Politechnika Warszawska

Metody Numeryczne

Projekt zaliczeniowy

Jan Gołda

Listopad 2020

Spis treści

1 Wstęp

Celem projektu opisanego w tym raporcie było świadome zaimplementowanie oraz wykorzystanie różnorodnych technik z zakresu metod numerycznych w celu symulacji oraz optymalizacji procesu fizycznego.

1.1 Opis zjawiska

Symulowane zjawisko polega na hartowaniu metalowych prętów, a konkretniej na etapie hartowania polegającym na ich chłodzeniu. Do tego celu wykorzystuje się zbiorniki ze specjalnym olejem, który w procesie wymiany ciepła gwałtownie chłodzi zanurzane w nim pręty.

Wymiana ta może być przybliżona następującym opisem fizycznym (jest to opis znaczeni uproszczony, pomijający takie aspekty jak wymiana ciepła ze zbiornikiem, otoczeniem itp.):

$$\frac{m_b c_b}{hA}\frac{dT_b}{dt} + T_b = T_w \tag{1a}$$

$$\frac{m_w c_w}{hA}\frac{dT_w}{dt} + T_w = T_b \tag{1b}$$

Gdzie korzystamy z oznaczeń:

m_b, m_c
: Masa wyrażona w kg, odpowiednio pręta i oleju chłodzącego.

c_b, c_w
: Ciepło właściwe wyrażane w $\frac{J}{kg \cdot K}$ opisujące energię potrzebną do zmiany temperatury odpowiednio pręta i oleju w jednostce masy o jednostkę ciepła.

T_b, T_w
: Temperatura wyrażana w $°C$ odpowiednio prętu i oleju

A
: Powierzchnia wymiany ciepła pomiędzy prętem a olejem wyrażana w m^2

h
: Przewodność cieplna wyrażana w $\frac{J}{s \cdot m^2}$ opisująca zdolnośc substancji do przekazywania ciepła

2 Wykorzystane algorytmy

W rozdziale tym opisane zostały wykorzystane algorytmy oraz przedstawiona została ich implementacja.

2.1 Metoda Eulera

Do symulacji badanego zjawiska polegającego na wymianie ciepła pomiędzy prętem a cieczą chłodzącą wykorzystano numeryczny algorytm rozwiązywania równań różniczkowych nazywany metodą Eulera.

Zrealizowany został on w dwóch wersjach, różniących się sposobem obliczania Δy. W pierwszej, zwanej podstawową, zmiana y obliczana jest według wzoru:

$$\Delta y = hf\left(x_n, y_n\right) \tag{2}$$

Natomiast w drugiej, zwanej zmodyfikowaną wykorzystuje się:

$$\Delta y = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2} f\left(x_n, y_n\right)\right) \tag{3}$$

2.1.1 Metoda podstawowa

Proces wymiany ciepła zadany został jako układ dwóch równań różniczkowych zwyczajnych:

$$\frac{m_b c_b}{hA}\frac{dT_b}{dt} + T_b = T_w \tag{4a}$$

$$\frac{m_w c_w}{hA}\frac{dT_w}{dt} + T_w = T_b \tag{4b}$$

Który przekształcony może zostać do:

$$dT_b\left(x\right) = \frac{hA}{m_b c_b} x dt \tag{5a}$$

$$dT_w\left(x\right) = \frac{hA}{m_w c_w} x dt \tag{5b}$$

Korzystając z tego możemy zapisać wzór na krok metody Eulera:

$$T_b^{(i+1)} = T_b^{(i)} + dT_b\left(T_w^{(i)} - T_b^{(i)}\right) \tag{6a}$$

$$T_w^{(i+1)} = T_w^{(i)} + dT_w\left(T_b^{(i)} - T_w^{(i)}\right) \tag{6b}$$

2.1.2 Metoda zmodyfikowana

Wzory na zmodyfikowaną metodę Eulera możemy wyznaczyć na podstawie równania (3) wykorzystując dT_b oraz dT_w wyznaczone w (5):

$$T_b^{(i+\frac{1}{2})} = T_b^{(i)} + \frac{1}{2} dT_b\left(T_w^{(i)} - T_b^{(i)}\right) \tag{7a}$$

$$T_w^{(i+\frac{1}{2})} = T_w^{(i)} + \frac{1}{2} dT_w\left(T_b^{(i)} - T_w^{(i)}\right) \tag{7b}$$

$$T_b^{(i+1)} = T_b^{(i)} + dT_b\left(T_w^{(i+\frac{1}{2})} - T_b^{(i+\frac{1}{2})}\right) \tag{8a}$$

$$T_w^{(i+1)} = T_w^{(i)} + dT_w\left(T_b^{(i+\frac{1}{2})} - T_w^{(i+\frac{1}{2})}\right) \tag{8b}$$

2.1.3 Implementacja

Poniższa funkcja implementuje powyższe dwie metody wykorzystując do tego bibliotekę NumPy oraz mechanizm generatorów, który dostępny jest w języku Python.

```
def heat_exchange_simulation(Tb, Tw, mb, mw, cb, cw, k, a, dt, improved=True):
    h = k if callable(k) else (lambda dT: k)

    t = 0.0
    T = np.array([Tb, Tw])
    m = np.array([mb, mw])
    c = np.array([cb, cw])

    dT = lambda T: h(np.diff(T)) * a / m / c * np.diff(T) * np.array([1, -1])

    while True:
        yield t, *T
        t += dt

        if improved:
            Tp = T + dt/2 * dT(T)
            T += dt * dy(Tp)
        else:
            T += dt * dy(T)
```

Kod źródłowy 1: Metoda Eulera

W praktyce powyższy kod został następnie przekształcony do postaci obiektowej, co zmniejszyło jego czytelność ale ułatwiło wykorzystanie w eksperymentach.

2.2 Aproksymacja wielomianowa

Jednym z algorytmów wykorzystanych w projekcie był algorytm aproksymacji wielomianowej metodą najmniejszych kwadratów.

Metoda ta polega na dopasowaniu wielomianu $W_m(x)$ zadanego stopnia m tak aby błąd aproksymacji wyrażony jako suma kwadratów odległości w punktach x_i pomiędzy y_i a $W_m(x_i)$ był jak najmniejszy.

Tak sformułowane zadanie można wyrazić równaniem macierzowym:

$$M^T M A = M^T Y \tag{9}$$

Po przekształceniu:

$$A = \left(M^T M\right)^{-1} M^T Y \tag{10}$$

Gdzie:

$$M = \begin{bmatrix} x_0^0 & x_0^1 & \cdots & x_0^m \\ x_1^0 & x_1^1 & \cdots & x_1^m \\ \vdots & \vdots & \ddots & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^m \end{bmatrix} \quad Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{11}$$

W efekcie otrzymując macierz A zawierającą współczynniki poszukiwanego wielomianu stopnia m.

2.2.1 Implementacja

```python
class PolynomialApproximation:

    def __init__(self, samples, degree):
        self._samples = samples
        self._degree = degree

    def __call__(self, x):
        return (self._matrix_A * x ** np.arange(self._degree + 1)).sum()

    @cached_property
    def _matrix_A(self):
        M = self._matrix_M
        Y = self._matrix_Y
        return np.linalg.inv(M.T.dot(M)).dot(M.T).dot(Y)

    @cached_property
    def _matrix_M(self):
        return self._samples[:, :1] ** np.arange(self._degree + 1)

    @cached_property
    def _matrix_Y(self):
        return self._samples[:, 1]
```

Kod źródłowy 2: Aproksymacja wielomianowa

2.3 Interpolacja funkcjami sklejanymi

W przypadku interpolacji funkcjami sklejanymi skorzystałem z wersji algorytmu, z którym miałem do czynienia już wcześniej. Pozwala on na tworzenie funkcji sklejanych bez ograniczenia co do odległości pomiędzy węzłami interpolacji.

