

# Lagranżowskie sieci neuronowe

Aleksander Kaluta

13 września 2024

## 1 Badania

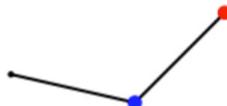
Model bez LNN, Model LNN z danymi numerycznymi, model LNN z danymi rzeczywistymi

### 1.1 Uzyskanie danych za pomocą metod numerycznych

W pierwszej kolejności ustalono kąty oraz prędkości początkowe.

```
x0 = np.array([3*np.pi/7, 3*np.pi/4, 0, 0], dtype=np.float32)
```

W podanym przykładzie pierwszy element wahadła tworzy kąt  $\frac{3}{7}\pi$  z linią pionową, natomiast drugi element kąt  $\frac{3}{4}\pi$ . Obie części wahadła charakteryzują się zerową prędkością początkową.



Następnie, przy użyciu funkcji odeint, która bazuje na metodzie Rungego-Kutty w celu rozwiązyania układów równań różniczkowych zwyczajnych, obliczono rozwiązanie (7). W efekcie uzyskano wartości kątów oraz prędkości kątowych dla przyjętego kroku czasowego t.

```
t = np.linspace(0, 100, num=1501)
x_train = jax.device_get(solve_analytical(x0, t))
```

Aby móc zastosować Recurrent Neural Network (RNN), konieczne jest przekształcenie danych z układu współrzędnych kątowych na współrzędne kartezjańskie. Taka transformacja umożliwia modelowi skuteczniejsze przetwarzanie danych, które w tym przypadku są związane z ruchem wahadła. Przekształcenie współrzędnych kątowych na kartezjańskie odbywa się przy użyciu zależności trygonometrycznych.

```

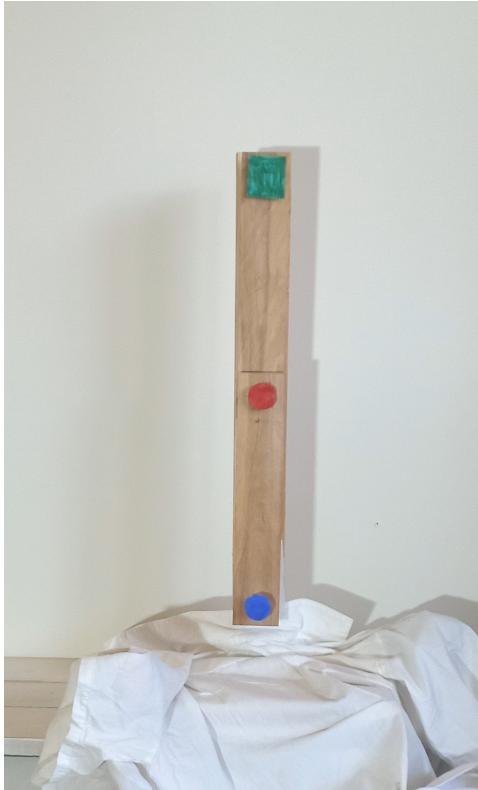
def to_cartesian(theta1, w1, theta2, w2, L1, L2):
    """ Transforms theta and omega to cartesian coordinates
    and velocities x1, y1, x2, y2, vx1, vy1, vx2, vy2
    """
    x1 = L1 * np.sin(theta1)
    y1 = -L1 * np.cos(theta1)
    x2 = x1 + L2 * np.sin(theta2)
    y2 = y1 - L2 * np.cos(theta2)
    vx1 = L1*np.cos(theta1)*w1
    vy1 = L1*np.sin(theta1)*w1
    vx2 = vx1 + L2*np.cos(theta2)*w2
    vy2 = vy1 + L2*np.sin(theta2)*w2
    return x1, y1, x2, y2, vx1, vy1, vx2, vy2

theta1, theta2, w1, w2 = x_train[:, 0], x_train[:, 1], x_train[:, 2], x_train[:, 3]
x1_new1, y1_new1, x2_new1, y2_new1 = to_cartesian(theta1, w1, theta2, w2, L1, L2)[:4]

```

## 1.2 Uzyskanie danych za pomocą estymacji pozycji rze- czywistego wahadła podwójnego

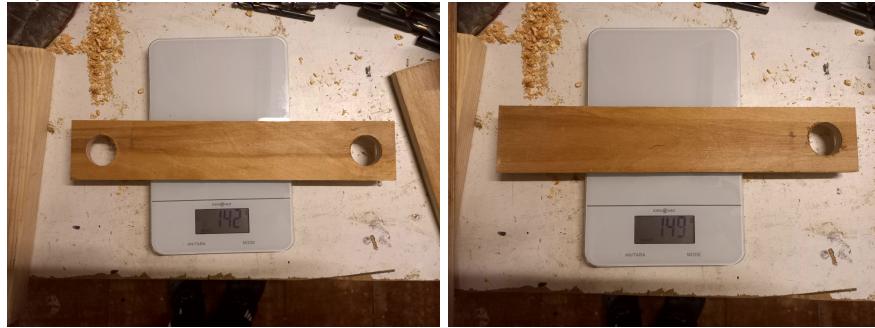
W celu realizacji tego zadania został stworzony fizyczny model podwójnego wahadła.



