Interpolacja wielomianowa - projekt

Natalia Wojtania i Grzegorz Chojnacki

20 listopada 2020

1 Zadanie

1.1 Tytuł

Tytuł zadania to "Dwutlenek węgla".

1.2 Treść

Program, który oszacuje tempo przyrostu dwutlenku węgla w atmosferze Ziemi. Węzły mają przedstawiać ilość wyemitowanego do atmosfery CO_2 w ciągu roku lub w innym przedziale czasowym.

1.3 Metoda

W programie należy wykorzystać metodę Newtona.

1.3.1 Opis metody

Mając zadany układ punktów $\{(x_j, y_j), j = 0, 1, 2, 3..., n\}$, gdzie $x_0, x_1, x_2, ..., x_n$ są węzłami interpolacyjnymi, a $y_0, y_1, ..., y_n$ wartościami, poszukujemy wielomianu interpolacyjnego $P \in \sqcap_n$ spełniającego warunki $P(x_i) = y_i, i = 0, 1, 2, ..., n$ w postaci :

$$P(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + ... + b_n(x - x_0) \cdot ... \cdot (x - x_{n-1}).$$

Z wyżej wymienionych warunków otrzymamy układ z niewiadomymi $b_0, b_1, ..., b_n$. Z pierwszego równania $P(x_0) = y_0 = b_0$, następnie $P(x_1) = y_1 = b_0 + b_1(x - x_0)$, stąd $b_1 = \frac{y_1 - y_0}{x_1 - x_0}$ itd.

1.3.2 Przykład

x_i	0	2	3
y_i	1	11	19

$$P(x) = b_0 + b_1(x-0) + b_2(x-0)(x-2)$$

Z warunku $P(0) = 1$ mamy $b_0 = 1$, z $P(2) = 11$ mamy $b_1 = 5$, z $P(3) = 19$ mamy $b_2 = 1$.
Stąd $P(x) = 1 + 5(x-0) + 1(x-0)(x-2) = x^2 + 3x + 1$.

2 Opis implementacji algorytmu

Implementacja realizująca metodę Newtona.

2.1 Dane wejściowe

Na wejściu program pobiera od użytkownika wartości punktów $P_j(x, y), j = 0, 1, 2, ..., n$, gdzie 'Rok pomiarów' to x, a 'Przyrost CO_2 [Mt]' to y. Realizacja wprowadzenia danych możliwa jest na dwa sposoby. Poprzez bezpośrednie wpisanie wartości lub zaimportowanie danych z pliku JSON.

2.2 Struktury danych

Każdy wielomian to lista współczynników, gdzie poszczególny indeks odpowiada potędze x przy danym współczynniku. Zerowy element obrazuje x^0(x do potęgi zerowej), pierwszy element odpowiada x^1 itd.

PRZYKŁAD : [2,-1,2,-2] oznacza $-2x^3 + 2x^2 - x + 2$.

2.2.1 Wielomian

Program korzysta ze struktury klasy *Polynomial*, która wspiera operacje:

- Dodawanie: Przewidziane dla wyrazów wolnych jak i wielomianów. Odpowiednie elementy list obu wielomianów są dodawane.

  PRZYKŁAD : [0, 0, 3] + [1, 2, 0, 4] = [1, 2, 3, 4]
  oznacza $3x^2 + 4x^3 + 2x + 1 = 4x^3 + 3x^2 + 2x + 1$

- Mnożenie: Podczas mnożenia razy x^n (x do potęgi n-tej) każdy wykładnik jest podnoszony o potęgę n, a lista przesuwa się o n miejsc. W przypadku mnożenia wielomianu razy wielomian, drugi wielomian mnożymy przez każdy składnik pierwszego i sumujemy.

  PRZYKŁAD : [3, 3] · [-3, 3] = [-9,0,9]
  oznacza $(3x + 3)(3x - 3) = 9x^2 - 9$

- Wyliczanie wartości w punkcie: Za pomocą metody Hornera.

2.2.2 Punkt

Wykorzystywane są również proste obiekty modelujące punkty. Zawierają tylko 2 pola x i y.

