Элементы вычислительной геометрии

- Выбор типа данных для представления точек (плоскости).
- Задача поиска точки пересечения (если она есть) двух заданных отрезков на плоскости.
- Задача вычисления угла между двумя плоскими прямыми.
- Задача определения, лежат ли две заданные точки плоскости по одну сторону от заданной границы области; в частности - по одну сторону от заданной прямой.
- Задача определения, лежит ли заданная точка плоскости внутри заданного выпуклого многоугольника.
- Задача определения, является ли заданный плоский многоугольник выпуклым.
- Задача построения выпуклой оболочки заданного множества точек плоскости.
- Задача вычисления ориентированной площади плоского многоугольника (любого).

Выбор типа данных для представления точек плоскости

В языке Jula имеется несколько возможностей определить тип данных для представления точек плоскости.

- Точки можно представлять двухэлементными векторами `Vector{<:Real}`, например:

```
P=[0,0]
```

- Точку можно задать как двухэлементный кортеж `Tuple{T,T} where T<:Real`, например:

```
P=(0,0)
```

- Точку можно задать как структуру с двумя однотипными числовыми полями:

```
struct Vector_2D{T<:Real}
    x::T
    y::T
end
```

наример:

```
P=Vector_2D(0,0)
```

- Точку можно задать как именованный кортеж `NamedTuple{(:x, :y), Tuple{T,T}} where T<:Real`. Приэтом для этого типа удобно будет определить псевдоним:

```
Vector2D{T<:Real}=NamedTuple{(:x, :y), Tuple{T,T}}
```

Задать значение типа Vector2D можно, наример, так:

```
P = (x=0,y=0)
```

или так:

```
P = Vector2D{Int}((0,0))
```

Последнее означает также, что кортеж из двух однотипных вещественных аргументов может быть легко преобразован к типу Vector2D.

Замечание. Точку можно задать как значение одного из параметрических типов, определенных пакете StaticArrays.jl, а именно - как значение типа

`SVector`, или одного из типов, производных от параметрического типа `FieldVector`.

Однако оставим этот способ за скобками, поскольку изучение этого пакета отвлекло бы нас от главной цели. Отметим лишь, что так называемые статические массивы, определяемые в этом пакете, т.е. массивы, размеры которых фиксированы и не могут быть изменены, в некоторых случаях могут быть более эффективны по сравнению с динамическими массивами, но это относится только к массивам небольших размеров.

Мы остановимся на способе представления точки плоскости с помощью именованных кортежей (`NamedTuple`).

Отметим, что тип `NamedTuple` представляет собой просто некоторую "обёртку" вокруг типа `Tuple` с соответствующим (фиксированным) числом элементов.

Обращаться к элементам именованного кортежа можно как с помощью индексов, например, `A[1], A[2]`, или с помощью их имен, например, `A.x, A.y`.

Координаты точек плоскости - суть координаты соответствующих радиус-векторов. Поэтому не будем делать различий между точками и их радиус-векторами. В соответствии с этим определим необходимые операции для типа `Vector2D`, представляющего радиус-вектора точек плоскости.

```
using LinearAlgebra

Vector2D{T<:Real} = NamedTuple{(:x, :y), Tuple{T,T}}

Base. +(a::Vector2D{T},b::Vector2D{T}) where T =
    Vector2D{T}(Tuple(a) .+ Tuple(b))

Base. -(a::Vector2D{T}, b::Vector2D{T}) where T =
    Vector2D{T}(Tuple(a) .- Tuple(b))

Base. *(α::T, a::Vector2D{T}) where T = Vector2D{T}(α.*Tuple(a))
```

```
LinearAlgebra.norm(a::Vector2D) = norm(Tuple(a))
# norm(a) - длина вектора, эта функция опредедена в LinearAlgebra

LinearAlgebra.dot(a::Vector2D{T}, b::Vector2D{T}) where T =
    dot(Tuple(a), Tuple(b))
# dot(a,b)=|a||b|cos(a,b) - скалярное произведение, эта функция
определена в LinearAlgebra

Base. cos(a::Vector2D{T}, b::Vector2D{T}) where T =
    dot(a,b)/norm(a)/norm(b)

xdot(a::Vector2D{T}, b::Vector2D{T}) where T = a.x*b.y-a.y*b.x
# xdot(a,b)=|a||b|sin(a,b) - косое произведение

Base.sin(a::Vector2D{T}, b::Vector2D{T}) where T =
    xdot(a,b)/norm(a)/norm(b)

Base.angle(a::Vector2D{T}, b::Vector2D{T}) where T =
    atan(sin(a,b),cos(a,b))

Base.sign(a::Vector2D{T}, b::Vector2D{T}) where T = sign(sin(a,b))
```

