Descenso al fondo de un cráter en marte

Juan Manuel Hernández Solano, José Carlos Yamuni Contreras, Miguel Steven Nguyen

March 16, 2023

1 Búsqueda local para el descenso al fondo de un cráter

[2]:
```
# Read crater dada
crater_map = np.load('crater_map.npy')
NR, NC = crater_map.shape
SCALE = 10.045
```

[4]:
```
# Define the rover class
class Rover():
    """
    This class represents the current position of the rover in relation
    to its surroundings.
    """

    def __init__(self, crater_map, position, prev_positions):
        """
        We will take in the map of the area and the rover's current position.
        ---
        crater_map: numpy array
        start_position: tuple like (x, y) WITH MATRIX COORDINATES
        end_position: tuple like (x, y)
        ---
        Everytime the rover moves, it adds to a list its previous positions.
        """
        self.crater_map = copy.copy(crater_map)
        self.position = position
        self.current_row = self.position[0]
        self.current_column = self.position[1]
        self.prev_positions = prev_positions
        self.prev_positions.append(self.position)

    def show(self):
        """ Plots the points where the rover has traveled """
        fig = plt.figure(figsize=(16, 9))
        ax = fig.add_subplot()
        sns.heatmap(crater_map, cmap='hot')
        ax.set_xlim(200, 400)
        ax.set_ylim(450, 700)
        for i in range(len(self.prev_positions) - 1):
            ax.plot(
                [self.prev_positions[i][1], self.prev_positions[i + 1][1]],
                [self.prev_positions[i][0], self.prev_positions[i + 1][0]],
                color='cyan'
            );
        return ax

    def cost(self):
        """
        Returns current height. Lower is better
        """

        return self.crater_map[self.current_row][self.current_column]

    def neighbour(self):
        """
        Returns a random neighbour from a given state. The neighbour must have␣
↪a height
        difference less than 2m.
        """
        current_height = self.crater_map[self.current_row][self.current_column]
        directions = []

        # UL
        if abs(current_height - self.crater_map[self.current_row - 1][self.
↪current_column - 1]) <= 2:
            directions.append((self.current_row - 1, self.current_column - 1))
        # UU
        if abs(current_height - self.crater_map[self.current_row - 1][self.
↪current_column]) <= 2:
            directions.append((self.current_row - 1, self.current_column))
        # UR
        if abs(current_height - self.crater_map[self.current_row - 1][self.
↪current_column + 1]) <= 2:
            directions.append((self.current_row - 1, self.current_column + 1))
        # LL
        if abs(current_height - self.crater_map[self.current_row][self.
↪current_column - 1]) <= 2:
            directions.append((self.current_row, self.current_column - 1))
        # RR
        if abs(current_height - self.crater_map[self.current_row][self.
↪current_column + 1]) <= 2:
            directions.append((self.current_row, self.current_column + 1))
        # DL
        if abs(current_height - self.crater_map[self.current_row + 1][self.
↪current_column - 1]) <= 2:
            directions.append((self.current_row + 1, self.current_column - 1))
        # DD
        if abs(current_height - self.crater_map[self.current_row + 1][self.
↪current_column]) <= 2:
            directions.append((self.current_row + 1, self.current_column))
        # DR
        if abs(current_height - self.crater_map[self.current_row + 1][self.
↪current_column + 1]) <= 2:
            directions.append((self.current_row + 1, self.current_column + 1))

        new_position = random.choice(directions)
        self.prev_positions.append(new_position)
        new_rover = Rover(self.crater_map, new_position, copy.copy(self.
↪prev_positions))
        return new_rover
```

Nuestra clase del Rover implementa los métodos normalmente utilizados en las búsquedas locales; regresa vecinos y costos.

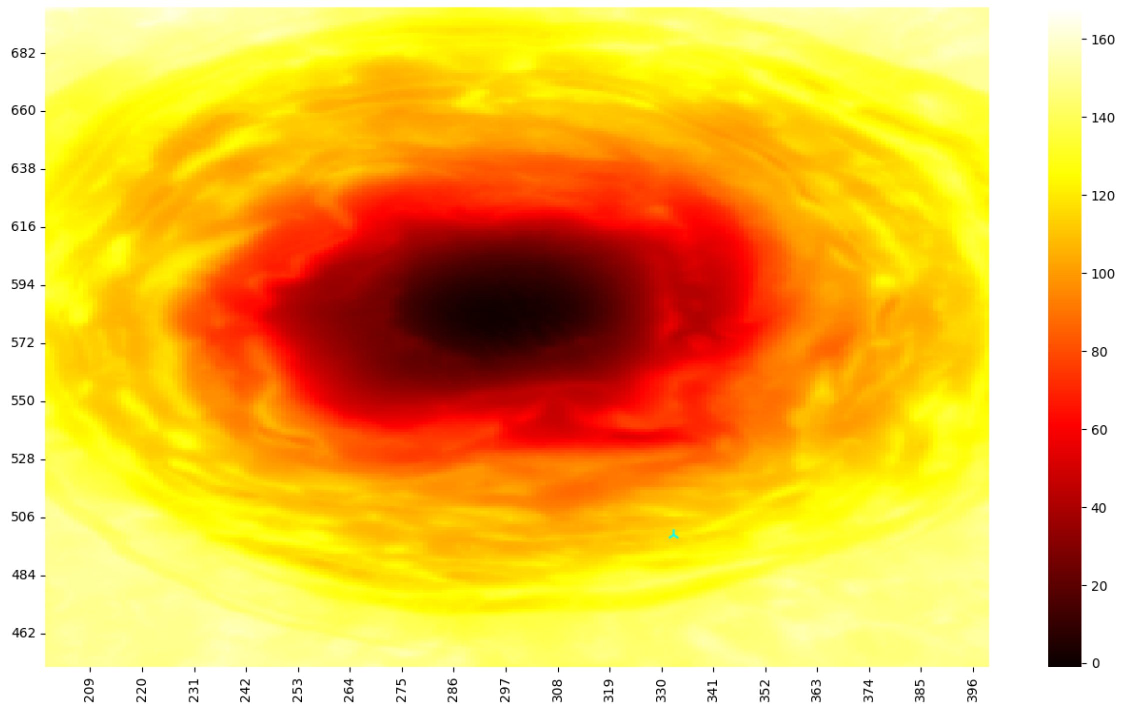
2 Greedy Search

2.1 Given Initial Positions

[5]:
```
# Convert initial position to matrix coordinates
initial_x = 3350
initial_y = 5800
initial_row = get_row(NR, initial_y, SCALE)
initial_column = get_column(initial_x, SCALE)
```

