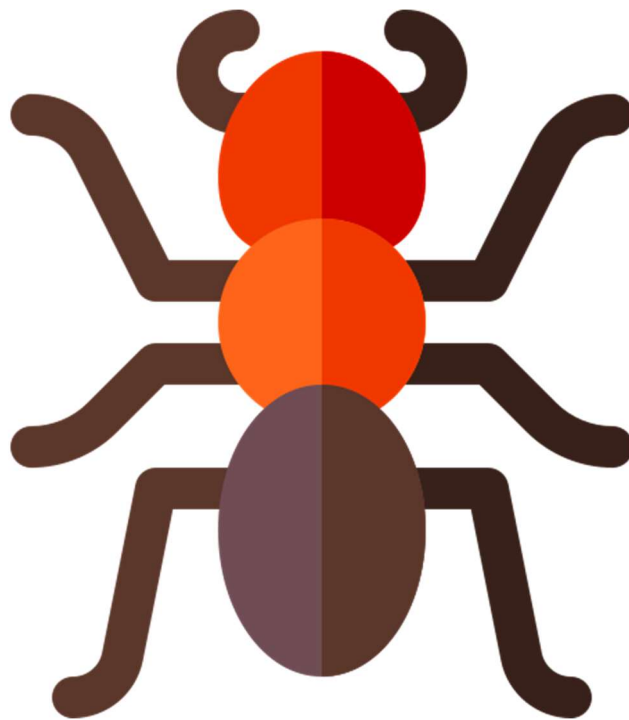


*ŁUKASZ STANISZEWSKI*

# SYMULATOR MRÓWKI LANGTONA

OFICJALNA DOKUMENTACJA



# SPIS TREŚCI

ROZDZIAŁ I. OPIS PROJEKTU .....	3
<b>WSTĘP</b> .....	3
<b>ZASADY DZIAŁANIA</b> .....	3
<b>WYMAGANIA PROJEKTU</b> .....	3
ROZDZIAŁ II. STRUKTURA PROJEKTU .....	4
<b>OGÓLNY PODZIAŁ</b> .....	4
<b>SCHEMAT DZIAŁANIA</b> .....	4
<b>NAJWAŻNIEJSZE UŻYTE BIBLIOTEKI</b> .....	5
<b>STRUKTURA MODUŁÓW I KLAS</b> .....	5
<b>NAJISTOTNIEJSZE FUNKCJE</b> .....	5
<b>ALGORYTYMIKA</b> .....	6
FAZA WSTĘPNA .....	6
FAZA SYMULACJI .....	6
ROZDZIAŁ III. INSTRUKCJA UŻYTKOWNIKA .....	7
<b>WSTĘP</b> .....	7
<b>START KONSOLOWY</b> .....	7
<b>START GUI</b> .....	8
<b>OPCJA TWORZENIA PLIKU (1)</b> .....	8
<b>OPCJA ŁADOWANIA PLIKU</b> .....	9
<b>OPCJA TWORZENIA PLIKU Z PRAWDOPODOBIENSTWEM</b> .....	9
<b>WYNIK SYMULACJI</b> .....	9
ROZDZIAŁ IV. PRZEPROWADZONE TESTY .....	10
<b>TESTY JEDNOSTKOWE</b> .....	10
<b>WAŻNY PRZYKŁAD</b> .....	10
<b>TEST SYMULACJI</b> .....	11
ROZDZIAŁ V. EWENTUALNY ROZWÓJ .....	11
ROZDZIAŁ VI. WYKORZYSTANE ZASOBY .....	11

## ROZDZIAŁ I. OPIS PROJEKTU

### WSTĘP

Projekt polega na implementacji automatu komórkowego, tzw. **Mrówki Langtona**.