Задача определения угла между прямыми (между направляющими векторами)

Для решения данной задачи нами была определена (переопределена) функцию `angle`, которая вычисляет значение угла между векторами в диапазоне значений $[-\pi;\pi)$ радиан, по заданным значениям синуса и косинуса этого угла. В общем случае заначения только синуса или только косинуса не достаточно для однозначного определения угла.

Так, если искомый угол заведомо находится в диапазоне от `0` до `180` градусов, то он однозначно определен своим косинусом.

Если же искомый угол заведомо по абсолютной величине не превосходит `90` градусов, то он однозначно определен своим синусом.

В общем же случае, когда угол лежит в интервале [-pi; pi) для его определения необходимо вычислять и косинус и синус этого угла.

Для вычисления угла по значениям его синуса и косинуса нами использована встроенная функция `atan` (арктангенс).

Вообще, имеются два варианта атой функции:

```
atan(y/x)
atan(y,x)
```

В обоих случаях этой функцией возвращается полярный угол (`x`, `y` - координаты полярного вектора), однако в первом случае этот угол принадлежит интервалу $[-\pi/2; \pi/2)$, а во втором - $[-\pi; \pi)$.

При этом значения косинуса и синуса угла основываются на функциях, вычисляющих скалярное произведение векторов и так называемое косое произведение векторов, соответственно.

Скалярное и косое произведение векторов

Для вычисления скалярного произведения векторов нами была определена функция

```
LinearAlgebra.dot(A::Vector2D{T}, B::Vector2D{T}) where T =
dot(Tuple(A), Tuple(B))
```

В данном случае эта функция возвращает сумму попарных произведений соответствующих координат двух заданных векторов:

```
A.x*B.x + A.y*B.y
```

Косое произведение связано с векторным произведением векторов. От векторного произведения двух векторов оно отличается тем, что

результатом будет не сам вектор (ортогональный плоскости, определяемой этими двумя векторами), а лишь коэффициент при орте этого вектора.

Рассмотрим выражение векторного произведения через координаты векторов, лежащих в плоскости `OXY`

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & 0 \\ b_x & b_y & 0 \end{vmatrix} = (a_x \cdot b_y - b_x \cdot a_y) \cdot \vec{k}$$

Откуда получаем выражение искомого косого произведения:

$$a_x \cdot b_y - b_x \cdot a_y$$

Так нами была определена соответствующая функция

```
xdot(A::Vector2D{T}, B::Vector2D{T}) where T = A.x*B.y-A.y*B.x
```

Определение знака угла

Часто бывает, что не нужно определять само значение угла, достаточно бывает лишь определить знак угла, или иметь возможность сравнить два угла по величине. В последнем случае можно использовать непосредственно значения косиниусов или сунусов сравниваемых углов, если только значения рассматриваемых углов не выходят за пределы участков монотонности этих функций.

Определенная (переопределенная) нами функция `sign` позволяет вычислять знак угла между заданными векторами (не вычисляя сам угол), т.е. определять, принадлежит ли угол интервалу $(0, \pi)$, или - интервалу $(0, -\pi)$, при этом при равенстве угла 0 или π, функция `sign` возвращает значение 0.

Задача поиска точки пересечения двух отрезков на плоскости

Будем считать, что отрезки задаются координатами своих концов.

Точки плоскости будем представлять именованными кортежами типа `Vector2D`, определенного нами ранее.