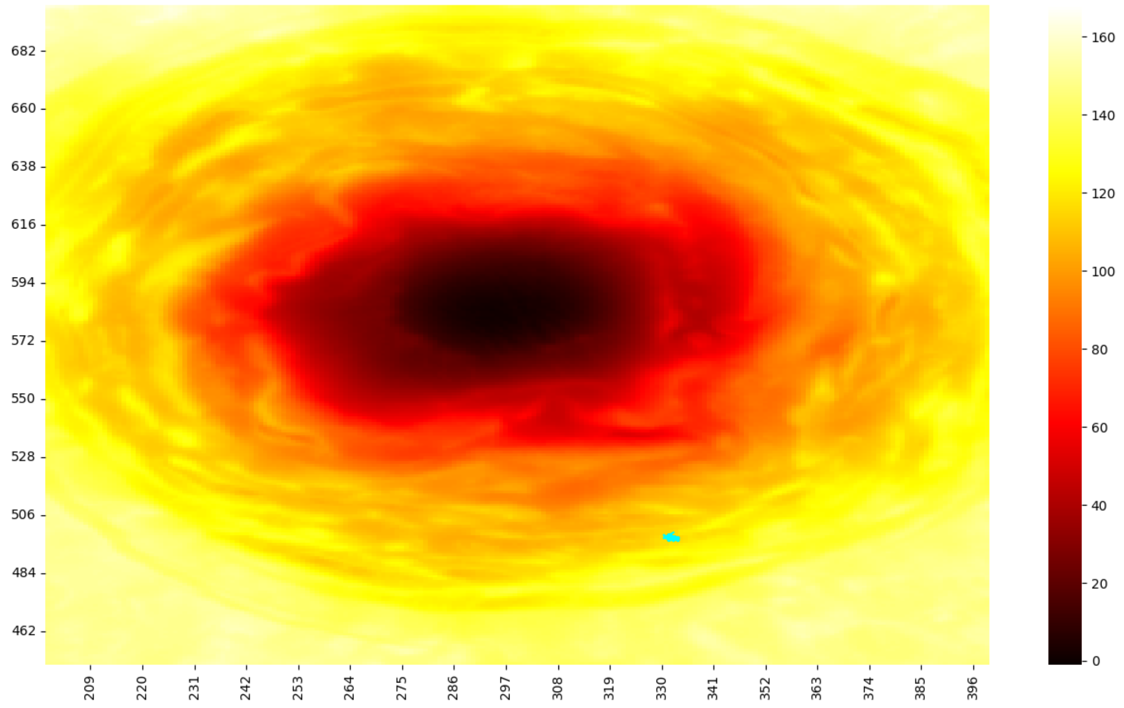
[6]:
```
initial_rover = Rover(crater_map, (initial_row, initial_column), [])
initial_rover.show();
plt.scatter(333, 500, marker='2', s=80, color='cyan')
print(f"Initial cost: {initial_rover.cost()}")
```

```
Initial cost: 117.02547363281272
```



[7]:
```
# Hill climbing
hill_climb_rover = initial_rover

cost = hill_climb_rover.cost()
step = 0

print(f"Iteration: {step} | Cost: {cost}")
while cost > 0 and step < 100:
    # Get new neighbour and cost
    neighbour = hill_climb_rover.neighbour()
    new_cost = neighbour.cost()

    # Check if cost is less
    if new_cost < cost:
        hill_climb_rover = neighbour
        cost = new_cost

    step += 1

    if step % 10 == 0:
        print(f"Iteration: {step} | Cost: {cost}")

print("\nFinal Route")
hill_climb_rover.show();
```

```
Iteration: 0 | Cost: 117.02547363281272
Iteration: 10 | Cost: 114.60082763671898
Iteration: 20 | Cost: 114.60082763671898
Iteration: 30 | Cost: 114.60082763671898
Iteration: 40 | Cost: 114.60082763671898
Iteration: 50 | Cost: 114.60082763671898
Iteration: 60 | Cost: 114.60082763671898
Iteration: 70 | Cost: 114.60082763671898
Iteration: 80 | Cost: 114.60082763671898
Iteration: 90 | Cost: 114.60082763671898
Iteration: 100 | Cost: 114.60082763671898

Final Route
```

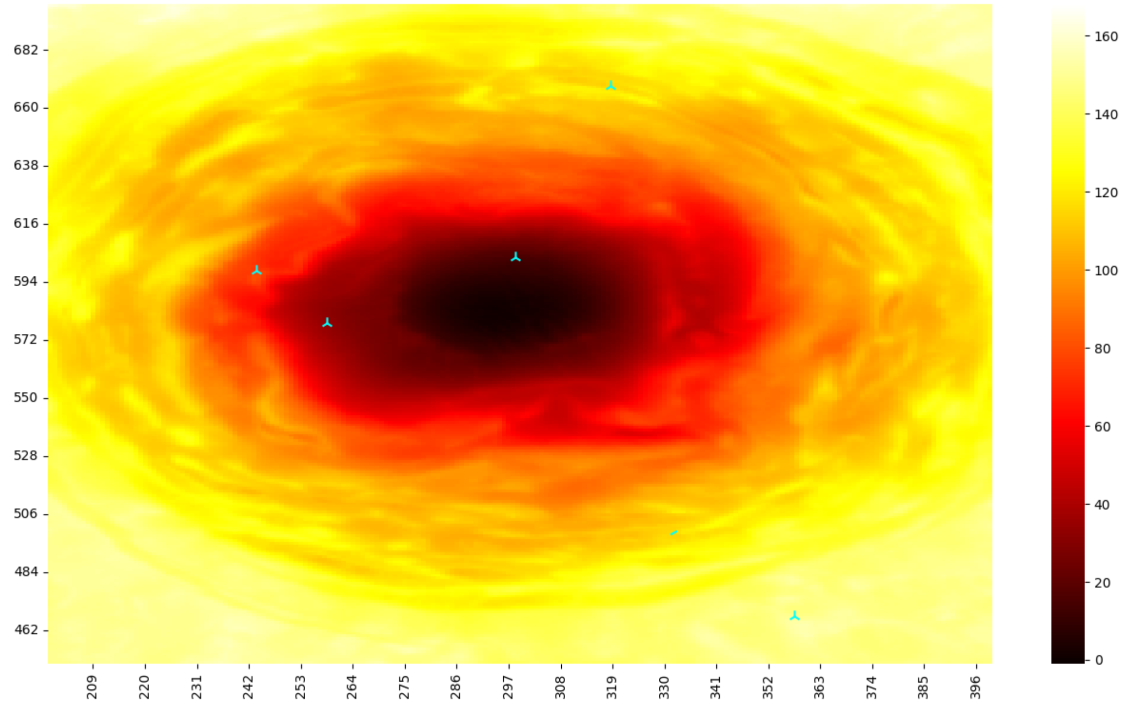


Podemos observar que el Rover no llega entrar al cráter. Esta búsqueda no es capáz de explorar suficiente posiciones.

3 Custom initial positions

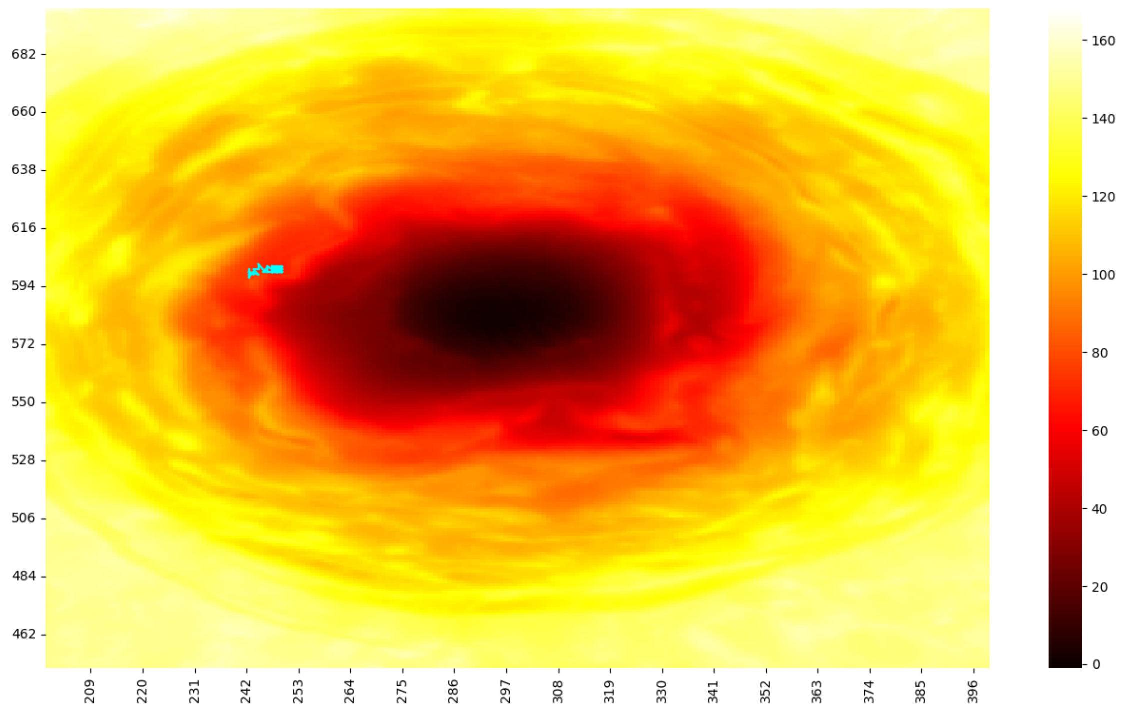
[8]:
```
custom_points = [
    (2450, 4800),
    (3000, 4750),
    (3200, 4100),
    (2600, 5000),
    (3600, 6120)
]
```