2.3 Funkcje pomocnicze

W programie wykorzystywane są takie funkcje jak:

1. map: transformacja każdego elementu danej listy
2. reduce: sprowadzenie listy do pojedynczej wartośći, przy pomocy funkcji działającej na kolejnych elementach danej listy
3. filter: filtrowanie danych w liście, spełniających określony warunek

2.4 Przebieg działania

Program wyświetla komunikat: 'Wprowadź listę punktów poniżej'. Jeśli zostały wprowadzone prawidłowe dane, to na bieżąco wyświetlany jest odpowiedni wielomian. Próba ręcznego wprowadzenia nieprawidłowych danych, które weryfikowane są w programie poprzez funkcję *getPoints* skutkuje zignorowaniem błędnego punktu. Podobnie dzieje się w sytuacji pozostawienia pustego pola. Program sprawdza czy 'Rok pamiarów' nie powtarza się.

Dane dostarczone z pliku JSON program sprawdza poprzez funkcję *parsePoints* oraz wyświetla komunikat "Błąd wczytywania pliku" w przypadku niepowodzenia.

Następnie funkcja *recalculate* zajmuje się przekazaniem punktów, w celu dalszego rachunku, a także wyświetleniem wyniku.

Funkcja *getPolynomial* klasy *NewtonEvaluator* zwraca wielomian, licząc wcześniej niewiadome b_0, b_1...; mając tylko jeden punkt zwracana jest od razu wartość y. W przeciwnym wypadku zwracany jest wyliczany wielomian.
Z listy punktów generowana jest lista list, która przypomina piramidę, gdzie każda lista zawiera o jeden element mniej. Każda lista stanowi grupę punktów, na podstawie których wyliczane są niewiadome $b_0, b_1, ..., b_n$.

$$b_n = \frac{y_n - P(x_{n-1})}{(x_n - x_0)(x_n - x_1)...(x_n - x_{n-1})}$$

$$P(x_n) = b_0 - b_1(x_n - x_0) - b_2(x_n - x_0)(x_n - x_1) - ... - b_{n-1}(x_n - x_0)(x_n - x_1)...(x_n - x_{n-1})$$

Wykorzystywane są funkcje rekurencyjne *getB* oraz P , w których przypadkiem bazowym jest lista z jednym punktem, dla której zwracana jest wartość y tego punktu. W przeciwnym wypadku obie te funkcje dostają listę punktów do xn i wyliczają wyniki na podstawie powyższych wzorów. Korzystają z funkcji pomocnicznej *pointProduct*, która przekształca listę punktów na listę wielomianów w postaci $x_n \to (x - x_n)$. Następnie wylicza ich iloczyn.

Aby wyznaczyć wielomian musimy zsumować niewiadomą b_0 i iloczyn pozostałych niewiadomych z odpowiadającymi im grupami wielomianowymi.

Z uwagi na występowanie rekurencji i dużą powtarzalność wykonywanych operacji w najbardziej kluczowych fragmentach, wykorzystywana jest technika spamiętywania.

Wynikiem działania programu jest wielomian interpolacyjny () obrazujący oszacowanie tempa przyrostu dwutlenku węgla.

2.5 Najważniejsze fragmenty programu

newtonEvaluator.js

```
class NewtonEvaluator {
  constructor(points) { this.points = points }

  getPolynomial() {
    if (this.points.length === 1) return new Polynomial([this.points[0].y])

    const [b0, ...bs] = this.getBs()
    const polynomials = init(this.points)
      .map(Polynomial.point)
      .map(getListSlicesFromStart)
      .map(group => group.reduce(Polynomial.product))

    return polynomials
      .map((polynomial, index) => polynomial.multiply(bs[index]))
      .reduce(Polynomial.sum)
      .add(b0)
  }

  getBs() {
    const pointProduct = (points) => points
      .map(Polynomial.point)
      .reduce(Polynomial.product)

    const P = memoized((points) => {
      if (points.length === 1) return new Polynomial([points[0].y])
      else {
        const rest = init(points)
        return P(rest).add(pointProduct(rest).multiply(getB(points)))
      }
    })

    const getB = memoized((points) => {
      if (points.length === 1) return points[0].y
      else {
```

