroka@put.poznan.pl

Egzamin

- Data: 13 lutego 2025 (czwartek), godz. 17:00.
- Forma: egzamin pisemny – Sala CW-10
- Czas trwania: 1.5 h
- Co zabrać ze sobą: dokument tożsamości, kalkulator, długopis(y), ew. kartkę papieru i przyrządy do rysowania.
- Wyniki:
  - Online – w systemie eKursy – w ciągu tygodnia (do 20.02.2025)
  - Możliwość obejrzenia pracy – 20.02.2025 – pokój P205 (Polanka).
- Poprawa egzaminu (tylko dla osób z oceną niedostateczną lub nieobecnych): 24 lutego 2025, godz. 14:00, sala CW-10 – identyczna forma jak pierwszy termin.

Plan wykładu

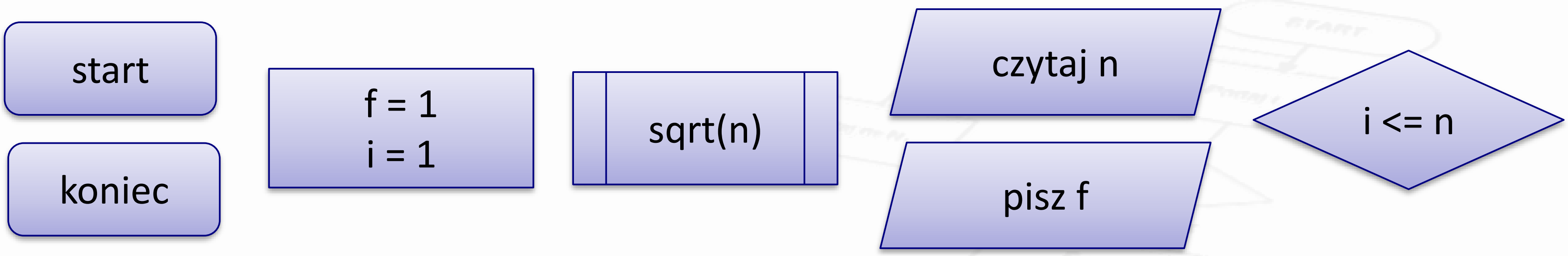
- Zapis algorytmów (schematy blokowe, pseudokod)
- Systemy liczbowe
- Podstawy języka Python
  - Operatory i kolejność działań
  - Instrukcje warunkowe, pętle i tablice NumPy
  - Funkcje
  - Tabelaryzacja i graficzna prezentacja funkcji (wykresy)
- Złożoność algorytmów i rekurencja.
- Elementy algebry liniowej – rozwiązywanie układów równań.
- Metody iteracyjne przybliżone
  - Szeregi Taylora-Maclaurina
  - Rozwiązywanie równań nieliniowych
  - Rozwiązywanie układów równań w sposób przybliżony.
- Pochodne i całki
- Interpolacja i aproksymacja funkcji

Cześć 1

Zapis algorytmów

Sposoby zapisu algorytmów

- Pseudokod
  - Tekstowy opis zawierający sekwencję operacji, które musimy wykonać w celu rozwiązania problemu
  - Brak ustandaryzowanej formy.
  - Może wykorzystywać słowa kluczowe, np.: **set**, **for**, **if**,…
- Schematy blokowe
  - Reprezentacja graficzna algorytmu przy użyciu bloków określonego kształtu

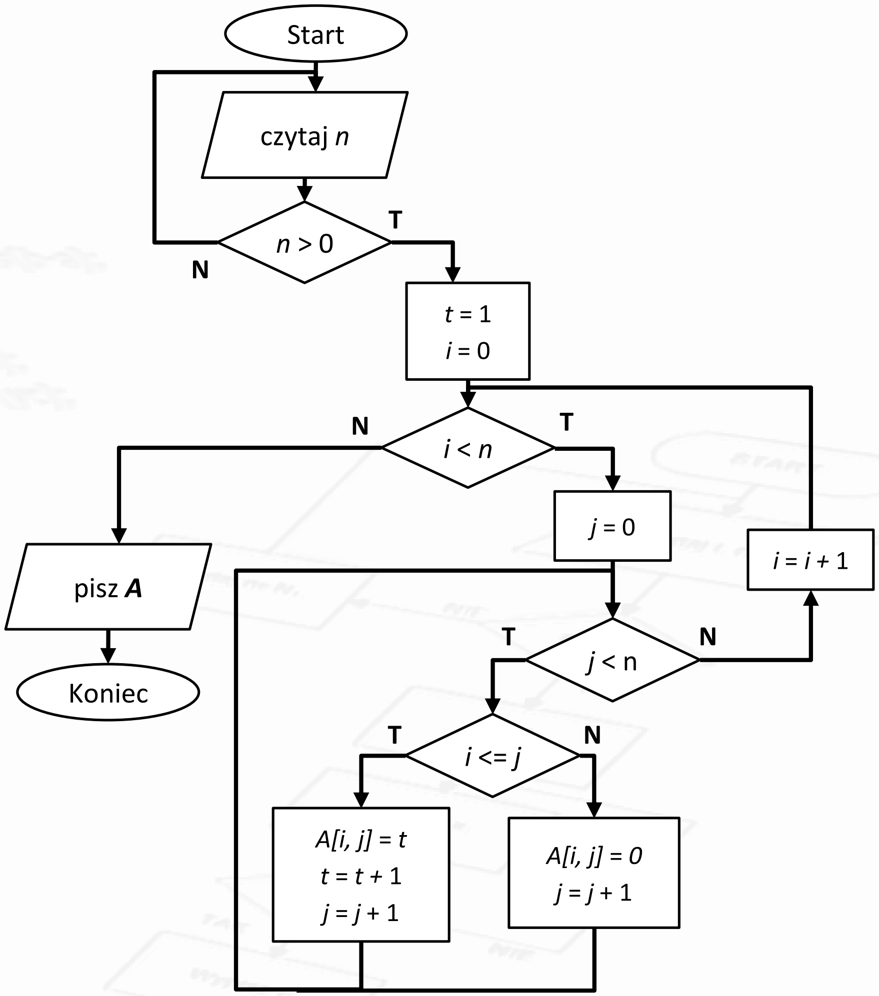


- Implementacja w wybranym języku programowania

Sposoby zapisu algorytmów

- Tworzenie macierzy – uzupełnij schemat wypełnania macierzy górnotrójkątnej o wymiarze *n* kolejnymi liczbami naturalnymi (od 1):
  - Np. dla *n*=4

$$\boldsymbol{A}=\begin{bmatrix}1 & 2 & 3 & 4\\ 0 & 5 & 6 & 7\\ 0 & 0 & 8 & 9\\ 0 & 0 & 0 & 10\end{bmatrix}$$