Na początkowym etapie z arkusza sklejki wycięto dwie części o odpowiednich wymiarach. Następnie, przy użyciu ..., wycięto otwory o średnicy ... w celu zamontowania łożysk.



Wszystkie elementy modelu zostały dokładnie zważone i zmierzone, co pozwoliło na precyzyjne określenie parametrów potrzebnych do późniejszych obliczeń i symulacji.





W drugim ramieniu wahadła zamontowano dodatkowe śrubki w celu wyrównania masy obu ramion, co zapewnia ich odpowiedni ruch.

Zamierzamy również stworzyć program do wykrywania określonych kolorów na nagranym materiale wideo, który pozwoli na analizę ruchu wahadła. W tym celu końce ramion wahadła zostały pomalowane na kolory: zielony, czerwony oraz niebieski. Dzięki temu, po odczytaniu pozycji tych kolorów na obrazie, możliwe będzie uzyskanie współrzędnych ramion wahadła. Następnie, zarejestrowano filmy przedstawiające ruch wahadła z różnymi punktami początkowymi. Program, który analizuje te nagrania, działa w następujący sposób:

W słowniku color\_ranges zdefiniowano zakresy kolorów w przestrzeni HSV (Hue, Saturation, Value) dla kolorów czerwonego, zielonego oraz niebieskiego.

```
color_ranges = {
    'red': [(0, 120, 70), (10, 255, 255)],
    'green': [(36, 100, 100), (86, 255, 255)],
    'blue': [(94, 80, 2), (126, 255, 255)]
}
```

Program najpierw konwertuje każdą klatkę wideo z przestrzeni kolorów BGR do przestrzeni HSV, co umożliwia bardziej efektywne wykrywanie kolorów, ponieważ przestrzeń HSV lepiej oddaje różnice w barwach i nasyceniu niż BGR.

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Następnie tworzy maski dla zdefiniowanych wcześniej kolorów (czerwonego, zielonego i niebieskiego). Czyli piksele odpowiadające danemu kolorowi mają wartość 1, podczas gdy pozostałe piksele przyjmują wartość 0.

```
mask = cv2.inRange(hsv, lower_bound, upper_bound)
```

W kolejnym kroku program znajduje kontury obszarów, które są zgodne z maskami. Dla każdego wykrytego konturu o powierzchni większej niż 500 pikseli program wylicza współrzędne środka prostokąta otaczającego kontur. W końcowym etapie zwraca listę współrzędnych środków wszystkich wykrytych konturów, co pozwala na określenie pozycji końców ramion wahadła w każdej klatce filmu.

```

contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
centers = []

for contour in contours:
    x, y, w, h = cv2.boundingRect(contour)

    center_x = x + w // 2
    center_y = y + h // 2
    centers.append((center_x, center_y))

return centers

```

Program oblicza medianę współrzędnych środków wszystkich wykrytych obszarów dla każdego koloru (czerwonego, zielonego i niebieskiego), co pozwala na bardziej stabilne i precyzyjne wyznaczenie pozycji końców ramion wahadła, eliminując wpływ potencjalnych szumów i błędnych detekcji. Wyliczone mediany są następnie dodawane do odpowiednich list dla wszystkich kolorów. Dzięki temu uzyskujemy sekwencję współrzędnych, które odzwierciedlają ruch ramion wahadła w czasie.

```

for color_name, centers in all_coordinates.items():
    if centers:
        medians[color_name] = (
            int(np.median([c[0] for c in centers])),
            int(np.median([c[1] for c in centers])))
    else:
        medians[color_name] = (None, None)

Red center: (221, 493), Green center: (343, 400), Blue center: (106, 373)
Red center: (220, 494), Green center: (341, 402), Blue center: (96, 382)
Red center: (224, 497), Green center: (341, 402), Blue center: (87, 400)
Red center: (229, 502), Green center: (341, 402), Blue center: (75, 427)
Red center: (235, 508), Green center: (338, 403), Blue center: (78, 463)
Red center: (243, 517), Green center: (340, 406), Blue center: (82, 507)
Red center: (252, 521), Green center: (342, 402), Blue center: (94, 556)
Red center: (259, 527), Green center: (340, 406), Blue center: (None, None)
Red center: (265, 536), Green center: (338, 401), Blue center: (None, None)

```

Niestety, ze względu na szybki ruch wahadła, w niektórych miejscach program nie zdołał wykryć kolorów, co powoduje brakujące dane.