Уравнение прямой, проходящей через точки A, B может быть записано в виде:

$$\frac{x - A.x}{B.x - A.x} = \frac{y - A.y}{B.y - A.y}$$

или в более универсальном виде:

$$(x - A.x)(B.y - A.y) = (y - A.y)(B.x - A.x)$$

(теперь уже не надо будет заботиться о неравенстве нулю знаменателей).

Раскрыв здесь скобки, и приведя подобные члены, перейдем к канонической записи уравнения прямой, в котором все коэффициенты выражены через координаты заданных точек A и B:

$$(B.y - A.y)x + (A.x - B.x)y = A.y(A.x - B.x) + A.x(B.y - A.y)$$

Чтобы проверить, что отрезки пересекаются и найти их точку пересечения, потребуется

- найти точку пересечения (если она есть) прямых, содержащих данные отрезки;
- проверить принадлежность найденной точки пересечения обоим отрезкам.

Пусть $[A_1, B_1]$ и $[A_2, B_2]$ - два рассматриваемых отрезка, концы которых представлены типом `Vector2D`.

Тогда для нахождения точки пересечения содержащих их прямых требуется решить (если решение существует) следующую систему двух линейных уравнений:

$$(B_1.y - A_1.y)x + (A_1.x - B_1.x)y = A_1.y(A_1.x - B_1.x) + A_1.x(B_1.y - A_1.y)$$

$$(B_2.y - A_2.y)x + (A_2.x - B_2.x)y = A_2.y(A_2.x - B_2.x) + A_2.x(B_2.y - A_2.y)$$

Для представления отрезка введем новый тип данных

```
Segment2D{T<:Real} = NamedTuple{(:A, :B), NTuple{2,Vector2D{T}}}
```

Задать значение типа `Segment2D` можно так:

```
s = (A=(x=0,y=0), B=(x=1,y=2))
```

или так

```
s = Segment2D{Int}((Vector2D{Int}((0,0)), Vector2D{Int}((1,2))))
```

Последнее означает также, что кортеж типа `NTuple{2,Vector2D}` может быть легко преобразован к типу `Segment2D`.

Для типа `Segment2D` определим функцию двух аргументов, возвращающую точку пересечения заданных отрезков, или значение `noting`, если пересечения нет.

```
function intersect(s1::Segment2D{T},s2::Segment2D{T}) where T
    A = [s1.B[2]-s1.A[2]  s1.A[1]-s1.B[1]
         s2.B[2]-s2.A[2]  s2.A[1]-s2.B[1]]

    b = [s1.A[2]*(s1.A[1]-s1.B[1]) + s1.A[1]*(s1.B[2]-s1.A[2])
         s2.A[2]*(s2.A[1]-s2.B[1]) + s2.A[1]*(s2.B[2]-s2.A[2]]
    #try
    x,y = A\b # если A - вырождена, то выбрасывается исключение
    #catch
        #return nothing
    #end

    if isinner((;x, y), s1)==false || isinner((;x, y), s2)==false
        return nothing
    end

    return (;x, y) #Vector2D{T}((x,y))
```

```
end

isinner(P::Vector2D, s::Segment2D) =
    (s.A.x <= P.x <= s.B.x || s.A.x >= P.x >= s.B.x)  &&
    (s.A.y <= P.y <= s.B.y || s.A.y >= P.y >= s.B.y)
```

Замечание 1.
При написании этой функции мы исходили из того, что уравнение вида `A*z=b` имеет единственное решение (т.е. что матрица `A` - не вырожденная). Но, как мы знаем, если матрица вырожденная, то такое уравнение может или не иметь вообще решений, или иметь их бесконечно много. Таким образом, если матрица окажется вырожденной, то при выполнении операции `A\b` в нашей функции произойдет ошибка.

Мы сознательно пошли на это, чтобы не усложнять алгоритм. При этом, если, например, координаты точек брать случайными, то события, когда наша матрица окажется вырожденной будут исчезающе редкими, что не трудно понять из общих вероятностных соображений (либо проверить экспериментально).

Однако при желании мы всегда будем иметь возможность перехватить возможную исключительную ситуацию в функции intersect, поместив её вызов в try-блок конструкции try-catch-end (см. документацию Julia).