Sposoby zapisu algorytmów

- Mnożenie wektorów (iloczyn skalarny) – dokończ pseudokod:

1. **Czytaj** wektory *a* i *b*.
2. **Przypisz** *n* = len (*a*) i *m* = len(*b*)
3. **Jeżeli** *n* == *m*:

   Idź do 4.

   W przeciwnym razie**:**

   Zakończ algorytm
4. **Przypisz** *c* = 0.
5. **Dla** *i*=0 do *n*-1:

   Oblicz i przypisz c = c + a[i]*b[i]

Pisz *c*

Zakończ algorytm

Część 2

Systemy liczbowe

System dziesiętny na binarny

- Rozpocznij wczytując wartość dziesiętną x
- Dokonaj inicjalizacji indeksu $n = 0$ sekwencji binarnej (wektora)
- Dopóki $x > 0$ wykonuj
  - Znajdź wartość n-tego bitu ($b[n]$) jako reszta z dzielenia x przez 2 (dzielenie modulo)
  - Znajdź nową wartość x dzieląc ją przez 2 i zaokrąglając w dół.
  - Zwiększ indeks n
- Wyświetl sekwencję: $b[n - 1]$, $b[n - 2]$, …, $b[1]$
- Zakończ algorytm

Liczby ujemne

- Reprezentacja „znak – moduł”:

  <znak> <moduł>

  Znak obejmuje 1 bit i ma wartość „0” dla liczb dodatnich lub „1” dla ujemnych, moduł to wartość bezwzględna z liczby.

  Przykłady: $29_{(10)} = 0\ 11101_{(2)}$, $-31_{(10)} = 1\ 11111_{(2)}$

- Reprezentacja U2 liczb ujemnych: negujemy wszystkie bity modułu, a następnie dodajemy 1 do najmniej znaczącego bitu, np.:

  -31: $31_{(10)} = 0\ 11111$

  negacja: 1 00000

  +0 00001

  wynik: 1 00001

- Dodawanie w U2:

  $29+(-31) = -2 \qquad 0\ 11101_{(2)} + 1\ 00001_{(2)} = 1\ 11110_{(2)}$

- Zamiana na dziesiętny z U2:
  - Działamy tak samo jak dla liczb dodatnich, ale potęgę 2 dla najbardziej znaczącego bitu bierzemy ze znakiem -:

    $N = (-b_J*2^J) + (b_{J-1}*2^{J-1}) + \ldots + (b_1*2) + b_0$

Liczby rzeczywiste – r. stałopozycyjna

- Liczby rzeczywiste zwykle mają część ułamkową, np.:

  $0111.11_{(2)} = 7.75_{(10)}$

- Aby znaleźć reprezentację binarną części ułamkowej musimy znaleźć składowe potęgi 2 dla ujemnych wykładników:

  $N = (d_1 * 2^{-1}) + (d_2 * 2^{-2}) + \ldots + (d_J * 2^{-J})$

  Przykład:

  $0.875_{(10)} = 0.5 + 0.25 + 0.125 = (1*2^{-1}) + (1*2^{-2}) + (1*2^{-3}) = 0.111_{(2)}$

- Reprezentacja może być niedokładna ze względu na ograniczoną liczbę bitów, np. :
  - 0.3 z wykorzystaniem 6 bitów:

    $0.3_{(10)} = 0.01001_{(2)}$

  - Zamiana powrotna na dziesiętny:

    $0.01001_{(2)} = (1*2^{-2}) + (1*2^{-5}) = 0.25 + 0.03125 = 0.28125_{(10)}$

  - Błedy reprezentacji:

    bezwzględny: $0.3 - 0.28125 = 0.01875$

    względny: $0.01875/0.3 = 0.0625 = 6.25\%$

System binarny - przykłady

- Znajdź reprezentację **binarną następujących** liczb całkowitych :

$49_{(10)} = 0110001_{(2)}$
$-71_{(10)} = 10111001_{(2)}$

- Dodaj następujące liczby całkowite:

$01110110 + 10010110 = 00001100$
$00010111 + 11000110 = 11011101$

- Znajdź reprezentację stałopozucyjną następujących liczb z wykorzystaniem 10 bitów:

$0.4_{(10)} = 0.011001100_{(2)}$
$24.2_{(10)} = 011000.0011_{(2)}$
$-13.2_{(10)} = 10010.11010_{(2)}$

Część 3

Podstawy języka Python

Operatory i kolejność działań

Priorytet	Operator	Opis
1	()	Nawiasy
2	**	Potęgowanie
3	+x, -x, ~x	Operator unarny: plus, minus, negacja bitowa
4	*, /, //, %	Mnożenie, dzielenie, dzielenie całkowite, dzielenie modulo
5	+, -	Dodawanie, odejmowanie
6	<<, >>	Przesunięcie bitowe
7	&	Bitowy AND
8	^	Bitowy XOR
9	|	Bitowy OR
10	<, <=, >, >=, !=, ==	Porównanie
11	not	Logiczne not
12	and	Logiczne and
13	or	Logiczne or

Operatory i kolejność działań - ćwiczenie

- Wyznacz wyniki poniższych wyrażeń wiedząc, że: *a*=3, *b*=-3, *c*=2, i *d*=0:

(a+b)**3-c**d+a = 2

4*b/c**2*(c-b)**d = -3

a+b/d**c+(d-c)/a = Błąd (dzielenie przez 0)

a**(2+d*c)*c-b = 21

(((a-b)/c+3)/a+d*b/3)*3 = 6

a-b/c+3/a+d*b/3*3 = 5.5

a*b**(d or c) >> c = 6

(not d > c) and a > c << 2 or b < (not c) = True

Instrukcje warunkowe

- Wyrażenie **if … else**:

```
if wyrażenie_logiczne1:
    #kod dla wyrażenie_logiczne1 True
elif wyrażenie_logiczne2:
    #kod dla wyrażenie_logiczne1 False i wyrażenie_logiczne2 True
else:
    #kod gdy oba False
```

- Operator **ternarny**:

```
wynik_dla_True if wyrażenie_logiczne else wynik_dla_False
```

- Wyrażenie **match**:

```
match zmienna:
    case pierwsza_wartość:
            #kod dla pierwszej wartości
    case druga_wartość:
            # kod dla drugiej wartości
    case _ :
            # kod dla przypadku ogólnego (żaden z powyższych)
```

Pętle

- Pętla **for**:
  - Wykorzystywana do powtarzania zbioru poleceń dla każdej wartości ze znanej sekwencji (pętla określona)
  - Składnia:

```
for iterator in sekwencja:
        #powtarzane instrukcje
```

- Pętla **while**:
  - Wykorzystywana do powtarzania zbioru poleceń tak długo, jak spełniony jest warunek wykonania pętli (pętla nieokreślona)
  - Składnia:

```
while <wyrażenie_logiczne>:
        #powtarzane instrukcje
```

- **Listy składane**:
  - Uproszczona składnia dla prostego wyznaczania sekwencji w sposób iteracyjny
  - Składnia:

```
wynik = [obliczenie_wyniku określenie_pętli]
```

Tablice NumPy

- **NumPy** umożliwia nam korzystanie z N-wymiarowych tablic:

```
import numpy as np

x = np.array([[2, 4, 5], [1, 2, 3]])
print(x)
```

- Tablica musi zawierać taką samą liczbę elementów w każdym wymiarze (np. wierszu), które muszą być tego samego typu.
- Dostęp do każdego elementu jest na podstawie podanych indeksów; indeksy mogą być wyrażone w postaci przedziałów (range) aby wybrać część tablicy.
- Tablica jednowymiarowa jest nazywana wektorem:
  - Wektor o równo odległych elementach można utworzyć korzystając z *arange* lub *linspace*:

```
row_vector = np.arange(1, 20, 2)
print(row_vector)
```

```
row_vector = np.linspace(1, 20, 10)
print(row_vector)
```

- Dwuwymiarowa tablica jest nazywana macierzą:
  - Jest szereg funkcji ułatwiających tworzenie szczególnych macierzy: *zeros, ones, eye, empty*.
- Standardowe operatory w przypadku tablic są stosowane elementowo
  - Aby wykonać mnożenie macierzowe możemy użyc funkcji *dot* lub operatora @.
  - W celu wykonania transpozycji używamy funkcji *transpose* lub właściwości *T*.

Zadanie – analiza kodu

- Przeanalizuj poniższy skrypt w Python'ie. Opisz jego działanie i przeznaczenie (1-2 zdania).

```
t = 1

c = np.zeros((a,a))
spacing = len(b)
for i in range(a):
    init_id = i % b[i % spacing]
    for j in range(init_id, a, b[i % spacing]):
        c[i,j] = t
        t+=b[i % spacing]
print(c)
```

- Podaj wynik działania jeśli wprowadzone są następujące wartości *a* i *b*:
  - a = 3, b = [2]

    ```
    [[1. 0. 3.]
     [0. 5. 0.]
     [7. 0. 9.]]
    ```

  - a = 4, b = [0]

    ```
    ZeroDivisionError: integer modulo by zero
    ```

  - a = 5, b = [2, 3]

    ```
    [[ 1.  0.  3.  0.  5.]
     [ 0.  7.  0.  0. 10.]
     [13.  0. 15.  0. 17.]
     [19.  0.  0. 22.  0.]
     [25.  0. 27.  0. 29.]]
    ```

Funkcje

- **Funkcja** – blok kodu (algorytm) o nadanej nazwie wykonujący określone zadanie; sekwencja poleceń, które wykonujemy w celu realizacji zadania
  - Do funkcji możemy przekazać dane jako **argumenty wejściowe**
  - Funkcja może zwracać wynik w postaci **argumentów wyjściowych**
  - Sekwencja operacji tworzy **ciało funkcji**
  - Funkcję uruchamiamy wprowadzając jej **nazwę** w kodzie razem z listą argumentów wejściowych i wyjściowych
- Funkcję tworzymy wprowadzając słowo kluczowe **def**, po którym następuje jej **nazwa** i lista **argumentów wejściowych**. Funkcja może zwracać wynik:

```
def nazwa_funkcji(arg1, arg2):
        #ciało funkcji
        return wynik
```

- Funkcję można zagnieździć wewnątrz ciała innej funkcji (nadrzędnej) – jest wtedy dostępna tylko w funkcji nadrzędnej
- Funkcja **lambda** (zwana też anonimową) jest funkcją zdefiniowaną bez nazwy, ale zapisaną w zmiennej.