Mając dane węzły interpolacji:

$$\{(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)\} \tag{12}$$

Poszukiwaliśmy funkcji sklejanej o następującej postaci:

$$f(x) = \begin{cases} f_1(x) & \text{for } x \in (-\infty, x_1) \\ \vdots & \\ f_k(x) & \text{for } x \in [x_{k-1}, x_k) \\ \vdots & \\ f_n(x) & \text{for } x \in [x_{n-1}, x_n) \end{cases} \tag{13}$$

Gdzie:

$$f_k(x) = a_k x^3 + b_k x^2 + c_k x + d_k \tag{14}$$

Aby znaleźć te funkcje rozwiązany został układ równań z $4n$ niewiadomymi:

$$A_{4n \times 4n} X_{4n} = B_{4n} \tag{15}$$

Równania te bazują na czterech warunkach.

2.3.1 Warunek 1: Każda funkcja przechodzi przez ograniczające ją węzły interpolacji

Warunek ten wyrażony może być jako (gdzie $k \in \{1, 2, ..., n\}$):

$$\begin{cases} f_k(x_{k-1}) = y_{k-1} \\ f_k(x_k) = y_k \end{cases} \tag{16}$$

Co daje nam:

$$\begin{cases} a_k x_{k-1}^3 + b_k x_{k-1}^2 + c_k x_{k-1} + d_k = y_{k-1} \\ a_k x_k^3 + b_k x_k^2 + c_k x_k + d_k = y_k \end{cases} \tag{17}$$

A zatem uzyskujemy macierz A_1 rozmiaru $(2n \times 4n)$:

$$A_1 = \begin{bmatrix} x_0^3 & x_0^2 & x_0^1 & x_0^0 & 0 & 0 & 0 & 0 & \cdots \\ x_1^3 & x_1^2 & x_1^1 & x_1^0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & x_1^3 & x_1^2 & x_1^1 & x_1^0 & \cdots \\ 0 & 0 & 0 & 0 & x_2^3 & x_2^2 & x_2^1 & x_2^0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{18}$$

Oraz macierz B_1 rozmiaru $(2n \times 1)$:

$$B_1 = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \end{bmatrix} \tag{19}$$

2.3.2 Warunek 2: Pierwsze pochodne są zgodne w punktach interpolacji

Co możemy wyrazić jako (gdzie $k \in \{1, 2, ..., n-1\}$):

$$\frac{d}{dx} f_k(x_k) = \frac{d}{dx} f_{k+1}(x_k) \tag{20}$$

Co daje nam:

$$3a_k x_k^2 + 2b_k x_k + c_k + 1 = 3a_{k+1} x_k^2 + 2b_{k+1} x_k + c_{k+1} + 1 \tag{21a}$$

$$3a_k x_k^2 + 2b_k x_k + c_k - 3a_{k+1} x_k^2 - 2b_{k+1} x_k - c_{k+1} = 0 \tag{21b}$$

A zatem uzyskujemy macierz A_2 rozmiaru $(n-1 \times 4n)$:

$$A_2 = \begin{bmatrix} 3x_1^2 & 2x_1^1 & x_1^0 & 0 & -3x_1^2 & -2x_1^1 & -x_1^0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 3x_2^2 & 2x_2^1 & x_2^0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{22}$$

Oraz macierz B_2 rozmiaru $(n-1 \times 1)$:

$$B_2 = \begin{bmatrix} 0 \\ 0 \\ \vdots \end{bmatrix} \tag{23}$$

2.3.3 Warunek 3: Drugie pochodne są zgodne w punktach interpolacji

Warunek ten wyrażony może być jako (gdzie $k \in \{1, 2, ..., n-1\}$):

$$\frac{d^2}{dx^2} f_k(x_k) = \frac{d^2}{dx^2} f_{k+1}(x_k) \tag{24}$$

Co daje nam:

$$6a_k x_k + 2b_k + 1 = 6a_{k+1} x_k + 2b_{k+1} + 1 \tag{25a}$$

$$6a_k x_k + 2b_k - 6a_{k+1} x_k - 2b_{k+1} = 0 \tag{25b}$$

A zatem uzyskujemy macierz A_3 rozmiaru $(n-1 \times 4n)$:

$$A_3 = \begin{bmatrix} 6x_1^1 & 2x_1^0 & 0 & 0 & -6x_1^1 & -2x_1^0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 6x_2^1 & 2x_2^0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{26}$$

Oraz macierz B_2 rozmiaru $(n-1 \times 1)$:

$$B_3 = \begin{bmatrix} 0 \\ 0 \\ \vdots \end{bmatrix} \tag{27}$$

2.3.4 Warunek 4: Druga pochodna zeruje się w pierwszym i ostatnim punkcie interpolacji

Co możemy wyrazić jako:

$$\begin{cases} \frac{d^2}{dx^2} f_1(x_0) = 0 \\ \frac{d^2}{dx^2} f_n(x_n) = 0 \end{cases} \tag{28}$$

Co daje nam:

$$\begin{cases} 6a_1 x_0 + 2b_1 + 1 = 0 \\ 6a_n x_n + 2b_n + 1 = 0 \end{cases} \tag{29}$$

A zatem uzyskujemy macierz A_4 rozmiaru $(2 \times 2n)$:

$$A_4 = \begin{bmatrix} 6x_0 & 2 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 6x_n & 2 & 1 \end{bmatrix} \tag{30}$$

Oraz macierz B_4 rozmiaru (2×1):

$$B_4 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{31}$$

2.3.5 Finalne równanie macierzowe

Finalne równanie macierzowe ma postać $AX = B$, gdzie:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \; X = \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ a_2 \\ b_2 \\ c_2 \\ d_2 \\ \vdots \end{bmatrix} \; B = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{bmatrix} \tag{32}$$

2.3.6 Implementacja

Finalna implementacja opisanego algorytmu została wykonana z wykorzystaniem biblioteki NumPy oraz podejścia obiektowego.

Ma ona około 100 linijek w związku z czym odsyłam do pliku `solution/interpolation.py` w załączonym kodzie źródłowym.

2.4 Metoda Simpsona

Metoda Simpsona, nazywana też metodą parabol, pozwala nam na wyznaczenie przybliżenia całki właściwej z funkcji $f(x)$ na przedziale $[a, b]$ poprzez podzielenie tego przedziału na m części (m - parzyste) na których możemy wyznaczyć przybliżenie całki korzystając z wielomianu drugiego stopnia:

$$\int_{x_i}^{x_{i+2}} f(x)dx \approx \frac{b-a}{3m}\Big(f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\Big) \tag{33}$$

A zatem sumując otrzymane całki otrzymujemy:

$$\int_a^b f(x)dx \approx \frac{b-a}{3m}\Big[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 4f(x_{m-1}) + f(x_m)\Big] \tag{34}$$

2.4.1 Implementacja

Równanie (34) możemy wyrazić jako sumę elementów przemnożonych przez siebie wektorów, co znacznie ułatwia implementację:

```
1 def simpson_integral(f, a, b, m):
2     xs = np.linspace(a, b, m + 1)
3     ys = np.array(list(map(f, xs)))
4 
5     constants = np.fromfunction(lambda i: 2 + 2 * (i % 2), shape=(m + 1,))
6     constants[[0, -1]] = 1
7 
8     return (b - a) / m / 3 * (constants * ys).sum()
```

Kod źródłowy 3: Metoda Simpsona

2.5 Generacja równo odległych węzłów

W projekcie wymagane było również wygenerowanie równo odległych pomiarów na podstawie zadanych danych. W tym celu wykorzystane zostały funkcje sklejane pierwszego rzędu, na podstawie których następnie uzyskano równo odległe węzły.