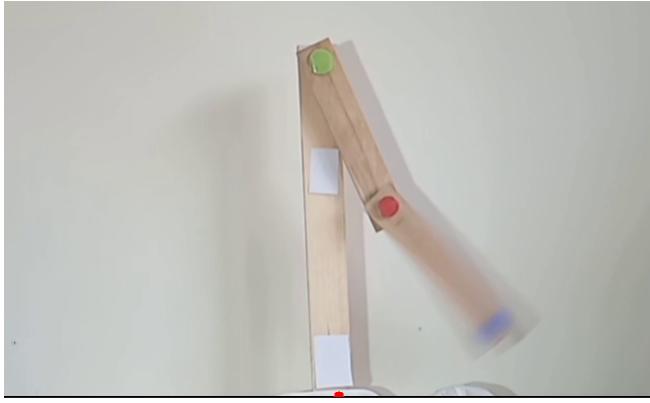
```
Red center: (259, 527), Green center: (340, 406), Blue center: (None, None)
```

Problem można rozwiązać ręcznie, jednak najpierw konieczne jest zidentyfikowanie filmu, w którym występuje najmniej niewykrytych klatek. Na pierwszy rzut oka wydaje się, że odpowiednim wyborem jest plik video3.mp4.

```
Liczba punktów nullowych niebieskiego koloru: 4
```

```
Liczba punktów nullowych czerwonego koloru: 39
```

Jednak po dokładniejszej analizie wyników programu okazuje się, że w wielu klatkach zamiast niebieskiego punktu program błędnie wykrywa białą narzutę jako niebieski kolor. To prowadzi do fałszywych detekcji i zniekształca dane o pozycjach wahadła w tych momentach.



Po przeanalizowaniu kolejnych materiałów wideo, okazało się, że jedynie w 3 klatkach wartości kolorów zostały wykryte błędnie, a w 14 klatkach kolory nie zostały wykryte w ogóle. W związku z tym, konieczne jest ręczne uzupełnienie lub poprawienie danych w łącznie 17 klatkach. Zapisujemy numery tych klatek, aby móc je sprawdzić ręcznie.

```
#video7
null_points_count_blue7 = sum(1 for point in blue_centers7 if point == (None, None))
null_points_count_red7 = sum(1 for point in red_centers7 if point == (None, None))
print(len(blue_centers7))
print(f'Liczba punktów nullowych niebieskiego koloru: {null_points_count_blue7}')
print(f'Liczba punktów nullowych czerwonego koloru: {null_points_count_red7}')
```

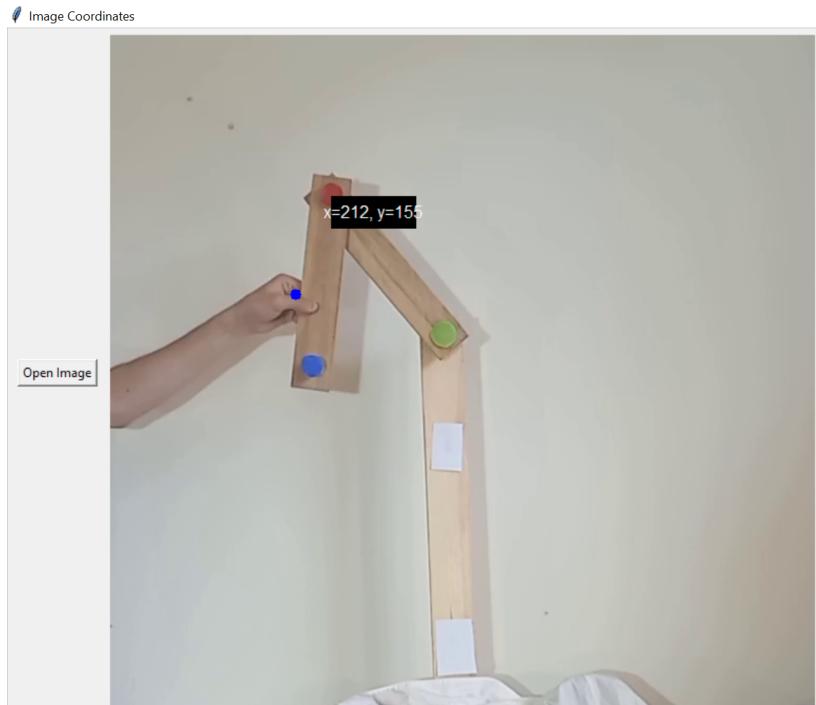
```
1480
Liczba punktów nullowych niebieskiego koloru: 3
Liczba punktów nullowych czerwonego koloru: 11
```

Video 7

Red: 0, 1, 2, 10, 11, 12, 13, 60, 61, 106, ~~111~~, 121,  
122, 166, 243

Blue: 18, 19, 20

W tym celu stworzymy w Tkinter aplikację, która umożliwia wyświetlanie współrzędnych klatek wideo. Dzięki niej ręcznie uzupełniamy brakujące dane lub korygujemy nieprawidłowo wykryte wartości.