Korygowanie błędów - ćwiczenie

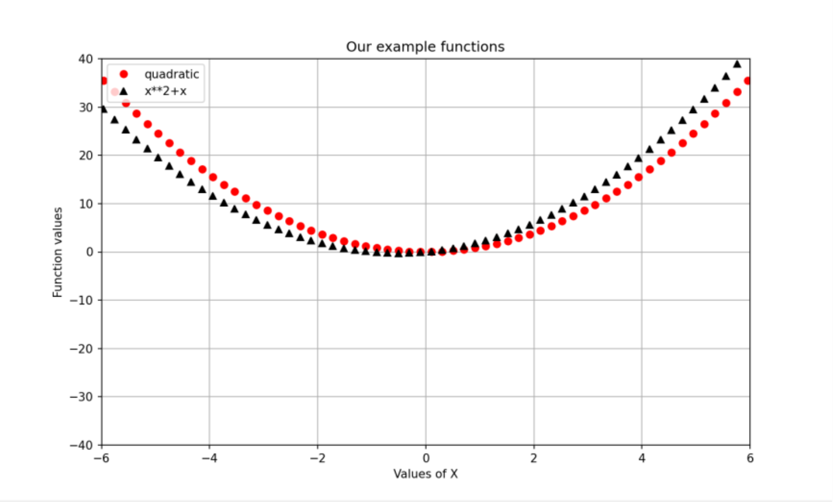
- W podanej definicji funkcji Python'ie, której zadaniem jest wykonanie mnożenia macierzowego, znajdują się 3 błędy (składniowy, uruchomieniowy/wyjątek i logiczny). Podkreśl linie zawierające błędy i podaj obok ich poprawne wersje (z poprawionymi błędami):

```
import numpy as np

def matMul[a,b]:        def matMul(a,b):
    n,m=a.shape
    o,p = b.shape
    assert(m==o)
    c = np.zeros((n,p))
    for i in np.arange(n):
        for j in np.arange(p):
            sum='0'              sum=0
            for k in np.arange(1,m):  for k in np.arange(m)
                sum+=a[i,k]*b[k,j]
            c[i,j]=sum
    return c
```

Graficzna prezentacja wyników

- Proces zapisu funkcji ciągłej w postaci dyskretnej (wektorów argumentów i wartości funkcji) jest nazwyany **tabelaryzacją.**
- Przedstawiamy zależność matematyczną w postaci pary tablic/list (wektorów) (**x**,**y**) o tej samej długości: **x** zawiera argumenty, **y** wartości funkcji.
- Tablicowaną funckję możemy następnie wykreślić w Python'ie korzystając z modułu *PyPlot* biblioteki *MatPlotLib*:
  - Wykres jest tworzony poleceniem plot(x,y), a następnie prezentowany na ekranie funkcją show().
  - Dla wykresu możemy określić styl prezentacji podając tekst formatujący styl linii i markerów ("<kolor><marker><styl linii>")
  - Do wykresu można też dodać tytuł, etykiety osi, legendę, linie siatki, czy ustalić zakres osi.
  - Na wykresie można umieścić kilka krzywych (kolejne polecenia plot) lub można je przestawić w postaci tablicowej korzystając z *subplot.*



Część 4

Złożoność i rekurencja

Złożoność

- **Złożoność algorytmu** oznacza zależność stopnia skomplikowania algorytmu w postaci liczby wykonywanych operacji od rozmiaru wejścia algorytmu (n).
  - Do głównych (elementarnych) i zliczanych operacji zaliczamy: dodawanie, odejmowanie, mnożenie, dzielenie, przypisania i wywołania funkcji.
- Do zapisu złożoności stosuję się notację dużego O: $\boldsymbol{O}(\log(n))$, $\boldsymbol{O}(n)$, $\boldsymbol{O}(n^k)$, $\boldsymbol{O}(n!)$
- **Ćwiczenie:** Oszacuj złożoność obliczeniową podanej funkcji. Podaj liczbę wykonywanych: dodawań, odejmowań, mnożeń, dzieleń i przypisani, a także ogólną złożoność w zapisie dużego O. Przyjmij rozmiar wejścia jako *n*.

```
def myFunction(n):
    a = np.zeros((n,n))
    s = np.zeros(n)
    for i in range(n):
        t = i/(i+1)
        a[i] = [t**j for j in range(0,n)]
        s[i] = 0
        for k in range(n):
            s[i]+=a[i][k]
    return (a,s)
```

Dodawań: n^2+n
Odejmowań: 0
Mnożeń: $n^2\log(n)$
(dla potęgowania: O(log(n)))
Dzieleń: n
Przypisań: $3n^2+4n$
Całkowita złożoność: **$O(n^2\log(n))$**

Rekurencja

- **Rekurencja** to proces powtarzania obliczeń elementów w sposób samopodobny, tj. każdy element jest wprost lub w sposób przybliżony podobny do swojej części
- W praktyce sprowadza się to do wyznaczania elementów (kolejnych wielkości) na podstawie wielkości obliczonych wcześniej w dokładnie taki sam sposób:

$$a_n = f(a_{n-1})$$

- Rekurencja programistyczna - mamy z nią do czynienia, gdy do obliczenia algorytm (funkcja) wywołuje ten sam algorytm, ale dla innych argumentów wejściowych.

```
def nazwaFunkcji(argumenty):
        if (przypadekPodstawowy):
                return wynik1
        elif (drugiPrzypadekBazowy):
                return wynik2
        else:   #przypadek rekurencyjny
                instrukcje
                return nazwaFunkcji(zmodyfikowaneArgumenty)
```

Część 5

Elementy algebry liniowej

Norma

- Normę wektora określonego stopnia wyznaczamy zgodnie ze wzorem:

$$\|v\|_2 = \sqrt{\sum_i v_i^2}$$

$$\|v\|_1 = \sum_i |v_i|$$

$$\|v\|_\infty = \sup_i |v_i|$$

- Normę macierzy liczymy dla wszystkich elementów:

$$\|M\|_p = \sqrt[p]{\left(\sum_i^m \sum_j^n |a_{ij}|^p\right)}$$

- W Python'ie służy do tego funkcja norm (w module *linalg* biblioteki *NumPy*).

```
import numpy as np
P = np.array([[1, 7], [2, 3], [5, 0]])
print(np.linalg.norm(P))
```

Układy równań linowych

- Postać ogólna:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
&\ldots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}
$$

- Dla rozwiązania układu w postaci górnotrójkątnej (lub dolnotrójkątnej) możemy użyć metody podstawienia wstecz (lub odpowiednio podstawienia w przód)
- Gdy macierz współczynników **A** jest nieosobliwa możemy użyć algorytmu eliminacji Gaussa do otrzymania postaci górnotrójkątnej
- Alternatywnie możemy użyć algorytmu Gaussa-Jordana do znalezienia rozwiązania.

