**Projekt zaliczeniowy: metody numeryczne**

Autor: **Filip Dąbrowski**

Nr indeksu: **341057**

Spis treści

[Wstęp 3](#_Toc216075319)

[Uruchomienie programu lokalnie 3](#_Toc216075320)

[Wybór technologii i architektury 3](#_Toc216075321)

[Komentowanie kodu 4](#_Toc216075322)

[Struktura programu / Architektura 5](#_Toc216075323)

[Katalog główny (root) 5](#_Toc216075324)

[readme.md 5](#_Toc216075325)

[requirements.txt 5](#_Toc216075326)

[parametry.py 6](#_Toc216075327)

[main.py 7](#_Toc216075328)

[Katalogi na wykresy 8](#_Toc216075329)

[Pliki \_\_init\_\_.py 8](#_Toc216075330)

[Katalog kod\_zrodlowy 9](#_Toc216075331)

[narzędzia.py 9](#_Toc216075332)

[Wymuszenia / funkcje\_wymuszen.py 10](#_Toc216075333)

[Całkowanie / funkcje\_calkowania.py 11](#_Toc216075334)

[Interpolacja / funkcje\_interpolacyjne.py 11](#_Toc216075335)

[Metody numeryczne / metoda\_eulera.py 13](#_Toc216075336)

[Metody numeryczne / ulepszona\_metoda\_eulera.py 14](#_Toc216075337)

[Metody numeryczne / inne\_metody.py / metoda\_bisekcji() 15](#_Toc216075338)

[Metody numeryczne / inne\_metody.py / metoda\_siecznych() 16](#_Toc216075339)

[Metody numeryczne / inne\_metody.py / metoda\_quasi\_newton() 17](#_Toc216075340)

[Obwody / liniowy.py 18](#_Toc216075341)

[Obwody / nieliniowy.py 19](#_Toc216075342)

[Opis modelu matematycznego oraz modelu matematycznego 20](#_Toc216075343)

[Część 1: symulator obwodu ze sprzężeniem indukcyjnym 20](#_Toc216075344)

[Model matematyczny 20](#_Toc216075345)

[Model numeryczny 20](#_Toc216075346)

[Implementacja 21](#_Toc216075347)

[Część 2: symulator obwodu z nieliniową indukcyjnością 23](#_Toc216075348)

[Model matematyczny 23](#_Toc216075349)

[Model numeryczny 23](#_Toc216075350)

[Implementacja 24](#_Toc216075351)

[Część 3: obliczanie mocy czynnej na rezystorach 26](#_Toc216075352)

[Model numeryczny 26](#_Toc216075353)

[Implementacja 27](#_Toc216075354)

[Część 4: znajdowanie częstotliwości dla mocy 406 W 29](#_Toc216075355)

[Model matematyczny 29](#_Toc216075356)

[Model numeryczny 29](#_Toc216075357)

[Implementacja 30](#_Toc216075358)

[Interpretacja wyników i odpowiedzi na pytania z projektu 32](#_Toc216075359)

[Cześć 1 32](#_Toc216075360)

[Wymagania projektowe 32](#_Toc216075361)

[Wymaganie nr 1: kod źródłowy 32](#_Toc216075362)

[Wymaganie nr 2: wykresy 32](#_Toc216075363)

[Dodatkowe wykresy weryfikacyjne 34](#_Toc216075364)

[Część 2 37](#_Toc216075365)

[Wymagania projektowe 37](#_Toc216075366)

[Raporty oraz interpretacja wyników 37](#_Toc216075367)

[Wykres charakterystyk M(u) – porównanie z projektem 38](#_Toc216075368)

[Najważniejsze różnice między metodami / wnioski 40](#_Toc216075369)

[Dodatkowa weryfikacja wartości z tabeli 40](#_Toc216075370)

[Interpretacja tabeli 40](#_Toc216075371)

[Część 3 42](#_Toc216075372)

[Wymagania projektowe 42](#_Toc216075373)

[Wyniki 42](#_Toc216075374)

[Interpretacja 43](#_Toc216075375)

[Cześć 4 44](#_Toc216075376)

[Wymagania projektowe 44](#_Toc216075377)

[Wyniki 44](#_Toc216075378)

[Podsumowanie 46](#_Toc216075379)

[Spis ilustracji 47](#_Toc216075380)

# Wstęp

Celem projektu była implementacja symulatora obwodu elektrycznego ze sprzężeniem indukcyjnym, umożliwiającego analizę stanu nieustalonego dla różnych sygnałów wymuszających. W ramach pracy opracowałem symulator, który realizuje wszystkie wymagania stawiane w projekcie.

## Uruchomienie programu lokalnie

Kroki uruchomienia:

1. Rozpakowanie pliku ZIP w dowolnej lokalizacji
2. Przejście do katalogu projektu za pomocą terminala
3. Utworzenie i aktywacja środowiska:

python -m venv venv

venv\Scripts\activate # Windows

# lub: source venv/bin/activate # Linux/Mac

1. Instalacja zależności

pip install -r requirements.txt

1. Możemy uruchomić aplikację

python main.py

Po otworzeniu pliku aplikacja wyświetla menu główne z którego możemy wybrać dowolny moduł (lub wykonanie wszystkich sekwencyjnie). Wykresy zapisywane są w podkatalogach gdzie w katalogu głównym projektu (root).

## Wybór technologii i architektury

Jako student informatyki, zdecydowałem się na implementację symulatora w języku Python, który uważam za przydatny w mojej przyszłej karierze zawodowej ze względu na szerokie zastosowanie w zagadnieniach z tematyki infrastruktury chmurowej, w której to chciałbym w przyszłości pracować. Uznałem więc projekt za doskonałą okazję, do potrenowania języka Python. Program został zaprojektowany jako modułowa aplikacja z tekstowym interfejsem użytkownika, co umożliwia interaktywne eksperymenty. Pełny kod źródłowy projektu jest dostępny w repozytorium GitHub: <https://github.com/filipdbr/symulator-obwodu-python>, a instrukcje konfiguracji środowiska i uruchomienia znajdują się w pliku README.md. Repozytorium jest prywatne. Jeżeli Pan Doktor chciałby je zobaczyć, z chęcią zaakceptuję prośbę o dostęp (proszę zawnioskować na GitHub). Instrukcję uruchomienia programu lokalnie umieszczę także w tym raporcie, w kolejnym paragrafie.

Wybrałem modularną architekturę ze względu na jej zgodność ze współczesnymi standardami inżynierii oprogramowania. Uznałem ją też za dobrze nadającą się do projektu, ponieważ takie metody jak chociażby metoda Eulera (prosta i ulepszona) są wykorzystywane w więcej niż jednej części projektu. Dzięki temu, nie istnieje potrzeba kopiowania i wklejania metody w kilku miejscach. Takie podejście ułatwia debugowanie, testowanie jednostkowe oraz wprowadzanie modyfikacji, a także zapewnia czystszy i bardziej elegancki kod poprzez separację odpowiedzialności. Choć kod źródłowy może być początkowo nieco trudniejszy do zrozumienia ze względu na rozproszenie logiki między niezależnymi modułami funkcjonalnymi, ostatecznie prowadzi to do lepszej organizacji i łatwiejszego rozwoju aplikacji.

Przy pracy z projektem nie stosowałem żadnych wbudowanych funkcji poza podstawowymi funkcjami pakietu *numpy*, takimi jak sin, pi itd. Do stworzenia wykresów wykorzystałem bibliotekę *matplotlib*. Zaimplementowane metody są więc zgodne z metodologią z podręcznika.

## Komentowanie kodu

Zdecydowałem się komentować kod jedynie w kluczowych momentach, ewentualnie zaznaczając bloki funkcjonalne. Podjąłem taką decyzję aby nie przesycić kodu komentarzami. Jednocześnie, starałem się stworzyć „samodokumentujący” się kod poprzez opisowe nazwy zmiennych oraz jawną logikę. Wyjątkiem jest zwykle niejawny import parametrów z pliku parametry.py – w tym przypadku zastosowałem czasem (z lenistwa) import całych modułów poprzez import \*. Uznałem jednak, że skoro przygotuję dosyć obszerny raport wyjaśniający działanie programu, nie ma sensu dwukrotnie wykonywać tej pracy. Ponadto kod z mniejszą ilością komentarzy wydaje mi się być czytelniejszy i bardziej elegancki.

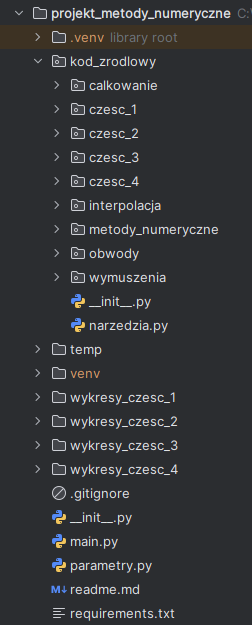
# Struktura programu / Architektura

Rozdział ten stanowi kluczowe wprowadzenie do konstrukcji programu, umożliwiając zrozumienie logicznego podziału odpowiedzialności. Takie przybliżenie architektury aplikacji jest niezbędne, aby w pełni przyswoić prezentowane w kolejnym rozdziale zasady modelowania matematycznego.

W moim programie każdy moduł odpowiada za konkretny aspekt funkcjonalności symulatora. Mam jednocześnie świadomość, że modularność mogłaby być jeszcze większa. Pisząc program czułem, że istnieje możliwość stworzenia funkcji tworzącej wykresy (zauważam tam dużą powtarzalność), jednak zabrakło mi czasu, a też nie to było sensem zadania.

## Katalog główny (root)

Poniżej przedstawiam strukturę katalogów:



Rysunek 1: struktura katalogów programu

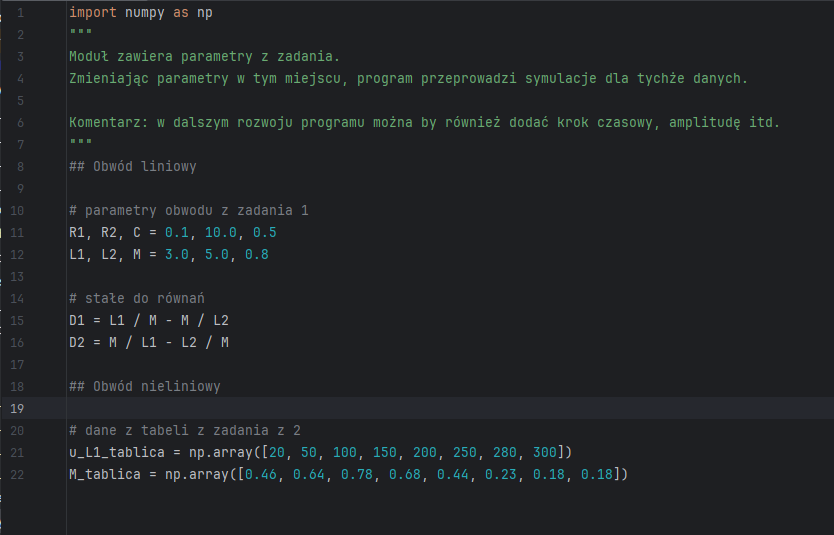
### readme.md

Plik markdown zawierający informacje o projekcie oraz instrukcje w jaki sposób stworzyć środowisko i uruchomić program. Standardowy plik umieszczany w dokumentacji każdego projektu.

### requirements.txt

Plik tekstowy używany do szybkiego importu zależności. Wygenerowałem go za pomocą komendy *pip freeze > requirements.txt*. Z jego użyciem możemy łatwo stworzyć środowisko projektu wywołując komendę *pip install -r requirements.txt* w cmd.

### parametry.py



Rysunek 2: parametry.py

Zdecydowałem się wyodrębnić moduł, w którym umieściłem kluczowe parametry z projektu. Inne moduły importują parametry, dzięki czemu są one zdefiniowane jedynie tutaj. Umożliwia to łatwe zarządzanie programem. Jeżeli chcemy wygenerować symulacje dla innych wartości parametrów, wystarczy zamienić je w tym miejscu.

Nie ma jednak możliwości dodawania parametrów ani zmiany ich nazw. Parametry obwodu liniowego muszą pozostać w takiej formie w jakiej są, możemy jedynie zmienić przypisane im wartości. W przypadku obwodu nieliniowego, tablice mogą zawierać dowolną liczbę elementów (natomiast w innych module występuje ograniczenie do u\_max = 320 więc dodawanie wartości powyżej 320 V nie będzie miało sensu.

Istnieje możliwość zwiększenia modułu, dodania tam takich zmiennych jak krok czasowy czy amplituda. Ja wybrałem rozwiązanie uproszczone i nie wprowadzałem już większej liczby parametrów w tym pliku.

### main.py



Głowna funkcja programu. Zapewnia prosty interfejs użytkownika w formie tekstowej. Daje możliwość wybrania 5 opcji – po jednej dla każdej części zadania oraz wywołanie wszystkich części na raz (wybór 5). Po zakończeniu działania program zapisze odpowiednie wykresy w podkatalogach w tym samym katalogu z którego wywoływany jest *main* (czyli w root). Będą to katalogi . Zasadniczo jedyną funkcją jaką wykonuje moduł jest wywoływanie funkcji wykonaj\_czesc\_1, wykonaj\_czesc\_2 itd.

Interfejs jest prosty, nie zawiera obsługi błędów poza jedynym, którym jest wybranie innej wartości niż wspomniane. Działa na prostej pętli while true.

### Katalogi na wykresy

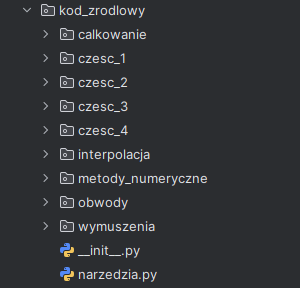
4 katalogi na wykresy. Zdecydowałem się aby były one widoczne i uzupełnione dla wygody Oceniającego, jednak jeżeli by tych folderów nie było, to program stworzy je i uzupełni odpowiednimi wykresami po ukończeniu swojego działania.

### Pliki \_\_init\_\_.py

Pliki \_\_init\_\_.py są niezbędne w Pythonie, ponieważ definiują katalog jako pakiet, co jest zgodne ze standardami tego języka. Działają one podobnie do plików nagłówkowych w innych językach, umożliwiając Pythonowi importowanie modułów zawartych w danym katalogu. Wykonanie kodu w \_\_init\_\_.py pozwala mi na podniesienie kluczowych funkcji, dzięki czemu są one łatwiej i czytelniej dostępne bezpośrednio z poziomu pakietu w innych modułach. Zasada ta dotyczy wszystkich plików \_\_init\_\_.py w projekcie, dlatego zrezygnuje z omawiania każdego z nich.

## Katalog kod\_zrodlowy

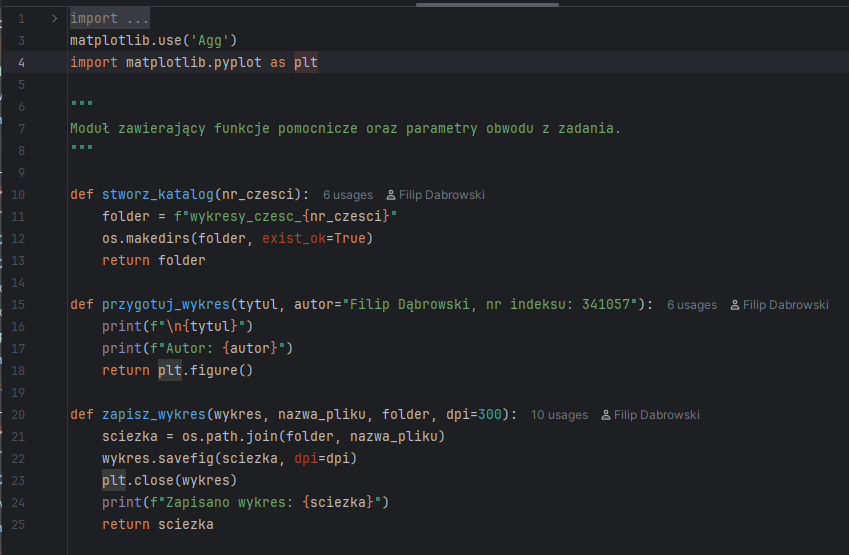
Najważniejszy katalog zawierający kod źródłowy programu:



Rysunek 3: struktura katalogu kod\_zrodlowy

Stworzyłem w nim osobne moduły funkcjonalne w odpowiednich podkatalogach. Dodatkowo widzimy 1 plik, który nazwałem narzędzia.