Задача определения, лежат ли две заданные точки плоскости по одну сторону от границы заданной области

Случай, когда граница граница представляет собой некоторую прямую, разделяющую плоскость на две полуплоскости

Пусть $s = [A, B]$, некоторый сегмент рассматриваемой прямой, и P, Q - пара точек, про которые требуется ответить, лежат ли они по одну сторону от прямой, или нет.

Определим направляющий вектор прямой $\vec{l} = \overrightarrow{AB}$. Тогда, точки P, Q лежат по одну сторону от прямой тогда и только тогда, когда угол между векторами $\vec{l}$, $\overrightarrow{AP}$ и угол между векторами $\vec{l}$, $\overrightarrow{AQ}$ имеют один и тот же знак (отложены в одну и ту же сторону от прямой).

Функция, возвращающая значение `true`, если точки лежат по одну сторону от заданной прямой может быть определена так:

```
function is_one(P::Vector2D{T}, Q::Vector2D{T}, s::Sedment2D{T}
) where T
    l = s.B-s.A
    return sin(l, P-s.A)*sin(l,Q-s.A)>0
end
```

Более общий способ

Ту же задачу можно решить и другим, более общим способом. Для этого должно быть задано уравнение границы некоторой области.

Пусть граница определена своим неявным уравнением

$$F(x, y) = 0$$

где $F(x, y)$ - некоторая непрерывная функция.

В частности это уравнение может представлять собой и уравнение прямой, проходящей через две точки A, B:

$$(x - A.x)(B.y - A.y) - (y - A.y)(B.x - A.x) = 0$$

Тогда неравенство

$$F(x, y) < 0$$

разбивает все точки плоскости на 2 области: для двух точек P, Q лежащих в одной и той же области данное неравенство выполняется или не выполняется одновременно.

Тогда, если определить:

```
is_one_area(F::Function, P::Vector2D{T}, Q::Vector2D{T}) where T =
    (F(P...)*F(Q...)>0)
```

Здесь предполагается, что функция F - есть функция двух вещественных переменных, причем уравнение $F(x,y)=0$, определяет границу рассматриваемой плоской области.

В частности, если речь идет о прямолинейной границе, определяемой некоторым сегментом (s::Segment2D), то функция F будет определена выражением

```
(x-s.A.x)(s.B.y-s.A.y)-(y-s.A.y)(s.B.x-s.A.x)
```

Задача определения, является ли заданный плоский многоугольник выпуклым

Возможен следующий алгоритм, с оценкой сложности $O(n)$, где n - число вершин многоугольника.

Опять будем исходить из дополнительного предположения о том, что вершины перечисляются во входном массиве, в каком-то одном из двух возможных, порядке обхода границы.

Тогда можно последовательно перебирая вершины многоугольника вычислять соответствующие его углы. У выпуклого многоугольника все эти углы будут меньше 180 градусов (или, если будем вычислять внешние углы, то все они - больше 180 градусов). А у не выпуклого многоугольника обязательно найдутся, как углы меньшие, так и большие 180 градусов.

Задача определения, лежит ли заданная точка плоскости внутри заданного многоугольника