2.5.1 Funkcje sklejane pierwszego rzędu

Algorytm ten polega na prostej liniowej interpolacji wartości w przedziale na podstawie jego znanych skrajnych wartości.

```
class LinearSpline:

    def __init__(self, samples):
        self._samples = samples

    def __call__(self, x):
        if not self._samples[0, 0] <= x <= self._samples[-1, 0]:
            return np.nan

        i = np.searchsorted(self._samples[:, 0], x)
        dx = self._samples[i, 0] - self._samples[i - 1, 0]
        dy = self._samples[i, 1] - self._samples[i - 1, 1]
        a = dy / dx
        return self._samples[i - 1, 1] + a * (x - self._samples[i - 1, 0])
```

Kod źródłowy 4: Funkcje sklejane pierwszego rzędu

2.5.2 Tabularyzacja funkcji

Mając daną funkcję f zdefiniowana na przedziale $[a, b]$ możemy łatwo wyznaczyć n równo odległych pomiarów korzystając z poniższego kodu:

```
def tabularize(f, a, b, n):
    xs = np.linspace(a, b, n)
    ys = np.array(list(map(f, xs)))
    return np.column_stack([xs, ys])
```

Kod źródłowy 5: Tabularyzacja funkcji

2.6 Metoda Newtona-Raphsona

Metoda Newtona-Raphsona jest iteracyjnym algorytmem pozwalającym na znalezienie (przybliżonego) pierwiastka x_p funkcji f należącej do klasy C^2 w zadanych przedziale, w którym to pierwsza oraz druga pochodna tej funkcji ma stały znak.

Klasyczna metoda opiera się na wyznaczeniu ciągu iteracyjnego x_n który dąży do naszego szukanego x_p. Ciąg ten zadany jest wzorem:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{35}$$

W tym projekcie metoda ta wykorzystywana będzie dla funkcji dla których nie znamy dokładnych wartości $f'(x)$ w związku z czym zostaną one przybliżone wg wzoru:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{36}$$

2.6.1 Stabilność numeryczna

Niestety metoda w wersji przytoczonej powyżej jest niestabilna numerycznie, co berze się z dzielenia f przez f' w (36). Gdy mamy do czynienia z funkcją, która przyjmuje w x_n wartość znacznie odstającą rzędem wielkości od pochodnej w tym punkcie zaczynamy obserwować duże błędy obliczeniowe.

Aby zniwelować ten problem poniższa implementacja uzupełniona została o parametr skali S, który (odpowiednio dobrany) pozwala zniwelować ten problem poprzez odpowiednie przeskalowanie wartości badanej funkcji podczas wyznaczania zmiany x:

$$x_{n+1} = x_n - S\frac{f(x_n)}{f'(x_n)} \tag{37a}$$

$$f'(x) \approx \frac{S(f(x + \Delta x) - f(x))}{\Delta x} \tag{37b}$$

2.6.2 Implementacja

```
class NewtonRaphson:

    def __init__(self, f, x0, dx, scale=1.0):
        self._f = f
        self._x = x0
        self._dx = dx
        self._scale = scale

        self._history = [(self.x, self.y)]

    def step(self):
        y1 = self._f(self._x)
        y2 = self._f(self._x + self._dx)
        dy = (y2 - y1) * self._scale
        df = dy / self._dx

        self._x -= y1 / df * self._scale

    ...
```

Kod źródłowy 6: Metoda Newtona-Raphsona

3 Część 1: Symulacja procesu wymiany ciepła

Rozdział ten opisuje pierwszą część projektu, w której to zaimplementowano i przetestowano model symulujący wymianę ciepła pomiędzy prętem a cieczą chłodzącą. Szczegółowy opis implementacji znajduje się w sekcji 2.1, natomiast testy i wyniki symulacji znajdują się poniżej.

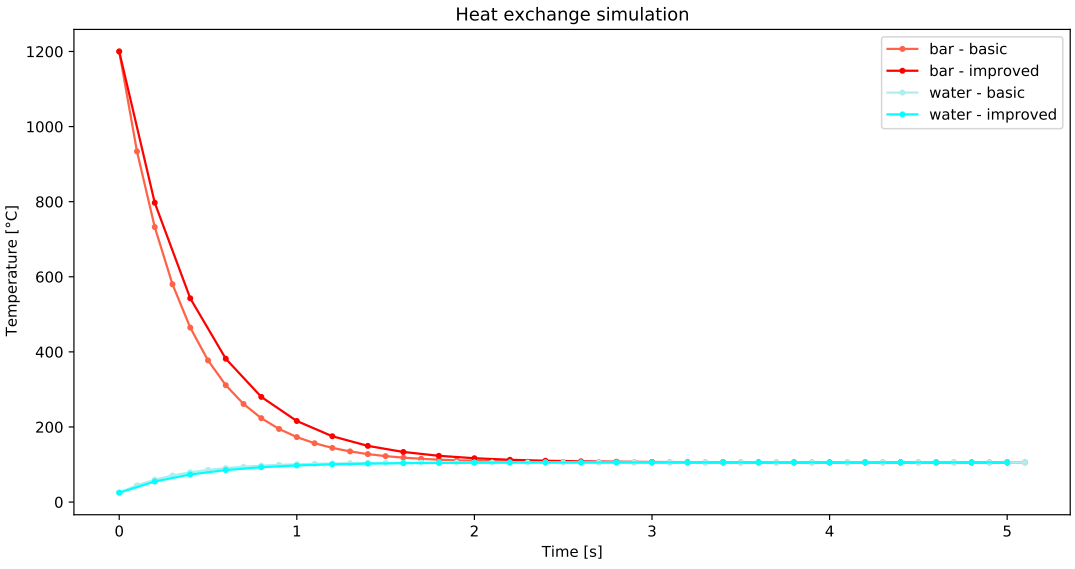
Warto zwrócić uwagę, że w tym rozdziale testowane są dwa warianty metody Eulera, metoda podstawa oraz metoda zmodyfikowana. Jako że metoda zmodyfikowana w każdym kroku symulacji dwa razy oblicza pochodną (efektywnie wykonując dwa kroki), we wszystkich testach w tym rozdziale metoda zmodyfikowana uruchamiana jest z dwukrotnie większym krokiem symulacji, co pozwala na poprawne porównanie tych metod.

