Exercício 2 - GC 2021.1

Carolina Herbster

Questão 1

A. θ(a, b) = θ(b) - θ(a)
   Seja a o vetor (x_a, y_a), e $a = y_a/x_a$. A implementação de θ(a) se dá por:
   - $y_a + 1 - (1/a)$, se y > 0, x > 0, a ≥ 1
   - a, se y > 0, x > 0, a < 1
   - $4 - \theta((-x_a, y_a))$, se y > 0, x < 0
   - $8 - \theta((x_a, -y_a))$, se y < 0

   Em código Python:

```
def square_pseudo_angle(v):
    if v[1] > 0:
        if v[0] > 0:
            a = v[1]/v[0]
            if a >= 1:
                return v[1]+1-(1/a)
            else:
                return a
        else:
            return 4 - square_pseudo_angle([-v[0], v[1]])
    else:
        return 8 - square_pseudo_angle([v[0], -v[1]])
```

B. Se θ(a, b) > 0, então a está mais à esquerda (no sentido anti-horário) de b

Questão 2

Outra possível fórmula do pseudo-ângulo é a dada no livro-texto “Introdução à Geometria Computacional”, a qual se dá por:

$$\text{pseudo-ângulo}\ (x, y) = 1 - \frac{x \cdot y}{\|x\| \, \|y\|}.$$

O código Python para essa fórmula (usando a biblioteca numpy), é:

```
def pseudo_angle_dot(a,b):
    return 1 - (np.dot(a,b) / (np.linalg.norm(a)*np.linalg.norm(b)))
```

Em relação ao método anterior de pseudo-ângulo, esse método é mais custoso computacionalmente, o que se reflete nos tempos de execução para ordenar uma lista de 100 vetores por ângulo, usando ambos os métodos:

```
Average time to sort using square pseudo angle: 0.004371700000774581
Average time to sort using cosine pseudo angle: 0.027907699999559554
```

Questão 3

Utilizando certas ferramentas da linguagem de programação Python, como a compreensão de listas e a função “zip”, que recebe dois vetores e retorna um iterador com os pares no mesmo índice de cada vetor, podemos implementar com facilidade as operações básicas:

```
def vec_sum(a,b):
    return np.array([a+b for [a,b] in zip(a,b)])

def vec_subtract(a,b):
    return np.array([a-b for [a,b] in zip(a,b)])

def vec_dot(a,b):
    return sum([a*b for [a,b] in zip(a,b)])
```

Questão 4

Implementando o produto vetorial em duas dimensões de acordo com a definição, temos:

```
# Receives two two-dimensional vectors, a and b, interpreted as three-dimensional vectors
lying on the XY plane.
# Returns the cross product between them
def cross(a,b):
    return a[0]*b[1] - a[1]*b[0]
```

Após implementarmos o produto vetorial, podemos utilizá-lo para implementar outras operações, como a interseção e a área orientada

```
# Receives two line segments, ab and cd, on the XY plane and returns if they intercept or not
def intersect(a,b,c,d):
    ab = np.subtract(b, a)
    ac = np.subtract(c, a)
    ad = np.subtract(d, a)

    cd = np.subtract(d, c)
    ca = np.subtract(a, c)
```

```
    cb = np.subtract(b, c)

    p1 = cross(ab, ac) * cross(ab, ad)
    p2 = cross(cd, ca) * cross(cd, cb)

    return p1 < 0 and p2 < 0

# Receives three points, a, b and c, on the XY plane, and returns the oriented area of their
parallelogram
def oriented_area(a,b,c):
    o = np.array([0,0])

    oa = np.subtract(a,o)
    ob = np.subtract(b,o)
    oc = np.subtract(c,o)

    return 0.5 * (cross(oa, ob) + cross(ob, oc) + cross(oc, oa))
```

Questão 5

Algoritmo do tiro:

```
# Receives a point P and a sequence of points p = [p_0, p_1, ... , p_n, p_(n+1)], that forms a
closed polygon,
# where p_(n+1) == p_1. Returns -1 if outside, 0 if in frontier, 1 if inside
def point_in_polygon_intersection(P, p):
    n = len(p)-1
    N = 0 # Number of intersections
    [x0, y0] = [P[0], P[1]]
    Pn = np.add(P, [1,0]) # We will test the horizontal line that passes by P
    for i in range(0,n):
        xi = p[i,0]
        yi = p[i,1]
        xip1 = p[i+1,0]
        yip1 = p[i+1,1]

        if not math.isclose(yi, yip1): # Is not an horizontal line
            [x, y] = line_intersection(p[i], p[i+1], P, Pn) # Check the intersection between
test line and one line of the poly
            if math.isclose(x, x0): # If the inter point is the same as the test point, itself
lies on the polygon frontier
                return 0
            elif x > x0 and point_in_line([x,y], p[i],p[i+1]):
                N += 1
        elif point_in_line(P, p[i], p[i+1]):
            return 0
    odd = N % 2 == 1
    return 1 if odd else -1
```