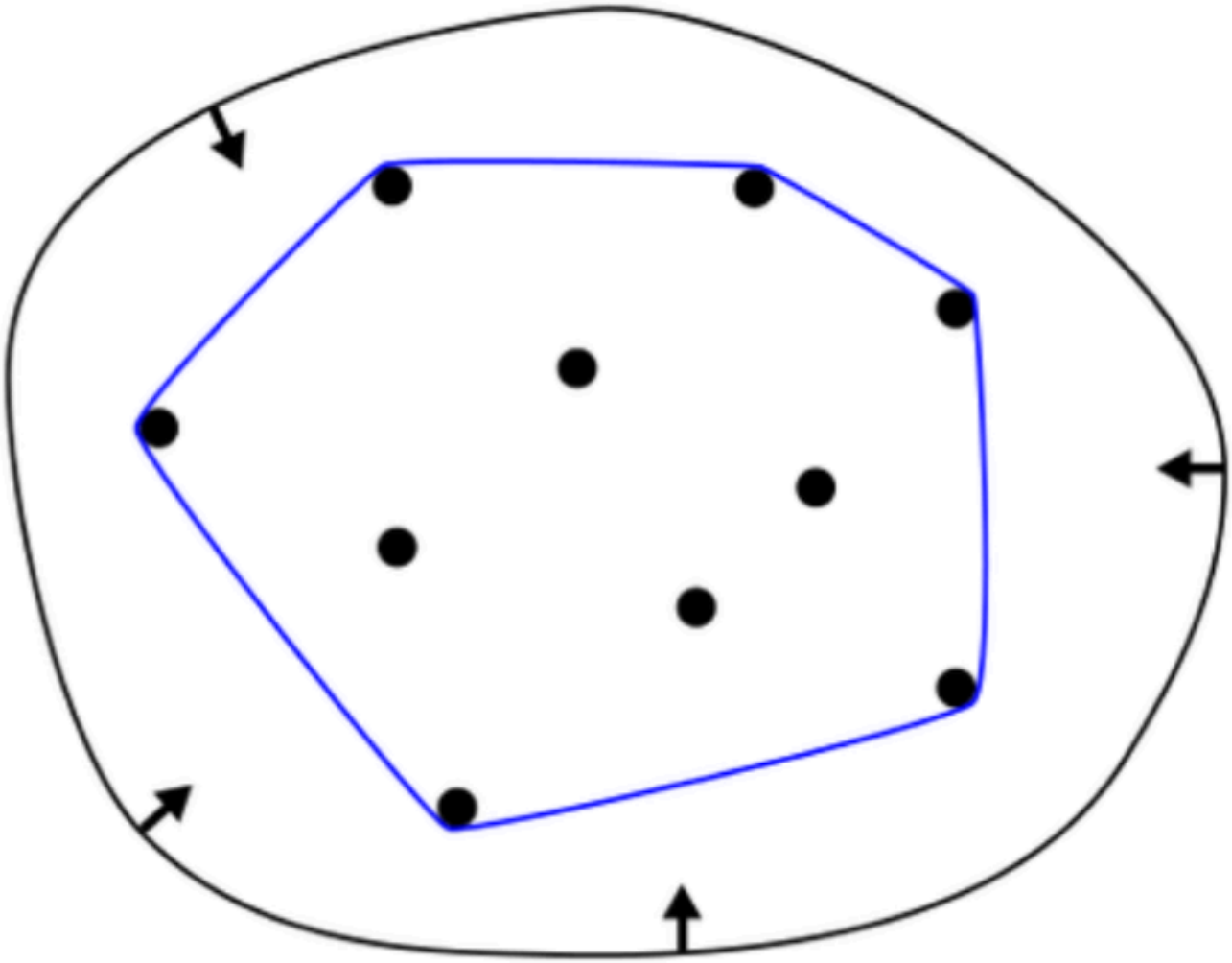
Указание. Вычислить алгебраическую (т.е. с учетом знака) сумму углов, между направлениями из заданной точки на каждые две соседние вершины многоугольника. Далее воспользоваться тем, что, если полученная сумма окажется равной нулю, то точка лежит вне многоугольника, а если она окажется равной 360 градусам, то - внутри.

Замечание. С учетом того, что вычисления будут осуществляться в арифметике с плавающей точкой, может возникнуть проблема сравнения суммы углов с 0 или с 2π. Однако, поскольку эта сумма не может иметь промежуточных значений между указанными величинами, то сравнивать можно с значением π: если сумма окажется окажется меньше, то это будет означать, что она равна 0, а если - больше, то - что она равна $2pi$.

Задача построения выпуклой оболочки заданного множества точек плоскости

Задача ставится так. Дан некоторый массив координат точек плоскости. Требуется определить, какие из этих точек являются вершинами наименьшего выпуклого многоугольника, содержащего все остальные точки. Это наименьший многоугольник и называется **выпуклой оболочкой** заданного набора точек плоскости.