### narzędzia.py



Rysunek 4: narzędzia.py

Zawiera 3 funkcje pomocnicze:

1. stworz\_katalog tworzy katalogi na wykresy,
2. przygotuj\_wykres ma dwa główne cele: inicjalizuje i zwraca nowy obiekt Figure z biblioteki Matplotlib, który służy jako główny kontener do rysowania wykresu. Jednocześnie funkcja ta odpowiada za wyświetlenie metadanych (tytułu wykresu i informacji o autorze) w konsoli
3. zapisz\_wykres wykonuje zapis. Jeżeli chcielibyśmy zmienić domyślne miejsce zapisuj wykresów, należałoby podstawić jakąś inną ścieżkę.

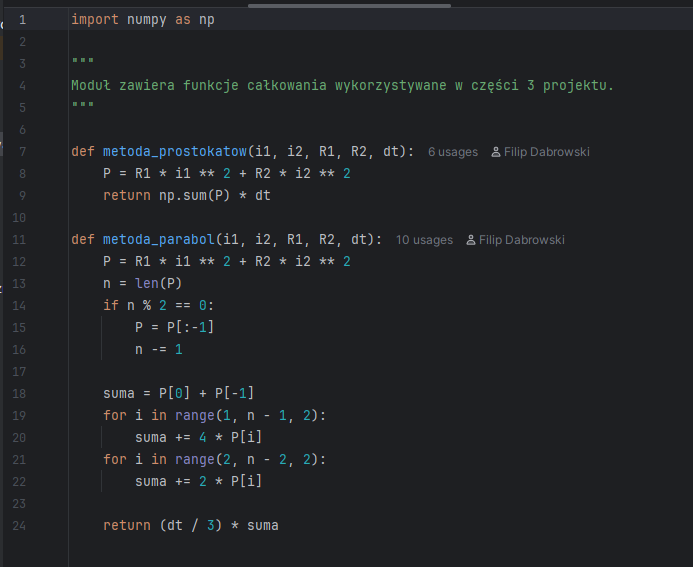
Przejdźmy teraz do podkatalogów. Większość z nich zawiera jedynie 1 plik (nie wliczając \_\_init\_\_.py). Wyjątkiem jest katalog *metody\_numeryczne*. Wybrałem nazwy nieco niezręcznie, ponieważ w katalogach całkowanie czy interpolacja również zawierają się metody numeryczne, jednak zdecydowałem się już na koniec tego nie zmieniać bojąc się ewentualnych komplikacji. Dokładne przełożenie się modelu matematycznego na implementację numeryczną wyjaśnię w rozdziale Model matematyczny obwodu i Model matematyczny obwodu.

### Wymuszenia / funkcje\_wymuszen.py

Moduł zawiera wszystkie funkcje wymuszeń używane w projekcie. Dodatkowo zaimplementowałem słownik powalający w prosty i intuicyjny sposób odwoływać się do metod.



### Całkowanie / funkcje\_calkowania.py



Rysunek 5: funkcje\_calkowania.py

Plik zawiera 2 metody które zostały wykorzystane w części 3 projektu (oraz metoda parabol również w części 4):

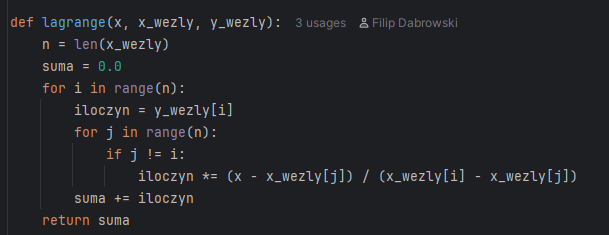
1. metoda\_prostokatow() - przybliża energię wydzieloną na rezystorach poprzez sumowanie chwilowej mocy (P) pomnożonej przez krok czasowy (dt) co jest numerycznym szacowaniem całki z mocy w danym interwale.
2. metoda\_parabol() - szacuje energię wydzieloną w obwodzie, stosując regułę Simpsona (metodę parabol). Polega na obliczeniu chwilowej mocy (P), dostosowaniu danych, by ich liczba była nieparzysta, a następnie zastosowaniu ważonej sumy wartości P (punkty środkowe mają wagę 4, a pozostałe wewnętrzne wagę 2) i pomnożeniu jej przez stałą dt/3 w celu uzyskania dokładniejszego przybliżenia całki (całkowitej energii).

### Interpolacja / funkcje\_interpolacyjne.py

Zawiera 3 metody:

#### legrange()

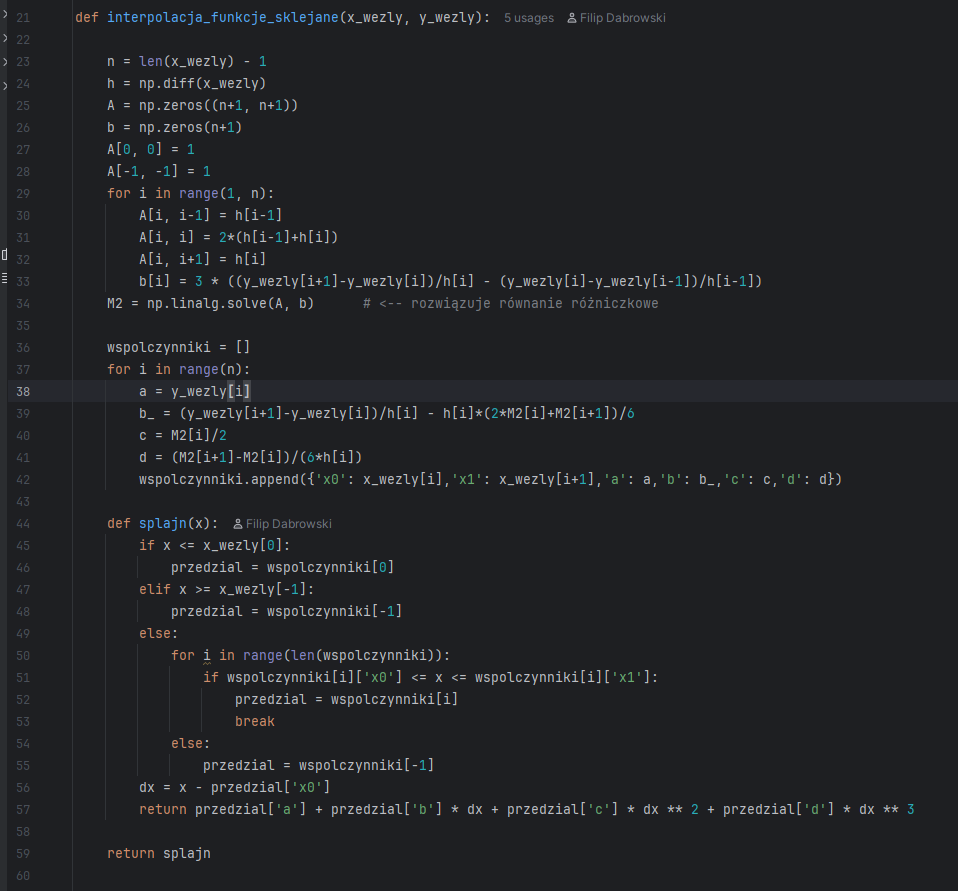
Interpolacja wielomianem Legrange’a:



Rysunek 6: legrange()

Implementuje wielomian interpolacyjny Lagrange'a - polega na znajdowaniu unikalnego wielomianu, który dokładnie przechodzi przez wszystkie podane węzły danych (x\_wezly, y\_wezly), pozwalając na oszacowanie wartości y dla dowolnego argumentu x wewnątrz ich zakresu.

#### Interpolacja / interpolacja\_funkcje\_sklejane()



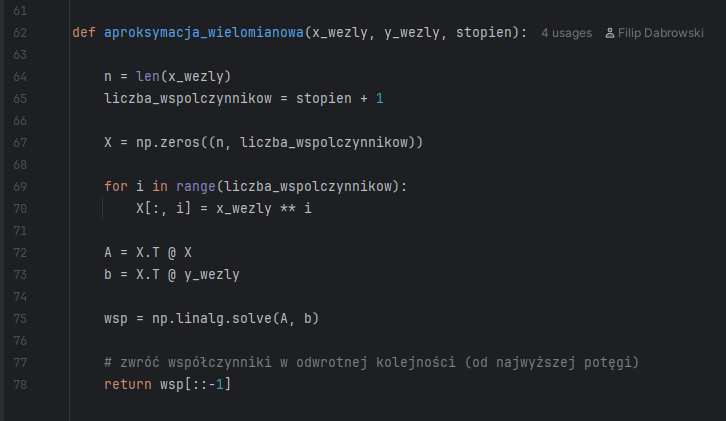
Rysunek 7: interpolacja\_funkcje\_sklejane()

Metoda implementuje interpolację funkcjami sklejanymi (splajnami) trzeciego stopnia, które minimalizują krzywiznę i zapewniają gładkie przejścia między węzłami. Najpierw konstruuje i rozwiązuje układ równań liniowych , aby wyznaczyć wartości drugich pochodnych M2 w węzłach. Następnie, na podstawie tych pochodnych, oblicza współczynniki wielomianów sześciennych dla każdego przedziału i zwraca funkcję wewnętrzną *splajn*(), która oblicza wartość y dla dowolnego x za pomocą odpowiednio dobranej krzywej sześciennej.

W funkcji tej używam metody z pakietu numpy*: np.linalg.solve()* która rozwiązuje równanie różniczkowe. Jest to metoda nieco wykraczająca poza podstawowe – mam nadzieje że takie rozwiązanie nie będzie karane.

#### Interpolacja / aproksymacja\_wielomianowa()

Nieco niefortunna nazwa ponieważ katalog nazwałem interpolacja, a wewnątrz znajduje się ekstrapolacja. Przepraszam za taką niespójność, ale nie chcę już tego zmieniać. Metoda zaimplementowana jest następująco:



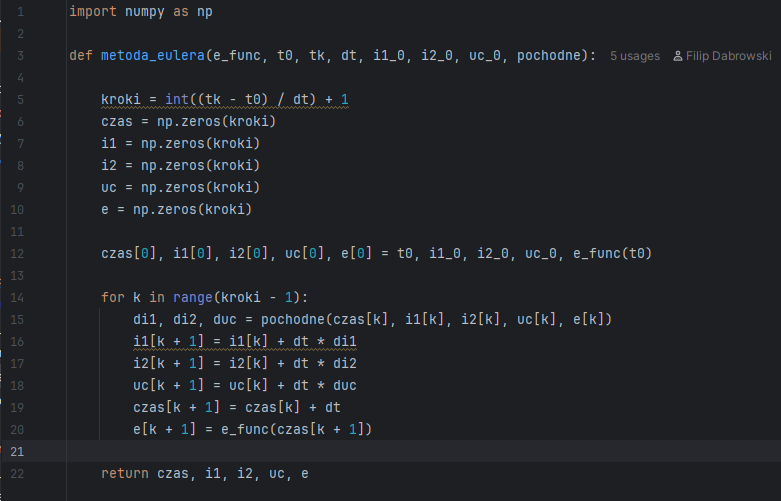
Rysunek 8: aproksymacja\_wielomianowa()

Funkcja realizuje metodę najmniejszych kwadratów w celu znalezienia wielomianu określonego stopnia, który najlepiej pasuje do zadanego zbioru węzłów (x\_wezly, y\_wezly). Osiąga to poprzez konstrukcję i rozwiązanie układu równań liniowych , wynikającego z macierzy Vandermonde'a (X) i jej transpozycji, co minimalizuje sumę kwadratów błędów między wielomianem a danymi.

Funkcję zaimplementowałem w ten sposób, że przyjmuj stopień jako argument. Dzięki temu nie ma potrzeby tworzenia oddzielnych funkcji dla aproksymacji wielomianowej 3 i 5 stopnia – wystarczy podstawić 3 lub 5 jako argument. Możemy w ten sposób wykonać aproksymację wielomianową dowolnego rzędu. W metodzie nie zaimplementowałem obsługi błędów, ponieważ wiedziałem że argumenty w moim kodzie będą właściwe (warunek: ). Gdybym jednak rozwijał „konsumencki” symulator, należałoby taką obsługę błędów wprowadzić.

### Metody numeryczne / metoda\_eulera.py

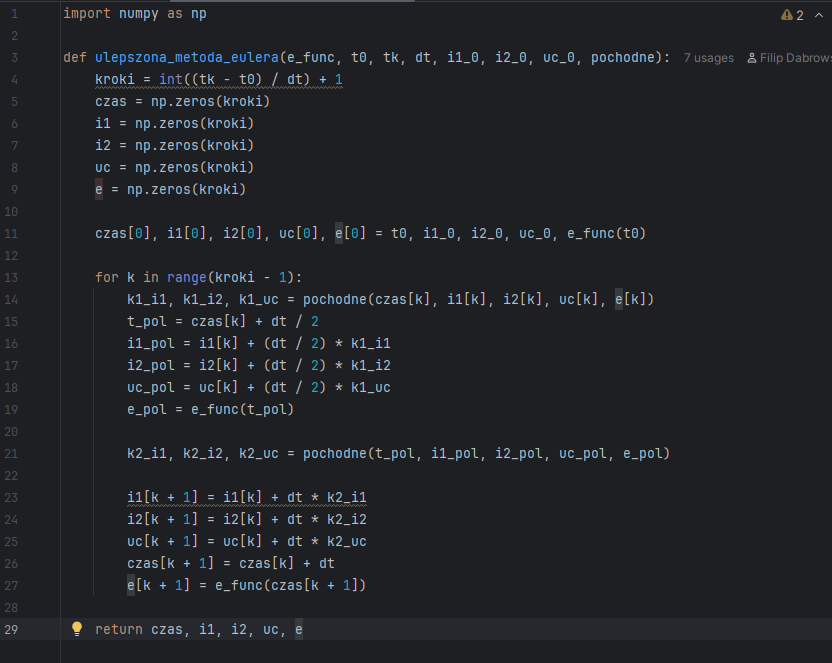
Funkcja realizuje numeryczne rozwiązywanie równań różniczkowych zwyczajnych, wykorzystując wartości pochodnych obliczone w bieżącym kroku czasowym () do przybliżenia wartości funkcji w następnym kroku (). Metoda działa iteracyjnie: w każdej pętli aktualizuje stan zmiennych () poprzez dodanie iloczynu pochodnej (zmiany chwilowej) i kroku czasowego (dt).