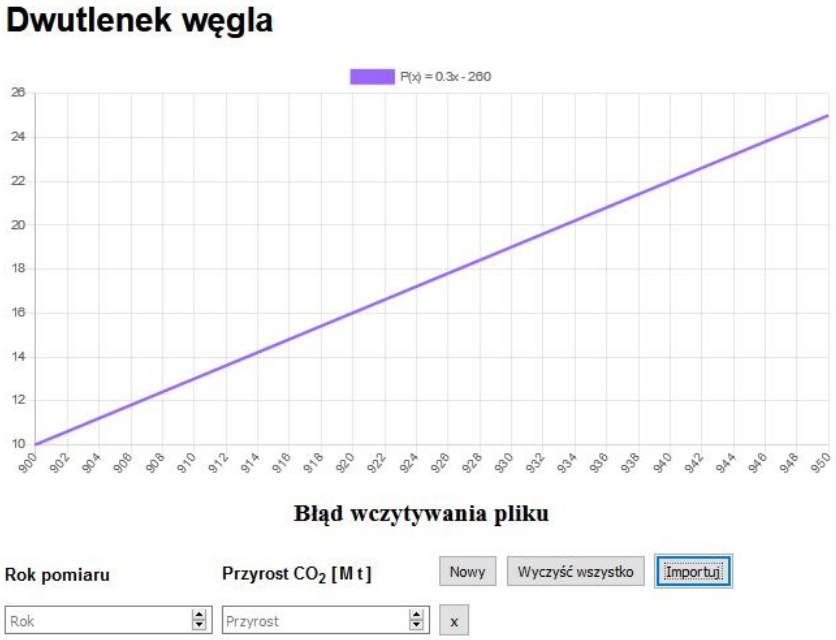
```
        const [current, rest] = [last(points), init(points)]
        return (current.y - P(rest).at(current.x)) /
         pointProduct(rest).at(current.x)
      }
    })

    return this.points.map(getListSlicesFromStart).map(getB)
  }
}
```

polynomial.js

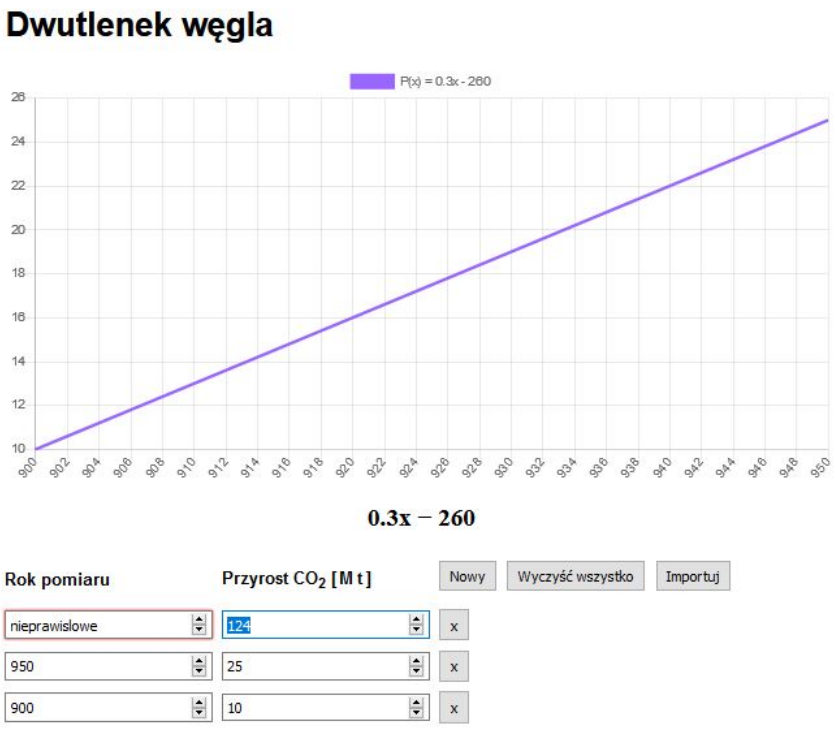
```
class Polynomial {
  static one  = new Polynomial([1])
  static zero = new Polynomial([0])
  static point = (p) => new Polynomial([-p.x, 1])
  static product = (acc, polynomial) => acc.multiply(polynomial)
  static sum     = (acc, polynomial) => acc.add(polynomial)

  terms = []

  constructor(terms) {
    const trimmed = terms => {
      if (terms.length === 0) return [0]
      else if (last(terms) !== 0) return terms
      else return trimmed(init(terms))
    }

    if (terms.length === 0) throw new Error('Term list is empty')
    else this.terms = trimmed(terms)
  }

  at(x) {
    return this.terms.reduceRight((acc, term) => acc * x + term)
  }

  add(that) {
```

```
    if (typeof that == 'number') return this.addNumber(that)
    else return this.addPolynomial(that)
  }

  addNumber(that) {
    const [head, ...tail] = this.terms
    return new Polynomial([head + that, ...tail])
  }

  addPolynomial(that) {
    const [longer, shorter] = (this.terms.length >= that.terms.length)
    ? [this, that]
    : [that, this]


    const addedTerms = zip(longer.terms, shorter.terms).map(pairSum)

    return new Polynomial(addedTerms)
  }

  multiply(that) {

    const padLeft = (arr, padding) => new Array(padding).fill(0).concat(arr)

    const multiplyByTerm = (thisTerm, power, that) => {
      const multiplied = that.terms.map(thatTerm => thatTerm * thisTerm)
      return padLeft(multiplied, power)
    }

    if (typeof that === "number") return this.multiply(new Polynomial([that]))
    else return this.terms
      .map((term, power) => multiplyByTerm(term, power, that))
      .map(terms => new Polynomial(terms))
      .reduce(Polynomial.sum)
  }

}
```