Red [0] = (212, 155)  
 Red [1] = (213, 153)  
 Red [2] = (212, 159)  
 Red [10] = (172, 350)  
 Red [11] = (204, 395)  
 Red [12] = (257, 434)  
 Red [13] = (316, 445)  
 Red [60] = (248, 427)  
 Red [61] = (197, 391)

Red [106] = (401, 422)  
 Red [121] = (324, 444)  
 Red [122] = (275, 436)  
 Red [166] = (286, 441)  
 Red [243] = (328, 445)  
 Blue [18] = (461, 468)  
 Blue [19] = (521, 449)  
 Blue [20] = (572, 423)

Po wprowadzeniu poprawek, dla pewności sprawdzamy ponownie, czy nie ma jeszcze brakujących lub błędnych wartości, aby zapewnić kompletność i poprawność danych.

```
null_points_count_red7 = np.sum(np.any(np.isnan(red_centers7), axis=1))
print(f'Liczba punktów nullowych czerwonego koloru: {null_points_count_red7}')

Liczba punktów nullowych czerwonego koloru: 0
null_points_count_blue7 = np.sum(np.any(np.isnan(blue_centers7), axis=1))
print(f'Liczba punktów nullowych niebieskiego koloru: {null_points_count_blue7}')

Liczba punktów nullowych niebieskiego koloru: 0
```

Zielony punkt został wybrany jako punkt zerowy układu odniesienia, dlatego aby precyzyjnie określić jego pozycję, obliczamy medianę wszystkich współrzęd-

nych zielonego punktu z każdej klatki. Mediana ta będzie reprezentować stałą pozycję punktu zerowego.

```
green_center7.append(np.nanmedian(green_centers7[:,0]))
green_center7.append(np.nanmedian(green_centers7[:,1]))
green_center7
```

[323.0, 285.0]

Po ustaleniu punktu odniesienia możemy przystąpić do normalizacji danych. Punkt odpowiadający pierwszemu ramieniu wahadła jest wyznaczony jako znormalizowana różnica między współrzędnymi wykrytego czerwonego punktu a współrzędnymi zielonego punktu.

```
red2=preprocessing.normalize(green_center7-red_centers7)
red2
```

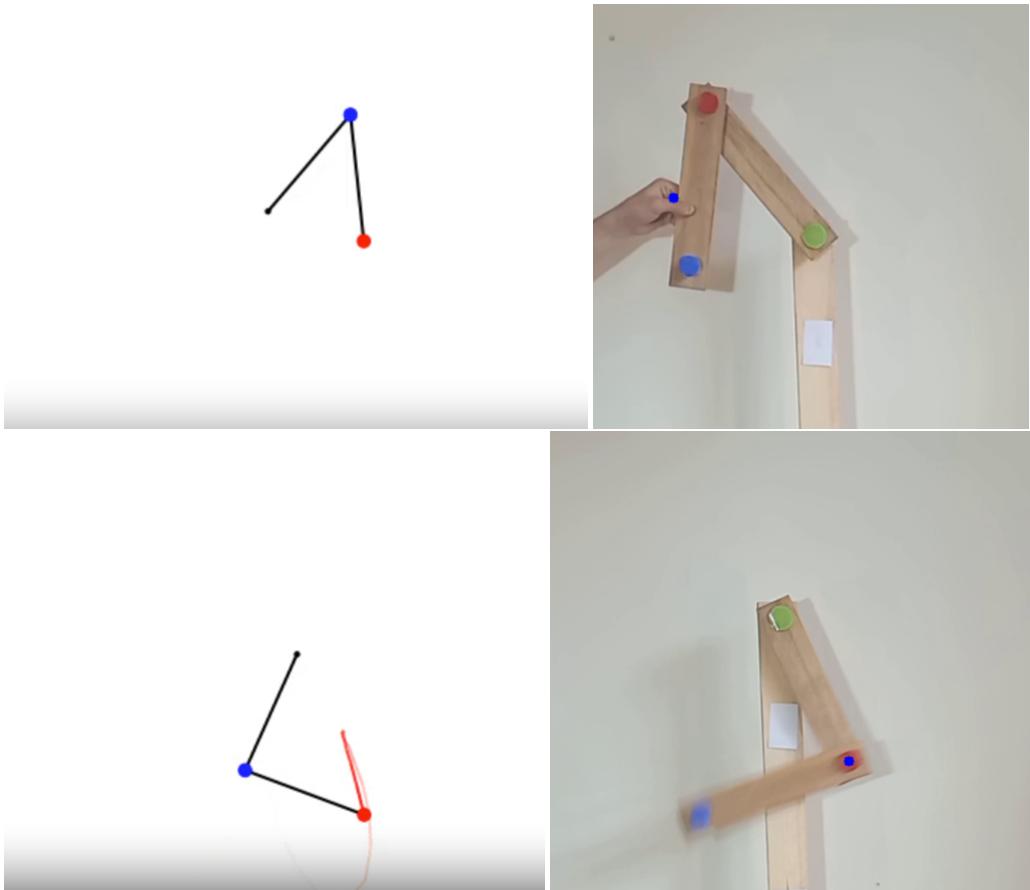
array([[ 0.64934491, 0.76049404],
 [ 0.6401844 , 0.76822128],
 [ 0.64646424, 0.76294429],
 ...,
 [-0.18768705, -0.98222888],
 [-0.17378533, -0.98478356],
 [-0.1505811 , -0.98859766]])