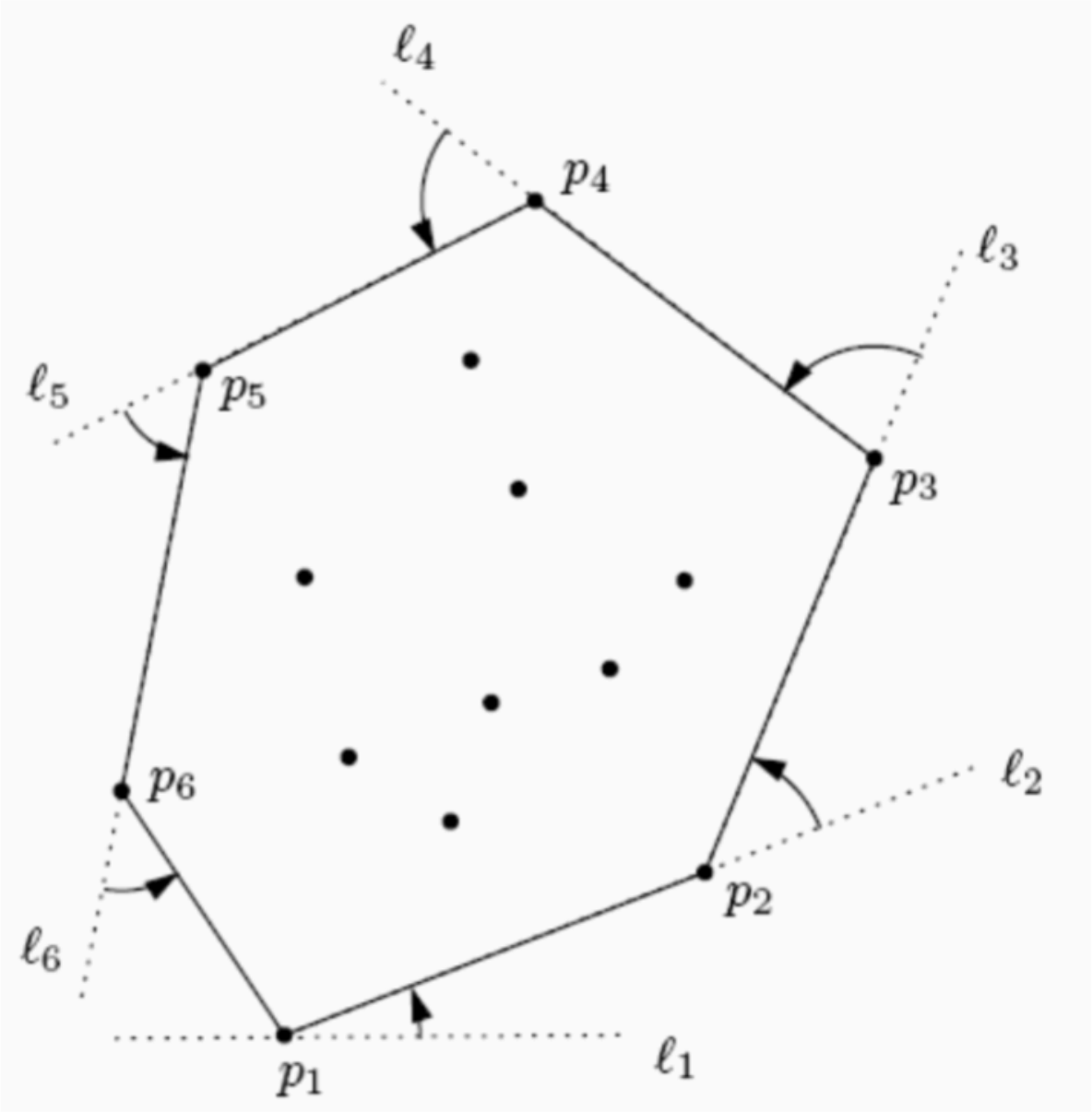
Понятие выпуклой оболочки поясняет следующий рисунок



