Geometric Algorithms
Assignment 1

Laura Baakman (s1869140)

September 12, 2014

A

The Euclidean distance between the n-dimensional points $\boldsymbol{a}$ and $\boldsymbol{b}$ is defined as:

$$d(\boldsymbol{a}, \boldsymbol{b}) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}. \tag{1}$$

My implementation of this formula is provided in Listing 1.
The length of the polygonal line created by drawing line segments between consecutive points from the list $[\boldsymbol{p_0}, \ldots, \boldsymbol{p_n}]$ is:

$$\sum_{i=0}^{n} d(\boldsymbol{p_i}, \boldsymbol{p_{i+1}}). \tag{2}$$

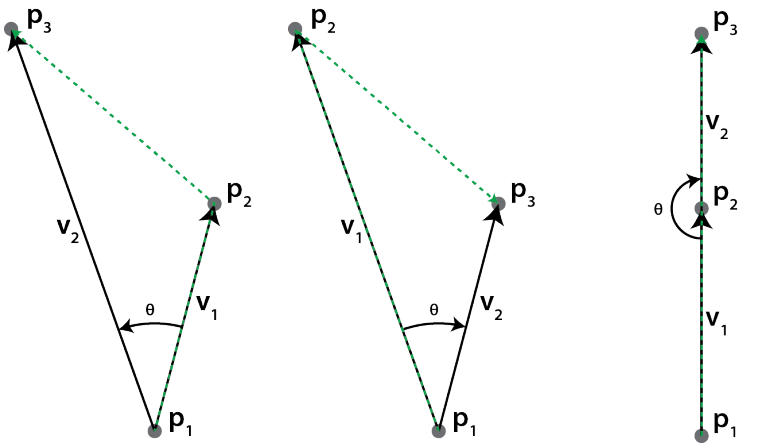
The first step of my implementation computes the list `pairs`, which contains all pairs of points between which the distance should be computed. Actually computing this distance, using the function `euclidean_distance` and summing the results gives the length of the requested line, see Listing 2.

Listing 1: Function that computes the Euclidean distance between two points.

```
def euclidean_distance(a, b):
    """Compute the euclidean distance between the n-dimensional points
    a and b."""
    return(
        sqrt(
            sum(
                [(a_i - b_i)**2 for a_i, b_i in zip(a, b)]
            )
        )
    )
```

Listing 2: Function that computes the length of the polygonal path between consecutive points.

```
def length_of_connecting_path(points):
    """Compute the length of the polygonal line that connects
    consecutive points."""
    pairs = zip(points, points[1:])
    return(
        sum(
            [euclidean_distance(a, b) for (a, b) in pairs]
        )
    )
```



(a) left turn **(b)** right turn **(c)** no turn

Figure 1: A path, the green dashed line, through the points P_1, P_2 and P_3 making (a) a left, (b) a right turn and (c) no turn.

Listing 3: Mathematica code used to compute the value of q.

```
p1 = {p1x, p1y, 0};
p2 = {p2x, p2y, 0};
p3 = {p3x, p3y, 0};

v1 = p2 - p1;
v2 = p3 - p2;

Cross[v1, v2]
```

B

Figure 1 presents the three possible types of paths that form a straight line through the points $\boldsymbol{P_1}$, $\boldsymbol{P_2}$ and $\boldsymbol{P_3}$. Converting these three 2D points to three dimensional ones, by adding a z-coordinate that is zero, allows us to use the cross product to determine the angle θ, using the definition of the cross product:

$$\boldsymbol{v_1} \times \boldsymbol{v_2} = \|\boldsymbol{v_1}\| \cdot \|\boldsymbol{v_2}\| \cdot \boldsymbol{n} \cdot \sin\theta. \tag{3}$$

Where the $\boldsymbol{v_1} = \boldsymbol{P_2} - \boldsymbol{P_1}$, $\boldsymbol{v_2} = \boldsymbol{P_3} - \boldsymbol{P_1}$ and $\boldsymbol{n} = [0, 0, 1]$ represents the normal of the plane that contains the points.

Since all values but the last of the vector $\boldsymbol{n}$ are zero the result of (3) will be of the form $[0, 0, q]$, where q is influenced by $\sin\theta$. The norms of the vectors $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ are greater than or equal to zero by definition. And does not influence the value of q. From this we can conclude that the sign, or the lack thereof of q is completely dependent upon $\sin\theta$. As we are only interested in the direction of the angle, not in its size we can use this to determine the type of path.