[9]:
```
initial_rover.show();
for i in custom_points:
    row = get_row(NR, i[1], SCALE)
    column = get_column(i[0], SCALE)
    plt.scatter(column, row, marker='2', s=80, color='cyan')
```



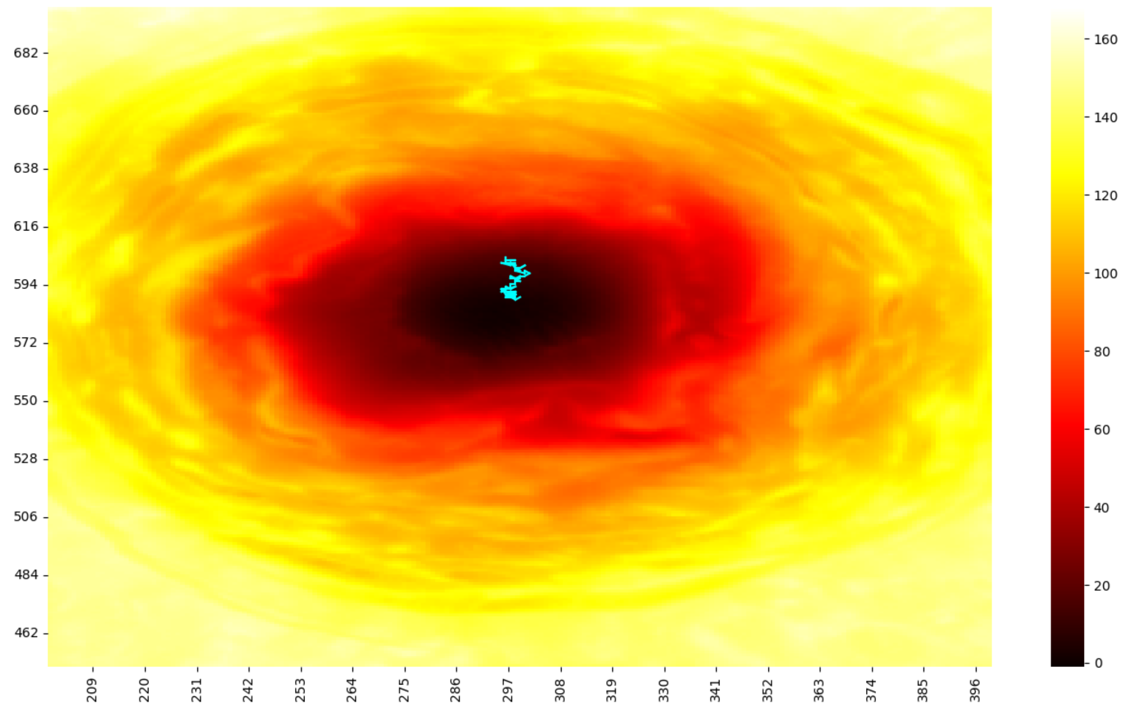
[10]:
```
initial_row = get_row(NR, custom_points[0][1], SCALE)
initial_column = get_column(custom_points[0][0], SCALE)
initial_rover = Rover(crater_map, (initial_row, initial_column), [])
hill_climb_rover = initial_rover

cost = hill_climb_rover.cost()
step = 0

print(f"Iteration: {step} | Cost: {cost}")
while cost > 0 and step < 100:
    # Get new neighbour and cost
    neighbour = hill_climb_rover.neighbour()
    new_cost = neighbour.cost()

    # Check if cost is less
    if new_cost < cost:
        hill_climb_rover = neighbour
        cost = new_cost

    step += 1

    if step % 10 == 0:
        print(f"Iteration: {step} | Cost: {cost}")

print("\nFinal Route")
hill_climb_rover.show();
```

```
Iteration: 0 | Cost: 73.97523925781272
Iteration: 10 | Cost: 72.76457519531272
Iteration: 20 | Cost: 68.30317626953146
Iteration: 30 | Cost: 67.68876464843771
Iteration: 40 | Cost: 67.47635498046897
Iteration: 50 | Cost: 67.47635498046897
Iteration: 60 | Cost: 67.47635498046897
Iteration: 70 | Cost: 67.47635498046897
Iteration: 80 | Cost: 67.47635498046897
Iteration: 90 | Cost: 67.47635498046897
Iteration: 100 | Cost: 67.47635498046897

Final Route
```



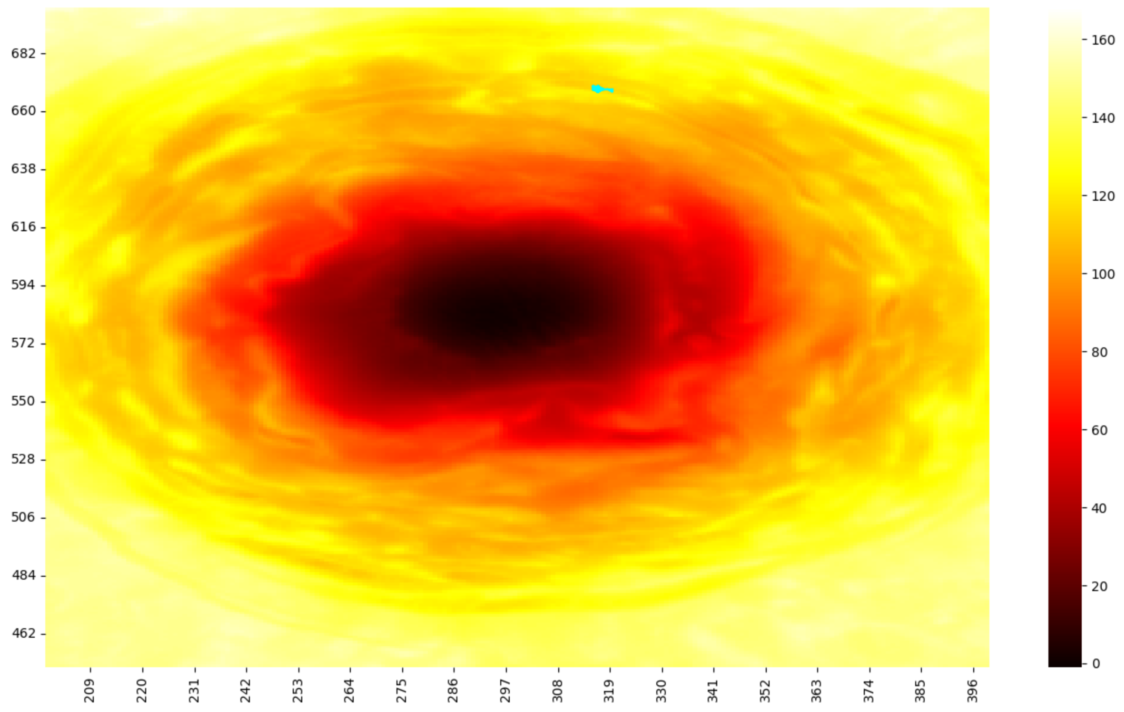
[11]:
```python
initial_row = get_row(NR, custom_points[1][1], SCALE)
initial_column = get_column(custom_points[1][0], SCALE)
initial_rover = Rover(crater_map, (initial_row, initial_column), [])

hill_climb_rover = initial_rover

cost = hill_climb_rover.cost()
step = 0

print(f"Iteration: {step} | Cost: {cost}")
while cost > 0 and step < 100:
    # Get new neighbour and cost
    neighbour = hill_climb_rover.neighbour()
    new_cost = neighbour.cost()

    # Check if cost is less
    if new_cost < cost:
        hill_climb_rover = neighbour
        cost = new_cost

    step += 1

    if step % 10 == 0:
        print(f"Iteration: {step} | Cost: {cost}")

print("\nFinal Route")
hill_climb_rover.show();
```

```
Iteration: 0 | Cost: 15.682854003906469
Iteration: 10 | Cost: 14.277929687500219
Iteration: 20 | Cost: 13.301357421875219
Iteration: 30 | Cost: 11.097604980468969
Iteration: 40 | Cost: 8.40298339843772
Iteration: 50 | Cost: 4.640405273437718
Iteration: 60 | Cost: 4.395268554687719
Iteration: 70 | Cost: 3.304389648437718
Iteration: 80 | Cost: 3.304389648437718
Iteration: 90 | Cost: 3.304389648437718
Iteration: 100 | Cost: 2.7348779296877184

Final Route
```