Układy równań linowych

- Podstawienie wstecz:
  - Zaczynamy od ostatniego równania, z którego wyznaczamy x_n.
  - Kontynuujemy „w górę” wykorzystując znane wartości x_k do wyznaczenia nowych wartości:

$$x_n = \frac{b_n}{a_{nn}}$$

$$x_{n-1} = \frac{b_{n-1} - a_{n-1n} x_n}{a_{n-1n-1}}$$

$$x_k = \frac{b_k - \sum_{i=k+1}^{n} a_{ki} x_i}{a_{kk}}$$

- Dla macierzy dolnotrójkątnej mamy analogiczną metodę podstawienia w przód:
  - Zaczynamy od pierwszego równania i wyznaczamy x_1.
  - Kontynuujemy „w dół” wyznaczając kolejne x_k.

Układy równań linowych

- Algorytm eliminiacji Gaussa:
  - Krok 0: stwórz macierz rozszerzoną **[A|b]** łącząc macierz A i wektor prawej strony b.
  - Krok 1: Jeśli to konieczne (0 na przekątnej) zamień wiersze macierzy [**A**|**b**] aby wartość $a_{11} \neq 0$. Następnie usuń x_1 w wierszach 2 do n odejmując wiersz 1 przemnożony przez stałą $m_{r,1}$.
  - Kroki p=2...n-1: Kontynuuj z x_p. Jeśli konieczne to zamień wiersze w [**A**|**b**] aby wartość $a_{pp} \neq 0$. Następnie usuwaj x_p z wierszy $p+1$ do n odejmując wiersz p przemnożony przez stałą $m_{r,p}$
  - Krok n: Rozdziel macierz rozszerzoną na macierz kwadratową górnotrójkątną U i wektor prawej strony v, a następnie zastosuj podstawienie wstecz.
- Układ można też rozwiązań algorytmem Gaussa-Jordana:
  - Początkowo stosujemy tę samą metodę co w eliminacji Gaussa, jednak normalizujemy kolejne wiersze, aby wartości na przekątnej wynosiły 1.
  - Następnie, po otrzymaniu postaci górnotrójkątnej, kontynuujemy w analogiczny sposób, ale wstecz – zerując wartości w „górnym trójkącie” macierzy.
  - W wyniku tego otrzymujemy z lewej strony macierzy połączonej macierz jednostkową, a z prawej strony wektor (lub macierz) rozwiązania układu.

Układy równań linowych - ćwiczenie

- Znajdź równoważny układ górnotrójkątny dla podanego układu równań:

$2x_1 + x_2 + 4x_3 + 5x_4 = 11$

$2x_1 - 3x_2 + x_3 - 2x_4 = 1$

$-x_2 + 3x_3 + 2x_4 = 5$

$x_1 - 2x_2 + x_3 = 4$

- A następnie użyj podstawienia wstecz do rozwiązania układu

Wynik:

$2x_1 + x_2 + 4x_3 + 5x_4 = 11$	$x_1 = 1$
$-x_2 - 3x_3 - 2x_4 = 5$	$x_2 = -2$
$8.5x_3 - 7.5x_4 = -14$	$x_3 = -1$
$-1.7647x_4 = -5.2941$	$x_4 = 3$

- Rozwiąż też powyższy układ korzystając z algorytmu Gaussa-Jordana.

Część 6

Metody iteracyjne przybliżone

Metody iteracyjne przybliżone

- Metody **iteracyjne** to algorytmy wykorzystujące działania iteracyjne, a więc powtarzanie sekwencji operacji w pętli
- Dla złożonych problemów stosuje się rozwiązania iteracyjne, ograniczając liczbę powtórzeń i otrzymując wynik **przybliżony**
- Obliczenia zatrzymujemy gdy:
  - Osiągnęliśmy określoną maksymalną liczbę iteracji.
  - Gdy osiągnęliśmy wystarczającą (założoną) dokładność wyniku:
    - Bład bezwzględny:

$$\Delta = |\, \textit{wynik\_dokładny} - \textit{obliczony\_wynik} \,|$$

    - Błąd względny

$$\delta = \left| \frac{\text{wynik\_dokładny} - \text{obliczony\_wynik}}{\text{wynik\_dokładny}} \right|$$

Szeregi Taylora-Maclaurina - ćwiczenie

- Znajdź wartość funkcji *cos*(*x*) korzystając z jej rozwinięcia w szereg Taylora-Maclaurina:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!}$$

  - Wyprowadź zależność rekurencyjną do obliczenia *a*(*i*) na podstawie *a*(*i*-1)
  - Oblicz przybliżoną wartość cos(2) i błąd bezwzględny zakładając że cos(2) = -0.416146 dla:
    - Wykonanych 2 iteracji
    - Wykonanych 3 iteracji
    - Wykonanych 5 iteracji

$$a_i = (-1)^i \frac{x^{2i}}{(2i)!} \qquad a_{i-1} = (-1)^{i-1} \frac{x^{2(i-1)}}{(2(i-1))!} = (-1)^{i-1} \frac{x^{2i-2}}{(2i-2)!}$$

$$m = \frac{a_i}{a_{i-1}} = \frac{(-1)^i \frac{x^{2i}}{(2i)!}}{(-1)^{i-1} \frac{x^{2i-2}}{(2i-2)!}} = (-1) \frac{x^{2i} \cdot (2i-2)!}{x^{2i-2} \cdot (2i)!} = (-1) \frac{x^2 \cdot (2i-2)!}{2i \cdot (2i-1) \cdot (2i-2)!} = -\frac{x^2}{(2i-1) \cdot 2i}$$

$$a_i = m \cdot a_{i-1} = -\frac{x^2}{(2i-1) \cdot 2i} \cdot a_{i-1}$$

2 iteracje: -1
błąd: 0.583853
3 iteracje: -0.333333
błąd: 0.082813
5 iteracji: -0.415873
błąd: 0.000274