If the points are collinear the angle θ is π and which results in $q = 0$. If q is negative the path makes a left turn, if q is positive the path makes a right turn.

Using Mathematica, see Listing 3, we get an expression for the value of q:

$$q = -\boldsymbol{P_{1}}_y\boldsymbol{P_{2}}_x + \boldsymbol{P_{1}}_x\boldsymbol{P_{2}}_y + \boldsymbol{P_{1}}_y\boldsymbol{P_{3}}_x - \boldsymbol{P_{2}}_y\boldsymbol{P_{3}}_x - \boldsymbol{P_{1}}_x\boldsymbol{P_{3}}_y + \boldsymbol{P_{2}}_x\boldsymbol{P_{3}}_y, \tag{4}$$

where $\boldsymbol{P_{1}}_y$ represents the y-coordinate of the point $\boldsymbol{P_1}$.

C

Convex Hull

To find the convex hull of the generated set we use the ConvexHull algorithm given in the exercise. Since the code necessary to compute `L_upper` and `L_lower` is comparative it has been extracted to a method called `half_convex_hull` provided in Listing 4. The function `half_convex_hull` uses the method `make_right)turn` that determines if the path defined by the last three points in the set `L`

Listing 4: Method that computes the upper (or lower) half of the convex hull.

```
def half_convex_hull(L, for_range, points_sorted, make_right_turn):
    """
    Compute the upper or lower part of the convex hull.

    Args:
        L: The initial set to be used for the this half of the convex
    hull.
        for_range: The range of points_sorted to be considered.
        points_sorted: The sorted list of points of which the half
    convex hull is computed.
        make_right_turn: The function that defines whether a path makes
     a right turn.
    """
    for i in for_range:
        L.append(points_sorted[i])
        while (
            len(L) > 2 and
            not make_right_turn(L[-3], L[-2], L[-1])
        ):
            L.pop(-2)
    return L
```

Listing 5: Method that computes the type of a path defined by `p1`, `p2` and `p3`.

```
def make_right_turn(p1, p2, p3):
    """Return true if the line drawn through p1, p2 and p3 makes a
    right turn."""
    q = -(p1[1]*p2[0]) + p1[0]*p2[1] + p1[1]*p3[0] - p2[1]*p3[0] - p1
    [0]*p3[1] + p2[0]*p3[1]
    return (q > 0)
```

makes a right turn. This method, uses the value of q defined in (4), see Listing 5. Both

This method is called twice to compute the convex hull, see Listing 6.

Both the method `half_convex_hull` and `convex_hull` accept an argument called `make_right_turn`, this is done so that these methods can be used with a different method to determine the turn of a path.

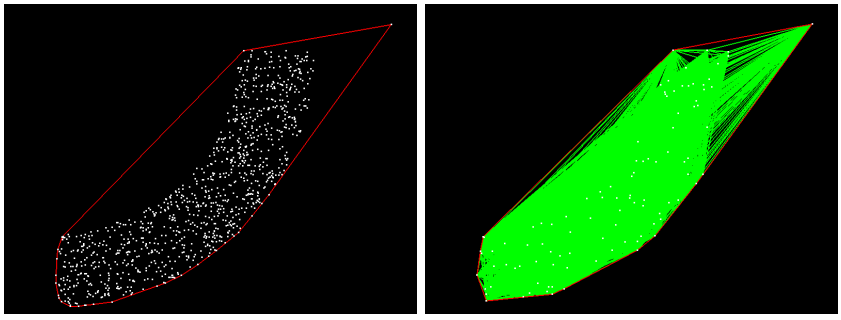
See Figure 2a for a visualization of a convex hull of 1000 points, generated with the provided `generate_points()` method. Figure 2b shows the convex hull of 100 points and all possible straight lines between points in the set in green, showing that the convex hull is indeed convex.

Listing 6: Method that computes the convex hull of the inputted set of points.

```
def convex_hull(cv_points, make_right_turn=make_right_turn):
    """Compute the convex hull of the passed points using the passed
    points."""
    cv_points.sort()

    L_upper = half_convex_hull(
        cv_points[0:2],
        range(2, len(cv_points)),
        cv_points,
        make_right_turn
    )
    L_lower = half_convex_hull(
        cv_points[-2:],
        reversed(range(0, len(cv_points) - 2)),
        cv_points,
        make_right_turn
    )

    return(L_upper + L_lower[1:len(cv_points) - 1])
```



(a) Convex hull of 1000 points. **(b)** Convex hull of 100.