[12]:
```python
initial_row = get_row(NR, custom_points[2][1], SCALE)
initial_column = get_column(custom_points[2][0], SCALE)
initial_rover = Rover(crater_map, (initial_row, initial_column), [])

hill_climb_rover = initial_rover

cost = hill_climb_rover.cost()
step = 0

print(f"Iteration: {step} | Cost: {cost}")
while cost > 0 and step < 100:
    # Get new neighbour and cost
    neighbour = hill_climb_rover.neighbour()
    new_cost = neighbour.cost()

    # Check if cost is less
    if new_cost < cost:
        hill_climb_rover = neighbour
        cost = new_cost

    step += 1

    if step % 10 == 0:
        print(f"Iteration: {step} | Cost: {cost}")
print("\nFinal Route")
hill_climb_rover.show();
```

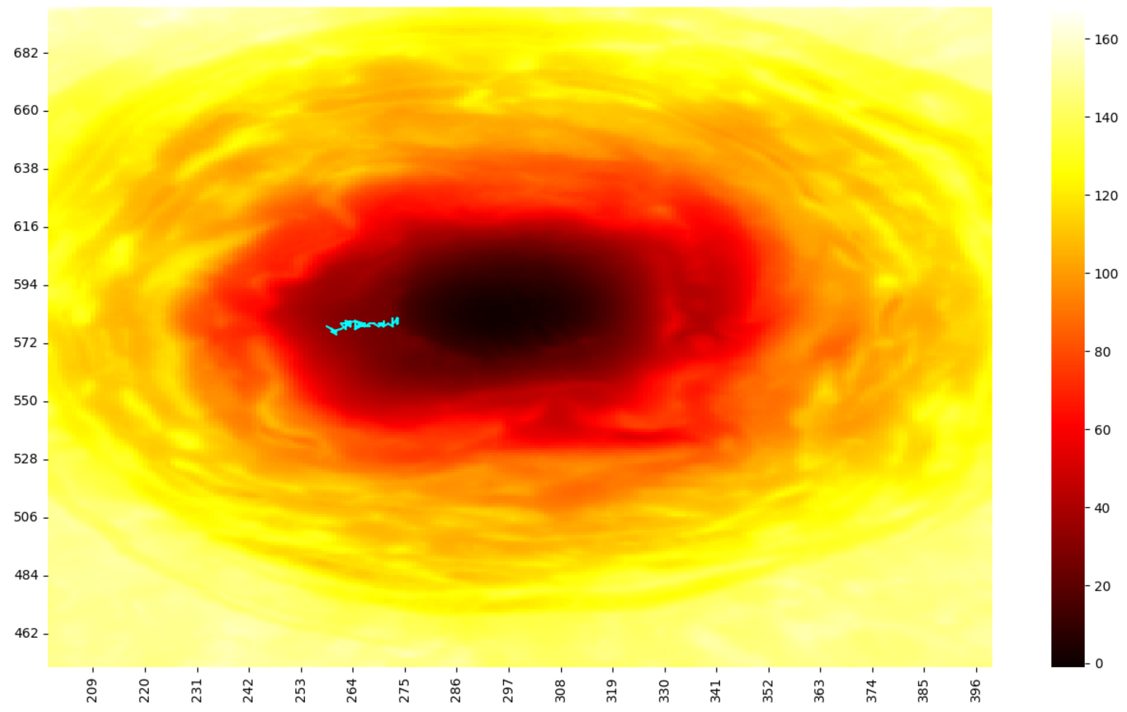
```
Iteration: 0 | Cost: 115.38589355468771
Iteration: 10 | Cost: 113.31362304687522
Iteration: 20 | Cost: 113.31362304687522
Iteration: 30 | Cost: 113.31362304687522
Iteration: 40 | Cost: 113.31362304687522
Iteration: 50 | Cost: 113.31362304687522
Iteration: 60 | Cost: 113.31362304687522
Iteration: 70 | Cost: 113.31362304687522
Iteration: 80 | Cost: 113.31362304687522
Iteration: 90 | Cost: 113.31362304687522
Iteration: 100 | Cost: 113.31362304687522

Final Route
```



[13]:
```
initial_row = get_row(NR, custom_points[3][1], SCALE)
initial_column = get_column(custom_points[3][0], SCALE)
initial_rover = Rover(crater_map, (initial_row, initial_column), [])

hill_climb_rover = initial_rover
```

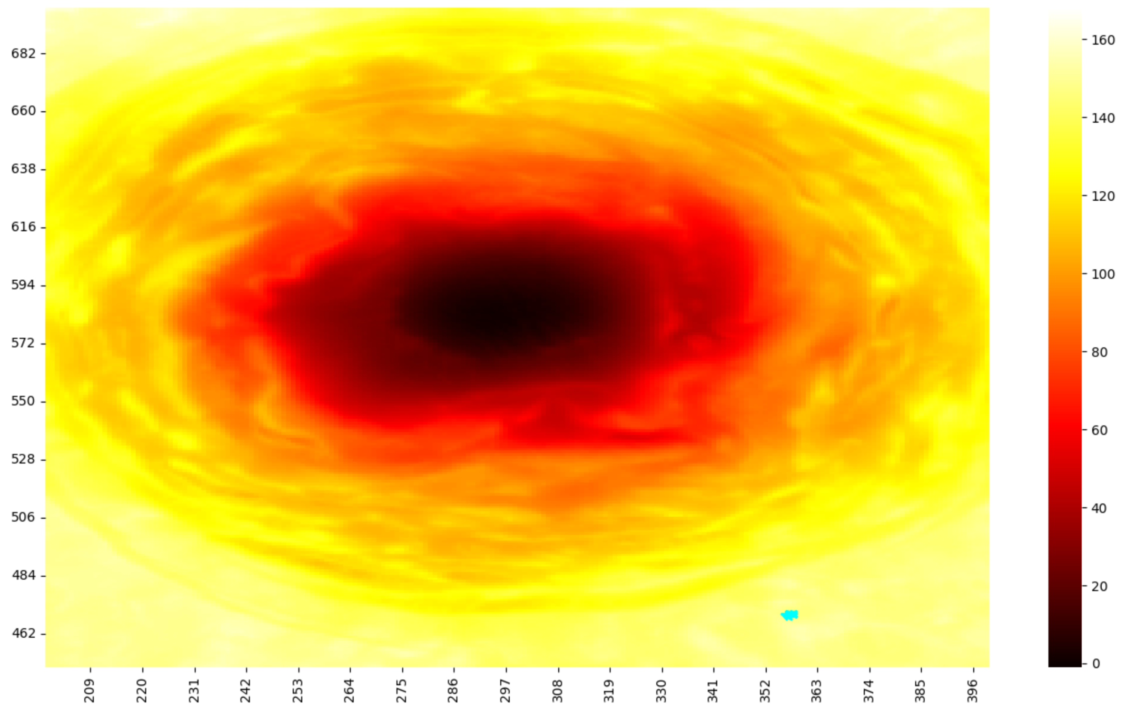
```python
cost = hill_climb_rover.cost()
step = 0

print(f"Iteration: {step} | Cost: {cost}")
while cost > 0 and step < 100:
    # Get new neighbour and cost
    neighbour = hill_climb_rover.neighbour()
    new_cost = neighbour.cost()

    # Check if cost is less
    if new_cost < cost:
        hill_climb_rover = neighbour
        cost = new_cost

    step += 1

    if step % 10 == 0:
        print(f"Iteration: {step} | Cost: {cost}")

print("\nFinal Route")
hill_climb_rover.show();
```

```
Iteration: 0 | Cost: 33.10188720703147
Iteration: 10 | Cost: 28.33530761718772
Iteration: 20 | Cost: 27.71760498046897
Iteration: 30 | Cost: 27.33277832031272
Iteration: 40 | Cost: 24.977692871093968
Iteration: 50 | Cost: 21.05169921875022
Iteration: 60 | Cost: 18.39313720703147
Iteration: 70 | Cost: 18.39313720703147
Iteration: 80 | Cost: 18.39313720703147
Iteration: 90 | Cost: 18.39313720703147
Iteration: 100 | Cost: 18.39313720703147

Final Route
```



```
[14]: initial_row = get_row(NR, custom_points[4][1], SCALE)
      initial_column = get_column(custom_points[4][0], SCALE)
      initial_rover = Rover(crater_map, (initial_row, initial_column), [])

      hill_climb_rover = initial_rover

      cost = hill_climb_rover.cost()
      step = 0

      print(f"Iteration: {step} | Cost: {cost}")
      while cost > 0 and step < 100:
          # Get new neighbour and cost
          neighbour = hill_climb_rover.neighbour()
          new_cost = neighbour.cost()

          # Check if cost is less
          if new_cost < cost:
              hill_climb_rover = neighbour
              cost = new_cost

          step += 1

          if step % 10 == 0:
              print(f"Iteration: {step} | Cost: {cost}")
      print("\nFinal Route")
      hill_climb_rover.show();
```

```
Iteration: 0 | Cost: 147.14847412109395
Iteration: 10 | Cost: 145.75047851562522
Iteration: 20 | Cost: 145.75047851562522
Iteration: 30 | Cost: 145.75047851562522
Iteration: 40 | Cost: 145.75047851562522
Iteration: 50 | Cost: 145.75047851562522
Iteration: 60 | Cost: 145.75047851562522
Iteration: 70 | Cost: 145.75047851562522
Iteration: 80 | Cost: 145.75047851562522
Iteration: 90 | Cost: 145.75047851562522
Iteration: 100 | Cost: 145.75047851562522

Final Route
```