3.1 Wstępna weryfikacja rozwiązania

Weryfikację poprawności rozwiązania rozpoczęto od przeprowadzenia próbnej symulacji mającej na celu wstępne porównanie jej przebiegu z przebiegiem podanym w opisie projektu. Parametry symulacji były następujące (poprzez zapis $dt = 0.1, 0.2$ rozumiemy krok 0.1 dla metody podstawowej oraz krok 0.2 dla metody zmodyfikowanej:

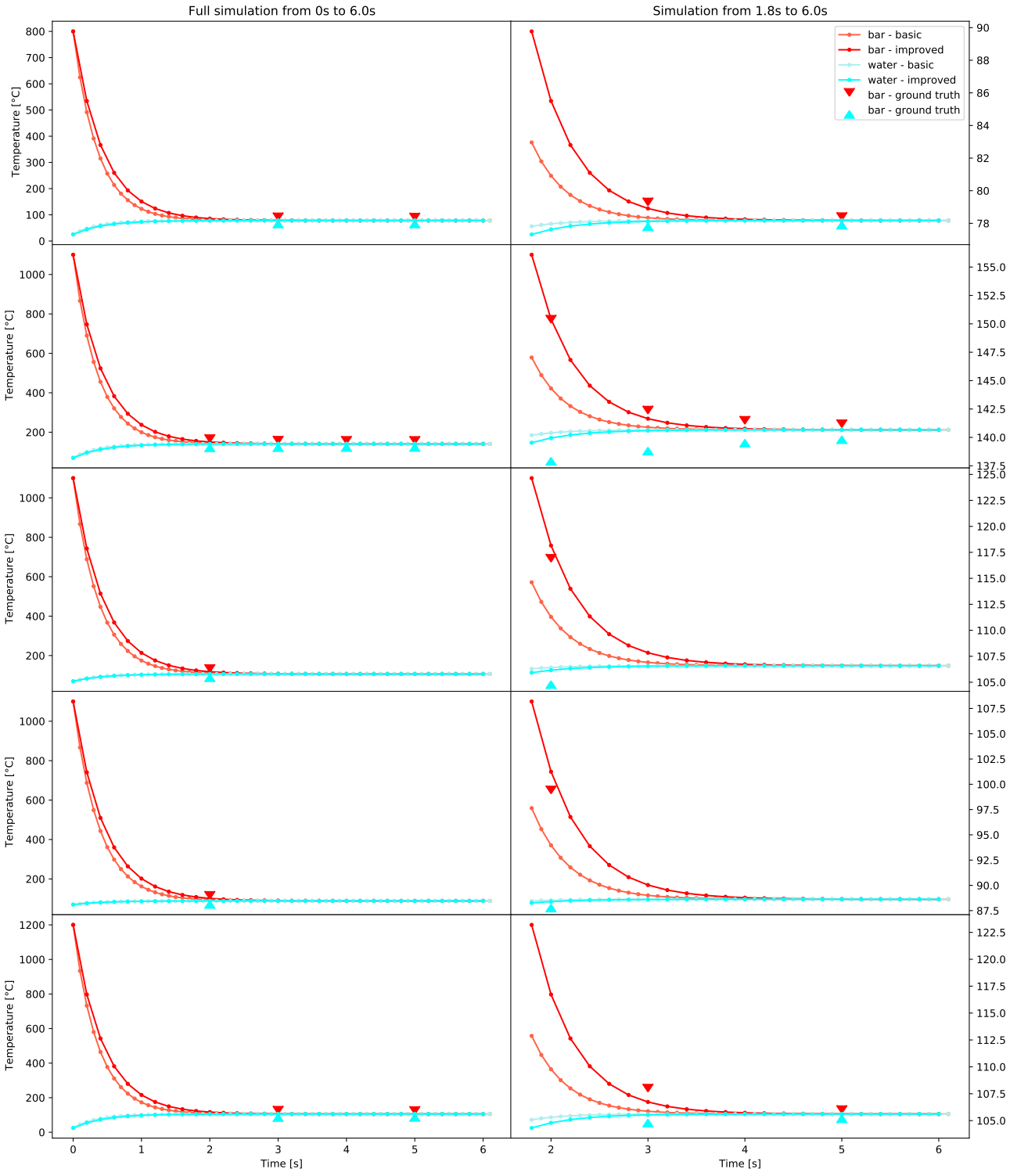
$$T_b = 1200.0 \qquad m_b = 0.2 \qquad c_b = 3.85$$
$$T_w = 25.0 \qquad m_w = 2.5 \qquad c_w = 4.1813$$
$$h = 160 \qquad a = 0.0109 \qquad dt = 0.1, 0.2$$

Uzyskany przebieg symulacji prezentuje się następująco:



Rysunek 1: Przebieg testowej symulacji wymiany ciepła

Analiza rysunku 1 pozwoliła wstępnie potwierdzić poprawność zaimplementowanego algorytmu.



Rysunek 2: Przebiegi symulacji dla danych pomiarowych

3.2 Przebiegi dla zadanych eksperymentów pomiarowych

W celu dokładniejszej weryfikacji poprawności implementacji wykorzystano dane pomiarowe przedstawione w tabeli 1 dla których to wygenerowano przebiegi symulacji z krokami czasowymi odpowiednio $dt = 0.1$ oraz $dt = 0.2$, czego rezultaty widoczne na rysunku 2.

Tablica 1: Dane z eksperymentów pomiarowych

	$\mathbf{T_b}$	$\mathbf{T_w}$	$\mathbf{m_b}$	$\mathbf{m_w}$	$\mathbf{c_b}$	$\mathbf{c_w}$	**h**	**a**	**t**	$\mathbf{T_b(t)}$	$\mathbf{T_w(t)}$
2	800.0	25.0	0.2	2.5	3.85	4.1813	160.0	0.0109	3.0	79.1	78.0
5	800.0	25.0	0.2	2.5	3.85	4.1813	160.0	0.0109	5.0	78.2	78.1
6	1100.0	70.0	0.2	2.5	3.85	4.1813	160.0	0.0109	2.0	150.1	138.2
3	1100.0	70.0	0.2	2.5	3.85	4.1813	160.0	0.0109	3.0	142.1	139.1
9	1100.0	70.0	0.2	2.5	3.85	4.1813	160.0	0.0109	4.0	141.2	139.8
10	1100.0	70.0	0.2	2.5	3.85	4.1813	160.0	0.0109	5.0	140.9	140.1
7	1100.0	70.0	0.2	5.0	3.85	4.1813	160.0	0.0109	2.0	116.6	105.1
8	1100.0	70.0	0.2	10.0	3.85	4.1813	160.0	0.0109	2.0	99.1	88.1
1	1200.0	25.0	0.2	2.5	3.85	4.1813	160.0	0.0109	3.0	107.7	105.1
4	1200.0	25.0	0.2	2.5	3.85	4.1813	160.0	0.0109	5.0	105.7	105.5

Wstępna analiza przebiegów widocznych w lewej kolumnie na rysunku 2 pokazuje, że symulacja przebiega zgodnie z oczekiwaniami. Jednak po dokładniejszej analizie z wykorzystaniem prawej kolumny (są to te same, przybliżone wykresy) widzimy, że nie w każdym przypadku uzyskane wyniki są idealne.

Dokładna analiza błędów znajduje się w następnym podrozdziale, natomiast na tą chwilę warto zauważyć że zmodyfikowana/ulepszona wersja algorytmu Eulera rzeczywiście jest lepsza, ponieważ w porównaniu do wersji podstawowej daje wyniki bliższe danym pomiarowym.

3.3 Analiza błędów

W tabeli 2 przedstawione zostało dokładne porównanie wyników uzyskanych dwoma metodami w odniesieniu do danych pomiarowych z tabeli 1.

Analizując wartości w kolumnach `Error` widzimy, że każdym przypadku metoda zmodyfikowana jest lepsza od metody podstawowej. W niektórych przypadkach różnica ta jest bardzo mała, ale są też takie w których metoda zmodyfikowana jest dwukrotnie lepsza.