Rysunek 9: metoda\_eulera()

### Metody numeryczne / ulepszona\_metoda\_eulera.py

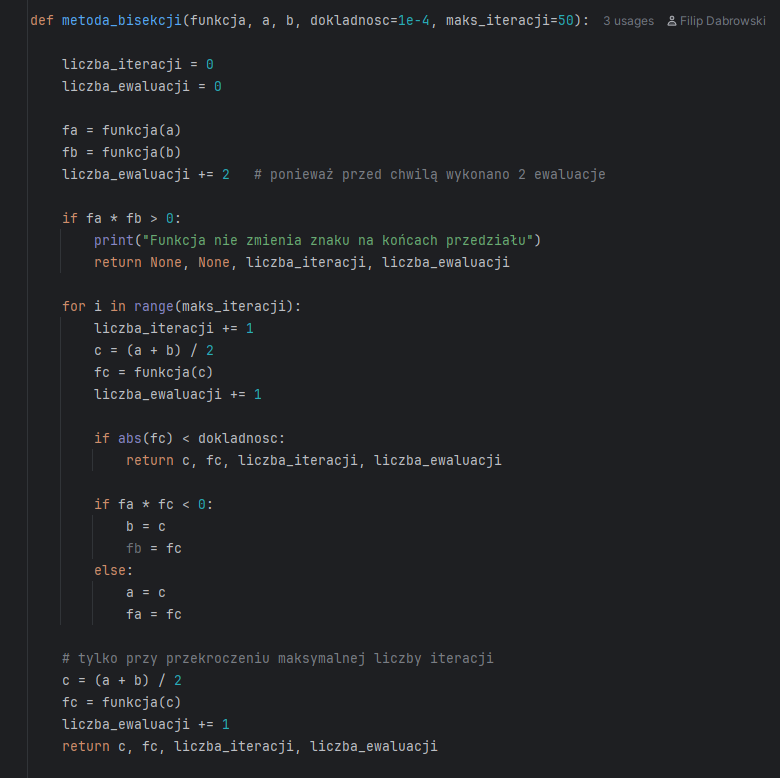
Funkcja numerycznie rozwiązuje równania różniczkowe, znacząco poprawiając precyzję względem zwykłej metody Eulera. Osiąga to, obliczając najpierw wstępne przybliżenie wartości pochodnych w punkcie środkowym interwału (współczynnik k1), a następnie używając tych poprawek do obliczenia dokładniejszego oszacowania pochodnych w środku interwału (k2), które finalnie wykorzystuje do uaktualnienia zmiennych stanu () w kroku :



Rysunek 10: ulepszona\_metoda\_eulera()

### Metody numeryczne / inne\_metody.py / metoda\_bisekcji()

Metoda służy do znajdowania pierwiastków (miejsc zerowych) danej funkcji. Działanie metody polega na wielokrotnym dzieleniu na pół zadanego przedziału [a, b], w którym funkcja zmienia znak, a następnie wybieraniu tej połówki, na której zmiana znaku nadal występuje, co prowadzi do stopniowego zawężania przedziału, aż do osiągnięcia zadanej dokładności lub maksymalnej liczby iteracji:



Rysunek 11: metoda\_bisekcji()

### Metody numeryczne / inne\_metody.py / metoda\_siecznych()

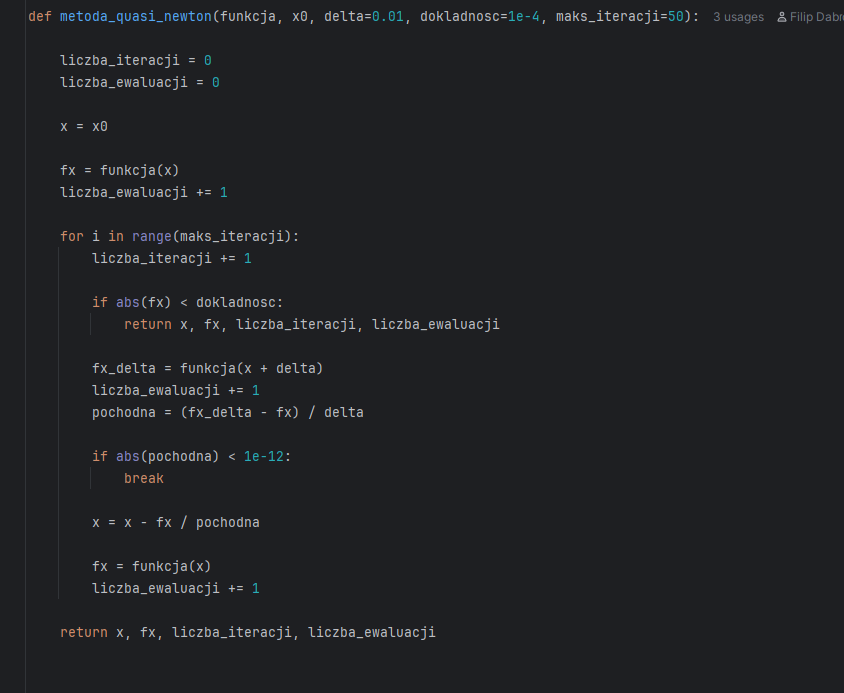
Funkcja służy do znajdowania pierwiastków funkcji poprzez aproksymację jej przebiegu za pomocą kolejnych siecznych. Metoda działa iteracyjnie: na podstawie dwóch ostatnich punktów oraz wyznacza punkt przecięcia siecznej z osią x (nowy punkt ), a następnie używa go jako kolejnego przybliżenia pierwiastka, powtarzając proces aż do osiągnięcia zadanej dokładności.



Rysunek 12: metoda\_siecznych()

### Metody numeryczne / inne\_metody.py / metoda\_quasi\_newton()

Ta funkcja implementuje metodę quasi-Newtona do znajdowania pierwiastków funkcji. Działa iteracyjnie: wykorzystuje bieżącą wartość funkcji oraz numerycznie przybliżoną pochodną (obliczoną za pomocą małej różnicy i parametru ) do wyznaczenia następnego, lepszego przybliżenia pierwiastka, powtarzając proces aż do osiągnięcia zadanej dokładności.

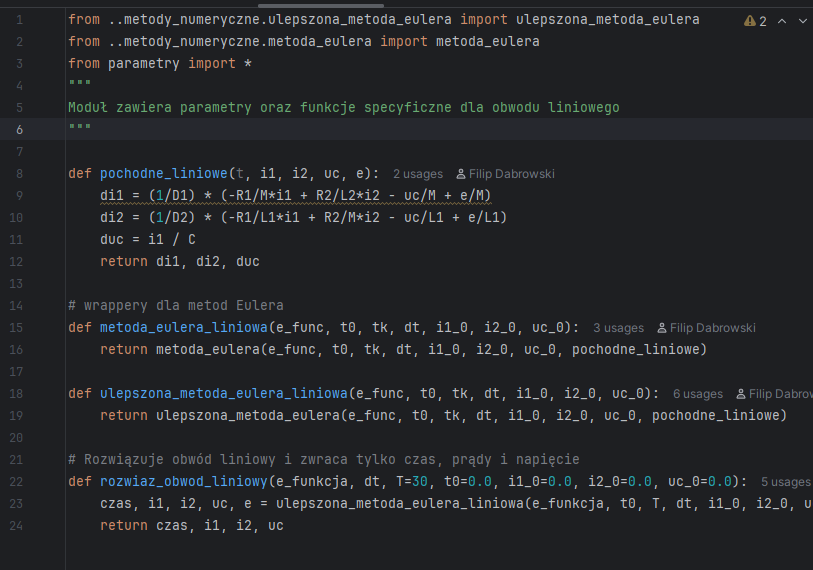


Rysunek 13: metoda\_quasi\_newton()

### Obwody / liniowy.py

Moduł jest zestawem narzędzi do symulacji obwodu liniowego i jest odpowiedzialny za zdefiniowanie modelu matematycznego oraz udostępnienie interfejsów do jego numerycznego rozwiązywania.

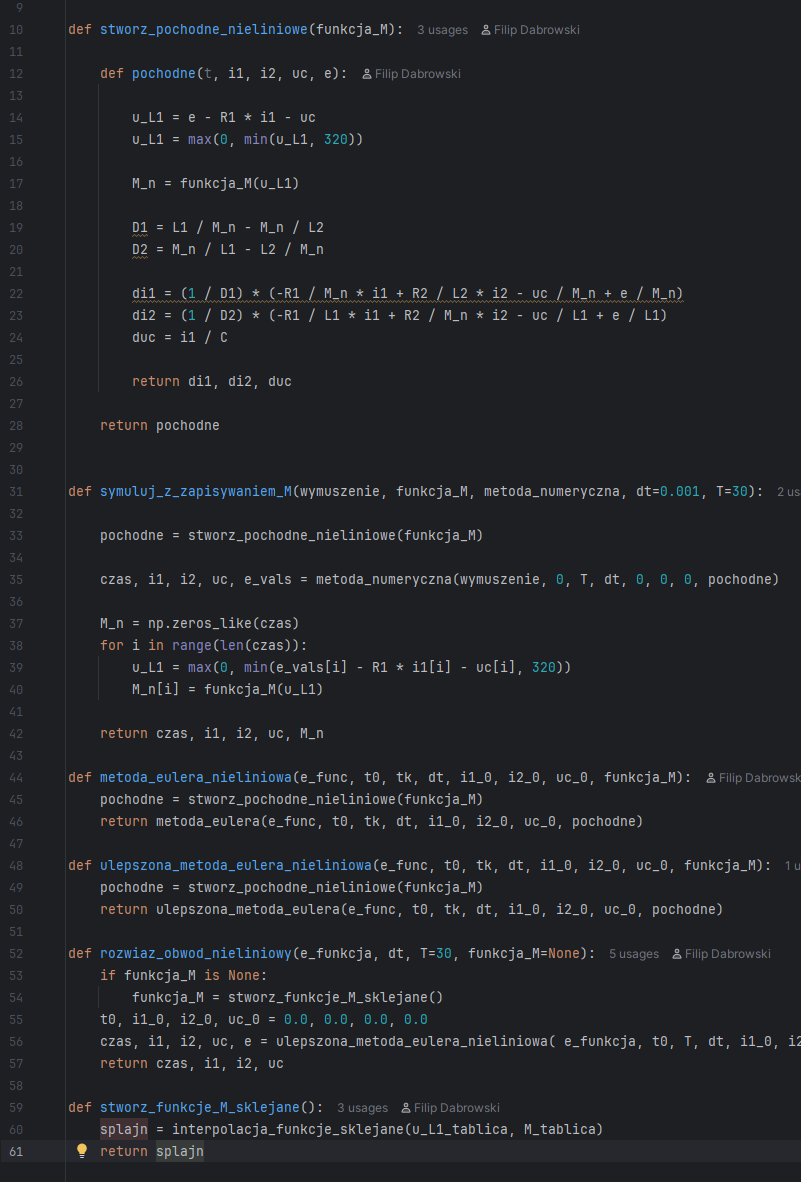
Moduł definiuje funkcję *pochodne\_liniowe()*, która implementuje układ równań różniczkowych opisujących dynamikę liniowego obwodu elektrycznego w oparciu o globalnie zaimportowane parametry. Następnie udostępnia wygodne funkcje opakowujące (wrappery) (*metoda\_eulera\_liniowa()* i *ulepszona\_metoda\_eulera\_liniowa()*), które stosują te specyficzne pochodne do standardowych numerycznych metod rozwiązywania równań różniczkowych. Finalnie funkcja *rozwiaz\_obwod\_liniowy()* wykonuje pełną symulację za pomocą ulepszonej metody Eulera.



Rysunek 14: liniowy.py

### Obwody / nieliniowy.py

Ten moduł jest z kolei zestawem narzędzi do symulacji obwodu nieliniowego Moduł koncentruje się na symulacji obwodu nieliniowego, którego kluczową nieliniowością jest indukcyjność wzajemna (M) zależna od napięcia. Centralną rolę pełni funkcja *stworz\_pochodne\_nieliniowe()*, która dynamicznie generuje funkcję pochodnych dla obwodu, uwzględniając na bieżąco wartość M uzyskaną z interpolacji. Pozostałe funkcje modułu służą jako interfejsy do wywoływania metod Eulera i ulepszonej metody Eulera z tymi nieliniowymi pochodnymi, co ostatecznie pozwala na dokładne śledzenie ewolucji nieliniowego obwodu w czasie.



Rysunek 15: nieliniowy.py

# Opis modelu matematycznego oraz modelu matematycznego

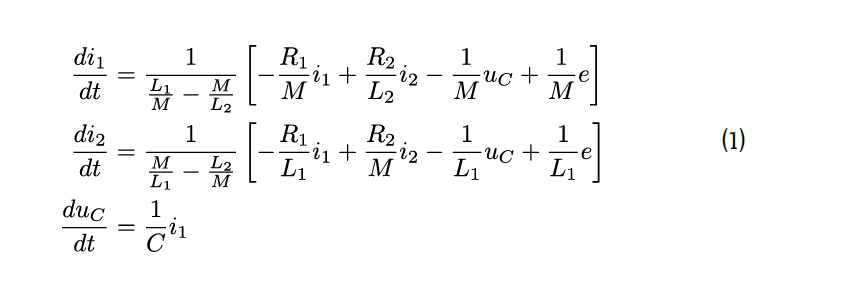
Przyjąłem następującą strukturę raportu: dla każdej części zadania, równolegle przedstawię model matematyczny oraz model numeryczny wraz z implementacją. Takie podejście ma na celu wyraźne ukazanie przełożenia teoretycznego modelu na praktyczny algorytm oraz podkreślenie, jak metody numeryczne łączą matematykę i informatykę.

## Część 1: symulator obwodu ze sprzężeniem indukcyjnym

Celem części pierwszej projektu było zaimplementowanie symulatora stanu nieustalonego obwodu elektrycznego ze sprzężeniem indukcyjnym w postaci liniowej. Obwód ten jest opisany układem równań różniczkowych zwyczajnych pierwszego rzędu, które rozwiązałem numerycznie przy użyciu metody Eulera oraz ulepszonej metody Eulera. Symulacja miała na celu wyznaczenie przebiegów czasowych prądów ,  oraz napięcia  dla czterech zadanych wymuszeń napięciowych.

### Model matematyczny

Obwód elektryczny ze sprzężeniem indukcyjnym (transformator) został opisany za pomocą trzech równań różniczkowych, wynikających z praw Kirchhoffa:



Rysunek 16: równania różniczkowe, część 1

### Model numeryczny

Do rozwiązania układu równań różniczkowych zastosowałem dwie metody numeryczne.

#### Metoda Eulera

Dla równania  krok obliczeniowy ma postać:

gdzie  – krok całkowania, , a  – wektor prawej strony układu równań.

#### Ulepszona metoda Eulera

Metoda ta jest metodą predyktor-korektor.

Predyktor:

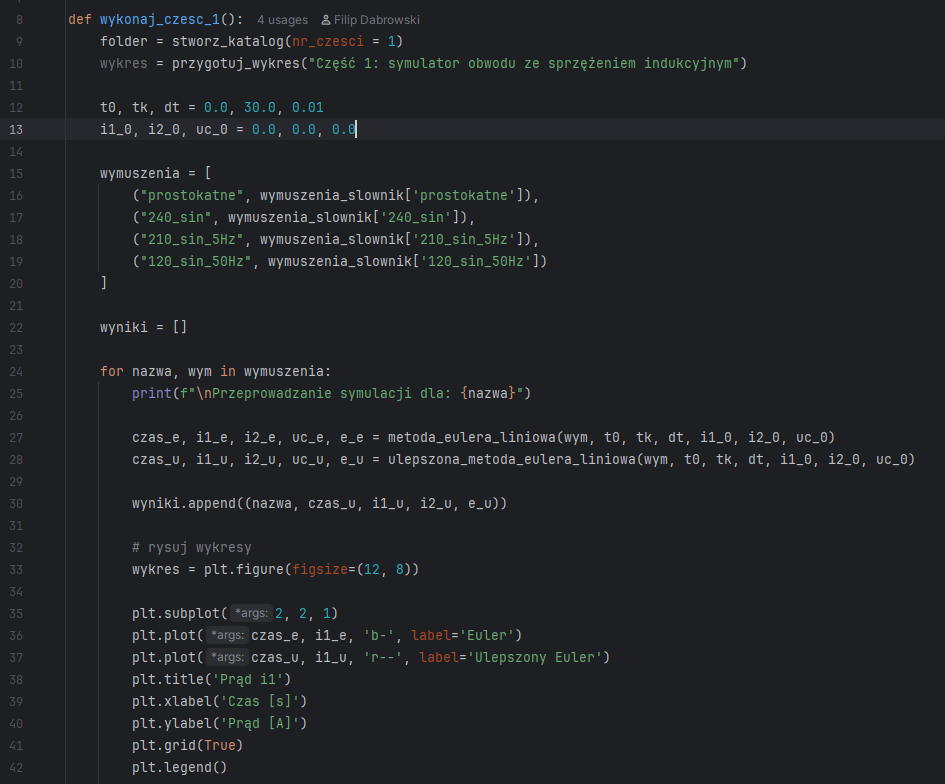
Korektor:

Jest to metoda rzędu drugiego, bardziej dokładna niż podstawowa metoda Eulera. Przedział czasowy symulacji:  s. Krok całkowania  został dobrany eksperymentalnie (domyślnie  s) tak, aby zapewnić stabilność i dokładność rozwiązań.

Obliczenia polegają na iteracyjnym stosowaniu wybranych wzorów: rozpoczynając od warunku początkowego , krok po kroku (w odstępach h) wyznaczamy kolejne wartości , które przybliżają rozwiązanie układu wzdłuż całego przedziału czasowego. Metoda Eulera używa tylko pochodnej z początku kroku , natomiast ulepszona metoda Heuna uśrednia pochodne z początku i z przybliżonego końca, co znacznie zwiększa dokładność końcowego rezultatu.

### Implementacja

Implementacja rozwiązania znajduje się w pliku projekt\_czesc\_1.py w katalogu czesc\_1. Zawiera on zasadniczo 2 funkcje które wywołują metody eulera (*metoda\_eulera\_liniowa()*) oraz ulepszoną metodę eulera (*ulepszona\_metoda\_eulera\_liniowa()*) a następnie rysują wykresy. To samo odbywa się jeszcze raz w funkcji *wykonaj\_porownanie()* dla wymuszenia sin(t). Poniżej prezentuję kod oraz jego bardziej szczegółowe omówienie:



Rysunek 17: projekt\_czesc\_1.py

Kod realizujący pierwszą część projektu działa według warstwowej struktury. Jego główną część stanowi funkcja *wykonaj\_czesc\_1()*. Rozpoczyna ona od przygotowania infrastruktury - tworzy folder na wyniki i definiuje podstawowe parametry symulacji, takie jak 30-sekundowy przedział czasowy z krokiem całkowania 0,01 sekundy oraz zerowe warunki początkowe dla prądów i napięcia.