*Rys 1 - trasa mrówki po wykonaniu 7000 kroków*

### ZASADY DZIAŁANIA

1. Mrówka porusza się na planszy o określonych wymiarach, podzielonej na kwadratowe komórki (pola) w dwóch możliwych kolorach: czarnym i białym
2. Jeśli mrówka znajduje się na polu białym to obraca się w lewo (o kąt prosty), zmienia kolor pola na czarny i przechodzi na następną komórkę.
3. Jeśli mrówka znajduje się na polu czarnym to obraca się w prawo (o kąt prosty), zmienia kolor pola na biały i przechodzi na następną komórkę.
4. Jeśli mrówka doszła do końca planszy i próbuje za nią wyjść to przechodzi na losowo wybraną sąsiednią komórkę, która leży na planszy (np. jeśli mrówka znajduje się w dolnym lewym rogu planszy i próbuje przejść jedną komórkę w dół, to powinna wykonać losowy ruch w górę, lub w prawo).

### WYMAGANIA PROJEKTU

















1. Mrówka porusza się po obrazie, którego wartość początkowa to jedna z następujących możliwości:
  - a) biały obraz o wymiarach podanych przez użytkownika
  - b) obraz ładowany przez użytkownika (powinien być czarno-biały)
  - c) czarno-biały obraz, gdzie czarne piksele zostały losowo wygenerowane z zadaniem prawdopodobieństwem
2. Wyjściem programu jest zapisana seria obrazów obrazująca wynik 'spaceru' mrówki - jeden obraz na każdy ruch. Liczba kroków określona jest przez użytkownika.

## ROZDZIAŁ II. STRUKTURA PROJEKTU

### OGÓLNY PODZIAŁ

Strukturę omawianego projektu najwygodniej podzielić na kilka punktów:

1. **DOKUMENTACJA** – folder *doc*
2. **TESTY** – folder *tests* i zawarte w nim testy (czyt. rozdział IV)
3. **WYJĄTKI** – pliki *ConsoleExceptions.py* i *GuiExceptions.py*
4. **KLASY ODPOWIEDZIALNE ZA SYMULACJE** – pliki *Ant.py*, *BoardReader.py* i *Simulator.py*
5. **FUNKCJE KONWERTUJĄCE NA LINII PROJEKT-OBRAZ** – pliki *ImageInput.py* i *ImageOutput.py*
6. **CZĘŚĆ INTERFEJSOWA** – folder *gui\_files*, plik *LangtonGui.py*
7. **FOLDER WYNIKOWY** – folder *Steps*

 doc	12.01.2020 20:24	Folder plików
 gui_files	06.01.2020 00:01	Folder plików
 load	06.01.2020 00:01	Folder plików
 Steps	12.01.2020 20:26	Folder plików
 tests	06.01.2020 00:01	Folder plików
 <i>_init_</i>	06.01.2020 00:01	Python File
 <i>Ant</i>	06.01.2020 00:01	Python File
 <i>BoardReader</i>	06.01.2020 00:01	Python File
 <i>ConsoleExceptions</i>	07.01.2020 17:28	Python File
 <i>GuiExceptions</i>	06.01.2020 00:01	Python File
 <i>ImageInput</i>	07.01.2020 17:30	Python File
 <i>ImageOutput</i>	06.01.2020 00:01	Python File
 <i>InputBox</i>	06.01.2020 00:01	Python File
 <i>LangtonGui</i>	07.01.2020 17:31	Python File
 <i>run</i>	07.01.2020 17:26	Python File
 <i>Simulator</i>	06.01.2020 00:01	Python File

Rys. 2 - końcowy wygląd folderu z projektem

### SCHEMAT DZIAŁANIA

Schemat działania projektu obrazuje lista kroków:

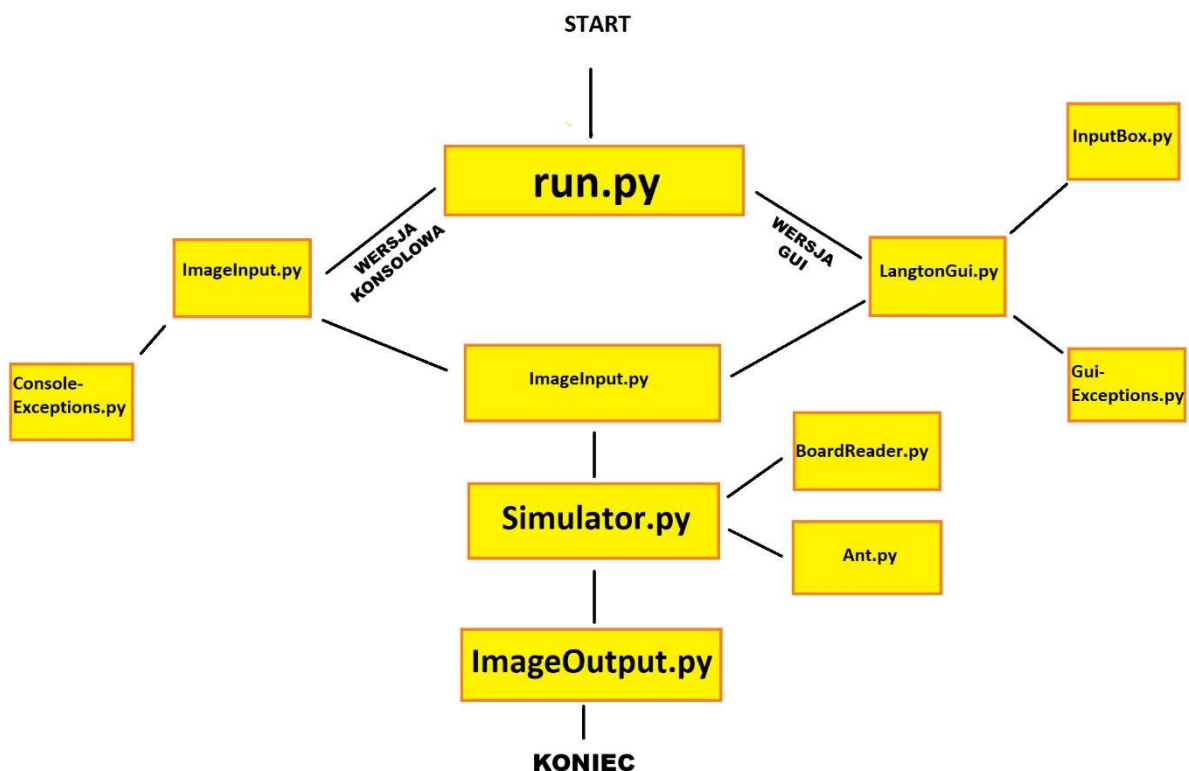
1. Uruchom projekt
2. Wybierz opcje
3. Wprowadź potrzebne dane
4. Program wykonuje symulacje
5. KONIEC

## NAJWAŻNIEJSZE UŻYTE BIBLIOTEKI

Użyte w projekcie biblioteki:

- 1) **Numpy** - korzystanie z klas do wygodnej pracy z wektorami dwuwymiarowymi
- 2) **Pillow** - biblioteka do obsługi obrazów
- 3) **Pygame** – biblioteka do tworzenia interfejsu graficznego
- 4) Dodatkowo częściej spotykane biblioteki typu **time**, **os**, **math** czy **sys**

## STRUKTURA MODUŁÓW I KLAS



## NAJISTOTNIEJSZE FUNKCJE

1. **Ant.go(kierunek)** – porusza mrowkę w dany kierunek
2. **BoardReader.create\_board\_from\_picture(obrazek)** – tworzy tablicę dwuwymiarową z danego obrazka
3. **ImageOutput.create\_picture\_from\_board(tablica)** – tworzy obrazek z danej tablicy
4. **Simulator.take\_step()** – funkcja odpowiedzialna za pojedynczy krok symulacji

## ALGORYTMIKA

### FAZA WSTĘPNA

Omówmy teraz dokładniej całe działanie projektu. Użytkownik uruchamiając plik `run.py` włącza symulator. W zależności od parametru w pliku `run.py` może uruchomić grę za pomocą:

- a) **KONSOLI**
- b) **GRAFICZNEGO INTERFEJSU UŻYTKOWNIKA (GUI)**

Teraz ma do wyboru wybranie 1 z 3 opcji:

- a) **STWORZENIA BIAŁEGO OBRAZKA** (podaje wysokość i szerokość oczekiwanej grafiki)
- b) **ZAŁADOWANIA SWOJEGO OBRAZKA O MAX WIELKOŚCI 300x300** (podaje ścieżkę do niego)
- c) **STWORZENIA BIAŁEGO OBRAZKA Z PRAWDOPODOBIENIESTWEM WYSTAPIENIA CZARNYCH PIXELI NA NIM** (zarówno wysokość i szerokość jak i prawdopodobieństwo jest podawane przez użytkownika)

Dodatkowo w każdej z 3 tych opcji musi podać również liczbę kroków jaką ma wykonać mrówka (jeżeli poda więcej niż 30000, mrówka wykona 30000 kroków).

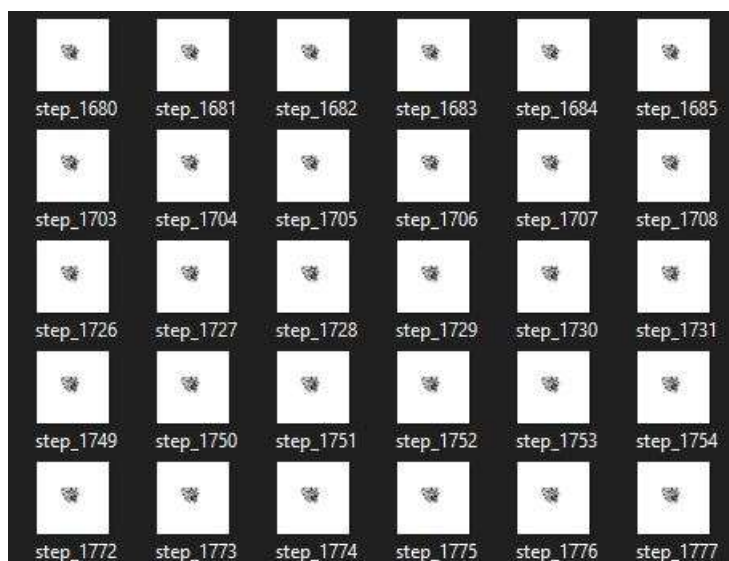
Następnie program sprawdza poprawność wprowadzonych danych – odpowiednie typy, wielkości itd.

Gdy nie wystąpi żaden wyjątek, przechodzimy do następnej fazy projektu.

### FAZA SYMULACJI

Fazę tę rozpoczyna program od skonwertowania wygenerowanego obrazka do tablicy dwuwymiarowej. Przedstawia ona plansze po jakiej się mrówka porusza – to właśnie w jej centrum zostaje umieszczona mrówka. W pętli wykonującej się N-krotnie (N – liczba kroków podana przez użytkownika we wcześniejszej fazie) mrówka robi kolejne kroki według algorytmu zawartego w I rozdziale dokumentacji. W międzyczasie każdy stan planszy jest przerabiany za pomocą odpowiedniej funkcji na czarno-biały obrazek, który jest zapisywany do folderu *Steps* z odpowiednią nazwą w postaci *step\_[nr kroku]*.

Po wykonaniu pętli, następuje koniec programu.