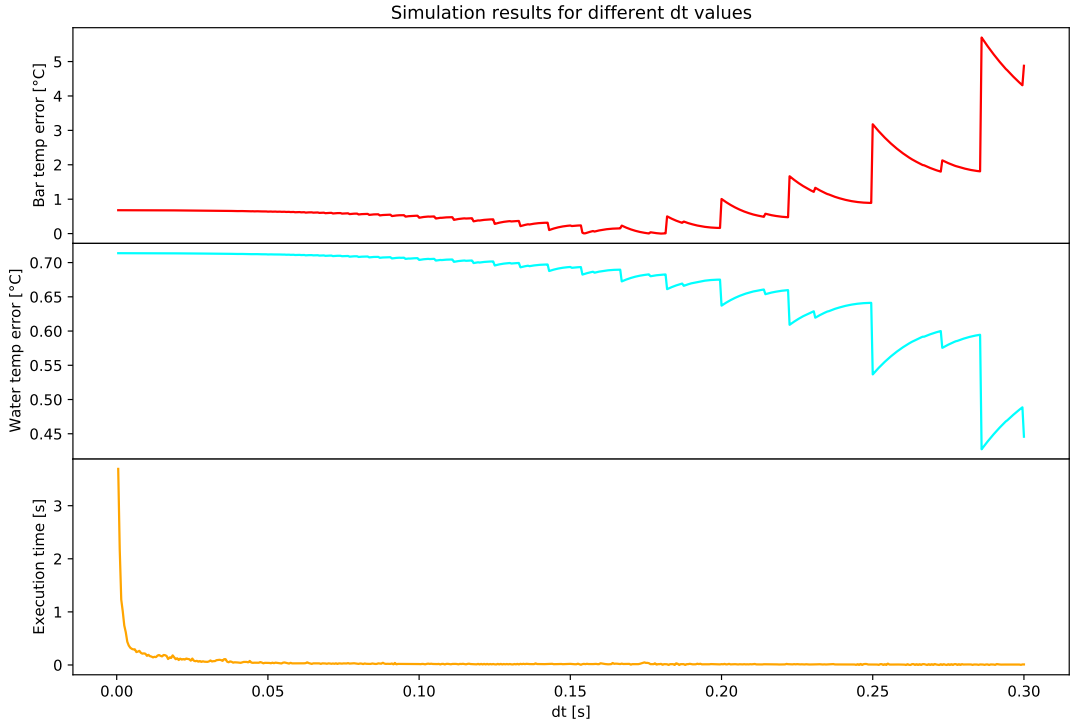
Tablica 2: Błędy symulacji względem danych pomiarowych

	$\mathbf{T_b}$					$\mathbf{T_w}$				
	gt	**Basic**		**Improved**		**gt**	**Basic**		**Improved**	
	Value	Value	Error	Value	Error	Value	Value	Error	Value	Error
2	79.1	78.34	0.7600	78.91	0.1907	78.0	78.16	0.1584	78.12	0.1165
5	78.2	78.17	0.0286	78.18	0.0216	78.1	78.17	0.0708	78.17	0.0703
6	150.1	144.31	5.7879	146.82	3.2793	138.2	140.40	2.1972	140.21	2.0124
3	142.1	140.89	1.2094	141.65	0.4528	139.1	140.65	1.5492	140.59	1.4935
9	141.2	140.68	0.5203	140.76	0.4353	139.8	140.66	0.8648	140.66	0.8585
10	140.9	140.67	0.2335	140.68	0.2242	140.1	140.67	0.5658	140.67	0.5651
7	116.6	111.29	5.3106	114.00	2.6028	105.1	106.41	1.3149	106.32	1.2151
8	99.1	93.96	5.1390	96.77	2.3340	88.1	88.53	0.4265	88.47	0.3749
1	107.7	105.87	1.8297	106.73	0.9665	105.1	105.60	0.4950	105.53	0.4314
4	105.7	105.61	0.0854	105.63	0.0747	105.5	105.61	0.1138	105.61	0.1131

3.4 Analiza wpływu parametrów

Algorytm ten posiada 9 parametrów, jednak tylko jeden, krok czasowy dt, jest parametrem związanym z samą symulacją, a nie z opisem zjawiska fizycznego. Dlatego też to na tym parametrze się skupiono, próbując tak dobrać jego wartość aby symulacja dawała jak najlepsze wyniki.

W ramach badania przetestowano dt z zakresu $[0.0005, 0.3]$.



Rysunek 3: Wyniki symulacji w zależności od dt

Z analizy wykresu 3 wyciągnięto następujące wnioski:

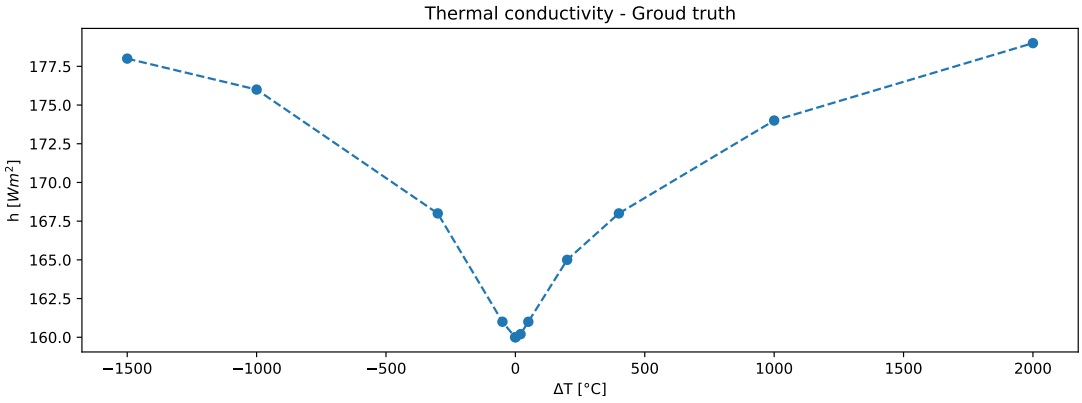
- Najlepsze wyniki symulacji temperatury prętu otrzymujemy dla $dt \in [0.15, 0.18]$.
- Dla wartości dt powyżej ok. 0.15 zaczynamy obserwować niestabilność numeryczną, jako że krok symulacji jest zbyt duży i tracimy dokładność przybliżenia.
- Z wykresu można by wywnioskować, że im większe dt tym mniejszy błąd Tw, jednak warto zwrócić uwagę na skalę wykresów. Błąd Tw jest nie znaczący w porównaniu do Tb. W związku z tym wartości $dt \in [0.15, 0.18]$ są najbardziej optymalne.

Tablica 3: Dane pomiarowe współczynnika ciepła h

$\mathbf{\Delta T[^\circ C]}$	-1500	-1000	-300	-50	-1	1	20	50	200	400	1000	2000
$\mathbf{h[Wm^{-2}]}$	178	176	168	161	160	160	160.2	161	165	168	174	179

4 Część 2: Wyznaczanie funkcji współczynnika ciepła

Celem tej części projektu było wyznaczenie funkcji opisującej współczynnik ciepła h w zależności od różnicy temperatur $|T_b - T_w|$ na podstawie zadanych danych pomiarowych widocznych w tabeli 3 oraz na wykresie 4.