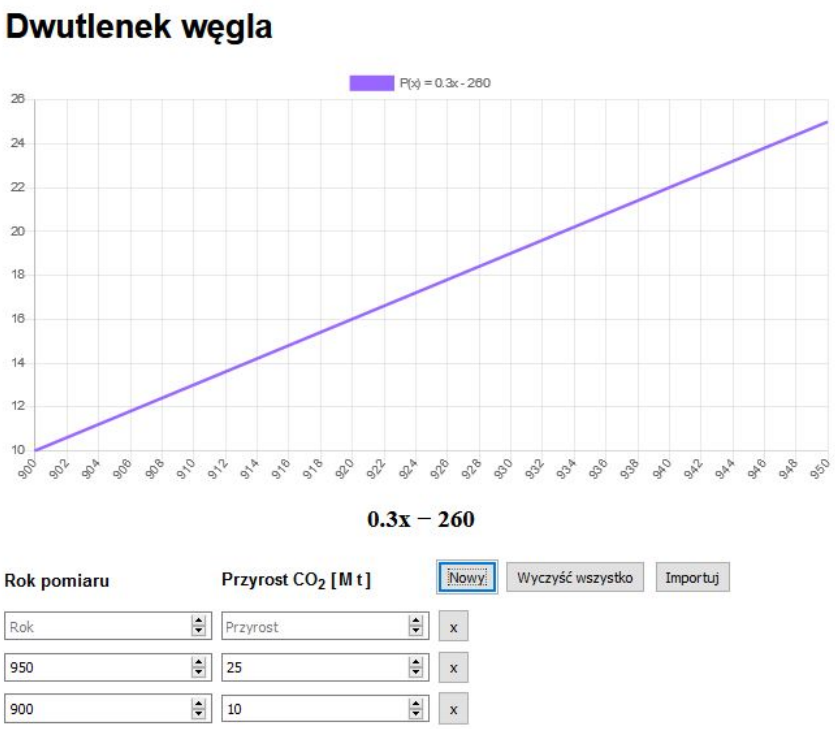
2.6 Widok działania programu



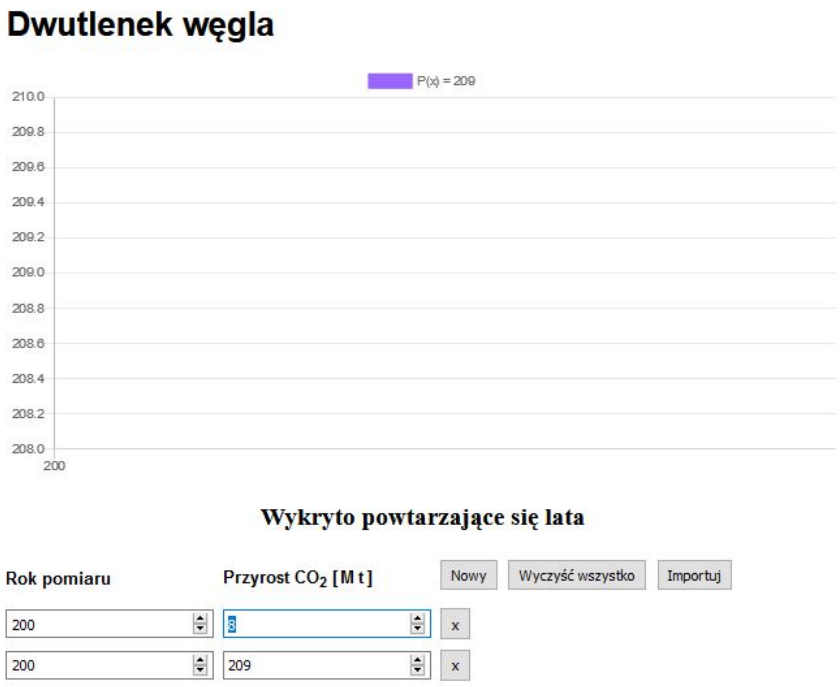
Rysunek 1: Błędne zaimportowanie pliku



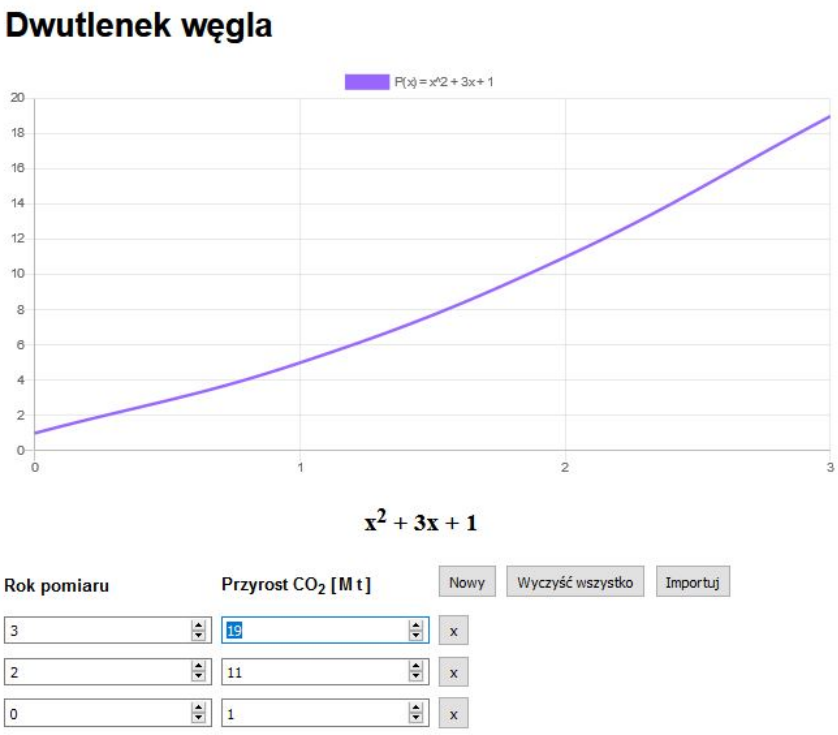