Rozwiązywanie równań nieliniowych

- Cztery metody:
  - Bisekcja:
    - Poszukujemy w przedziale <a, b>. Jeśli w przedziale jest pierwiastek (funkcja zmienia znak), to dzielimy go na pół, gdzie c to punkt podziału:

$$c = (a+b)/2$$

    - Kontynuujemy analogicznie z podprzedziałem zawierającym pierwiastek.
  - Falsi
    - Podobna do bisekcji, ale inny sposób podziału na podprzedziały:

$$c = b - \frac{f(b)(b-a)}{f(b) - f(a)}$$

  - Metoda siecznych (secant) – ten sam wzór podziału co w Falsi, ale nowy podprzedział zawsze tworzą dwa ostatnie punkty.
  - Metoda Newtona-Rhapsona
    Rozpoczynamy z pierwszym przybliżeniem pierwiastka x_0 i znajdujemy kolejne przybliżenie jako miejsce zerowe stycznej w x_0 zgodnie ze wzorem:

$$x_i = x_{i-1} - f(x_{i-1}) / f'(x_{i-1})$$

Rozwiązywanie równań nieliniowych - ćwiczenie

- Zilustruj działanie metody Newtona-Rhapsona znajdując miejsce zerowe funkcji:

$$y(x) = x^3 - 2x^2 + x - 5$$

- pokazując 5 początkowy iteracji. Przedstaw interpretację graficzną wykonywanych operacji i wyznacz kolejne przybliżenia. Rozpocznij z x_0= -1

Rozwiązanie:

W metodzie Newtona-Rhapsona znajdujemy kolejne przybliżenia jako:

$$x_i = x_{i-1} - f(x_{i-1}) / f'(x_{i-1})$$

Znajdujemy $f'(x)$:

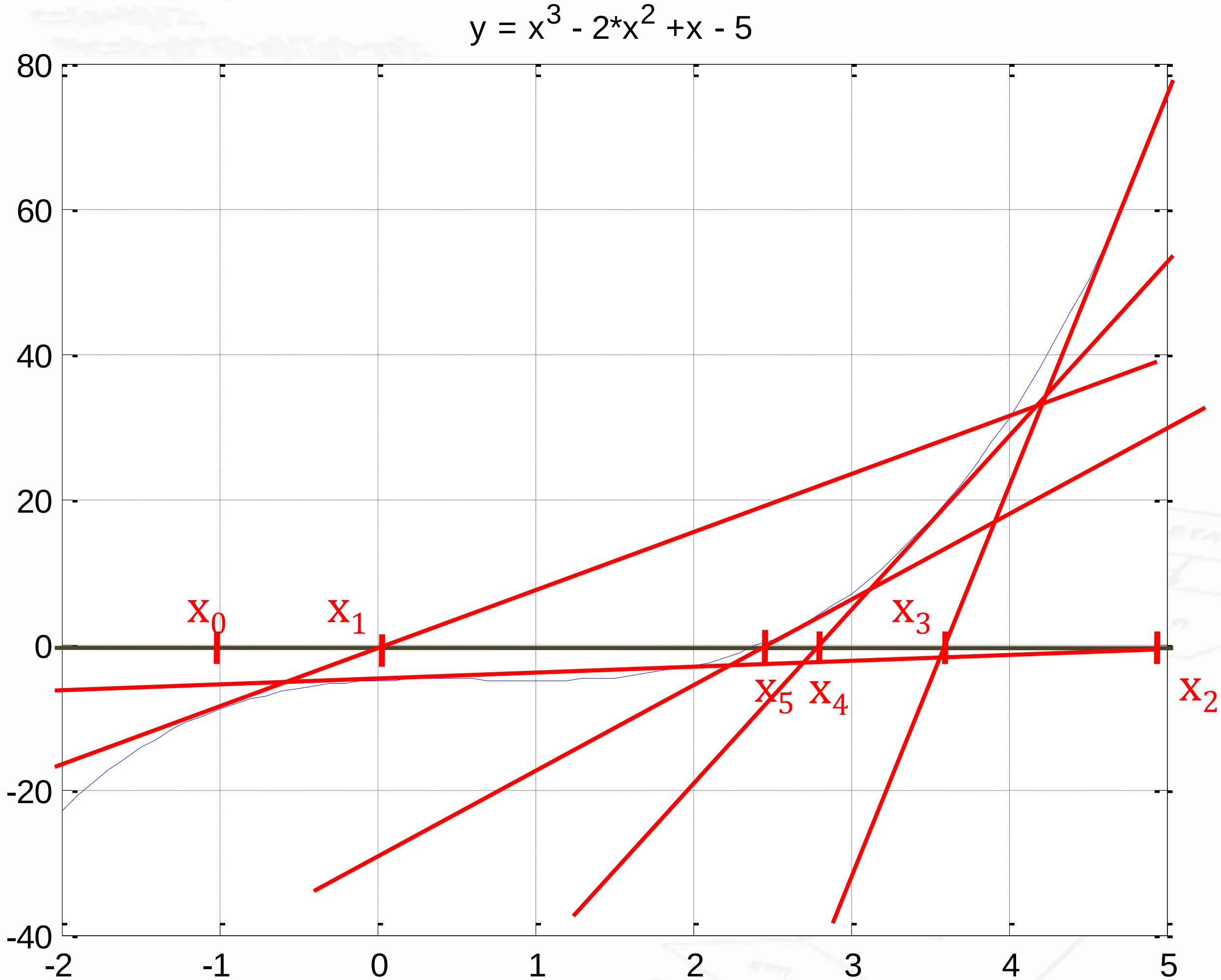
$$f'(x) = 3x^2 - 4x + 1$$

A kolejne znalezione przybliżenia to:

x_1 = -1 –(-9)/8 = 0.125 x_2 = 0.125 –(-4.9043)/0.5469 =9.0929

x_3 = 6.3161 x_4 = 4.4977 x_5 = 3.3529

Rozwiązywanie równań nieliniowych - ćwiczenie



Rozwiązywanie układów równań linowych

- Metody przybliżone – znajdujemy przybliżone rozwiązanie gdy macierz A jest ściśle diagonalnie dominująca.
- Dwie metody: Jakobiego i Gaussa-Seidla. W obu zaczynamy z wybranym pierwszym przybliżeniem, a następnie obliczamy iteracyjnie:
  - Metoda Jakobiego:
    - Do wyznaczenia nowego przybliżenia korzystamy tylko z wartości z poprzedniej iteracji:

$$x_j^{k+1} = \frac{b_j - a_{j1}x_1^k - a_{j2}x_2^k - \cdots - a_{jj-1}x_{j-1}^k - a_{jj+1}x_{j+1}^k - \cdots - a_{jn}x_n^k}{a_{jj}} = \frac{b_j - \sum_{\substack{i=1 \\ i \neq j}}^{n-1} a_{ji}x_i^k}{a_{jj}}$$