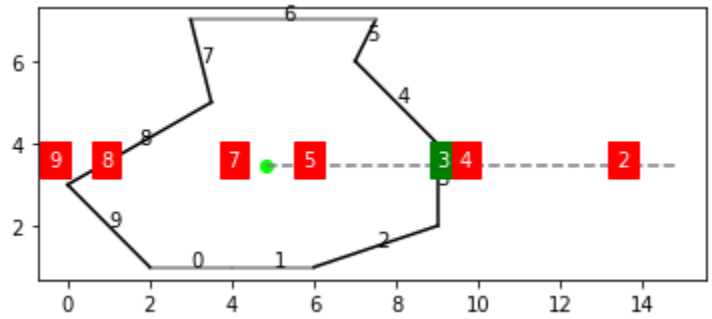
Algoritmo do índice de rotação:

```
def point_in_polygon_rotation(P, p):
    # Compute rotation index
    k = 0
    n = len(p)-1
    for i in range(0,n):
        Ppi = np.subtract(p[i], P)
        Ppip1 = np.subtract(p[i+1], P)
        or_angl = oriented_angle(Ppi, Ppip1)
        k += or_angl
    k *= 1/(2*math.pi)

    # Point is inside polygon if the rotation index is not zero
    return not math.isclose(k, 0, abs_tol=1e-5)
```

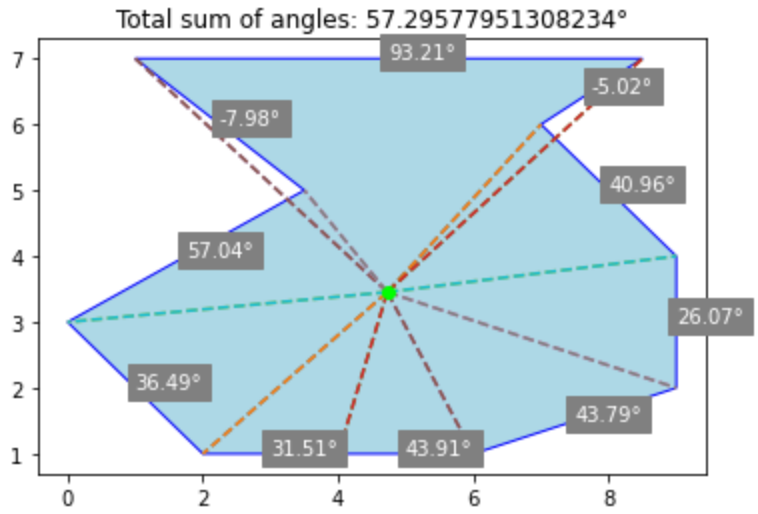
Questão 6

Podemos ilustrar o funcionamento do algoritmo do tiro através da seguinte imagem:



Nesta imagem, cada linha do polígono é numerada com a ordem em que é processada no algoritmo. As linhas horizontais são mostradas em cinza, e as linhas não horizontais, em preto. Para cada linha não horizontal, calculamos a interseção entre esta e a linha horizontal que passa pelo ponto de teste (em cinza tracejado, na imagem). O ponto de interseção, aqui numerado de acordo com a linha correspondente, é avaliado. Se o ponto de interseção estiver na própria linha e tiver a coordenada x maior que a coordenada do ponto de teste, então contamos aquela interseção. Consideramos então que o ponto está dentro do polígono se o número de interseções for ímpar.

Também podemos ilustrar o funcionamento do algoritmo do índice de rotação através desta imagem:



Nela, podemos observar as linhas partindo do ponto até cada vértice do polígono, onde os ângulos orientados entre linhas consecutivas são somados. Caso o valor total seja igual a zero, o ponto está fora do polígono. Caso contrário, está dentro. Nesse ponto, o valor total é diferente de zero, logo fica dentro.