Dla drugiego ramienia wahadła punkt ten obliczamy, dodając do współrzędnych pierwszego ramienia znormalizowaną różnicę pomiędzy współrzędnymi niebieskiego punktu a współrzędnymi czerwonego punktu. Dzięki temu otrzymujemy znormalizowane współrzędne obu ramion wahadła względem punktu zerowego.

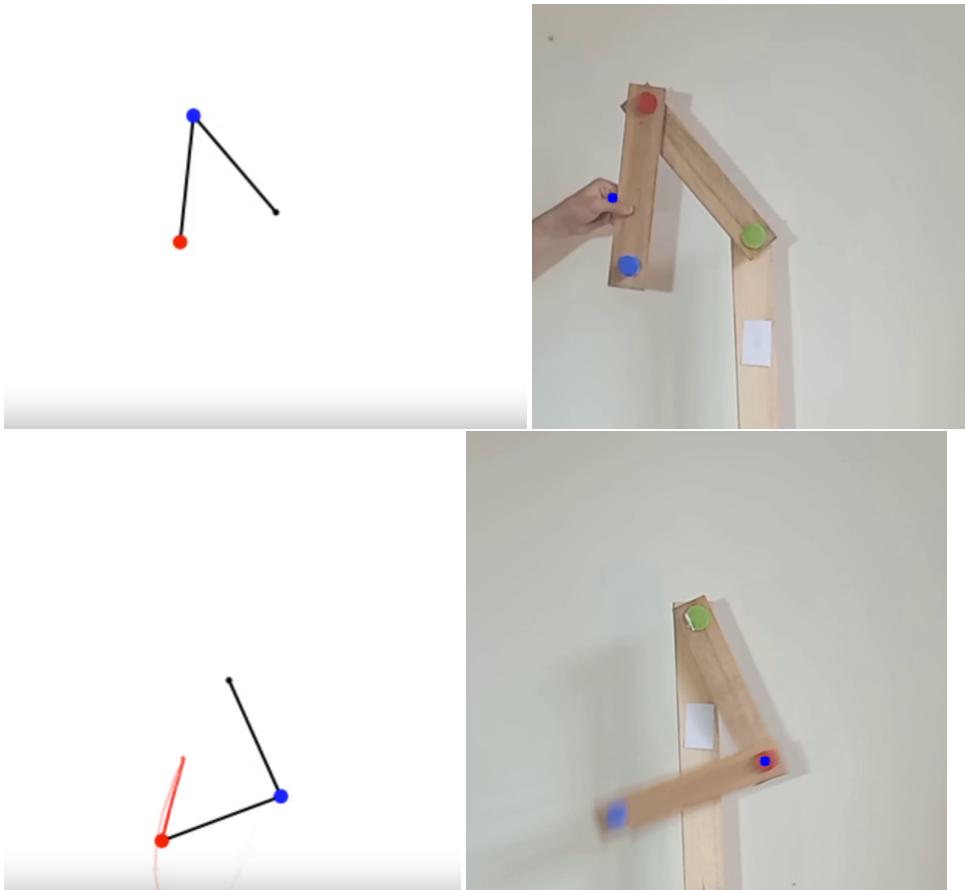
```
blue2=red2+preprocessing.normalize(red_centers7-blue_centers7)
blue2
```

array([[ 0.75435122, -0.23397751],
 [ 0.7559649 , -0.22505354],
 [ 0.75622646, -0.23101359],
 ...,
 [-0.58388212, -1.90039524],
 [-0.58106861, -1.89808544],
 [-0.56299227, -1.89959548]])

Na ostatnim etapie sprawdzono, czy utworzony model na podstawie zgromadzonych danych działa poprawnie. Okazało się, że model jest prawidłowy, jednak przedstawiony w lustrzanym odbiciu.