  - Metoda Gaussa-Seidla
    - Do wyznaczenia kolejnego przybliżenia wykorzystujemy też już te wartości z obecnej iteracji, które zostały obliczone:

$$x_j^{k+1} = \frac{b_j - a_{j1}x_1^{k+1} - a_{j2}x_2^{k+1} - \cdots - a_{jj-1}x_{j-1}^{k+1} - a_{jj+1}x_{j+1}^k - \cdots - a_{jn}x_n^k}{a_{jj}} = \frac{b_j - \sum_{i=1}^{j-1} a_{ji}x_i^{k+1} - \sum_{i=j+1}^{n-1} a_{ji}x_i^k}{a_{jj}}$$

Rozwiązywanie układów równań linowych - ćwiczenie

- Przedstaw 3 początkowe iteracje rozwiązywania podanego układu równań metodą Jakobiego. Załóż pierwsze przybliżenie x=[1, 0, 1, -1] :

$$6x_1 + x_2 - 1x_3 = 3$$
$$2x_1 - 5x_2 - 1x_4 = 7$$
$$-x_2 + 4x_3 + 2x_4 = 9$$
$$x_1 - 2x_2 + x_3 + 8x_4 = 5$$

Wynik:

$\mathbf{x} \approx [1, -1, 2, 0]$

Część 7

Pochodne i całki

Całki numeryczne

- Zakładając, że w przedziale [a,b] dane są następujące punkty

$$x_0 = a, x_1 = x_0 + \Delta x, x_2 = x_0 + 2\Delta x, \ldots, x_M = x_0 + M\Delta x = b$$

- Całkę numeryczną obliczamy jako:

$$Q[f] = \sum_{i=0}^{M-1} w_i f(x_i) = w_0 f(x_0) + w_1 f(x_1) + \cdots + w_{M-1} f(x_{M-1})$$

  - Zwykle zakłada się tę samą wartość w_i dla wszystkich i czyli $w_i = h$
- Trzy metody:
  - Prostokątów

$$x_i = a + ih + \frac{h}{2} = a + \frac{(2i-1)h}{2}, \quad i = 1,2,\ldots,M \qquad Q[f] = h\sum_{i=1}^{M} f\left(a + \frac{(2i-1)h}{2}\right)$$

  - Trapezów

$$Q[f] = \sum_{i=1}^{M} P_i = h\sum_{i=1}^{M} \frac{f(x_{i-1}) + f(x_i)}{2} = h\left(\frac{f(a)}{2} + \sum_{i=1}^{M-1} f(x_i) + \frac{f(b)}{2}\right)$$

  - Simpsona

$$h = \frac{b-a}{M} \qquad Q[f] = \frac{h}{3}\left(f(a) + f(b)\right) + \frac{2h}{3}\sum_{i=1}^{M-1} f(x_{2i}) + \frac{4h}{3}\sum_{i=1}^{M} f(x_{2i-1})$$

Całki numeryczne - ćwiczenie

- Wyznacz wartość następującej całki:

$$\int_0^5 \left(x^3 - 4x + 1\right)dx$$

korzystając z metody prostokątów. Użyj 10 podprzedziałów. Zilustruj działanie metody rysując odpowiednie kształty na wykresie.

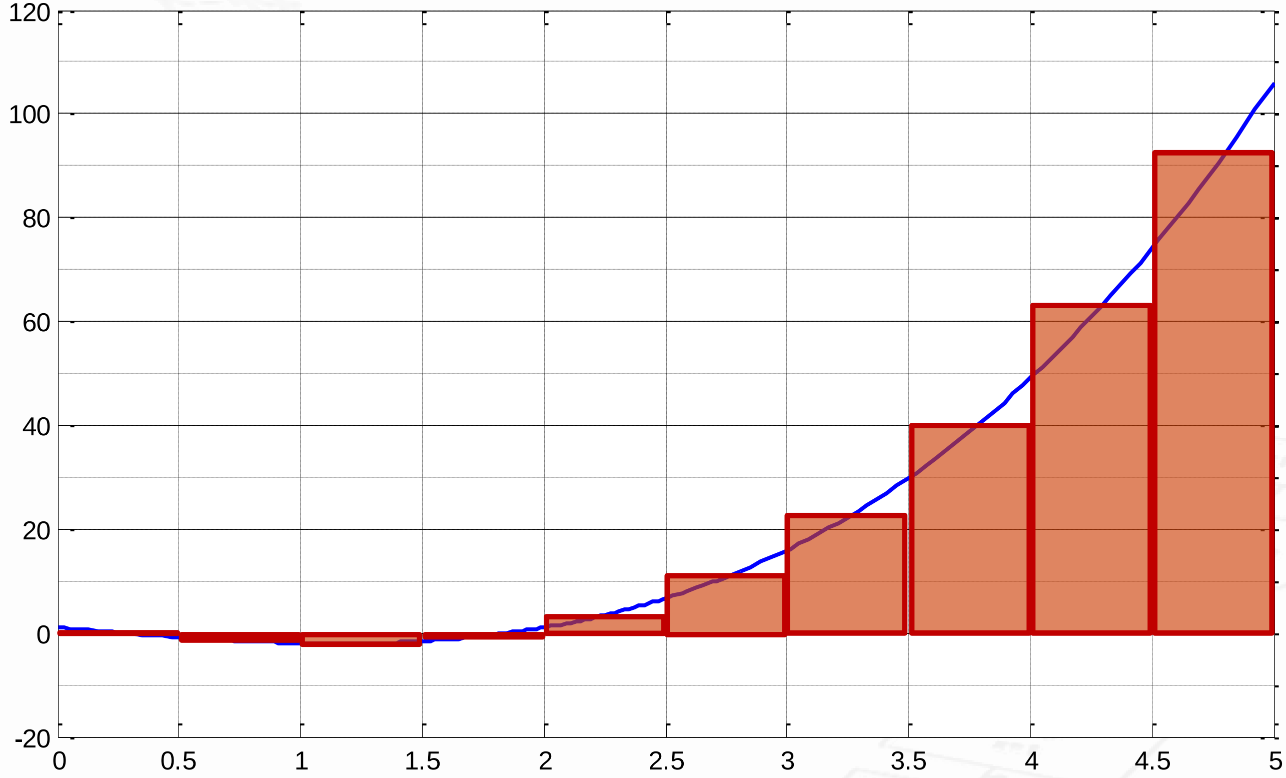
$$\Delta x = \frac{(b-a)}{M} = \frac{5-0}{10} = 0.5$$