Kluczowym elementem jest przetwarzanie czterech różnych wymuszeń pobieranych ze słownika. Dla każdego z nich, kod wywołuje dwie kluczowe funkcje całkujące: *metoda\_eulera\_liniowa()* oraz *ulepszona\_metoda\_eulera\_liniowa()*. Te funkcje to wrappery zdefiniowane w module *liniowy.py*, służą jako łącznik między ogólnymi rozwiązaniami numerycznymi a konkretnym modelem obwodu. Ich zadaniem jest przekazanie do ogólnych implementacji tychże metod odpowiedniej funkcji obliczającej pochodne, która została przygotowana specjalnie dla liniowego przypadku sprzężenia indukcyjnego.

Sam proces całkowania w metodzie Eulera, zaimplementowany w pliku *metoda\_eulera.py*, przebiega w pętli krok po kroku. W każdej iteracji, na podstawie aktualnych wartości prądów, napięcia i wymuszenia, obliczane są pochodne tych wielkości (czyli tempo ich zmian) za pomocą przekazanej funkcji. Następnie nowe wartości wyznaczane są przez proste dodanie do wartości bieżących iloczynu kroku czasowego i obliczonej pochodnej. Ulepszona metoda Eulera działa na podobnej zasadzie, ale wprowadza etap korekty – najpierw oblicza przybliżenie (predykcję), a potem wykorzystuje je do wyznaczenia bardziej dokładnej wartości końcowej.

Po otrzymaniu wyników z obu metod, program przystępuje do etapu wizualizacji. Dla każdego wymuszenia tworzony jest czteropanelowy wykres, który umożliwia bezpośrednie porównanie dokładności obu metod numerycznych poprzez nałożenie ich wyników na siebie. Dodatkowo, po zakończeniu wszystkich symulacji, generowane są dwa podsumowujące zestawienia graficzne: jedno ukazujące prądy dla wszystkich wymuszeń na wspólnym rysunku oraz drugie, służące celom weryfikacyjnym, porównujące szczegółowo wyniki obu metod dla standardowego wymuszenia testowego. Cały proces jest w pełni zautomatyzowany, a wszystkie wykresy są zapisywane do wcześniej utworzonego folderu.

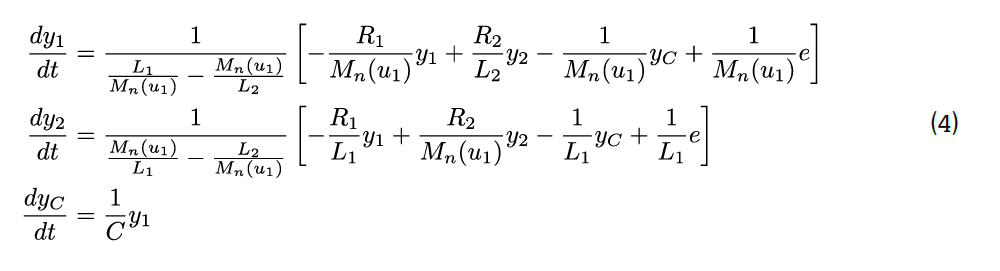
## Część 2: symulator obwodu z nieliniową indukcyjnością

W części drugiej projektu rozszerzyłem model obwodu o nieliniową charakterystykę sprzężenia indukcyjnego. Zadanie polegało na przeprowadzeniu symulacji stanu nieustalonego z uwzględnieniem, że indukcyjność wzajemna M zależy nieliniowo od napięcia na indukcyjności L1 w obwodzie pierwotnym. Zależność ta została podana w tabeli w projekcie. W celu wykorzystania tej zależności w równaniach różniczkowych konieczne było przybliżenie funkcji na podstawie punktów pomiarowych. Do tego celu zastosowałem cztery metody: interpolację wielomianową Lagrange'a, interpolację funkcjami sklejanymi trzeciego stopnia oraz aproksymację wielomianową stopnia 3 i 5. Następnie, z użyciem ulepszonej metody Eulera, przeprowadziłem symulacje dla dwóch wybranych wymuszeń.

### Model matematyczny

Równania różniczkowe opisujące obwód z nieliniową indukcyjnością wzajemną są podobne do tych z części pierwszej, jednak zamiast stałej wartości M, występuje teraz funkcja , która zależy od chwilowej wartości napięcia . Napięcie to oblicza się z prawa napięciowego jako:

Następnie, dla danej wartości , wyznacza się wartość indukcyjności wzajemnej z przybliżonej charakterystyki. Zmodyfikowany układ równań różniczkowych przyjmuje postać:



Rysunek 18: układ równań z części 2 projektu

gdzie .

### Model numeryczny

Do przybliżenia funkcji program wykorzystuje cztery metody numeryczne:

1. **Interpolacja Lagrange'a:** polega na konstrukcji wielomianu przechodzącego przez wszystkie punkty z tabeli. Dla danej wartości napięcia u, wartość M obliczana jest ze wzoru Lagrange'a.
2. **Interpolacja funkcjami sklejanymi trzeciego stopnia:** pomiędzy każdą parą punktów konstruowany jest wielomian trzeciego stopnia, tak aby w węzłach funkcja i jej pochodne (do drugiego rzędu) były ciągłe. Implementacja opiera się na rozwiązaniu układu równań liniowych dla drugich pochodnych, a następnie wyznaczeniu współczynników wielomianów sklejanych.
3. **Aproksymacja wielomianowa stopnia 3 i 5:** metoda najmniejszych kwadratów, w której dopasowuje się wielomian danego stopnia do punktów pomiarowych, minimalizując sumę kwadratów odchyłek. Rozwiązuje się układ równań normalnych , gdzie A jest macierzą Vandermonde'a.

Do całkowania równań różniczkowych ponownie użyłęm ulepszonej metody Eulera, która ze względu na swoją większą dokładność wydawała mi się bardziej odpowiednia do symulacji nieliniowego obwodu.

### Implementacja



Rysunek 19: projekt\_czesc\_2.py

Kod części drugiej znajduje się w pliku *projekt\_czesc\_2.py*. Główną funkcją jest *wykonaj\_czesc\_2().* Działanie programu można podzielić na kilka etapów:

Pierwszy etap to przygotowanie funkcji aproksymujących. Na podstawie tablicy danych (napięcie i odpowiadające wartości M) tworzone są cztery funkcje:

*M\_lagrange(u), M\_funkcje\_sklejane(u), M\_wielomian\_3(u), M\_wielomian\_5(u).* Każda z nich implementuje inną metodę przybliżenia.

Drugi etap to wizualizacja porównawcza metod. Generowany jest wykres przedstawiający punkty pomiarowe oraz przebiegi funkcji otrzymane za pomocą każdej z czterech metod. Wykres ten pozwala na ocenę jakości przybliżeń.

Trzeci etap obejmuje symulacje obwodu dla dwóch wymuszeń. Dla każdego z wybranych wymuszeń ( *oraz* ) przeprowadzane są cztery symulacje - po jednej dla każdej metody przybliżenia funkcji . Do symulacji używana jest funkcja *symuluj\_z\_zapisywaniem\_M()*, która korzysta z ulepszonej metody Eulera. Funkcja ta, oprócz rozwiązywania układu równań różniczkowych, zapisuje także chwilowe wartości indukcyjności M, co pozwala na późniejszą analizę.

Czwarty etap to generowanie wykresów wynikowych. Dla każdego wymuszenia tworzony jest czteropanelowy wykres przedstawiający przebiegi czasowe prądów i1, i2, napięcia uc oraz indukcyjności M dla wszystkich czterech metod. Wykresy są zapisywane do plików.

Ostatnim etapem jest tworzenie tabeli porównawczej. Program wyświetla w konsoli tabelę porównującą wartości M obliczone różnymi metodami dla napięć z oryginalnej tabeli. Pozwala to na ocenę dokładności interpolacji/aproksymacji w punktach węzłowych.

W trakcie implementacji części drugiej, z uwagi na niedopatrzenie, nieco zaniedbałem zasadę czystej modularności. Definicje funkcji aproksymujących znajdują się bezpośrednio w głównym pliku, zamiast być przeniesione do osobnych modułów. Wynika to z tego że na pomysł pisania modularnego kodu wpadłem dopiero przy wykonywaniu 3 części projektu, więc część 1 i 2 musiałem odpowiednio przerobić. Zwyczajnie zapomniałem to zrobić i zorientowałem się o tym zaniedbaniu dopiero teraz, w trakcie pisania raportu. Ponadto, niektóre funkcje opakowujące (jak te do symulacji nieliniowej) umieściłem w module liniowy.py, co może prowadzić do lekkiego zamieszania. Nie byłem pewien gdzie jest dla nich najlepsze miejsce.

## Część 3: obliczanie mocy czynnej na rezystorach

Zadaniem części trzeciej projektu było obliczenie całkowitej energii wydzielanej na rezystorach R1 i R2 w obwodzie w czasie 30 sekund. Wartość tę określa całka z mocy chwilowej, która zależy od kwadratów prądów płynących przez rezystory. Ćwiczenie obejmowało porównanie wyników dla obwodu liniowego (ze stałym M=0.8 H) i nieliniowego (gdzie M zależy od napięcia), zastosowanie dwóch różnych metod całkowania numerycznego (złożonej metody prostokątów i złożonej metody parabol Simpsona) oraz analizę wpływu długości kroku czasowego na dokładność obliczeń.

**Model matematyczny:** Moc chwilowa wydzielana na rezystorach R1 i R2 wyraża się wzorem:

gdzie  i  to przebiegi czasowe prądów obliczone wcześniej przez symulator.

Całkowita energia  wydzielona w przedziale czasowym od 0 do 30 sekund to całka z mocy chwilowej:

Wartość ta, podzielona przez czas 30 s, daje średnią moc czynną w watach:

W implementacji obliczana jest bezpośrednio wartość całki, co odpowiada całkowitej energii w dżulach. W raporcie i tabelach wyniki można przedstawić zarówno jako energię [J], jak i średnią moc [W].

### Model numeryczny

Do obliczenia całki numerycznie zastosowałem dwie metody:

**Złożona metoda prostokątów:** Całka przybliżana jest sumą pól prostokątów o podstawie i wysokości równej wartości funkcji podcałkowej w lewym końcu przedziału:

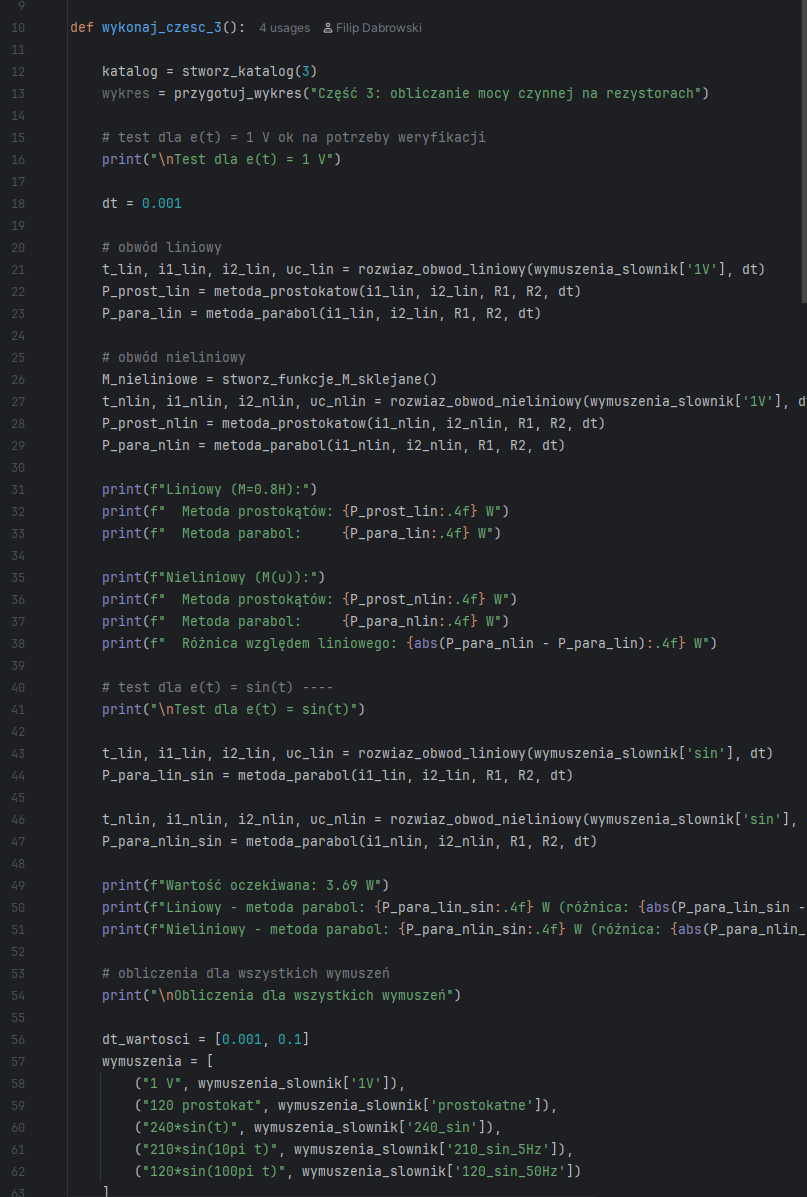
Implementacja w funkcji *metoda\_prostokatow()* mnoży sumę wartości mocy chwilowej we wszystkich punktach przez krok .

**Złożona metoda parabol (Simpsona):** Wykorzystuje kwadraturowe wzory Simpsona, które dla równomiernie rozłożonych punktów dają wyższą dokładność:

Funkcja *metoda\_parabol()* implementuje ten algorytm, dostosowując go do nieparzystej liczby przedziałów.

### Implementacja

Główny kod części trzeciej znajduje się w pliku projekt\_czesc\_3.py.



Rysunek 20: projekt\_czesc\_3.py

Działanie funkcji wykonaj\_czesc\_3() można podzielić na etapy:

Pierwszy etap to testy weryfikacyjne. Dla wymuszenia 1 V obliczana jest energia metodami prostokątów i parabol dla obwodu liniowego i nieliniowego. Dla wymuszenia obliczona wartość porównywana jest z oczekiwanym wynikiem 3.69 J (co w instrukcji jest podane jako moc, ale w kontekście obliczeń całki jest to energia). Testy te służą potwierdzeniu poprawności implementacji.

Drugi etap to główne obliczenia dla wszystkich pięciu wymuszeń (1 V, oraz 4 wymuszenia z części 1 projektu) i dwóch różnych kroków czasowych: (mały krok, wysoka dokładność) oraz (duży krok, mniejsza dokładność). Dla każdej kombinacji wymuszenia i kroku rozwiązywane są równania różniczkowe (używając ulepszonej metody Eulera) dla obu wariantów obwodu (liniowego i nieliniowego), a następnie oblicza całkę metodami prostokątów i parabol. Wyniki są zapisywane do list i prezentowane w przejrzystych tabelach w konsoli.

Trzeci etap to generowanie wykresów porównawczych. Dla wymuszenia tworzony jest zestaw czterech podwykresów pokazujących przebiegi prądów i mocy chwilowej dla obu wartości , z nałożeniem wyników dla obwodu liniowego i nieliniowego. Wykresy mocy zawierają wypełnienie pod krzywą, co wizualizuje obszar całkowania. Dodatkowo generowany jest wykres dla wymuszenia 1 V, który szczegółowo porównuje moc chwilową i prądy w obu przypadkach.

## Część 4: znajdowanie częstotliwości dla mocy 406 W

Zadaniem było wyznaczenie częstotliwości f sygnału wymuszającego, dla której całkowita moc czynna wydzielana na rezystorach R1 i R2 wynosi dokładnie 406 W. Przyjęto wymuszenie sinusoidalne postaci . Problem ten sprowadza się do rozwiązania nieliniowego równania , gdzie jest mocą całkowitą obliczaną numerycznie dla danej częstotliwości. Do rozwiązania zadania zastosowałęm trzy metody numeryczne: metodę bisekcji, metodę siecznych oraz metodę quasi-Newtona z numerycznym przybliżeniem pochodnej.

### Model matematyczny

Podstawą obliczeń jest funkcja celu F(f), zdefiniowana jako różnica między mocą całkowitą P(f) a zadaną wartością 406 W:

Moc całkowita P(f) dla danej częstotliwości f jest obliczana jako całka z mocy chwilowej w przedziale czasowym od 0 do 30 sekund:

gdzie przebiegi prądów i1(t) i i2(t) są rozwiązywane dla obwodu liniowego (M=0.8 H) z wymuszeniem .