Aby skorygować ten problem, wystarczyło dokonać odbicia lustrzanego poprzez zmianę znaku współrzędnej  $x$  dla wszystkich punktów. Po tej korekcie model działał poprawnie, a uzyskane dane wiernie odzwierciedlały rzeczywisty ruch podwójnego wahadła, zgodnie z estymacją jego pozycji.



### 1.3 Model przy użyciu RNN oparty na danych uzyskanych numerycznie, nauczony na współrzędnych

Po zebraniu i przekształceniu danych możemy przystąpić do tworzenia modelu. Korzystając z Recurrent Neural Network (RNN), musimy ustalić liczbę klatek, które sieć będzie analizowała do przewidzenia kolejnej klatki. W naszym przypadku timeframe wynosi 20, co oznacza, że sieć będzie analizowała 20 poprzednich układów współrzędnych, aby przewidzieć kolejne.

Dla tego celu przekształcamy tablicę z pierwotnymi danymi, która zawiera 1500 tablic (po 4 zmienne w każdej), w nową tablicę o rozmiarze 1480 tablic, gdzie każda ma wymiar 20x4. To oznacza, że na każde 20 zestawów współrzędnych sieć będzie próbowała przewidzieć kolejny 21. zestaw.

```

for i in range(len(x1_new) - timeframe):

    # Stack the next 'timeframe' positions of both masses to serve as the input sequence
    X.append(np.vstack((x1_new[i:i+timeframe], y1_new[i:i+timeframe], x2_new[i:i+timeframe], y2_new[i:i+timeframe])).T)

    # Record the positions of both masses 20 steps ahead from the current timestep to set as the prediction target
    Y.append([x1_new[i+timeframe], y1_new[i+timeframe], x2_new[i+timeframe], y2_new[i+timeframe]])

```

Następnie przystępujemy do tworzenia modelu. Wykorzystamy do tego LSTM (Long Short-Term Memory), który jest wariantem RNN, dobrze radzącym sobie z długoterminowymi zależnościami, oraz głęboką sieć neuronową, aby zwiększyć złożoność modelu.

```

model = Sequential([
    LSTM(64, input_shape=(timeframe, 4), return_sequences=True),
    LSTM(32), # Second LSTM Layer with 32 units
    Dense(4) # Output Layer for future x1, y1, x2, y2 positions
])

```

Działanie sieci można opisać, że sieć stara się z danych zawartych w pierwszych 20 układach przewidzieć, jak wygląda układ współrzędnych w 21. klatce. Dla przykładu, X[0] reprezentuje pierwsze 20 układów współrzędnych, natomiast Y[0] to 21. układ współrzędnych. W ten sposób sieć uczy się modelować dynamiczne zależności w ruchu wahadła i przewidywać jego przyszłe położenia na podstawie wcześniejszych stanów.

```

X[0]                                X[1]
array([[ 0.9749279 , -0.22252098,  1.6820347 ,  0.4845858 ],
       [ 0.978309 , -0.24186869,  1.6809305 ,  0.46170586],
       [ 0.95439947, -0.2985325 ,  1.6771235 ,  0.39260423],
       [ 0.92161924, -0.38809535,  1.6692172 ,  0.2760562 ],
       [ 0.8646444 , -0.5023843 ,  1.6549351 ,  0.11034787],
       [ 0.77764106, -0.6287085 ,  1.6305242 , -0.10660666],
       [ 0.6611618 , -0.75824337,  1.5888985 , -0.37723434],
       [ 0.5276897 , -0.84943724,  1.5168946 , -0.7028985 ],
       [ 0.404249 , -0.91464895,  1.3907369 , -1.0784829 ],
       [ 0.32850385, -0.94450265,  1.1738546 , -1.4787143 ],
       [ 0.33271384, -0.94302785,  0.81444037, -1.8193495 ],
       [ 0.36055085, -0.93273956,  0.31551835, -1.931725 ],
       [ 0.29231253, -0.95632285, -0.18669364, -1.8341343 ],
       [ 0.11806197, -0.9930062 , -0.62938297, -1.65733 ],
       [-0.13207965, -0.99123913, -0.0116328 , -1.4670398 ],
       [-0.4144933 , -0.9105236 , -1.3395425 , -1.289896 ],
       [-0.68307096, -0.730352 , -1.600534 , -1.1281732 ],
       [-0.8862997 , -0.4631121 , -1.7533457 , -0.96134007],
       [-0.9830793 , -0.1831804 , -1.7907777 , -0.77277637],
       [-0.99896413,  0.04550415, -1.7722844 , -0.58851147],
       [-0.97634894,  0.21620078, -1.7363052 , -0.43377334]], dtype=float32)
Y[0]
array([-0.97634894,  0.21620078, -1.7363052 , -0.43377334], dtype=float32)