Con la búsqueda codiciosa, podemos ver que los Rovers no llegan muy lejos, aunque estén casi al centro del crater.

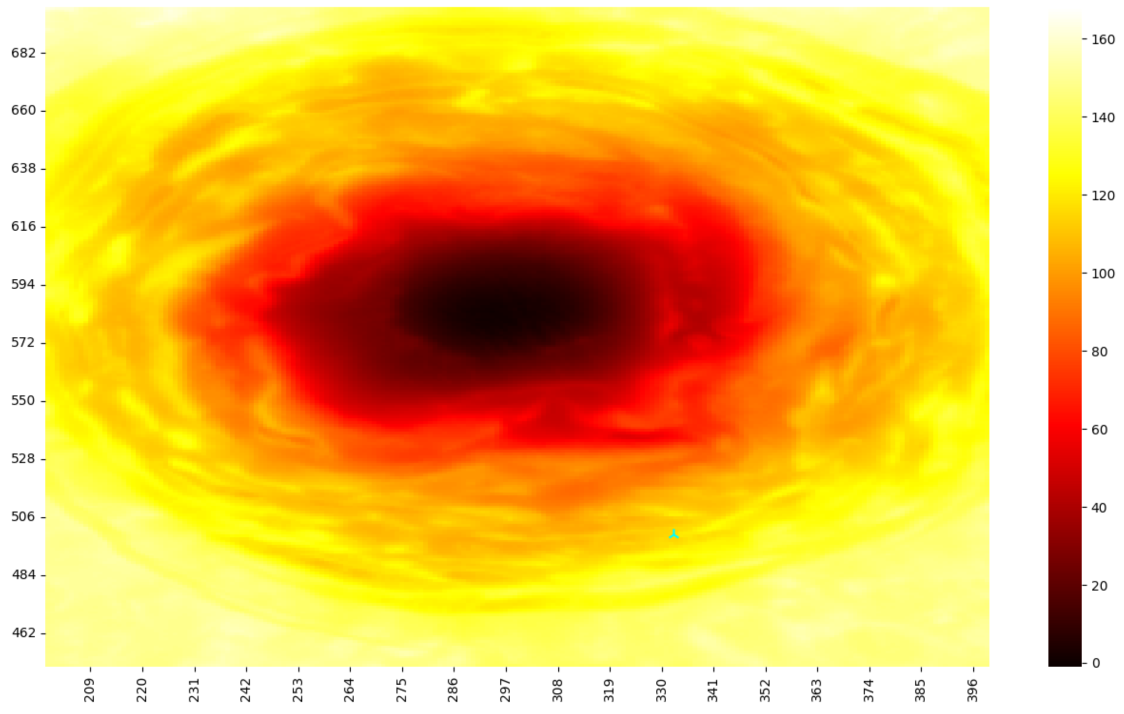
4 Simulated Annealing

4.1 Using given points

[15]:
```
# Convert initial position to matrix coordinates
initial_x = 3350
initial_y = 5800
initial_row = get_row(NR, initial_y, SCALE)
initial_column = get_column(initial_x, SCALE)
```

[16]:
```
initial_rover = Rover(crater_map, (initial_row, initial_column), [])
initial_rover.show();
plt.scatter(333, 500, marker='2', s=80, color='cyan')
print(f"Initial cost: {initial_rover.cost()}")
```

```
Initial cost: 117.02547363281272
```