### Model numeryczny

W celu znalezienia częstotliwości f, dla której F(f)=0, zaimplementowałem trzy metody znajdowania pierwiastków równań nieliniowych:

**Metoda bisekcji:** Algorytm wymaga przedziału początkowego [a, b], w którym funkcja F(f) zmienia znak. W każdej iteracji przedział jest dzielony na pół, a znak funkcji w środkowym punkcie decyduje o wyborze nowego przedziału. Metoda gwarantuje zbieżność przy spełnieniu warunku początkowego.

**Metoda siecznych:** Metoda wykorzystuje przybliżenie liniowe funkcji pomiędzy dwoma punktami. Iteracyjnie oblicza się punkt przecięcia siecznej z osią odciętych, używając wzoru:

Metoda ta jest szybsza od bisekcji, ale nie gwarantuje zbieżności dla każdego przypadku.

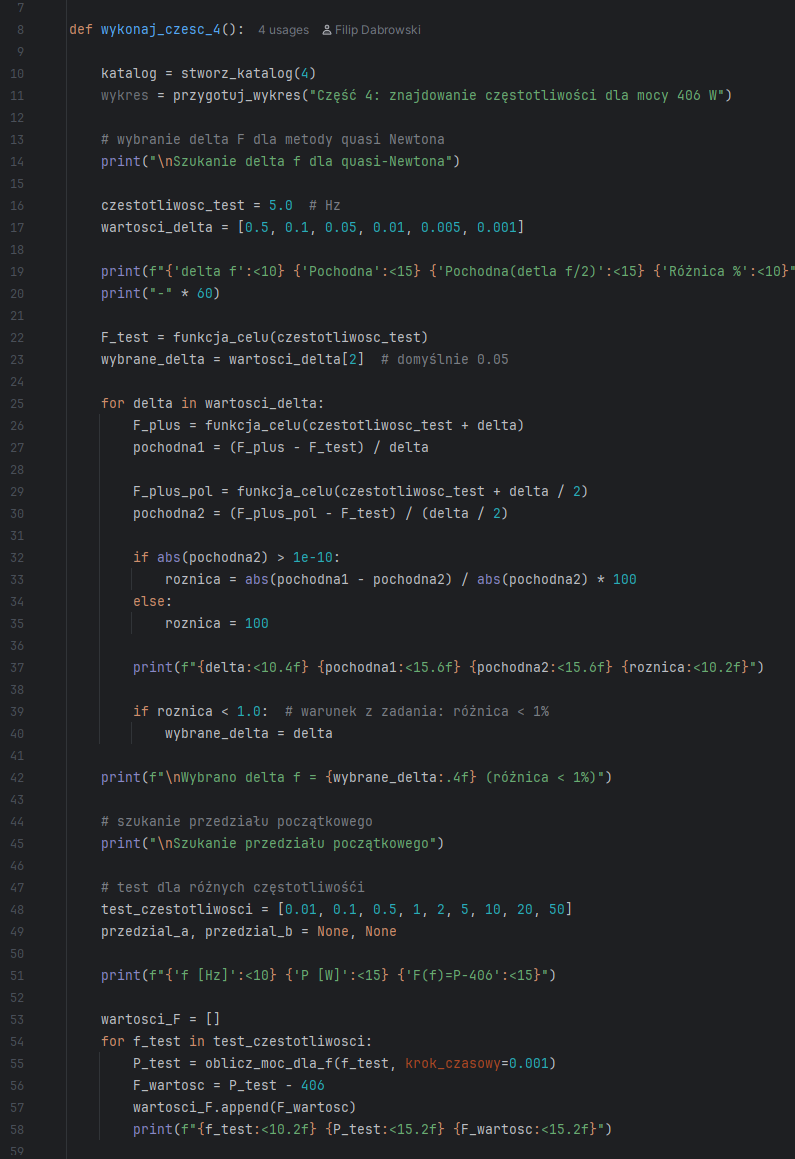
**Metoda quasi-Newtona:** Jest to zmodyfikowana metoda Newtona, w której pochodną funkcji przybliża się ilorazem różnicowym:

Iteracja przebiega według wzoru Newtona:

Kluczowym parametrem jest wartość , która musi być odpowiednio dobrana, aby przybliżenie pochodnej było wystarczająco dokładne.

### Implementacja

Kod części czwartej znajduje się w pliku projekt\_czesc\_4.py.



Rysunek 21: projekt\_czesc\_4.py

Pierwszy etap to dobranie parametru dla metody quasi-Newtona. Program testuje kilka wartości , porównując pochodne obliczone z krokiem i . Zgodnie z wymaganiami projektu, wybiera się taką wartość , dla której różnica między tymi przybliżeniami jest mniejsza niż 1%.

Drugi etap to znalezienie przedziału początkowego dla metody bisekcji. Program oblicza wartości funkcji dla kilku przykładowych częstotliwości, szukając przedziału, w którym funkcja zmienia znak. Jeśli taki przedział nie zostanie znaleziony, program szacuje zakres wokół częstotliwości dającej maksymalną moc.

Trzeci etap to zastosowanie trzech metod numerycznych. Dla każdej metody program śledzi liczbę iteracji oraz liczbę ewaluacji funkcji (każda ewaluacja wymaga rozwiązania układu równań różniczkowych i obliczenia całki). Wyniki są zbierane w tabeli porównawczej.

Czwarty etap to wizualizacja. Generowany jest wykres funkcji w znalezionym przedziale, na którym zaznaczono miejsca zerowe znalezione przez poszczególne metody.

# Interpretacja wyników i odpowiedzi na pytania z projektu

## Cześć 1

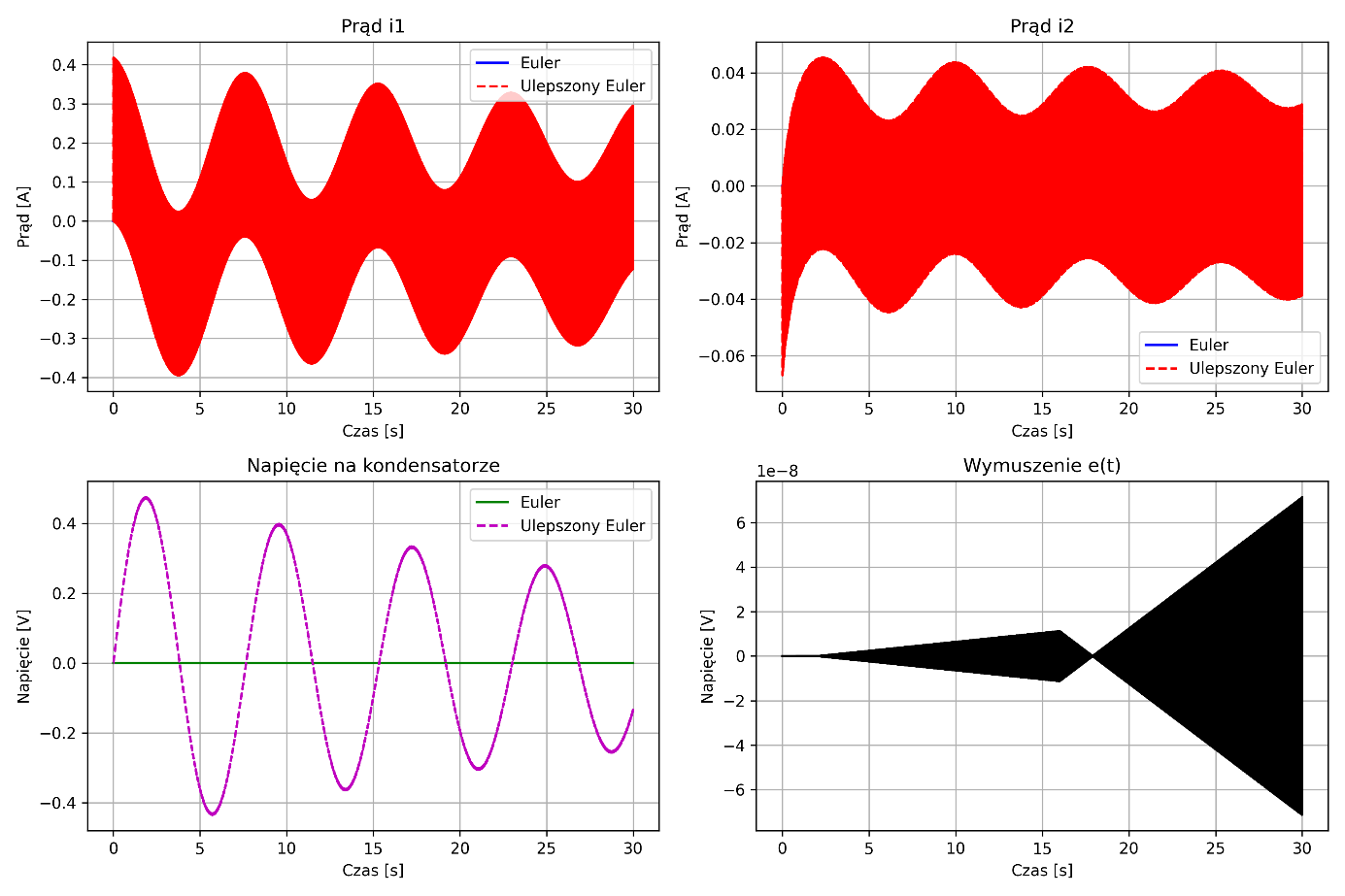
### Wymagania projektowe

W raporcie proszę przedstawić kod źródłowy programu oraz wykresy przebiegów prądów i oraz napięć i *.* Osobno dla poszczególnych metod przedstawionych w projekcie.

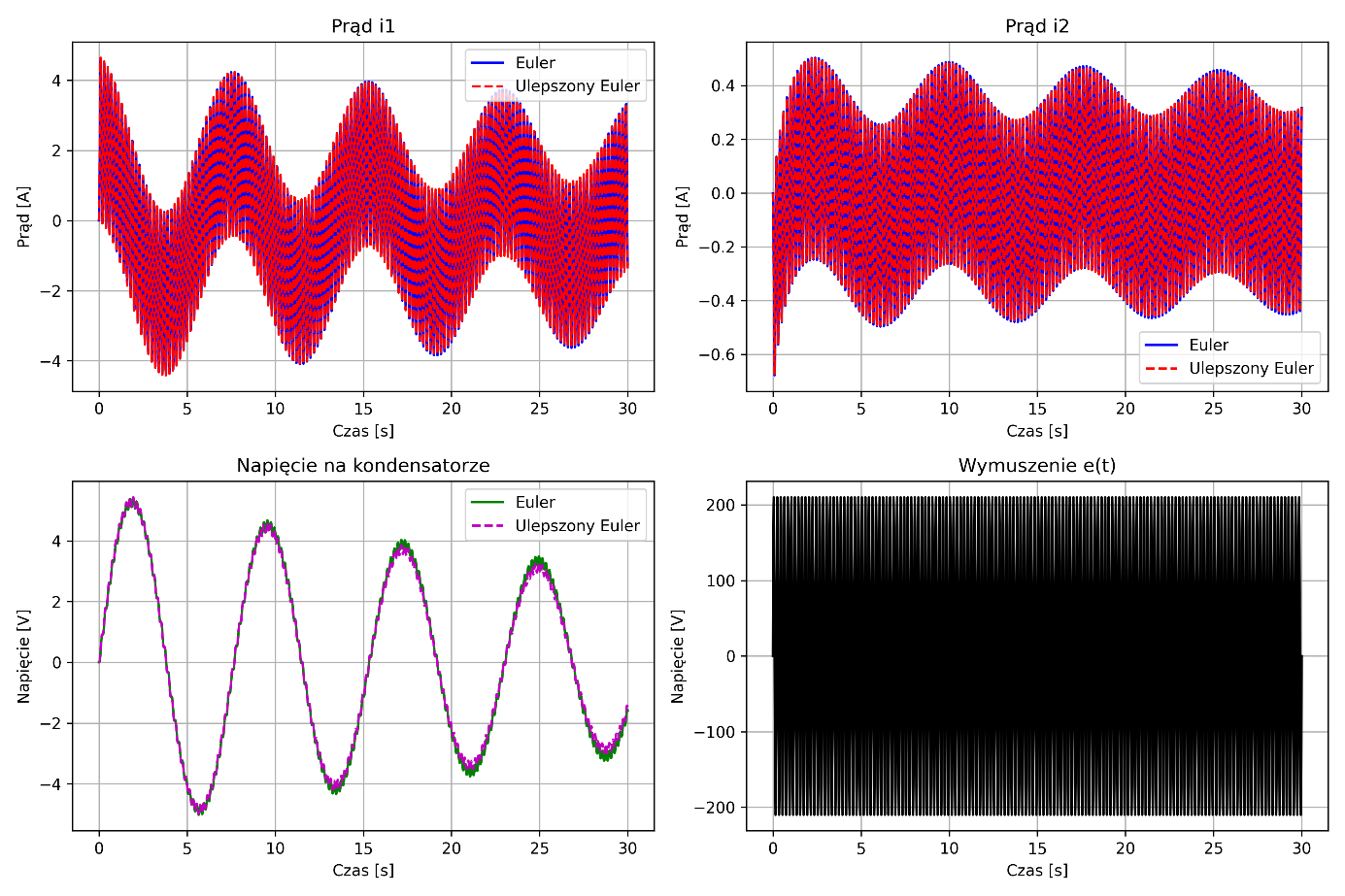
### Wymaganie nr 1: kod źródłowy

kod źródłowy – został on omówiony i częściowo wklejony. Z uwagi na fakt, że program zawiera wiele modułów, ciężkim zadaniem jest wkleić je wszystkie do raportu. Wkleiłem jednak najważniejsze fragmenty oraz przestawiłem model matematyczny, numeryczny oraz wytłumaczyłem implementacje.