Rys 3 – fragment przykładowego wyglądu folderu *Steps* po zakończonym programie

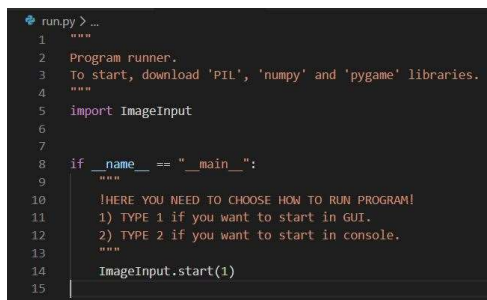
## ROZDZIAŁ III. INSTRUKCJA UŻYTKOWNIKA

### WSTĘP

Aby uruchomić program, należy wejść do folderu z projektem, za pomocą edytora tekstowego otworzyć plik **run.py** i w funkcji start() wpisać odpowiednio **1** lub **2**:

**1** – uruchomienie programu w gui

**2** – uruchomienie programu w konsoli

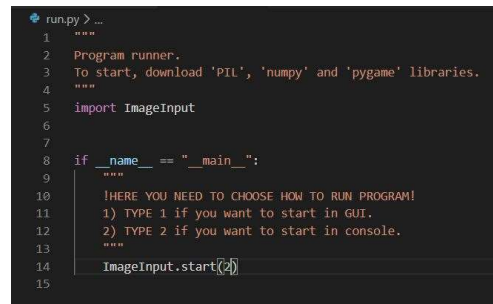


```

1  """
2  Program runner.
3  To start, download 'PIL', 'numpy' and 'pygame' libraries.
4  """
5  import ImageInput
6
7
8  if __name__ == "__main__":
9      """
10     HERE YOU NEED TO CHOOSE HOW TO RUN PROGRAM!
11     1) TYPE 1 if you want to start in GUI.
12     2) TYPE 2 if you want to start in console.
13     """
14     ImageInput.start(1)
15

```

Rys 4.1 - konfiguracja startu gui



```

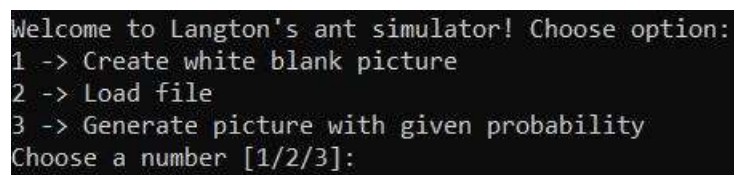
1  """
2  Program runner.
3  To start, download 'PIL', 'numpy' and 'pygame' libraries.
4  """
5  import ImageInput
6
7
8  if __name__ == "__main__":
9      """
10     HERE YOU NEED TO CHOOSE HOW TO RUN PROGRAM!
11     1) TYPE 1 if you want to start in GUI.
12     2) TYPE 2 if you want to start in console.
13     """
14     ImageInput.start(2)
15

```

Rys 4.2 - konfiguracja startu konsolowego

### START KONSOLOWY

Następnie po zapisaniu pliku **run.py** możemy go uruchomić. W konsoli powita nas ekran taki jak na zdjęciu.



```

Welcome to Langton's ant simulator! Choose option:
1 -> Create white blank picture
2 -> Load file
3 -> Generate picture with given probability
Choose a number [1/2/3]:

```

Rys 5 - ekran startowy w wersji konsolowej

## START GUI

Po uruchomieniu pliku **run.py** z parametrem **1**, pojawi nam się następujące okno.



Rys 6 - ekran startowy w wersji gui

## OPCJA TWORZENIA PLIKU (1)

Pierwszą z wybranych opcji, jest opcja tworzenia białego obrazka. Po wybraniu jej, należy wpisać odpowiednie dane, tak jak na obrazkach. Dane mają też swoje ograniczenia:

- 1) Wysokość [**height**] obrazka powinna być liczbą całkowitą z zakresu 0-300
- 2) Szerokość [**width**] obrazka powinna być liczbą całkowitą z zakresu 0-300
- 3) Liczba kroków [**steps**] powinna być liczbą z zakresu 1-30000

**LANGTON'S ANT  
SIMULATOR**

PLEASE TYPE NECESSARY DATA AND PRESS ENTER!

ENTER WIDTH [NUMBER]:  OK!

ENTER HEIGHT [NUMBER]:  OK!