Figure 2: Visualizations of the convex hull, shown in red, of a set of points, with in (b) not only the convex hull but also all possible straight lines between points in the set, shown in green.

Listing 7: Method that computes the area of an irregular polygon.

```
def area_irregular_polygon(points):
    """
    Compute the area  of an irregular polygon.

    The polygon is defined by the passed points, which are the polygons
     vertices.
    """
    points.append(points[0])
    first_sum = sum([x * y for ([x, _], [_, y]) in zip(points, points
    [1:])])
    second_sum = sum([x * y for ([x, _], [_, y]) in zip(points[1:],
    points)])
    return ((first_sum - second_sum) / 2)
```

Area

The area A of a irregular planar non-intersecting polygon with vertices (x_1, y_1), $\ldots, (x_n, y_n)$ is defined as:

$$\begin{aligned} A &= \frac{1}{2} \cdot \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \begin{vmatrix} x_2 & x_3 \\ y_2 & y_3 \end{vmatrix} + \ldots + \begin{vmatrix} x_n & x_1 \\ y_n & y_1 \end{vmatrix} \\ &= \frac{1}{2} \cdot (x_1 \cdot y_2 - x_2 \cdot y_1 + x_2 \cdot y_3 - x_3 \cdot y_2 \\ &\qquad + \ldots + \\ &\qquad x_{n-1} \cdot y_n - x_n \cdot y_{n-1} + x_n \cdot y_1 - x_1 \cdot y_n). \end{aligned} \tag{5}$$

Using this formula, the method to compute the area of a irregular polygon is implemented as in Listing 7.

D

For the purpose of this exercise I have defined a class `Line` that contains all methods pertaining to lines. Listing 8 presents the class definition and its constructors and factory methods. The factory method `randomWithFloat()` generates a random line, with the coefficients `a` and `b` as floats, `randomWithFraction()` does the same but uses `Fraction` objects.

Intersection

The x-coordinate intersection of two lines $f(x) = a_1 \cdot x + b_1$ and $g(x) = a_2 \cdot x + b_2$ can be found by solving $f(x) = g(x)$. Using Mathematica, see Listing 9, we get the solution of this equation:

$$x = \frac{-b_1 + b_2}{a_1 - a_2}. \tag{6}$$

Listing 8: Constructors and factory methods of the class Line.

```
class Line(object):

    """Class to represent a line of the form y = ax + b using two
    coefficients a, b."""

    def __init__(self, a, b):
        """Construct a Line object."""
        super(Line, self).__init__()
        self.a = a
        self.b = b

    @classmethod
    def randomWithFloat(cls, bounds=(-100, 100)):
        """
        Return a Line object, that is initialized with randomly chosen
    floats.

        By default the random chosen floats are between -100 and 100,
    passing a
        tuple (min, max) as the bounds parameter changes these values.
        """
        (minimum, maximum) = bounds
        return cls(
            uniform(minimum, maximum),
            uniform(minimum, maximum)
        )

    @classmethod
    def randomWithFraction(cls, bounds=(-100, 100)):
        """
        Return a Line object, that is initialized with randomly chosen
    fractions.

        By default the random chosen floats are between -100 and 100,
    passing
        a tuple (min, max)as the bounds parameter changes these values.
        """
        (minimum, maximum) = bounds
        return cls(
            Fraction.from_float(uniform(minimum, maximum)),
            Fraction.from_float(uniform(minimum, maximum))
        )
```

Listing 9: Mathematica code used to derive an expression for the x-coordinate of two lines.

```
Solve[a1 * x + b1 == a2 * x + b2, x]
```

Listing 10: The method `intersectionPoint()` of the class `Line`.

```
def intersectionPoint(self, l):
    """Determine at which point, if any, this line intersects line
l."""
    try:
        x = (-self.b + l.b) / (self.a - l.a)
        y = self.computeYCoordinate(x)
        return (x, y)
    except ZeroDivisionError:
        return None
```

Paying attention to the case where the lines are parallel, i.e $a_1 - a_2 = 0$, we use this equation to implement a method in the class `Line`, see Listing 10.