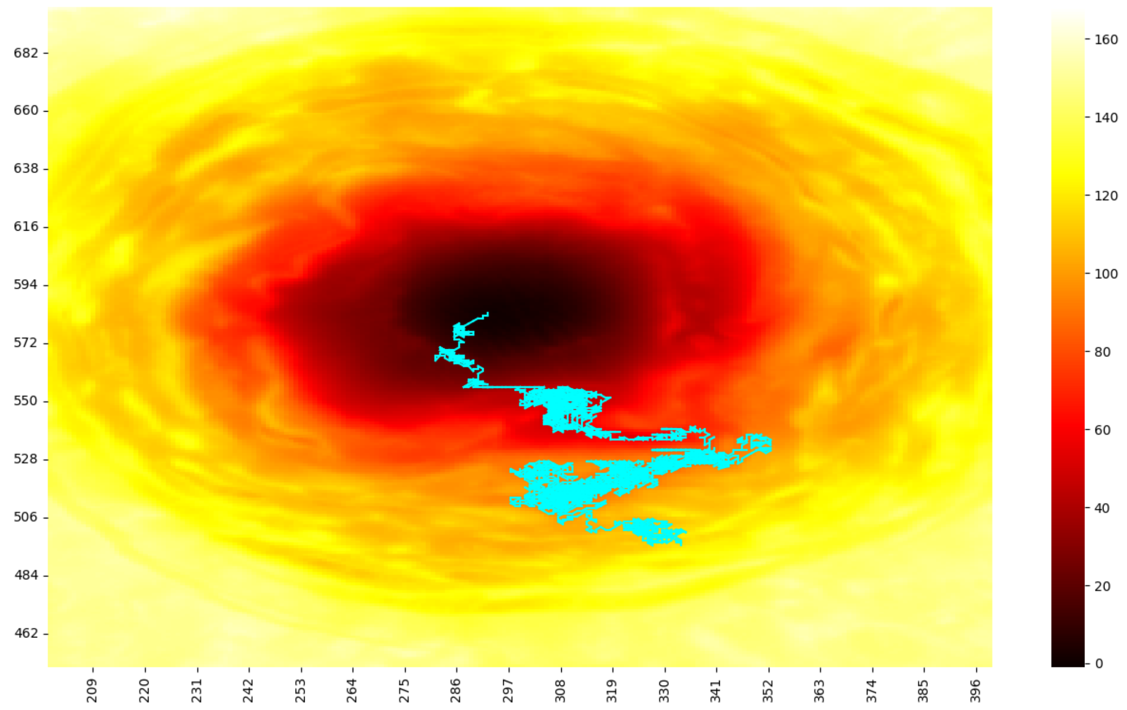


[17]:
```
random.seed(10)
```

[18]:
```
# Simulated Annealing
sim_ann_rover = initial_rover

cost = sim_ann_rover.cost()
step = 0
```

```python
alpha = 0.9995
t0 = 20
t = t0

print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
while t > 0.05 and cost > 1:
    # Calculate temperature
    t = t0 * math.pow(alpha, step)
    step += 1

    # Get random neighbour
    neighbour = sim_ann_rover.neighbour()
    new_cost = neighbour.cost()

    # Test neighbour
    if new_cost < cost:
        sim_ann_rover = neighbour
        cost = new_cost
    else:
        p = math.exp(-(new_cost - cost) / t)
        if p >= random.random():
            sim_ann_rover = neighbour
            cost = new_cost

    if step % 2500 == 0:
        print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")


print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
print("\nFinal Route")
sim_ann_rover.show();
```

```
Iteration: 0 | Cost: 117.02547363281272 | Temperature: 20.00000
Iteration: 2500 | Cost: 85.51915283203147 | Temperature: 5.73117
Iteration: 4810 | Cost: 0.6949072265627183 | Temperature: 1.80513

Final Route
```

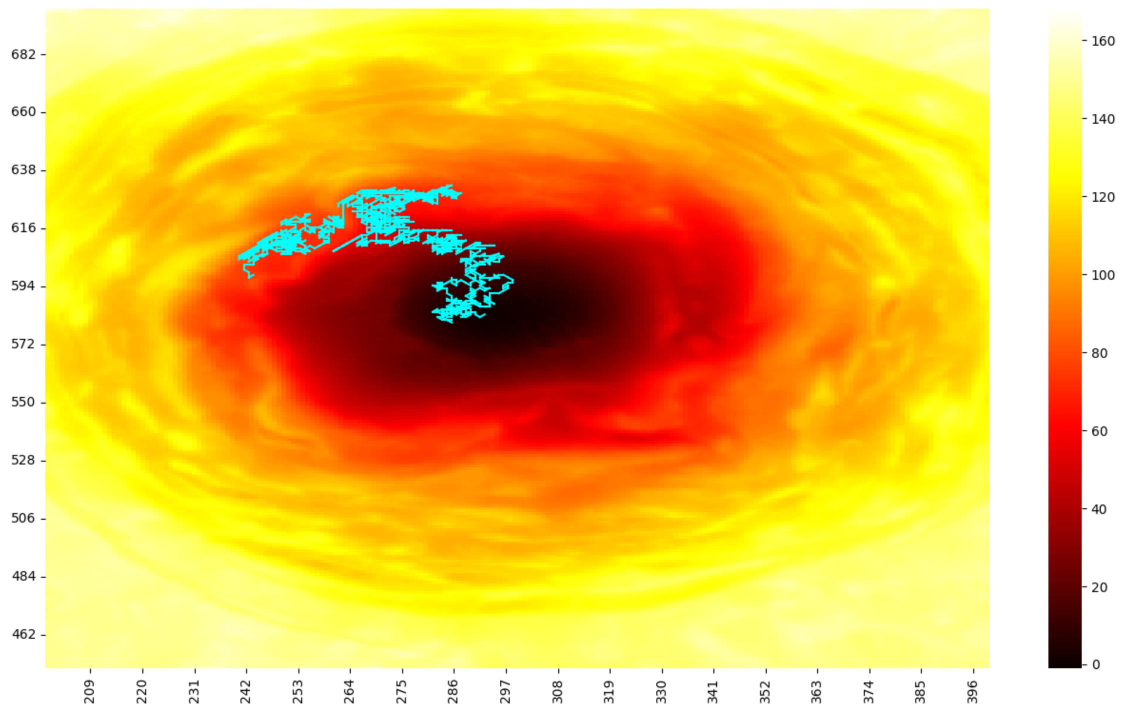


Para la posición inicial dada, el Rover llega al centro del cráter, aunque desperdicia energia despues de llegar hasta abajo. Esto se debe a los parámetros que definimos para temperatura y costo.

4.2 Custom Points

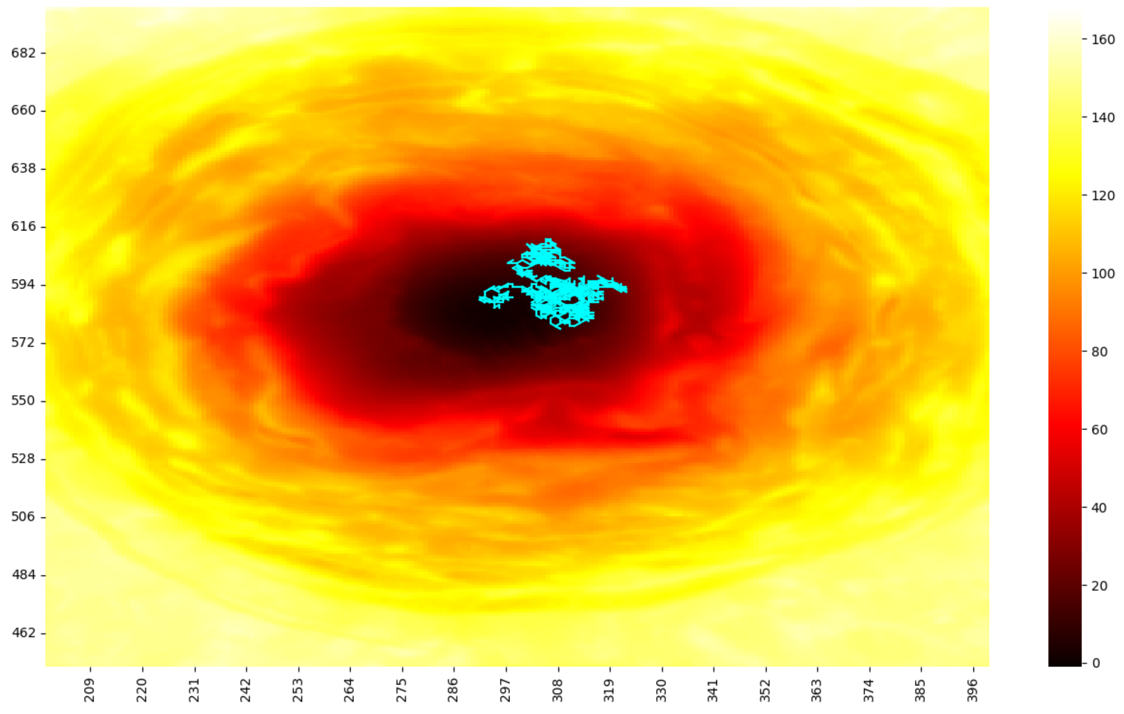
[19]:
```python
initial_row = get_row(NR, custom_points[0][1], SCALE)
initial_column = get_column(custom_points[0][0], SCALE)
initial_rover = Rover(crater_map, (initial_row, initial_column), [])

sim_ann_rover = initial_rover

cost = sim_ann_rover.cost()
step = 0

alpha = 0.9995
t0 = 20
t = t0

print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
while t > 0.05 and cost > 1:
    # Calculate temperature
    t = t0 * math.pow(alpha, step)
    step += 1

    # Get random neighbour
    neighbour = sim_ann_rover.neighbour()
    new_cost = neighbour.cost()

    # Test neighbour
    if new_cost < cost:
        sim_ann_rover = neighbour
        cost = new_cost
    else:
        p = math.exp(-(new_cost - cost) / t)
        if p >= random.random():
            sim_ann_rover = neighbour
            cost = new_cost

    if step % 2500 == 0:
        print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")


print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
print("\nFinal Route")
sim_ann_rover.show();
```

```
Iteration: 0 | Cost: 73.97523925781272 | Temperature: 20.00000
Iteration: 1916 | Cost: 0.6949072265627183 | Temperature: 7.67519

Final Route
```