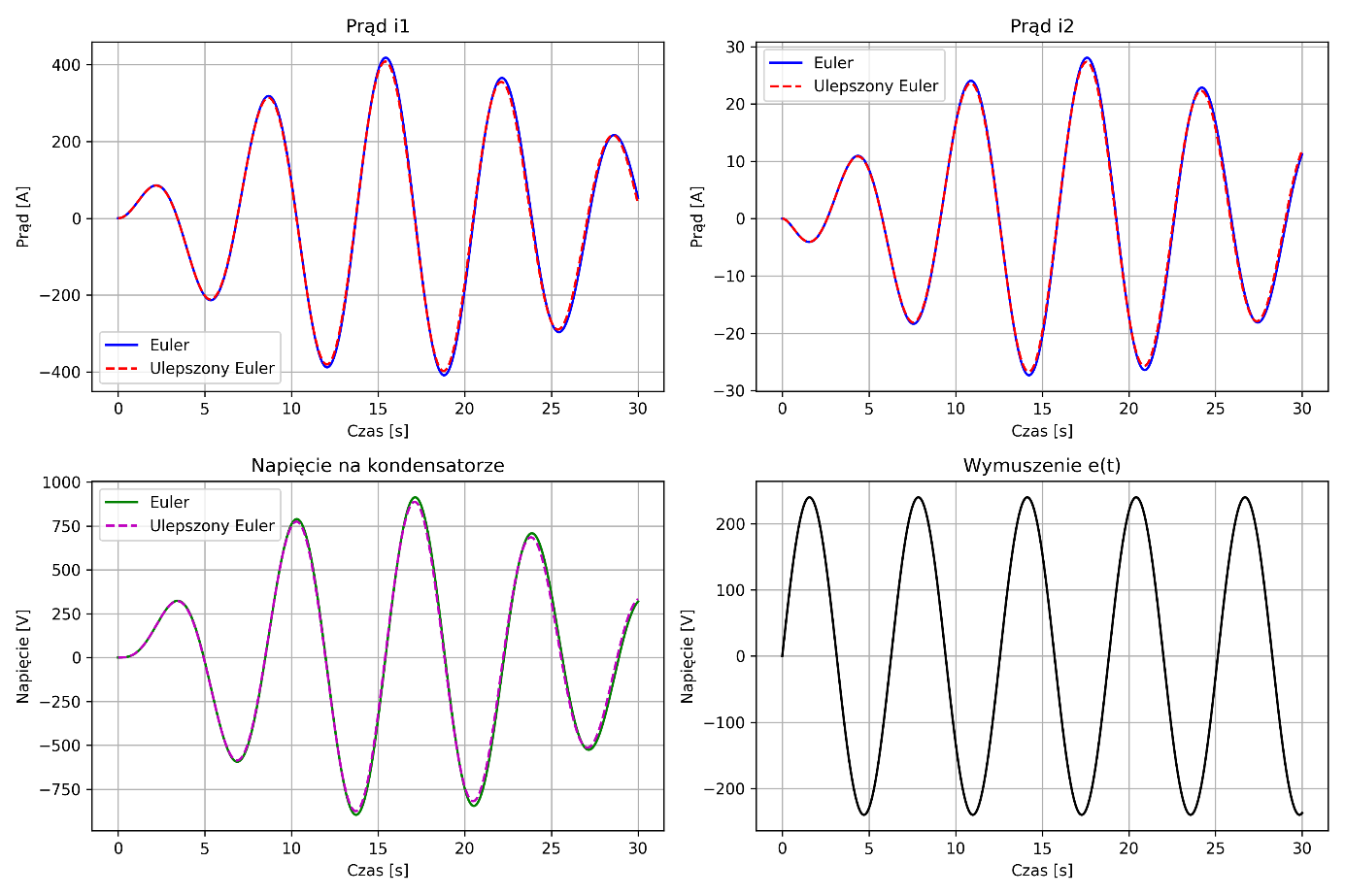
### Wymaganie nr 2: wykresy



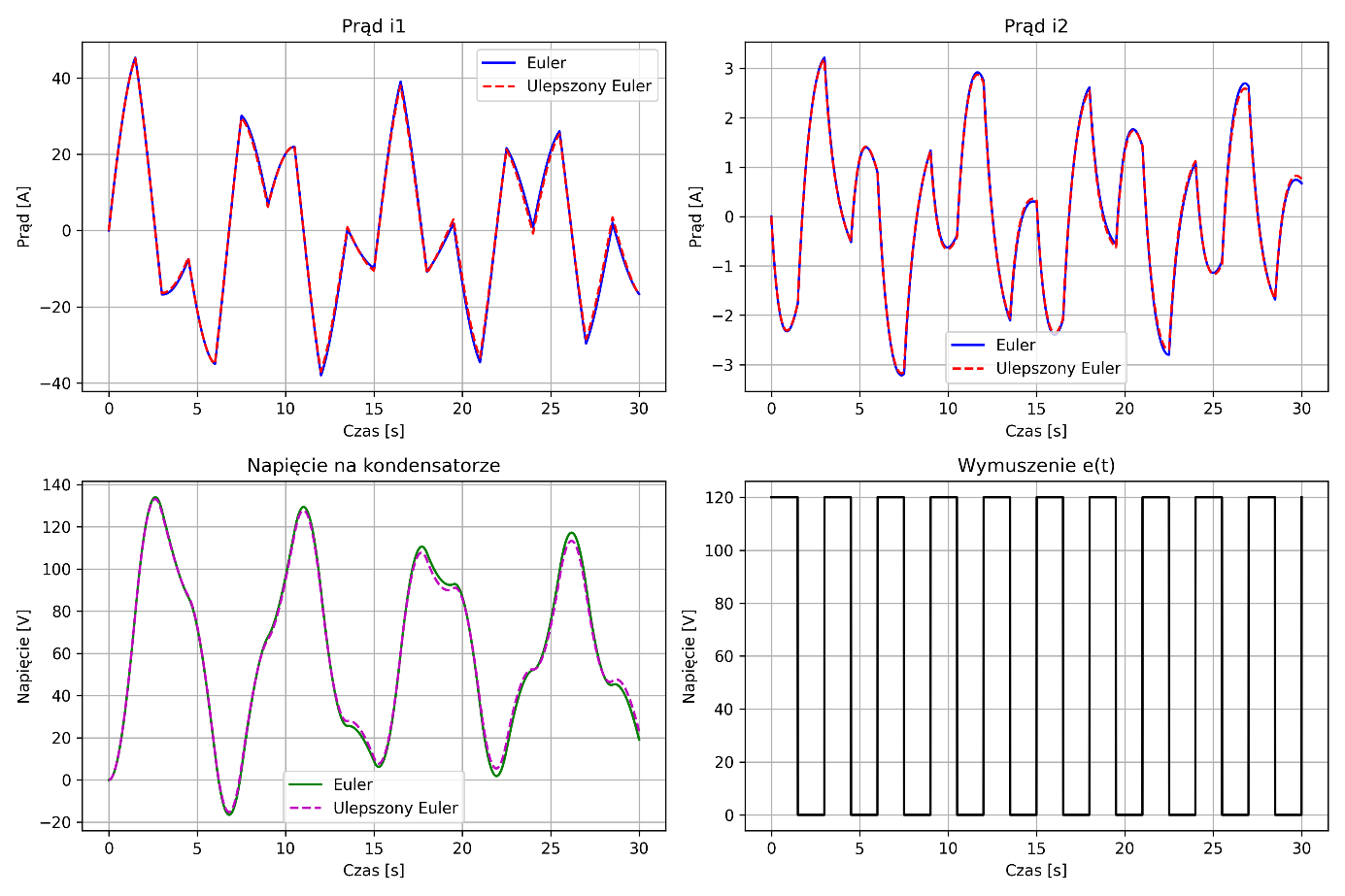
Rysunek 22: wykres dla wymuszenia e(t) = 120sin(2πft), dla f = 50 Hz



Rysunek 23: wykres dla wymuszenia e(t) = 210sin(2πft), dla f = 5 Hz



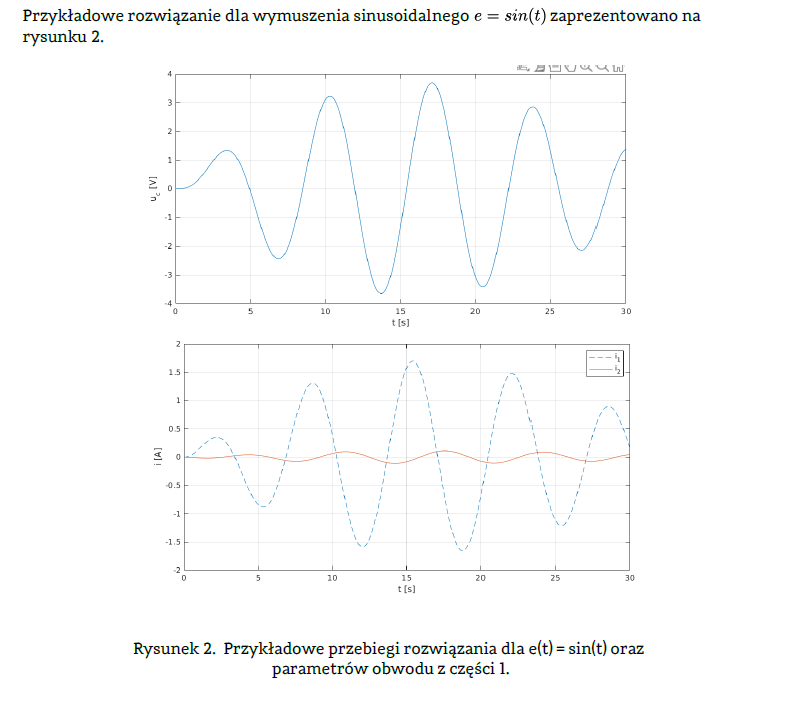
Rysunek 24: wykres dla wymuszenia e(t) = 240sin(t)



Rysunek 25: wykres dla wymuszenia prostokątnego

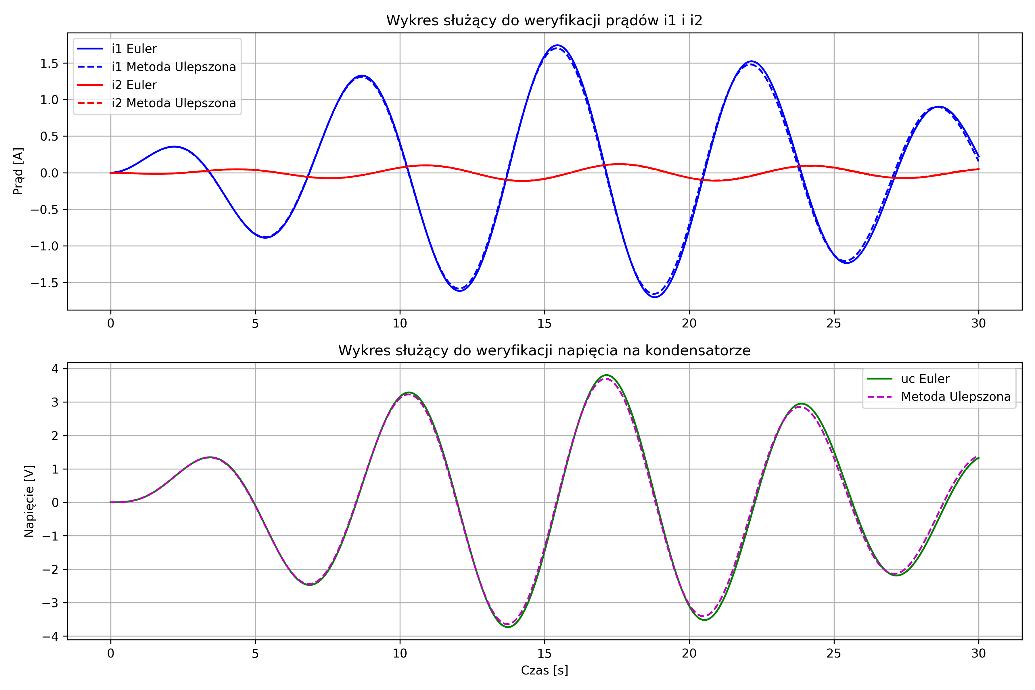
### Dodatkowe wykresy weryfikacyjne

W celu weryfikacji stworzyłem wykresy odtwarzające na bazie moich wyników wykresy przedstawione w projekcie. Możemy zauważyć (heurystycznie), że dają one bardzo zbliżone wyniki.

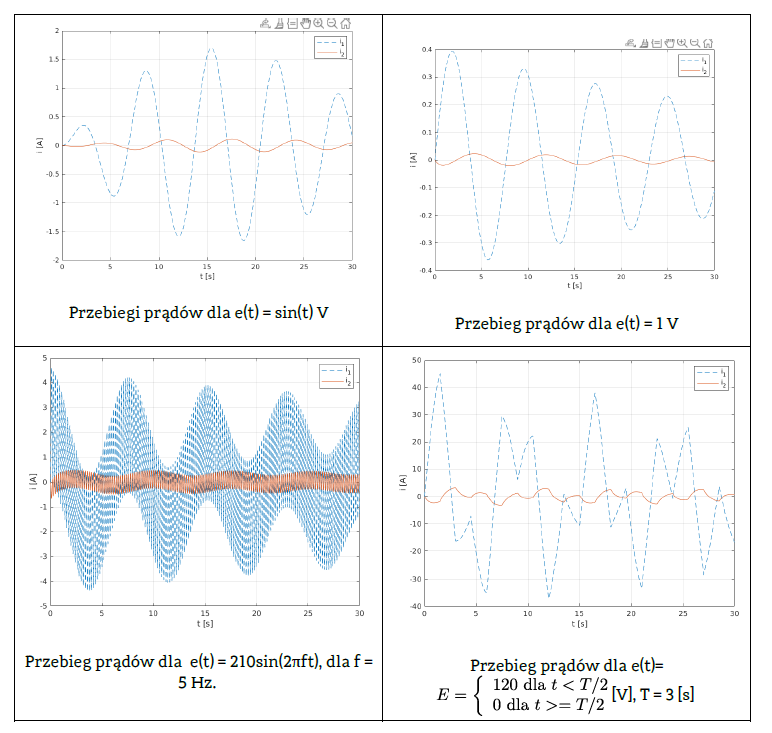


Rysunek 26: przykładowe rozwiązanie z projektu dla wymuszenia e = sin(t)

Poniżej umieszczam rozwiązanie wygenerowane przy wykorzystaniu mojego programu:

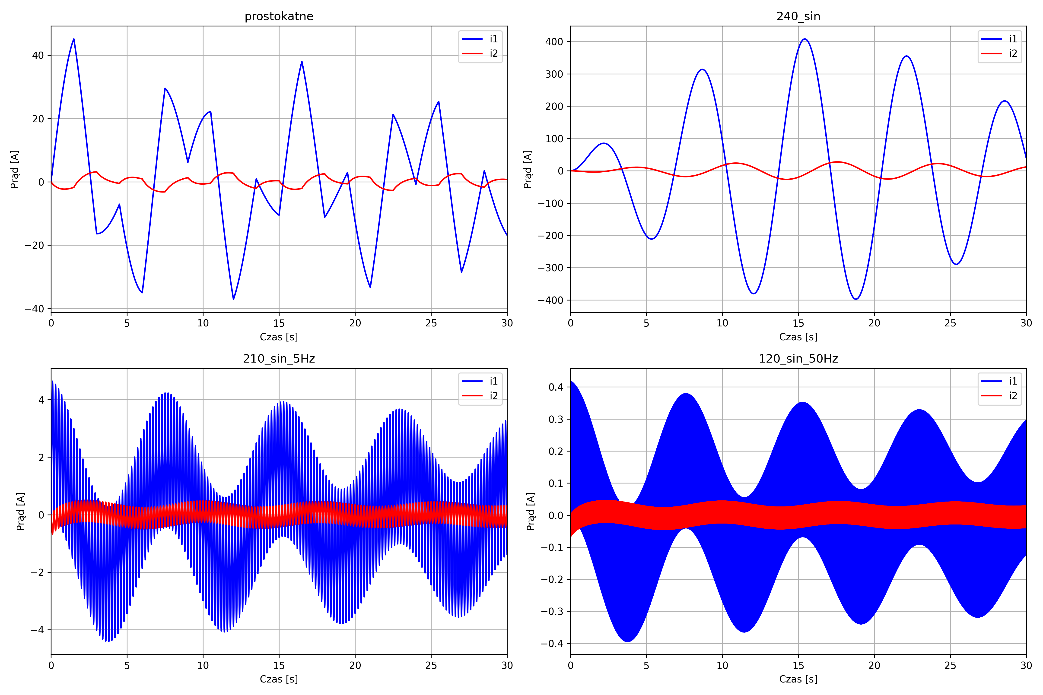


Rysunek 27: wykresy dla wymuszenia e = sin(t) wygenerowane przez mój program



Rysunek 28: przykładowe wyniki 4 wymuszeń z projektu

Moje wyniki (wykres zbiorczy dla 4 takich samych wymuszeń). Wykresy nie są takie same, jednak 2 z nich się pokrywają. **Pozwala mi to zweryfikować moje wyniki jako właściwe**:



Rysunek 29: : przykładowe wyniki 4 wymuszeń z mojego programu

## Część 2

### Wymagania projektowe

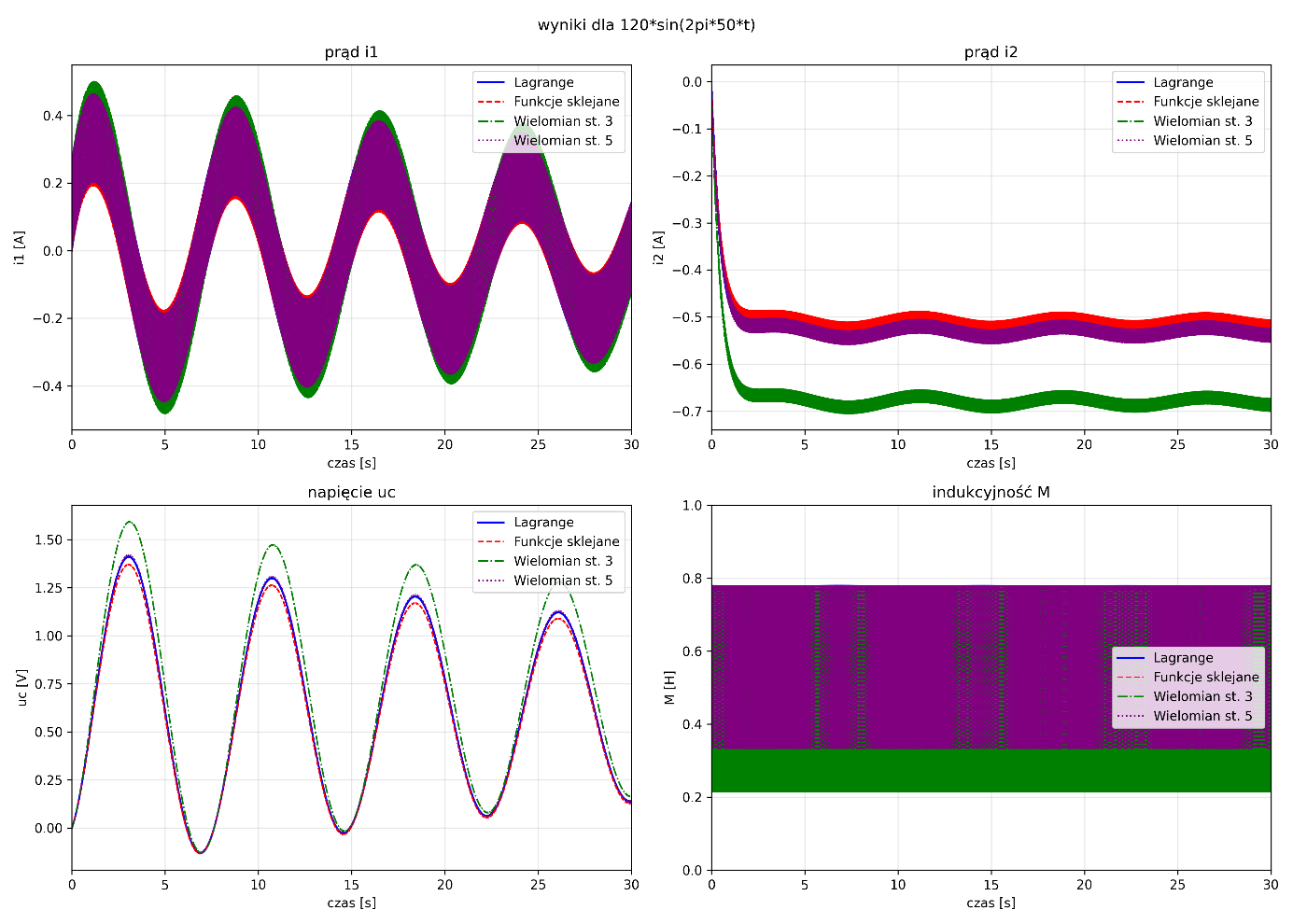
W ostatecznym wyniku w raporcie oczekuję 8 wykresów reprezentujących przebiegi czasowe:

* + dla
    - wykresów *i*1(t) (to powinien być ten sam okres co w części 1) dla każdej metody interpolacji / aproksymacji
    - wykresów *i*2(t) dla każdej metody interpolacji / aproksymacji
    - wykresów *u*R2(t) dla każdej metody interpolacji / aproksymacji
    - wykresów *u*c(t) dla każdej metody interpolacji / aproksymacji
  + dla
    - wykresów *i*1(t) (to powinien być ten sam okres co w części 1) dla każdej metody interpolacji / aproksymacji
    - wykresów *i*2(t) dla każdej metody interpolacji / aproksymacji
    - wykresów *u*R2(t) dla każdej metody interpolacji / aproksymacji
    - wykresów *u*c(t) dla każdej metody interpolacji / aproksymacji

Proszę zinterpretować wyniki. Czy są jakieś różnice? Czy różnice są znaczące? Skąd różnice mogłyby się wziąć?

### Raporty oraz interpretacja wyników

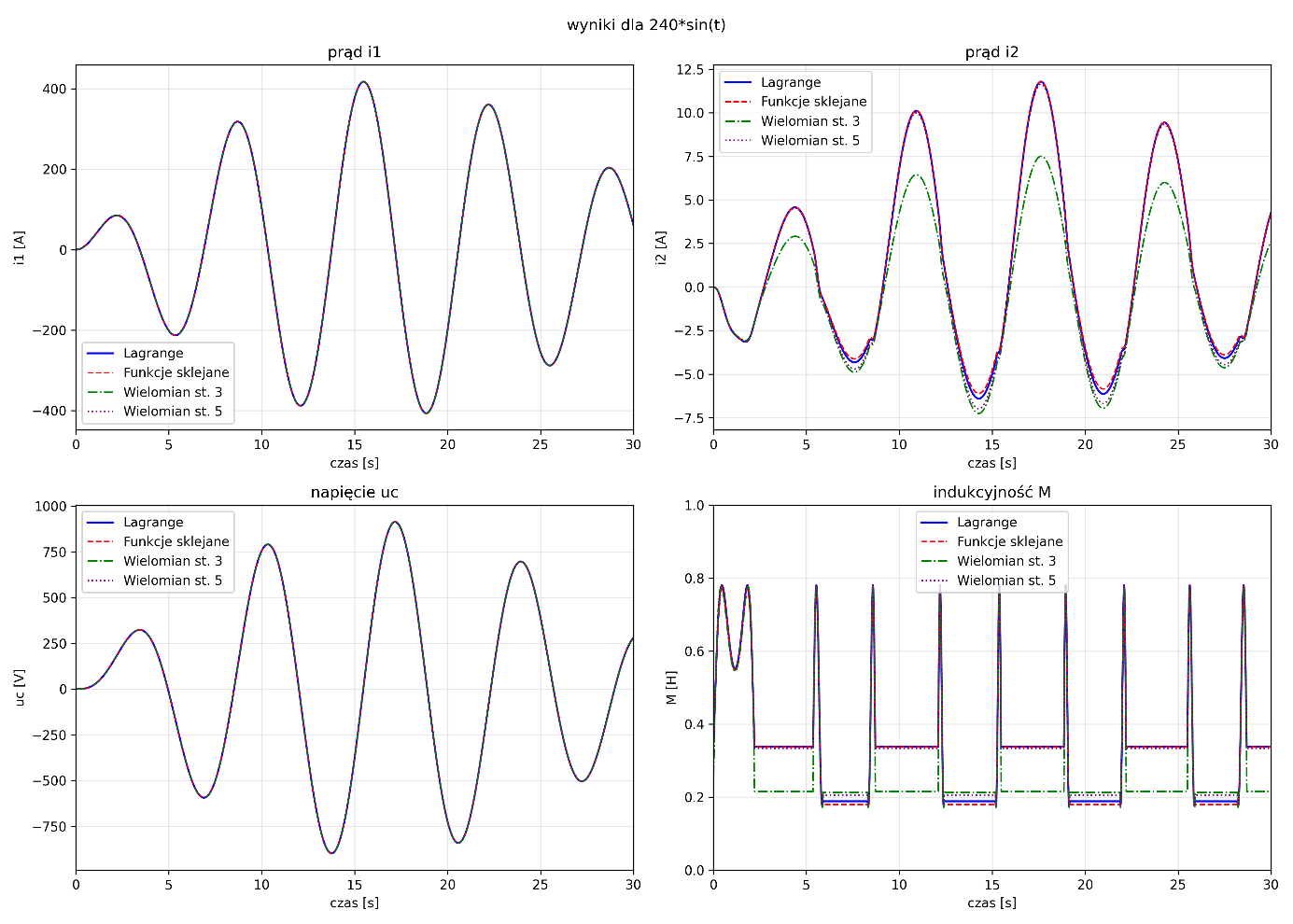
Dla każdego wymuszenia stworzony został czteropanelowy wykres przedstawiający przebiegi czasowe przebiegów prądów i oraz napięć i oraz indukcyjności dla wszystkich czterech metod (łącznie 8 wykresów jak w wymaganiach projektowych).



Rysunek 30: wyniki dla 120\*sin(2pi\*50\*t)

Na wykresach dla tego wymuszenia wszystkie metody dają bardzo podobne przebiegi prądów i₁(t), i₂(t) oraz napięcia , więc krzywe prawie całkowicie się pokrywają i różnice w amplitudach są tylko delikatnie zauważalne (zresztą podobnie jak w poprzednim wymuszeniu).

Przebieg M(t) wygląda dość „dziwnie”, bo zmiany są bardzo małe względem skali osi, przez co prawie nic na nim nie widać. Na tle pozostałych wyników można jedynie zauważyć, że metody globalne (zwłaszcza wielomian 3. stopnia) odcinają się nieco od bardziej gładkich wyników funkcji sklejanych



Rysunek 31: wyniki dla 240 \* sin(t)

Tutaj ponownie cztery metody dają bardzo zbliżone wyniki – amplitudy prądów i₁(t), i₂(t), napięcia oraz indukcyjności różnią się nieznacznie.

Delikatne różnice widoczne głównie w : Lagrange wykazuje oscylacje, funkcje sklejane mają najgładszy przebieg, wielomian 5. Stopnia również daje podobne wyniki do wcześniejszych metod. Zdecydowanie widzimy, że wielomian 3. Stopnia daje wyniki najbardziej odstające od innych.​

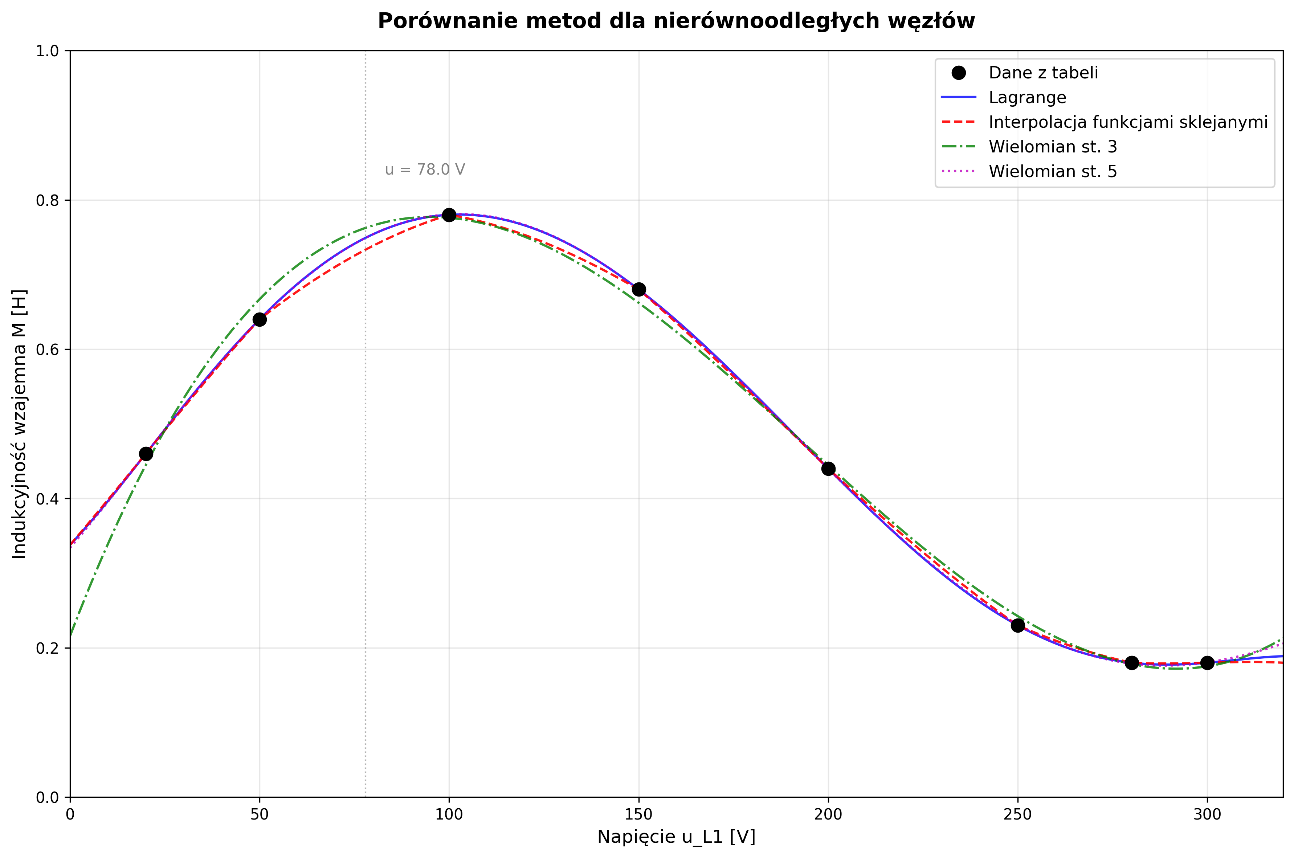
### Wykres charakterystyk M(u) – porównanie z projektem

Dodatkowo wykonałem wykres porównawczy aby odnieść się do wyników przedstawionych w projekcie.

Obraz zawierający tekst, linia, Wykres, diagram

Zawartość wygenerowana przez AI może być niepoprawna.

Rysunek 32: : wykres charakterystyki M(u) z projektu



Rysunek 33: wykres charakterystyki M(u) mojego programu

Na wykresie z projektu widzimy gładką krzywą która dokłądnie przechodzi przez punkty. Krzywa wygląda „naturalnie”, bez oscylacji. Dla mojej implementacji widzimy 4 metody opisywane wyżej, oto interpretacja:

**Lagrange (niebieska)** moim wykresie jest generalnie stabilna i podobna do przykładu z projektu. Istnieje jednak ryzyko że mogłaby odbiegać (zdaje się trochę odbiegać przy końcu). Lagrange nie jest najdokładniejszą z metod, istnieje ryzyko oscylacji widoczne delikatnie przy napięciu w przedziale ok [280, 320].

**Funkcje sklejane 3. stopnia (czerwony)** wydaje się najbliższy temu, co widoczne na wykresie z projektu. Gładka krzywa, wydaje się to być najbardziej fizyczny/realistyczny przebieg.

**Wielomian 3. stopnia (zielony)** uproszczona aproksymacja. Nie przechodzi idealnie przez wszystkie punkty. W środkowej części pasuje przyzwoicie, ale na początku i końcu wyraźnie rozmija się z danymi. Wynika to z tego, że jeden wielomian 3. stopnia nie odwzoruje dobrze charakterystyki, która ma wyraźny łuk i asymetrię.

**Wielomian 5. stopnia (różowy)** bardzo przyzwoite dopasowanie, wynik bardzo zbliżony do Lagrange’a. Nieco odbiega od Lagrange przy końcu.

### Najważniejsze różnice między metodami / wnioski

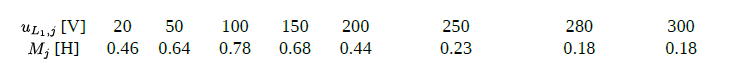
**Funkcje sklejane 3 stopnia:** Najbardziej stabilny, brak oscylacji. Daje realistyczny, użyteczny przebieg. Idealnie nadaje się do charakterystyk fizycznych (indukcyjności, straty, nieliniowości itd.).

**Lagrange:** przechodzi dokładnie przez wszystkie punkty. Zmienna gęstość węzłów może powodować lokalne odchyłki. Dla kilku punktów działa dobrze, dla większej liczby mogło by to wyglądać gorzej.

**Wielomian 3. stopnia:** Zbyt prosty, żeby dokładnie odwzorować tę charakterystykę. Najprostszy w implementacji jednak też najmniej dokładny.

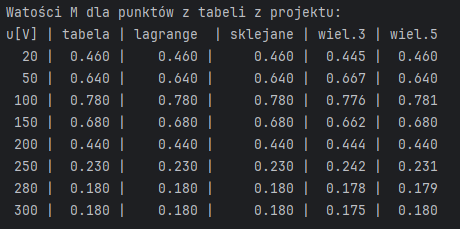
**Wielomian 5. stopnia:** Lepszy kompromis, ale nadal nie tak stabilny jak funkcje sklejane. Zachowanie na końcach przedziału jest najsłabszym punktem.

### Dodatkowa weryfikacja wartości z tabeli

W projekcie przedstawiono poniższą tabelę:  


Rysunek 34: tabela pomiarów par napięcia i indukcyjności wzajemnej z projektu

Przy wykorzystaniu mojego programu wygenerowałem tabelę zawierającą te węzły:



Rysunek 35: : tabela pomiarów par napięcia i indukcyjności wzajemnej dla 4 metod - z mojego programu

### Interpretacja tabeli

Wartości obliczone różnymi metodami dla napięć z pomiarów pokazują, że: interpolacja Lagrange'a oraz interpolacja funkcjami sklejanymi dają dokładnie takie same wartości jak dane wejściowe (tabela). To oczekiwane zachowanie, ponieważ obie metody interpolacyjne mają przechodzić dokładnie przez punkty pomiarowe.

Aproksymacja wielomianowa stopnia 3 wykazuje niewielkie odchyłki od wartości referencyjnych, największą przy napięciu 50 V (0,667 H vs 0,640 H - różnica 0,027 H). Aproksymacja wielomianowa stopnia 5 daje wyniki praktycznie identyczne z danymi wejściowymi, co wskazuje na jej bardzo dobre dopasowanie do punktów pomiarowych.