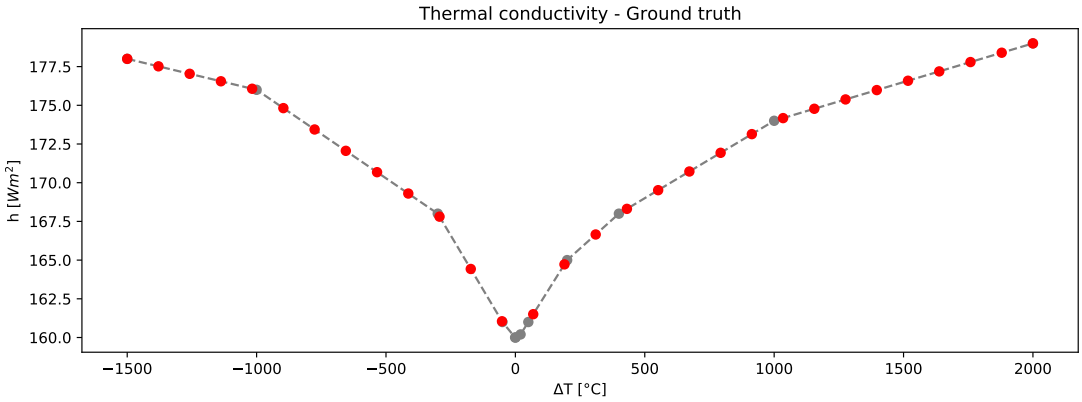


Rysunek 4: Dane pomiarowe współczynnika ciepła h

4.1 Generacja równoodległych węzłów

Podane dane pomiarowe współczynnika ciepła nie są równomiernie rozłożone na osi OX co może być problematyczne dla niektórych algorytmów interpolacji/aproksymacji. Algorytmy wykorzystane w tym projekcie nie mają takich ograniczeń mimo to zdecydowałem się na zaimplementowanie algorytmu generacji równoodległych węzłów w ramach ćwiczenia.

Implementacja, wykorzystująca funkcje sklejane pierwszego rzędu, została szczegółowo opisana w podrozdziale 2.5. Wizualizacja tego procesu dla 30 węzłów (istnieje pełna możliwość doboru tej wartości) znajduje się na rysunku 5

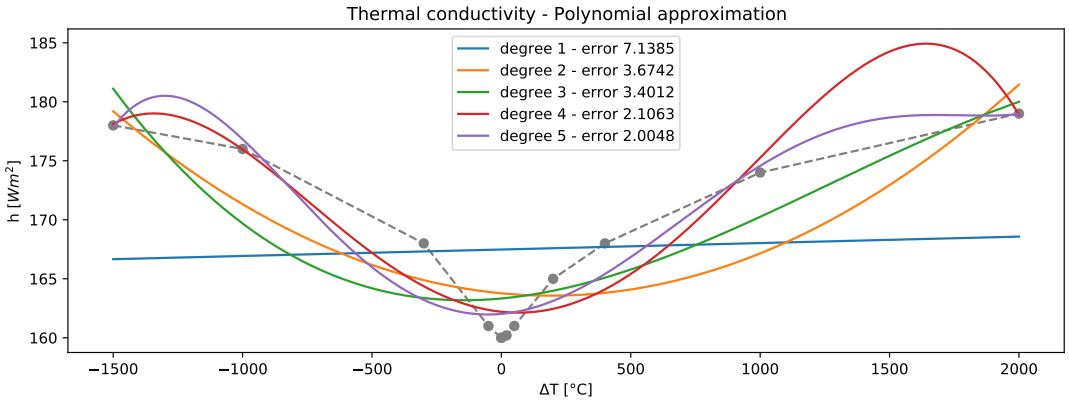


Rysunek 5: Generacja równoodległych węzłów współczynnika ciepła h

4.2 Aproksymacja wielomianowa

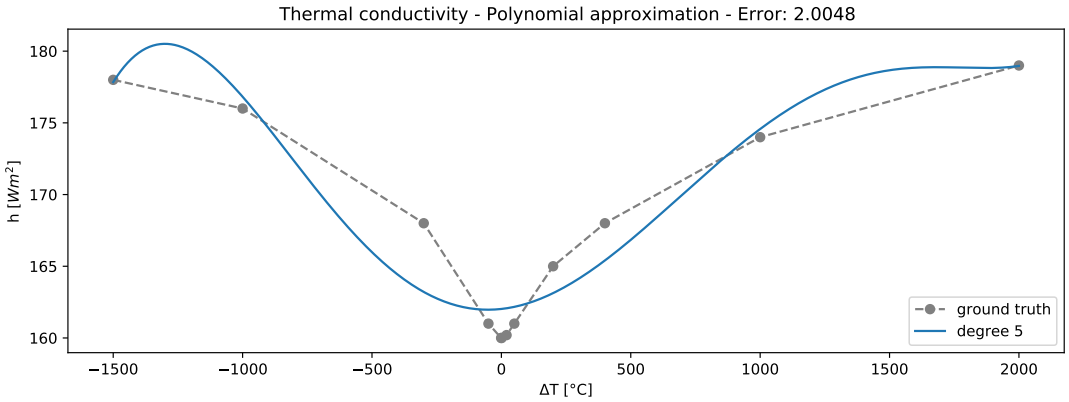
Pierwszym algorytmem zastosowanym do wyznaczenia przebiegu funkcji h był algorytm aproksymacji wielomianowej metodą najmniejszych kwadratów. Został on szczegółowo opisany i zaimplementowany w podrozdziale 2.2.

Na rysunku 6 przedstawiono wielomiany aproksymujące różnego stopnia wraz z błędami aproksymacji. Dla stopnia powyżej 5, funkcje te znacząco odstawały od oczekiwanej krzywej, dlatego zostały pominięte.



Rysunek 6: Wielomiany aproksymujące funkcję h

Najlepsze wyniki (najmniejszy błąd aproksymacji) uzyskano dla wielomianu stopnia 5. Dlatego w dalszych częściach to właśnie jego będziemy używać.

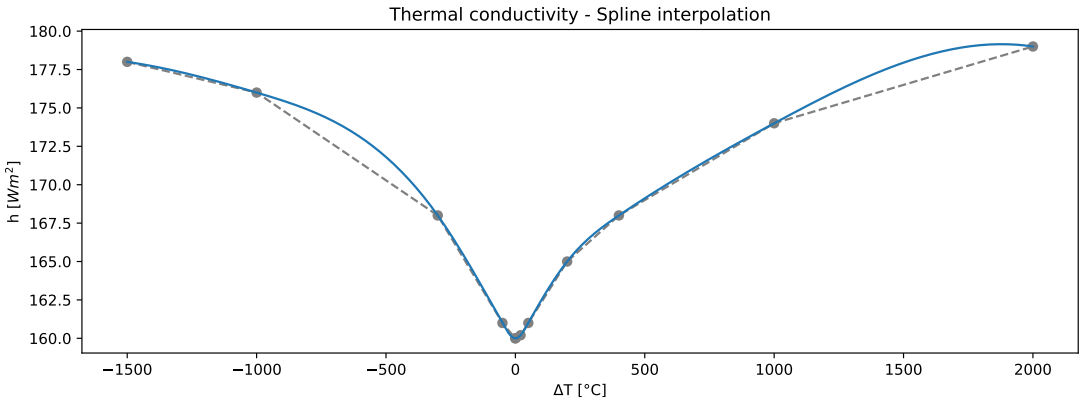


Rysunek 7: Wielomian stopnia 5 aproksymujący funkcję h

4.3 Interpolacja funkcjami sklejanymi

Drugim zastosowanym algorytmem był algorytm interpolacji funkcjami sklejanymi trzeciego stopnia. Zaimplementowana przeze mnie wersja (opisana w podrozdziale 2.3) nie wymaga aby węzły były równoodległe, a zatem można jej użyć bezpośrednio na dostępnych danych.

Jak widać na wykresie 8 funkcja sklejana dała zdecydowanie lepsze wyniki niż te uzyskane aproksymacją wielomianową. Dokładne ich porównanie znajduje się w następnym podpunkcie.