```

Do optymalizacji modelu wykorzystaliśmy popularną metodę gradientową Adam, która jest efektywnym algorytmem opartym na metodzie stochastycznego gradientu, odpowiednim dla dużych zbiorów danych. Funkcją kosztu, używaną do minimalizacji błędów predykcji modelu, jest błąd średniokwadratowy (MSE), który dobrze sprawdza się w problemach regresyjnych, takich jak przewidywanie współrzędnych.

```
model.compile(optimizer='adam', loss='mse')
```

Podczas trenowania modelu 20% naszych danych zostało wydzielonych jako zbiór walidacyjny, co pozwala na monitorowanie wydajności modelu i unikanie problemu nadmiernego dopasowania (overfittingu).

```
model.fit(X, Y, epochs=50, batch_size=32, validation_split=0.2)

Epoch 50/50
37/37 [=====] - 2s 64ms/step - loss: 0.0013 - val_loss: 0.0025
```

Aby zastosować model do przewidywania, musimy podać N kolejnych iteracji, a model będzie przewidywał następne N iteracji. Proces ten przebiega w podobny sposób, jak w fazie uczenia — dane wejściowe do modelu są ponownie konwertowane na tablice zawierające po 20 zestawów kolejnych klatek (timeframe). Dzięki temu model może przewidywać przyszłe położenia na podstawie wcześniejszych stanów ruchu.

```
X_var = [np.vstack((x1_var[j:j+timeframe], y1_var[j:j+timeframe], x2_var[j:j+timeframe],
                    y2_var[j:j+timeframe])).T for j in range(len(x1_var) - timeframe)]
```

```
predictions = model.predict(X_var)
```

```
X_var.shape
```

```
(1481, 20, 4)
```

```
predictions.shape
```

```
(1481, 4)
```

## 1.4 Model przy użyciu RNN oparty na danych uzyskanych numerycznie, nauczony na współrzędnych i prędkościach

Model tworzymy w taki sam sposób jak opisany w poprzednim rozdziale, z tą różnicą, że w tym przypadku do procesu uczenia modelu wykorzystujemy nie tylko współrzędne kątowe ramion wahadła, ale także ich prędkości kątowe. Dzięki temu model ma dostęp zarówno do informacji o położeniu, jak i o dynamicie ruchu. Wprowadzenie dodatkowego wymiaru (prędkości) jest, aby model lepiej zrozumiał zależności czasowe i zmiany w ruchu wahadła, czyli dokładniej prognozował kolejne pozycje układu w czasie.

```
for i in range(len(x1_new) - timeframe):
```

```
# Stack the next 'timeframe' positions of both masses to serve as the input sequence
```

```
X.append(np.vstack((x1_new[i:i+timeframe], y1_new[i:i+timeframe], x2_new[i:i+timeframe], y2_new[i:i+timeframe],
                     vx1_new[i:i+timeframe], vy1_new[i:i+timeframe], vx2_new[i:i+timeframe], vy2_new[i:i+timeframe])).T)
```

```
# Record the positions of both masses 20 steps ahead from the current timestep to set as the prediction target
```

```
Y.append([x1_new[i+timeframe], y1_new[i+timeframe], x2_new[i+timeframe], y2_new[i+timeframe],
          vx1_new[i+timeframe], vy1_new[i+timeframe], vx2_new[i+timeframe], vy2_new[i+timeframe]])
```

```
model = Sequential([
    LSTM(64, input_shape=(timeframe, 8), return_sequences=True), # First LSTM layer
    LSTM(32), # Second LSTM Layer with 32 units
    Dense(8) # Output Layer for future x1, y1, x2, y2, vx1, vy1, vx2, vy2
])
```

```

X_var = [np.vstack((x1_var[j:j+timeframe], y1_var[j:j+timeframe], x2_var[j:j+timeframe],
                    y2_var[j:j+timeframe], vx1_var[j:j+timeframe], vy1_var[j:j+timeframe],
                    vx2_var[j:j+timeframe], vy2_var[j:j+timeframe])).T for j in range(len(x1_var) - timeframe)]
predictions = model.predict(X_var)

```

## 1.5 Model przy użyciu LNN oparty na danych uzyskanych numerycznie

Dysponując kątami i prędkościami kątowymi obliczonymi numerycznie, możemy zastosować te wartości w równaniu (7) w celu obliczenia przyspieszeń.

```

def f_analytical(state, t=0, m1=1, m2=1, l1=1, l2=1, g=9.8):
    t1, t2, w1, w2 = state
    a1 = (l2 / l1) * (m2 / (m1 + m2)) * jnp.cos(t1 - t2)
    a2 = (l1 / l2) * jnp.cos(t1 - t2)
    f1 = -(l2 / l1) * (m2 / (m1 + m2)) * (w1**2) * jnp.sin(t1 - t2) - \
        (g / l1) * jnp.sin(t1)
    f2 = (l1 / l2) * (w2**2) * jnp.sin(t1 - t2) - (g / l2) * jnp.sin(t2)
    g1 = (f1 - a1 * f2) / (1 - a1 * a2)
    g2 = (f2 - a2 * f1) / (1 - a1 * a2)
    return jnp.stack([w1, w2, g1, g2])

t = np.arange(N, dtype=np.float32) # time steps 0 to N
#time x_train = jax.device_get(solve_analytical(x0, t)) # dynamic
#time xt_train = jax.device_get(jax.vmap(f_analytical))(x_train))