[20]:
```python
initial_row = get_row(NR, custom_points[1][1], SCALE)
initial_column = get_column(custom_points[1][0], SCALE)
initial_rover = Rover(crater_map, (initial_row, initial_column), [])

sim_ann_rover = initial_rover

cost = sim_ann_rover.cost()
step = 0

alpha = 0.9995
t0 = 20
t = t0

print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
while t > 0.05 and cost > 1:
    # Calculate temperature
    t = t0 * math.pow(alpha, step)
    step += 1

    # Get random neighbour
    neighbour = sim_ann_rover.neighbour()
    new_cost = neighbour.cost()

    # Test neighbour
    if new_cost < cost:
        sim_ann_rover = neighbour
        cost = new_cost
    else:
        p = math.exp(-(new_cost - cost) / t)
        if p >= random.random():
            sim_ann_rover = neighbour
            cost = new_cost

    if step % 2500 == 0:
        print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")


print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
print("\nFinal Route")
sim_ann_rover.show();
```

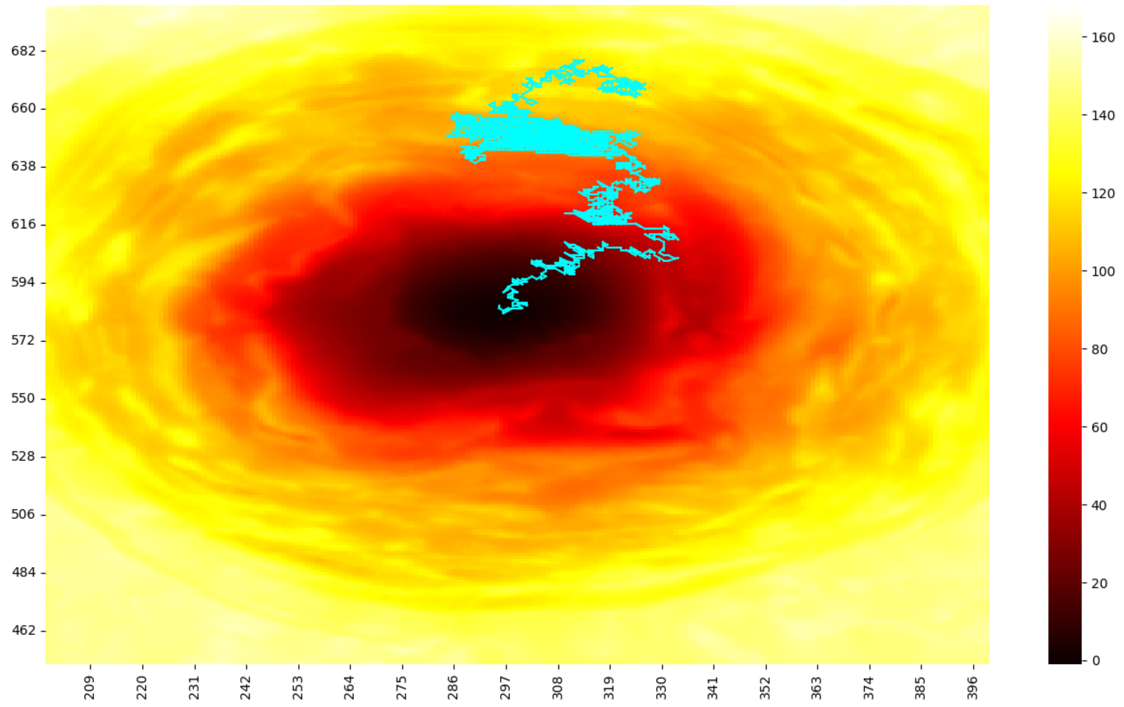
```
Iteration: 0 | Cost: 15.682854003906469 | Temperature: 20.00000
Iteration: 1131 | Cost: 0.9429150390627182 | Temperature: 11.36560

Final Route
```



```
[21]: initial_row = get_row(NR, custom_points[2][1], SCALE)
      initial_column = get_column(custom_points[2][0], SCALE)
      initial_rover = Rover(crater_map, (initial_row, initial_column), [])

      sim_ann_rover = initial_rover

      cost = sim_ann_rover.cost()
      step = 0

      alpha = 0.9995
      t0 = 20
      t = t0

      print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
      while t > 0.05 and cost > 1:
          # Calculate temperature
          t = t0 * math.pow(alpha, step)
          step += 1

          # Get random neighbour
          neighbour = sim_ann_rover.neighbour()
          new_cost = neighbour.cost()

          # Test neighbour
```

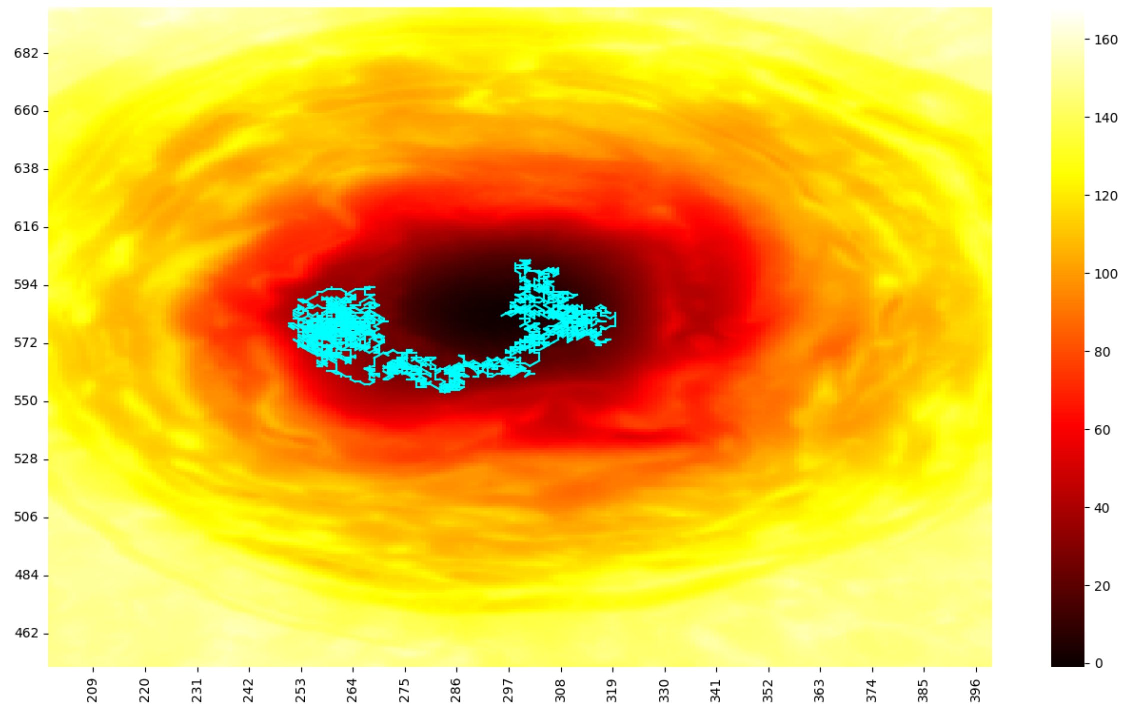
```
    if new_cost < cost:
        sim_ann_rover = neighbour
        cost = new_cost
    else:
        p = math.exp(-(new_cost - cost) / t)
        if p >= random.random():
            sim_ann_rover = neighbour
            cost = new_cost

    if step % 2500 == 0:
        print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")


print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
print("\nFinal Route")
sim_ann_rover.show();
```

```
Iteration: 0 | Cost: 115.38589355468771 | Temperature: 20.00000
Iteration: 2500 | Cost: 105.17575195312521 | Temperature: 5.73117
Iteration: 4773 | Cost: 0.9429150390627182 | Temperature: 1.83884

Final Route
```