## Część 3

### Wymagania projektowe

Proszę porównać wyniki całki reprezentującej sumaryczną energię wydzielającą się w elementach rezystancyjnych obwodu uzyskane różnymi metodami dla wszystkich zadanych wymuszeń sygnałów z części 1 oraz dla **dwóch różnych** kroków : bardzo krótkiego i bardzo długiego. Proszę w szczególności przedstawić uzupełnioną tabelę. Wyniki dla niektórych przebiegów powinny znacznie się różnić, dla niektórych nie. Bardzo długi krok powinien być długi, ale dawać jednak nadal w miarę sensowne wyniki graficzne. To nie powinno być rozwiązanie niestabilne! W raporcie proszę uwzględnić przebiegi czasowe rozwiązań dla poniższych przypadków.

### Wyniki



Rysunek 36: wyniki części 3

### Interpretacja

W tej części zadania porównałem złożoną metodę prostokątów oraz złożoną metodę parabol (Simpsona) dla dwóch wartości kroku całkowania: (krok bardzo krótki, wysokiej dokładności) oraz (krok długi, obniżonej dokładności). Symulacje wykonano dla pięciu różnych wymuszeń, zarówno dla wariantu liniowego, jak i nieliniowego obwodu.

**Dla bardzo małego kroku wyniki uzyskane obiema metodami całkowania są praktycznie identyczne.** Potwierdza to, że przy odpowiednio gęstej dyskretyzacji obie metody zapewniają wysoką dokładność. Świadczy to o poprawności implementacji algorytmów numerycznych.

W przypadku dużego kroku możemy zauważyć zróżnicowany wpływ na dokładność w zależności od charakteru wymuszenia. Dla sygnałów wolnozmiennych lub stałych, takich jak napięcie 1 V czy przebieg prostokątny, różnice między metodami są niewielkie, a uzyskane wartości mocy pozostają wiarygodne i zbliżone do wyników z małym krokiem. Jednak dla sygnałów szybkozmiennych, w szczególności sinusoidalnych o wyższych częstotliwościach, błąd związany z dużym krokiem staje się znaczący. **Najbardziej widocznym przykładem jest wymuszenie ) o częstotliwości 50 Hz. Dla , który jest pięciokrotnie dłuższy niż okres tego sygnału, obie metody całkowania zwracają wynik zerowy.** Jest to całkowicie błędne, gdyż krok czasowy jest zbyt duży, aby poprawnie próbkować tak szybko oscylujący sygnał, co prowadzi do utraty informacji o przebiegu.

**Dla wymuszenia (5 Hz) wpływ dużego kroku jest również wyraźny, choć mniej drastyczny. W przypadku liniowym obie metody dają zbliżone do siebie wyniki, ale wartości te są ponad pięciokrotnie wyższe od dokładnych wyników uzyskanych z małym krokiem.** W wariancie nieliniowym dodatkowo widać rozbieżność między wynikami z metody prostokątów i parabol, co wskazuje, że przy niedostatecznej liczbie próbek dokładniejsza metoda Simpsona może być bardziej podatna na błędy związane z niedokładnym odwzorowaniem kształtu funkcji.

Podsumowując, dokładność obliczeń całkowania numerycznego jest silnie uzależniona od relacji między krokiem całkowania a szybkością zmian całkowanej funkcji. Dla sygnałów wolnozmiennych nawet stosunkowo duży krok może dać poprawne wyniki, podczas dla sygnałów szybkozmiennych jest to niedopuszczalne i prowadzi do katastrofalnego błędu. Metoda parabol Simpsona, będąc metodą wyższego rzędu, teoretycznie oferuje większą dokładność, ale jej przewaga nad prostą metodą prostokątów jest widoczna tylko przy odpowiednio gęstym próbkowaniu. Wyniki wyraźnie pokazują również wpływ nieliniowości elementów obwodu na jego charakterystykę energetyczną.

## Cześć 4

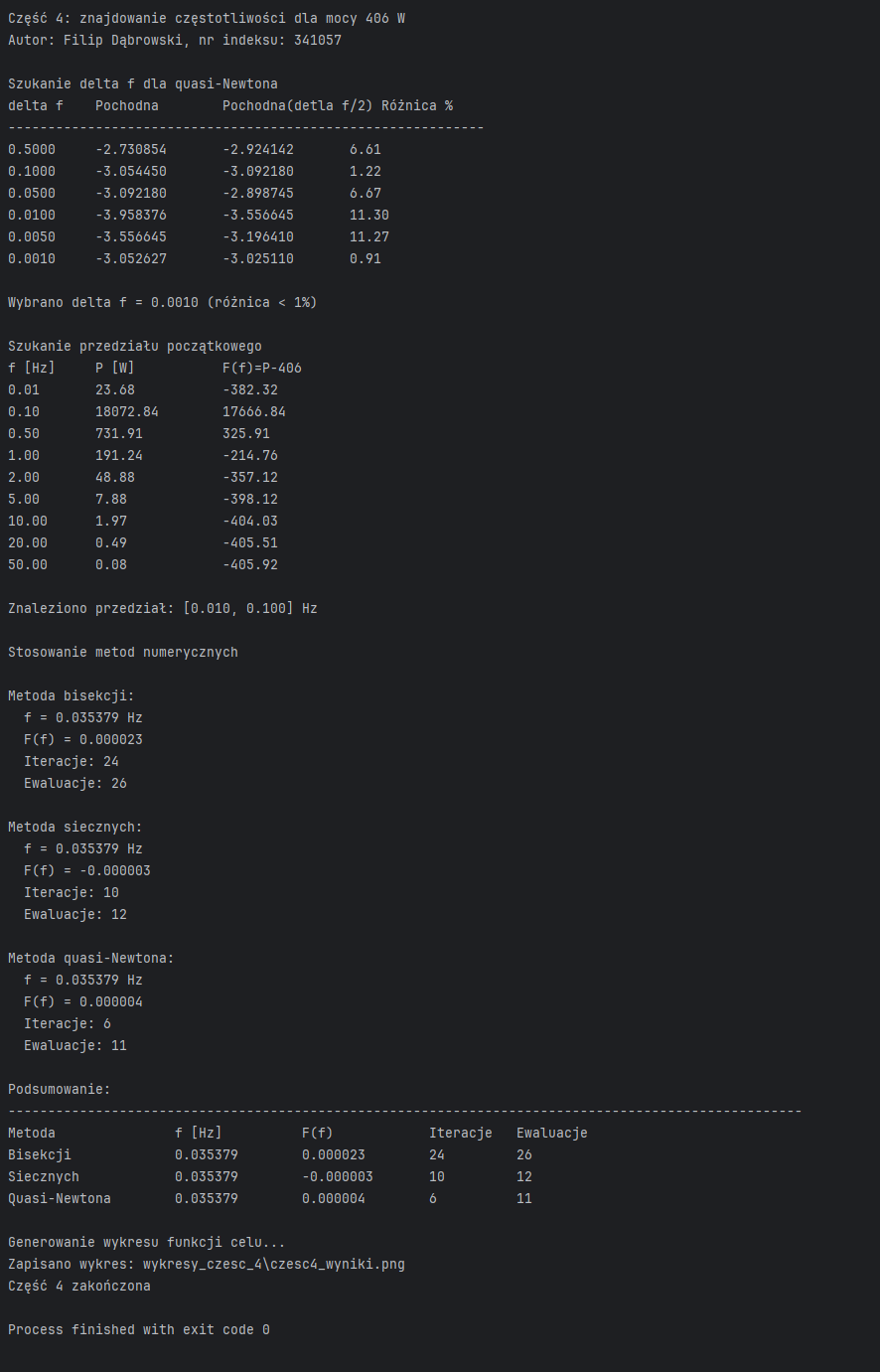
### Wymagania projektowe

W celu określenia i porównania efektywności metod w raporcie proszę przeanalizować liczbę iteracji potrzebnych do znalezienia rozwiązania poszczególnymi metodami oraz liczbę obliczeń funkcji oczekiwanej - mocy P dla wszystkich metod. **Przykładowa deklaracja funkcji, która będzie otrzymywać parametry obwodu jako argumenty zgodnie z następującą deklaracją: <- ta część jest niedokończona w zadaniu więc ją pominę.**

Należy zaimplementować trzy osobne warianty programu dla każdej z metod.

Dodatkowo należy uzupełnić tabelę.

### Wyniki



Rysunek 37: wyniki części 4

Uzupełniona tabela widoczna jest w zrzucie ekranu (podsumowanie). Jednak przepiszę ją jeszcze poniżej:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Metoda | Wartość rozwiązania -  częstotliwość *f* | Wartość funkcji *F* | Liczba iteracji metody | Liczba obliczeń mocy  P |
| Bisekcji | 0,035379 Hz | 0,000023 | 24 | 26 |
| Siecznych | 0,035379 Hz | -0,000003 | 10 | 12 |
| Quasi-Newtona | 0,035379 Hz | 0.000004 | 6 | 11 |

Wszystkie trzy metody - bisekcji, siecznych i Quasi-Newtona - zbiegły do praktycznie identycznego rozwiązania , co potwierdza poprawność ich implementacji. Różnice w wartościach funkcji celu na poziomie 10⁻⁵ świadczą o osiągnięciu zadanej dokładności.

#### Porównanie liczby iteracji i ewaluacji funkcji:

Metoda bisekcji wymagała 24 iteracji i 26 ewaluacji funkcji celu F(f) do osiągnięcia rozwiązania. Metoda siecznych potrzebowała 10 iteracji i 12 ewaluacji, natomiast metoda quasi-Newtona zaledwie 6 iteracji i 11 ewaluacji.

#### Analiza efektywności:

Liczba ewaluacji funkcji jest kluczowym wskaźnikiem efektywności w tym zadaniu, ponieważ każda ewaluacja wymaga rozwiązania pełnego układu równań różniczkowych (symulacji obwodu przez 30 sekund) oraz obliczenia całki metodą parabol. Jest to operacja obliczeniowo kosztowna.

Metoda quasi-Newtona okazała się najszybsza pod względem liczby iteracji (6), lecz pod względem liczby ewaluacji (11) była tylko nieznacznie lepsza od metody siecznych (12 ewaluacji). Metoda bisekcji była wyraźnie najmniej efektywna, wymagając ponad dwukrotnie więcej ewaluacji niż pozostałe metody.

Warto zauważyć, że w metodzie quasi-Newtona każda iteracja (poza początkową) wymaga co najmniej dwóch ewaluacji funkcji: jednej do obliczenia wartości funkcji w bieżącym punkcie i drugiej do obliczenia przybliżonej pochodnej. Mimo to, dzięki szybkiej zbieżności metody Newtona, całkowita liczba ewaluacji pozostaje niska.

#### Wnioski:

1. Metoda quasi-Newtona i metoda siecznych są znacznie bardziej efektywne niż metoda bisekcji do rozwiązywania tego typu problemów, gdyż wymagają mniej ewaluacji kosztownej funkcji celu.
2. Szybka zbieżność metody quasi-Newtona rekompensuje konieczność dodatkowych obliczeń pochodnej, co czyni ją konkurencyjną wobec metody siecznych.
3. Wybór metody w praktycznych zastosowaniach powinien uwzględniać nie tylko teoretyczną szybkość zbieżności, ale także koszt pojedynczej ewaluacji funkcji oraz konieczność określenia parametrów dodatkowych (jak Δf w metodzie quasi-Newtona).
4. W przypadku tego konkretnego problemu, gdzie funkcja celu jest gładka i dobrze uwarunkowana, wszystkie trzy metody okazały się skuteczne, lecz metody siecznych i quasi-Newtona są preferowane ze względu na niższy koszt obliczeniowy.

# Podsumowanie

W tym projekcie zbudowałem krok po kroku symulator obwodu elektrycznego, implementując kolejne metody numeryczne. Część pierwsza rozwiązuje układ równań różniczkowych, używając metod Eulera i ulepszonej metody Eulera. To pozwoliło mi zweryfikować poprawność podstawowego modelu liniowego.

Część druga rozszerzyła model o nieliniową charakterystykę sprzężenia. Zaimplementowałem wtedy cztery metody przybliżania danych z tabeli i porównałem ich wpływ na wyniki symulacji.

W części trzeciej przeszedłem do analizy energetycznej. Obliczałem całkowitą moc na rezystorach, stosując metody całkowania prostokątów i parabol. Przeanalizowałem przy tym, jak dobór kroku czasowego wpływa na dokładność wyników dla różnych typów wymuszeń.

Część czwarta połączyła wszystkie wcześniejsze elementy. Moim zadaniem było znalezienie częstotliwości, dla której moc wynosi 406 W. Zaimplementowałem i porównałem trzy metody znajdowania pierwiastków: bisekcji, siecznych i quasi-Newtona, oceniając ich efektywność przez pryzmat liczby wymaganych, kosztownych obliczeniowo ewaluacji.

Projekt był dla mnie praktycznym ćwiczeniem w konstrukcji złożonej aplikacji numerycznej - od podstawowego modelowania, przez integrację zaawansowanych algorytmów, aż po rozwiązanie konkretnego problemu inżynierskiego.

# Spis ilustracji

[Rysunek 1: struktura katalogów programu 5](#_Toc216075381)

[Rysunek 2: parametry.py 6](#_Toc216075382)

[Rysunek 3: struktura katalogu kod\_zrodlowy 9](#_Toc216075383)

[Rysunek 4: narzędzia.py 9](#_Toc216075384)

[Rysunek 5: funkcje\_calkowania.py 11](#_Toc216075385)

[Rysunek 6: legrange() 12](#_Toc216075386)

[Rysunek 7: interpolacja\_funkcje\_sklejane() 12](#_Toc216075387)

[Rysunek 8: aproksymacja\_wielomianowa() 13](#_Toc216075388)

[Rysunek 9: metoda\_eulera() 14](#_Toc216075389)

[Rysunek 10: ulepszona\_metoda\_eulera() 14](#_Toc216075390)

[Rysunek 11: metoda\_bisekcji() 15](#_Toc216075391)

[Rysunek 12: metoda\_siecznych() 16](#_Toc216075392)

[Rysunek 13: metoda\_quasi\_newton() 17](#_Toc216075393)

[Rysunek 14: liniowy.py 18](#_Toc216075394)

[Rysunek 15: nieliniowy.py 19](#_Toc216075395)

[Rysunek 16: równania różniczkowe, część 1 20](#_Toc216075396)

[Rysunek 17: projekt\_czesc\_1.py 21](#_Toc216075397)

[Rysunek 18: układ równań z części 2 projektu 23](#_Toc216075398)

[Rysunek 19: projekt\_czesc\_2.py 24](#_Toc216075399)

[Rysunek 20: projekt\_czesc\_3.py 27](#_Toc216075400)

[Rysunek 21: projekt\_czesc\_4.py 30](#_Toc216075401)

[Rysunek 22: wykres dla wymuszenia e(t) = 120sin(2πft), dla f = 50 Hz 32](#_Toc216075402)

[Rysunek 23: wykres dla wymuszenia e(t) = 210sin(2πft), dla f = 5 Hz 33](#_Toc216075403)

[Rysunek 24: wykres dla wymuszenia e(t) = 240sin(t) 33](#_Toc216075404)

[Rysunek 25: wykres dla wymuszenia prostokątnego 34](#_Toc216075405)

[Rysunek 26: przykładowe rozwiązanie z projektu dla wymuszenia e = sin(t) 34](#_Toc216075406)

[Rysunek 27: wykresy dla wymuszenia e = sin(t) wygenerowane przez mój program 35](#_Toc216075407)

[Rysunek 28: przykładowe wyniki 4 wymuszeń z projektu 35](#_Toc216075408)

[Rysunek 29: : przykładowe wyniki 4 wymuszeń z mojego programu 36](#_Toc216075409)

[Rysunek 30: wyniki dla 120\*sin(2pi\*50\*t) 37](#_Toc216075410)

[Rysunek 31: wyniki dla 240 \* sin(t) 38](#_Toc216075411)

[Rysunek 32: : wykres charakterystyki M(u) z projektu 39](#_Toc216075412)

[Rysunek 33: wykres charakterystyki M(u) mojego programu 39](#_Toc216075413)

[Rysunek 34: tabela pomiarów par napięcia i indukcyjności wzajemnej z projektu 40](#_Toc216075414)

[Rysunek 35: : tabela pomiarów par napięcia i indukcyjności wzajemnej dla 4 metod - z mojego programu 40](#_Toc216075415)

[Rysunek 36: wyniki części 3 42](#_Toc216075416)

[Rysunek 37: wyniki części 4 44](#_Toc216075417)