Rysunek 8: Funkcja sklejana interpolująca h

4.4 Różnica przebiegów

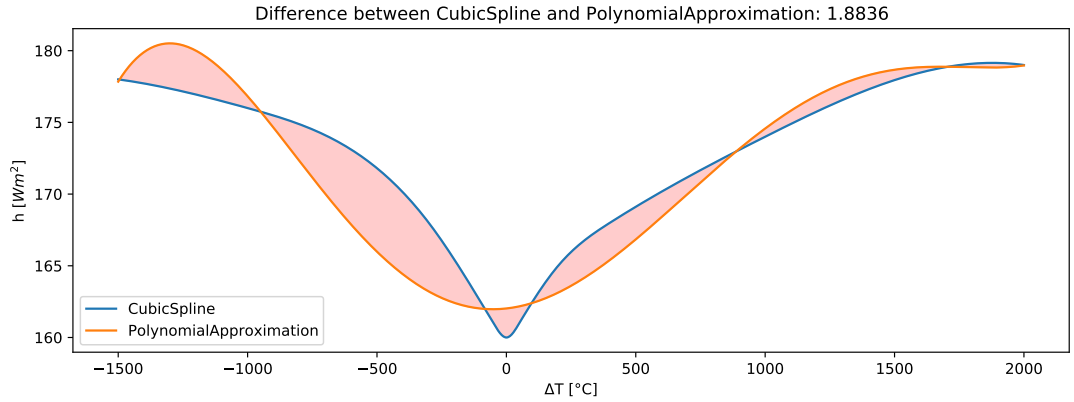
W celu porównania powyżej przedstawionych metod zastosowano miarę określającą średnią różnicę przebiegów dwóch funkcji f i g na przedziale $[a, b]$ zdefiniowaną jako:

$$e_{\text{avg}} = \frac{1}{b-a} \int_a^b |g(x) - f(x)| dx \tag{38}$$

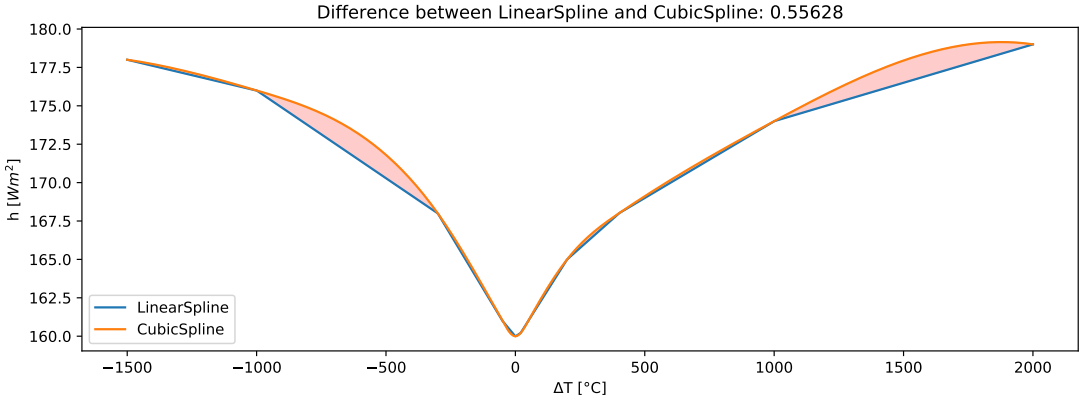
Przybliżenie powyższej całki wyznaczane było z wykorzystaniem metody Simpsona opisanej w 2.4.

Na rysunkach 9, 10 oraz 11 przedstawioan została wizualizacja e_{avg}.

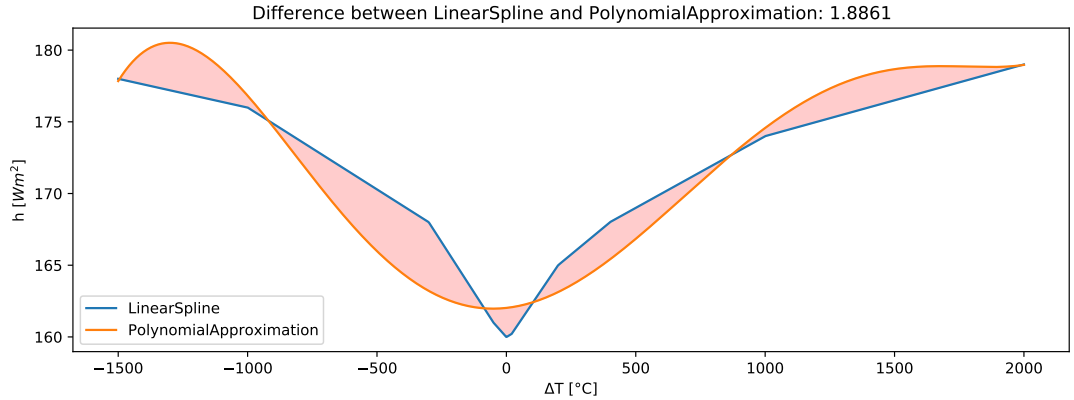
Interpolacja funkcjami sklejanymi uzyskała błąd względem interpolacji liniowej równy 0.55628, podczas gdy aproksymacja wielomianowa otrzymała 1.8861. W związku z czym możemy potwierdzić obserwację z poprzedniego punktu i uznać, że interpolacja funkcjami sklejanymi sprawdza się lepiej w przypadku tych danych.



Rysunek 9: Rożnica między CubicSpline oraz PolynomialApproximation



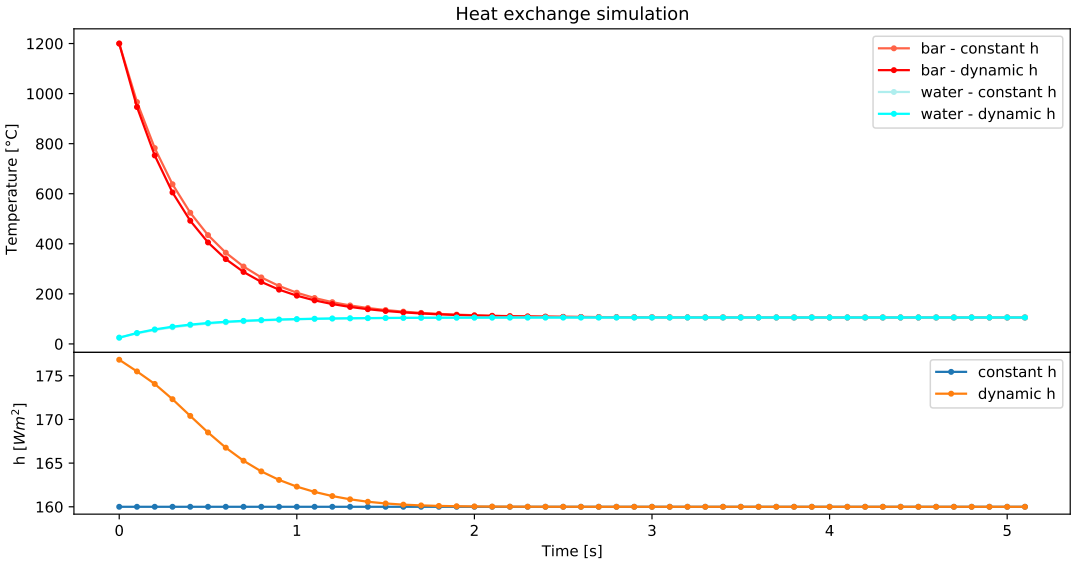
Rysunek 10: Rożnica między LinearSpline oraz CubicSpline



Rysunek 11: Rożnica między LinearSpline oraz PolynomialApproximation

4.5 Wpływ dynamicznego współczynnika h na symulację

W celu zbadania wpływu wykorzystania dynamicznego współczynnika h w symulacji (zamiast stałego, przyjętego wcześniej jako 160), przygotowano wykres przedstawiający przebiegi symulacji (rys. 12) oraz tabelę zawierającą porównanie błędów (tabela 4).