Point on Line

A point $\boldsymbol{P} = (q, z)$ lies on a line $y = ax + b$ if plugging $\boldsymbol{P}$ into the equation of the line returns a true statement, i.e. if the following holds:

$$a \cdot q + b - z = 0. \tag{7}$$

To test if a point is on a line I have implemented the method `pointOnLine()`, which returns `True` if the passed point is on the line, see Listing 11.

Test

Testing is done using the methods `testLinePair()` and `test()`. The first finds the intersection of a pair of lines. If the lines intersection, i.e. are not parallel, it is tested if the intersection point coincides with both of the lines, see Listing 12.

The method `test()` generates `N` lines and calls `testLinePair` for all unique line pairs with distinct lines. The code of this method is provided in Listing 13. The `Counter` object stores the values of `results` as keys and their counts as values in a dictionary.

Listing 11: The method `pointOnLine()` of the class `Line`.

```
def pointOnLine(self, (x, y)):
    """Determine if the point p = (x,y) lies on this line."""
    result = self.a * x + self.b - y
    return not result
```

Listing 12: The method `testLinePair()`.

```
def testLinePair(line1, line2):
    """
    Test if the intersection of line1 and line2 lies on both lines.

    Returns true if the intersectionPoint lies on both lines, false
    if it does not and None if there was no intersection.
    """
    intersection = line1.intersectionPoint(line2)
    if intersection:
        return (
            line1.pointOnLine(intersection) and
            line2.pointOnLine(intersection)
        )
    else:
        return None
```

Listing 13: The method `test()`.

```
def test(generationMethod, N=100):
    """
    Run the test that is part of this assignment.

    Args:
        generationMethod: The factory method of Line to be used.
        N: The number of lines to be generated, defaults to 100
    """
    lines = [generationMethod() for x in range(N)]

    results = []
    for idx, line1 in enumerate(lines):
        for line2 in lines[idx + 1:]:
            results.append(testLinePair(line1, line2))
    return Counter(results)
```

Set	A	B	C	D
N	25	17	11	5

Table 1: The results of assignment C with $\varepsilon = 0.05$, and N is the number of points on the convex hull.

Listing 14: Method that computes the convex hull of a globally defined set points, using the `convex_hull` method from assignment C. See Listing 6 for the implementation of that method.

```
def ConvexHull():
    global ch
    ch = convex_hull(points, rightTurnFiltered)
```

The test is executed for both fractions and floats. The results show that when fractions are used all intersection points lie on both of the lines, whereas with floats the 1980 of the 2970 did not coincide with their lines. No parallel were generated in either case.

E

Listing 15 presents the implementation of the method `rightTurnFiltered` which determines what, if any, type of turn a path makes. In principle this method is the same as the `make_right_turn()` method used in assignment C.

However when the vector $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ or `angle` is very small, q is recomputed with `Fraction`s. A value is considered very small when it is smaller than some epsilon that is defined globally.

To compute the convex hull we have reused the method `convex_hull()` that was defined for assignment C, see Listing 14.

The results of using the improved turn method are shown in Table 1.

Listing 15: Method that computes the type of a path defined by `p1`, `p2` and `p3`.

```
def rightTurnFiltered(a, b, c):
    """Return true if if the line drawn through a, b, and c makes a
    right turn."""
    def isSmall(number):
        """Function to check if a number is very small."""
        return abs(number) < epsilon

    def floatVectortoFractionVector(vector):
        return [Fraction.from_float(x) for x in vector]

    def crossProduct(p1, p2, p3):
        """Compute the crossProduct of the vectors p2 - p1 and p3 - p1.
    """
        return (
            -(p1[1]*p2[0]) + p1[0]*p2[1] +
            p1[1]*p3[0] - p2[1]*p3[0] -
            p1[0]*p3[1] + p2[0]*p3[1]
        )

    v1Norm = euclidean_distance(a, b)
    v2Norm = euclidean_distance(a, c)
    q = crossProduct(a, b, c)
    angle = q / (v1Norm * v2Norm)

    if (
        isSmall(v1Norm) or
        isSmall(v2Norm) or
        isSmall(angle)
    ):
        q = crossProduct(
            floatVectortoFractionVector(a),
            floatVectortoFractionVector(b),
            floatVectortoFractionVector(c)
        )
    return (q > 0)
```