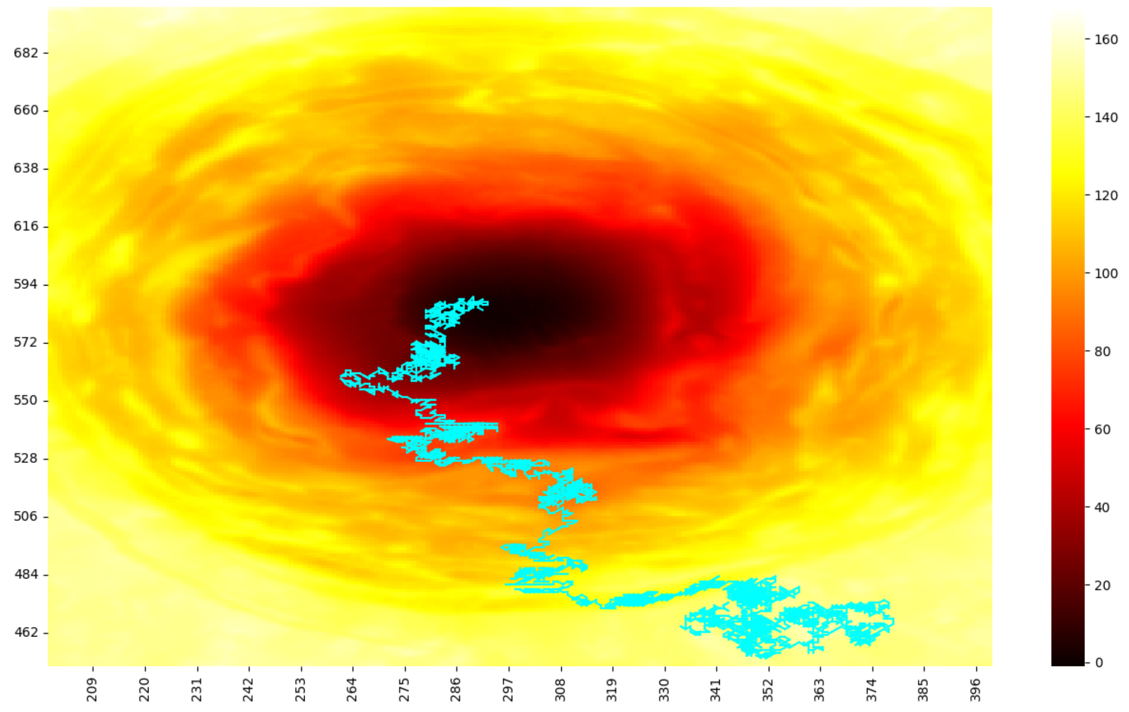
[22]:
```python
initial_row = get_row(NR, custom_points[3][1], SCALE)
initial_column = get_column(custom_points[3][0], SCALE)
initial_rover = Rover(crater_map, (initial_row, initial_column), [])

sim_ann_rover = initial_rover

cost = sim_ann_rover.cost()
step = 0

alpha = 0.9995
t0 = 20
t = t0

print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
while t > 0.05 and cost > 1:
    # Calculate temperature
    t = t0 * math.pow(alpha, step)
    step += 1

    # Get random neighbour
    neighbour = sim_ann_rover.neighbour()
    new_cost = neighbour.cost()

    # Test neighbour
    if new_cost < cost:
        sim_ann_rover = neighbour
        cost = new_cost
    else:
        p = math.exp(-(new_cost - cost) / t)
        if p >= random.random():
            sim_ann_rover = neighbour
            cost = new_cost

    if step % 2500 == 0:
        print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")


print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
print("\nFinal Route")
sim_ann_rover.show();
```

```
Iteration: 0 | Cost: 33.10188720703147 | Temperature: 20.00000
Iteration: 2500 | Cost: 15.136879882812718 | Temperature: 5.73117
Iteration: 3170 | Cost: 0.9780517578127182 | Temperature: 4.09938

Final Route
```



```
[23]: initial_row = get_row(NR, custom_points[4][1], SCALE)
initial_column = get_column(custom_points[4][0], SCALE)
initial_rover = Rover(crater_map, (initial_row, initial_column), [])

sim_ann_rover = initial_rover

cost = sim_ann_rover.cost()
step = 0

alpha = 0.9995
t0 = 20
t = t0

print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
while t > 0.05 and cost > 1:
    # Calculate temperature
    t = t0 * math.pow(alpha, step)
    step += 1

    # Get random neighbour
    neighbour = sim_ann_rover.neighbour()
    new_cost = neighbour.cost()

    # Test neighbour
    if new_cost < cost:
        sim_ann_rover = neighbour
        cost = new_cost
    else:
        p = math.exp(-(new_cost - cost) / t)
        if p >= random.random():
            sim_ann_rover = neighbour
            cost = new_cost

    if step % 2500 == 0:
        print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")


print(f"Iteration: {step} | Cost: {cost} | Temperature: {t:.5f}")
print("\nFinal Route")
sim_ann_rover.show();
```

```
Iteration: 0 | Cost: 147.14847412109395 | Temperature: 20.00000
Iteration: 2500 | Cost: 119.47951416015647 | Temperature: 5.73117
Iteration: 4804 | Cost: 0.9989306640627182 | Temperature: 1.81055

Final Route
```



En la mayoria de los casos, el recocido simulado es capaz de llegar al centro del crater. Sin embargo, debido a la naturaleza estocástica del algoritmo, no siempre vimos los mismos resultados al correr las simulaciones.

5 Conclusiones

Para que el Rover llegara al fondo del cráter de la superficie marciana, el problema fue planteado de la siguiente manera. Los estados son representados como coordenadas dentro del mapa, el cual es una matriz donde los números representan un color que a su vez indican la altura de los pixeles. Además, fue necesario escalar las coordenadas del mapa debido a que cada píxel representaba aproximadamente 10 metros en la imagen. Para comparar el problema con diferentes condiciones iniciales, también se incluyó una función que graficara los puntos recorridos sobre el mapa de la superficie. Como se dijo anteriormente, los estados son representados con coordenadas de la matriz. Por lo tanto, el costo de cada estado es simplemente el número de la coordenada que, como ya se dijo, es la altura en donde se encuentra el Rover. Los vecinos de los estados son definidos como los ocho pixeles colindantes con el de la posición actual, siempre y cuando tengan una diferencia de altura menor a dos con respecto al estado actual. Para que se pueda graficar el camino, se almacenan todos los vecinos escogidos en una lista.

Primero se trató de implementar el algoritmo de búsqueda voraz. Este algoritmo no garantiza en ningún contexto la completitud debido a que siempre escoge un vecino con mejor costo al anterior. Por lo tanto, si no se ha llegado a la meta y no existen vecino con una altura menor, el algoritmo hará que el Rover se estanque en esa posición. Dada la posición inicial establecida previamente, (3350, 5800), el Rover no estuvo ni cerca de llegar al cráter. Posteriormente, se probaron otras cinco posiciones iniciales, de las cuales solo en una se llegó al fondo del cráter. En la única vez que llegó, las condiciones iniciales ya estaban bastante cerca del fondo del cráter. Por lo tanto, se puede concluir que el algoritmo de búsqueda voraz, es poco efectivo a la hora de resolver problemas de búsqueda local debido a que no puede salir de máximos locales.

El segundo algoritmo implementado fue el de recocido simulado. Este algoritmo es mucho mejor que el anterior, ya que este acepta o rechaza el nuevo vecino que se escoge como solución en base a una función de probabilidad. Conforme pasan las iteraciones. A medida que avanzan las iteraciones, el parámetro de temperatura se reduce gradualmente, lo cual hace cada vez más difícil aceptar soluciones peores. Esto permite al algoritmo escapar de máximos locales, lo cual no puede hacer la búsqueda voraz. En este caso, con las condiciones iniciales preestablecidas el Rover si llegó al fondo del cráter. Al igual que con la búsqueda voraz, se probaron otras cinco posiciones iniciales. Sin embargo, en esta ocasión se llegó al fondo del cráter tres de cinco veces. Por lo tanto, se puede concluir que el recocido simulado, a pesar de no ser siempre completo dentro de un tiempo razonable, es bastante efectivo para problemas de búsqueda local. Las únicas desventajas que se deben de tomar en cuenta para la implementación de este método es su alto costo computacional y de tiempo.