```

Zbiór testowy uzyskuje się przy zastosowaniu dalszego kroku N.

```

t_test = np.arange(N, 2*N, dtype=np.float32) # time steps N to 2N
#time x_test = jax.device_get(solve_analytical(x0, t_test)) # dynamic
#time xt_test = jax.device_get(jax.vmap(f_analytical))(x_test)) #

```

Po zgromadzeniu wszystkich niezbędnych danych można przystąpić do konstrukcji sieci. Jej budowa będzie następująca:

```

init_random_params, nn_forward_fn = stax.serial(
    stax.Dense(128),
    stax.Softplus,
    stax.Dense(128),
    stax.Softplus,
    stax.Dense(1),
)

```

Wyjściem sieci będzie później obliczony pseudolagrangian.

```

def learned_lagrangian(params):
    def lagrangian(q, q_t):
        assert q.shape == (2,)
        state = normalize_dp(jnp.concatenate([q, q_t]))
        return jnp.squeeze(nn_forward_fn(params, state), axis=-1)
    return lagrangian

```

unkcją kosztów jest średni błąd kwadratowy między danymi uzyskanymi z równania (7), a danymi uzyskanymi z równania (12), przy uwzględnieniu wyjściowego pseudolagrangianu.

```

train_loss = loss(params, (x_train, xt_train))
test_loss = loss(params, (x_test, xt_test))

```

```

def loss(params, batch, time_step=None):
    state, targets = batch
    if time_step is not None:
        f = partial(equation_of_motion, learned_lagrangian(params))
        preds = jax.vmap(partial(rk4_step, f, t=0.0, h=time_step))(state)
    else:
        preds = jax.vmap(partial(equation_of_motion, learned_lagrangian(params)))(state)
    return jnp.mean((preds - targets) ** 2)

def equation_of_motion(lagrangian, state, t=None):
    q, q_t = jnp.split(state, 2)
    q_tt = (jnp.linalg.pinv(jax.hessian(lagrangian, 1)(q, q_t))
             @ (jax.grad(lagrangian, 0)(q, q_t)
                 - jax.jacobian(jax.jacobian(lagrangian, 1), 0)(q, q_t) @ q_t))
    return jnp.concatenate([q_t, q_tt])

```

Do procesu optymalizacji znów zastosujemy popularną metodę optymalizacji gradientowej, jaką jest adam.

```

opt_init, opt_update, get_params = optimizers.adam(
    lambda t: jnp.select([t < batch_size*(num_batches//3),
                          t < batch_size*(2*num_batches//3),
                          t > batch_size*(2*num_batches//3)],
                          [1e-3, 3e-4, 1e-4]))
opt_state = opt_init(init_params)

```

Po ustaleniu numeru i liczby batchy można przystąpić do uczenia modelu.

```

batch_size = 100
test_every = 10
num_batches = 500

for iteration in range(batch_size*num_batches + 1):
    if iteration % batch_size == 0:
        params = get_params(opt_state)
        train_loss = loss(params, (x_train, xt_train))
        train_losses.append(train_loss)
        test_loss = loss(params, (x_test, xt_test))
        test_losses.append(test_loss)
    if iteration % (batch_size*test_every) == 0:
        print(f"iteration={iteration}, train_loss={train_loss:.6f}, test_loss={test_loss:.6f}")
        opt_state = update_derivative(iteration, opt_state, (x_train, xt_train))

params = get_params(opt_state)
iteration=50000, train_loss=0.094098, test_loss=0.099857

```

Aby skorzystać z modelu, wystarczy podać współrzędne początkowe oraz pożądany krok czasowy. Model wykorzystuje obliczony pseudolagrangian do przewidywania następnych kroków, korzystając z równania (12).

```

x1 = np.array([3*np.pi/7, 3*np.pi/4, 0, 0], dtype=np.float32)
t2 = np.linspace(0, 20, num=301)
x1_model = jax.device_get(solve_lagrangian(learned_lagrangian(params), x1, t=t2))

```

```
def solve_lagrangian(lagrangian, initial_state, **kwargs):
    # We currently run odeint on CPUs only, because its cost is dominated by
    # control flow, which is slow on GPUs.
    @partial(jax.jit, backend='cpu')
    def f(initial_state):
        return odeint(partial(equation_of_motion, lagrangian),
                      initial_state, **kwargs)
    return f(initial_state)
```