ENTER STEPS TO DO [NUMBER]:

Rys 7.1 - ekran 1 opcji w gui

```
Choose a number [1/2/3]: 1
Give picture width: 100
Give picture height: 100
Give a number of steps to do (max=30000): 1000
```

Rys 7.2 - ekran 1 opcji w konsoli



## OPCJA ŁADOWANIA PLIKU

Drugą z możliwych opcji do wybrania jest ładowanie własnego pliku. Ładowany plik warto umieścić w folderze **load**, specjalnie do tego przeznaczonym. Nie ma wymogów co do kolorów na obrazku – zostanie on przekonwertowany na czarno-biały obiekt. Istotny jest jednak rozmiar obrazka. Ładowany obrazek nie może mieć szerokości lub wysokości większej niż 300 pixeli.

PLEASE TYPE NECESSARY DATA AND PRESS ENTER!

ENTER PATH TO FILE:

ENTER STEPS TO DO [NUMBER]:

Rys 8.1 - 2 opcja w gui

```
Choose a number [1/2/3]: 2
Give file path, name, and extension [ex. path/pic.jpg]: load/bad.png
Give a number of steps to do (max=30000): 1000
```

Rys 8.2 - opcja 2 w konsoli

## OPCJA TWORZENIA PLIKU Z PRAWDOPODOBIEŃSTWEM

Trzecią z możliwych opcji do wybrania jest tworzenie własnego pliku z prawdopodobieństwem wystąpienia czarnego pixela na nim. Poza atrybutami pojawiającymi się w opcji 1, występuje tu również atrybut **probability** – wymaga podania liczby rzeczywistej z zakresu 0-1.

PLEASE TYPE NECESSARY DATA AND PRESS ENTER!

ENTER WIDTH [NUMBER]:  OK!

ENTER HEIGHT [NUMBER]:  OK!

ENTER STEPS TO DO [NUMBER]:  OK!

ENTER PROBABILITY [NUMBER (0-1)]:

Rys 9.1 - opcja 3 w gui

```
Choose a number [1/2/3]: 3
Give probability [0-1]: 0.23
Give picture width: 1000
```

Rys 9.2 - opcja 3 w konsoli

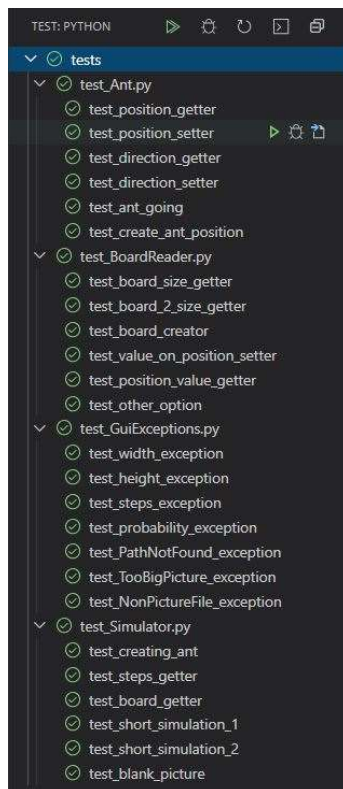
## WYNIK SYMULACJI

Po zakończeniu symulacji, należy wejść do folderu **steps**. Właśnie tam znajduje się wynik całej symulacji w postaci serii obrazków, tak jak pokazane to zostało na **rys 3**.

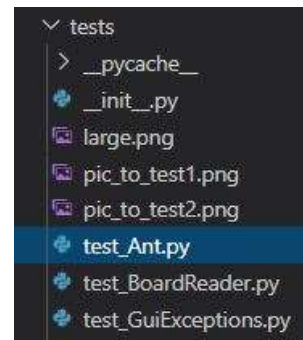
## ROZDZIAŁ IV. PRZEPROWADZONE TESTY

### TESTY JEDNOSTKOWE

Projekt ten zawiera liczne testy, sprawdzające prawidłowe działanie zawartych w nim klas, metod, funkcji czy atrybutów. Zaczynając od sprawdzenia pozycji mrówki na planszy aż po całe jej załadowanie z pliku.