Rysunek 12: Przebieg symulacji dla stałego i dynamicznego h

Tablica 4: Porównanie błędów symulacji dla stałego i dynamicznego h

	$\mathbf{T_b}$ **(const)**	$\mathbf{T_b}$ **(dynamic)**	$\mathbf{T_w}$ **(const)**	$\mathbf{T_b}$ **(dynamic)**
2	0.004995	0.005641	0.001686	0.001735
5	0.000329	0.000333	0.000904	0.000905
6	0.010571	0.016693	0.013660	0.014149
3	0.005098	0.005683	0.010881	0.010925
9	0.003329	0.003382	0.006159	0.006163
10	0.001630	0.001634	0.004036	0.004036
7	0.004014	0.013635	0.010814	0.011207
8	0.000571	0.011942	0.003755	0.004015
1	0.011852	0.012810	0.004322	0.004395
4	0.000765	0.000772	0.001076	0.001076

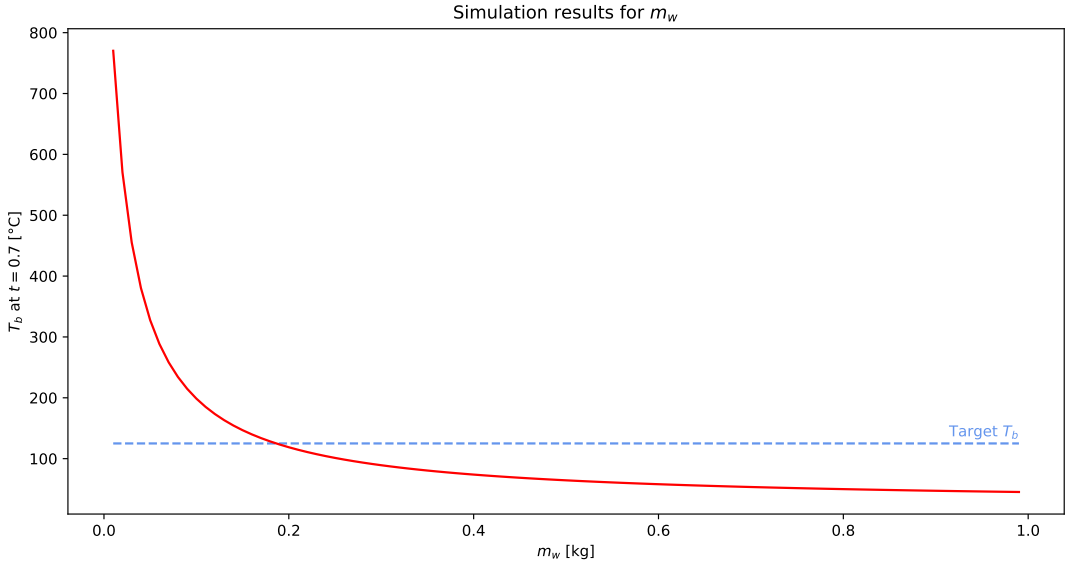
Zarówno przebiegi symulacji jak i błędy przedstawione w tabeli pokazują, że mimo iż obydwa podejścia dają bardzo zbliżone wyniki, to niestety symulacje z dynamicznym współczynnikiem h są nieznacznie gorsze. Przyczyn takiej sytuacji może być wiele, począwszy od sposobu uzyskania danych pomiarowych, a skończywszy na sposobie przeprowadzania interpolacji oraz symulacji, w związku z czym w projekcie tym pominięto poszukiwania tej przyczyny.

5 Część 3: Wyznaczanie minimalnej masy oleju

Celem tej części projektu było wyznaczenie minimalnej masy oleju (m_w) niezbędnej do wychłodzenia pręta do temperatury $125°C$ po czasie $0.7s$. W zadaniu tym przejęto następujące parametry:

$$T_b = 1200.0 \qquad m_b = 0.25 \qquad c_b = 0.29$$
$$T_w = 25.0 \qquad m_w = ? \qquad c_w = 4.1813$$
$$h = h(T_b - T_w) \qquad a = 0.0109 \qquad dt = 0.1$$

Zależność temperatury pręta T_b po czasie $0.7s$ od masy oleju m_w przedstawiona została na rysunku 13.



Rysunek 13: Zależność T_b od m_w po czasie $0.7s$

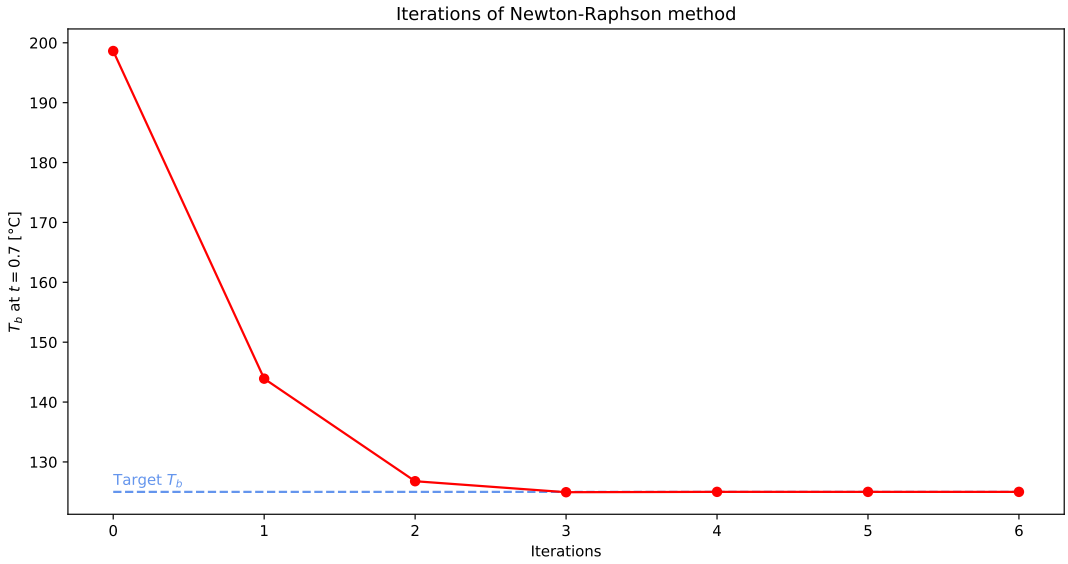
Celem tego zadania będzie znalezienie wartości m_w dla której $T_b = 125$, a zatem punktu przecięcia niebieskiej i czerwonej linii na wykresie.

Poszukiwanie tej wartości zrealizowane zostało z wykorzystaniem metody Newtona-Raphsona, która opisana została w podrozdziale 2.6.

Po przyjęciu masy początkowej równej $m_{w0} = 0.1$, kroku $\Delta m_w = 0.01$ oraz poszukiwanej dokładności $\epsilon = 10^{-5}$ po sześciu krokach iteracji otrzymano wynik:

$$m_w = 0.1863953911392946$$

Przebieg zbieżności tego procesu widoczny jest na rysunku 14.



Rysunek 14: Przebieg metody Newton-Raphson dla wyznaczania minimalnej masy oleju

6 Część 4: Optymalizacja procesu chłodzenia

Z powodu grubego niedoszacowania czasu potrzebnego na tą część projektu oraz spraw prywatnych w ostatnim okresie nie udało mi się zrealizować tej części projektu w stopniu nadającym się do prezentacji.

Spis rysunków

Spis tablic

Spis kodów źródłowych