Punkt	0	1	2	3	4	5	6	7	8	9
x	0.25	0.75	1.25	1.75	2.25	2.75	3.25	3.75	4.25	4.75
f(x)	0.02	-1.58	-2.05	-0.64	3.39	10.80	22.33	38.73	60.77	89.17

$$Q = \sum_{i=0}^{10} f(x_i)\Delta x \approx 110.47$$

Całki numeryczne - ćwiczenie

- Metoda prostokątów:



Pochodne

- Wyznaczamy numerycznie korzystając z różnic skończonych:
  - Dobieramy wartość $\{h_k\}$ z szeregu o $h_k \to 0$
  - Znajdujemy wartość zapewniającą najlepszą dokładność (granicę) przybliżając
    - Różnicą 2-punktową prostą:

$$D_k = \frac{f(a+h_k) - f(a)}{h_k}, \quad k = 1,2,\ldots,n,\ldots$$

    - Różnicą 2-punktową wsteczną:

$$D_k = \frac{f(a) - f(a-h_k)}{h_k}, \quad k = 1,2,\ldots,n,\ldots$$

    - Różnicą centralną 2-punktową:

$$D_k = \frac{f(x+h_k) - f(x-h_k)}{2h_k}, \quad k = 1,2,\ldots,n,\ldots$$

Część 8

Interpolacja i aproksymacja

Interpolacja

- Interesuje nas znalezienie ogólnej zależności (funkcji) w postaci wyrażenia matematycznego pozwalającego wyznaczyć wartość dla dowolnego argumentu z przedziału $[x_1, x_n]$ jeśli dane są wartości dyskretne (wybrane punkty).
- **Interpolacja** polega na wyznaczaniu w danym przedziale tzw. **funkcji interpolującej**, która przyjmuje w nim z góry zadane wartości w ustalonych punktach nazywanych **węzłami** (znane punkty należą do funkcji).
- Różne metody:
  - Interpolacja liniowa (na podstawie 2 punktów)

$$\hat{f}(x_*) = y_i + \frac{(y_{i+1} - y_i)(x_* - x_i)}{(x_{i+1} - x_i)}$$

  - Interpolacja funkcjami sklejanymi (np. sześcienna – wielomiany 3 stopnia)
  - Interpolacja wielomianowa (wielomian Lagrange'a).

Aproksymacja (regresja)

- **Aproksymacja** to przybliżanie funkcji zwanej **funkcją aproksymowaną** inną funkcją zwaną **funkcją aproksymującą**.
- Chcemy znaleźć funkcję określonej postaci (np. liniową), która "najlepiej" przybliża inną, bardziej skomplikowaną relację (funkcję). Najczęściej staramy się minimalizować błąd średniokwadratowy:

$$\min_{f(x)} E\left(f(x)\right) = \sum_{i=1}^{n} \left(y_i - f\left(x_i\right)\right)^2$$

- W regresji liniowej poszukujemy funkcji linowej postaci $y=a_0+a_1x$

$$a_1 = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}, \quad a_0 = \frac{\left(\sum_{i=1}^{n} x_i^2\right)\left(\sum_{i=1}^{n} y_i\right) - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} x_i y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

Interpolacja i regresja - ćwiczenie

- Dany jest zbiór punktów i zmierzonych wartości funkcji:

x	-4	-3	-2	-1	0	1	2	3	4
y	-8.55	-7.4	-5.7	-3.7	-0.75	0.9	2.9	5.7	7.15

- Wyznacz wartości funkcji dla x = -3.5 oraz x=1.8 korzystając z:
  - Interpolacji liniowej
  - Regresji liniowej

Interpolacja i regresja - ćwiczenie

- Interpolacja linowa:

x	-4	-3	-2	-1	0	1	2	3	4
y	-8.55	-7.4	-5.7	-3.7	-0.75	0.9	2.9	5.7	7.15

- Wybieramy najbliższe pary punktów i wyznaczamy wartość funkcji ze wzoru:

$$\hat{f}(x_*) = y_i + \frac{(y_{i+1} - y_i)(x_* - x_i)}{(x_{i+1} - x_i)}$$

  - Wynik dla x = -3.5:

$$y(-3.5) = -8.55 + \frac{(-7.4+8.55)(-3.5+4)}{(-3+4)} = -8.55 + 0.575 = -7.975$$

  - Dla x=1.8

$$y(1.8) = 0.9 + \frac{(2.9-0.9)(1.8-1)}{(2-1)} = 0.9 + 1.6 = 2.5$$

Interpolacja i regresja - ćwiczenie

- Regresja linowa:

x	-4	-3	-2	-1	0	1	2	3	4
y	-8.55	-7.4	-5.7	-3.7	-0.75	0.9	2.9	5.7	7.15

- Znajdujemy wartości współczynników funkcji liniowej dla wszystkich punktów: a_1=2.065, a_0 =-1.05
- Obliczamy wartości wyznaczonej funkcji linowej:
  - Wynik dla x = -3.5:

    y(-3.5) = -8.2775
  - Dla x=1.8:

    y(1.8) = 2.667

To wszystko w tym semestrze!

Do zobaczenia na egzaminie!