Rys 10 - wszystkie przeprowadzone testy (działają!)



Rys 11 - pliki z testami

### WAŻNY PRZYKŁAD

Istotną właściwością mrówki Langtona jest trasa, jaką wykonuje ona po zrobieniu 10000 kroków. Tworzy ona wtedy charakterystyczny kształt nazwany „autostradą”, nie inaczej jest w tym projekcie.



Rys 12 - droga wykonana przez mrówkę po 11000 krokach

## TEST SYMULACJI

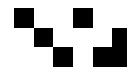
Podjąłem w projekcie próbę przeprowadzenia krótkiej symulacji na małym obrazku (np. o wymiarach 5x3), w której mogłem przewidzieć pierwsze 4 kroki mrówki. Następny krok program ze względu na mały obszar wybiera już losowo (stąd dalsze testy nie mają sensu – nie da się przewidzieć następnego kroku).

```

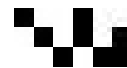
75 def test_short_simulation_2():
76     array = np.array([
77         [1, 0, 0, 1, 0, 0],
78         [0, 1, 0, 0, 0, 1],
79         [0, 0, 1, 0, 1, 1],
80     ])
81     assert simulator2.ant.position_x == 3
82     assert simulator2.ant.position_y == 1
83     simulator2.take_step()
84     array[1, 3] = 1
85     assert array[1, 3] == simulator2.board.get_pos_value(3, 1)
86     assert simulator2.ant.position_x == 2
87     assert simulator2.ant.position_y == 1
88     assert simulator2.ant.direction == "W"
89     simulator2.take_step()
90     array[1, 2] = 1
91     assert array[1, 2] == simulator2.board.get_pos_value(2, 1)
92     assert simulator2.ant.position_x == 2
93     assert simulator2.ant.position_y == 2
94     assert simulator2.ant.direction == "S"
95     simulator2.take_step()
96     array[2, 2] = 0
97     assert array[2, 2] == simulator2.board.get_pos_value(2, 2)
98     assert simulator2.ant.position_x == 1
99     assert simulator2.ant.position_y == 2
100    assert simulator2.ant.direction == "W"
101    simulator2.take_step()
102    array[2, 1] = 1
103    assert array[2, 1] == simulator2.board.get_pos_value(1, 2)
104    assert simulator2.ant.position_y != 3

```

Rys 13 - test symulacji



Rys 14.1 - załadowany obrazek jako tablica 0-1



Rys 14.2 - zmiana pozycji środkowego pixela



Rys 14.3 - mrowka skreca w lewo, idzie na pixel w 3 kolumnie i 2 wierszu



Rys 14.4 - mrowka skreca w lewo, idzie na pixel w 3 kolumnie i 3 wierszu



Rys 14.5 - mrowka skreca w prawo, idzie na pixel w 2 kolumnie i 3 rzędzie

Podobne symulacje przeprowadziłem również dla ładowanego obrazka 5x5 (4 kroki) i stworzonego białego obrazka 5x5 (10 kroków). Wszystkie symulacje wyszły prawidłowo.

## ROZDZIAŁ V. EWENTUALNY ROZWÓJ

Projekt można by było udoskonalić w możliwość pojawienia się kilku mrówek naraz na obrazie. O ile założenie te zostałyby sformułowane już w początkowej fazie projektowania (a tak naprawdę to w fazie planowania algorytmiki), zaimplementowanie pomysłu mogłoby dojść do skutku. Na obecnym etapie wdrożone algorytmy stworzone są z przeznaczeniem dla automatu jednej mrówki i ewentualna modyfikacja spowodowałaby zmianę struktury większości programu. Dlatego też kwestię tą postanowiłem umieścić w tym rozdziale dokumentacji.

## ROZDZIAŁ VI. WYKORZYSTANE ZASOBY

1. Ikona programu - [https://www.flaticon.com/free-icon/ant\\_2253561?term=ant&page=1&position=14](https://www.flaticon.com/free-icon/ant_2253561?term=ant&page=1&position=14)