Rysunek 2: Nieprawidłowo wprowadzone dane



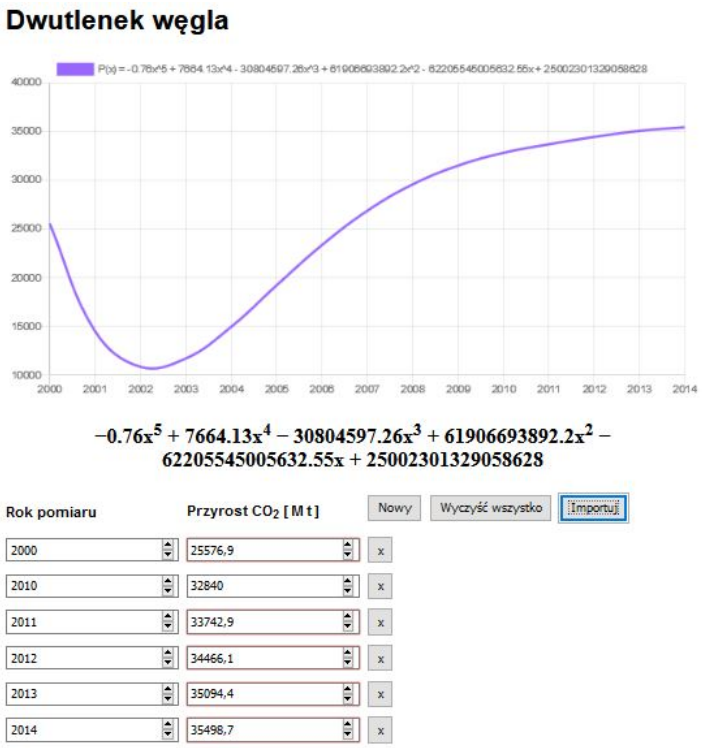
Rysunek 3: Ignorowane puste pole



Rysunek 4: Duplikat



Rysunek 5: Przykład opisany w dokumencie



Rysunek 6: Przykład z rzeczywistymi danymi