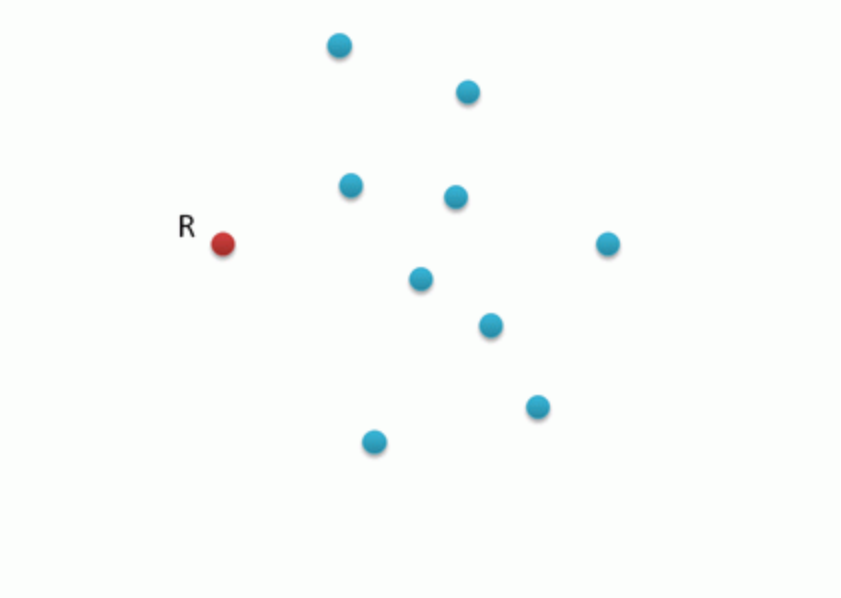
На рисунке показано, что если в точки "забить гвоздики", и поверх всех гвоздиков надеть хозяйственную резинку, то эта резинка примет форму выпуклой оболочки заданных точек.

Рассмотрим несколько популярных алгоритмов построения выпуклой оболочки.

Алгоритм Джарвиса


ℓ_4
p_4
ℓ_3
ℓ_5
p_5
p_3
p_6
ℓ_2
p_2
ℓ_6
ℓ_1
p_1

Визуализацию можно посмотреть



Этапы алгоритма

- В заданном массиве находится самая нижняя точка, обозначим её P_1, и выбирается начальное базовое направление l_1 (на рисунке вектор этого направления направлен по горизонтали вправо). Точка P_1 гарантированно войдет в выпуклую оболочку.
- Следующая точка, обозначим её P_2, которая гарантированно войдет в выпуклую оболочку, будет точка, направление на которую из точки P_1 с базовым направлением $\vec{l}_1$ будет составлять наименьший угол. После выбора этой точки следует выбрать новое базовое направлением $\vec{l}_2 = \overrightarrow{P_1P_2}$.
- Далее каждая $k+1-ая$ точка, добавляемая в выпуклую оболочку, определяется как точка, угол между направлением на которую из k-ой точки, и k-ым базовым направлением $\vec{l}_k$ является минимальным.

Алгоритм завершается когда очередная точка выпуклой оболочки не совпадет с точкой P_1.

Оценка сложности. Сложность алгоритма Джарвиса в наихудшем случае, когда почти все заданные точки войдут в выпуклую оболочку, оценивается как $O(N^2)$, где N - это число всех точек.

Однако часто число точек, вошедших в выпуклую оболочку, много меньше числа всех точек. Если число всех точек выпуклой оболочки можно оценить функцией $O(g(N))$, то то оценка общей сложности алгоритма Джарвиса будет иметь вид $O(N \cdot g(N))$.

Например, если предположить, что N точек более-менее равномерно заполняют некоторую выпуклую область плоскости, то число точек, входящих её в выпуклую оболочку можно оценить как $g(N) = O(\sqrt{N})$.

Алгоритм Грехома

Прежде всего вспомним понятие стека.

В программировании **стеком (stack)** называют абстрактный тип данных, представляющий набор элементов какого-либо типа, организованный по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

При работе со стеком используется протокол, состоящий из действий:

- проверить, является ли стек пустым;
- добавить (положить) в стек новый элемент; об этом говорят еще: поместить элемент на вершину стека;
- снять с вершины стека очередной элемент (возможно только, если стек не пустой);

Конкретная реализация стека может быть разной, например, он может быть реализован на основе динамического массива.

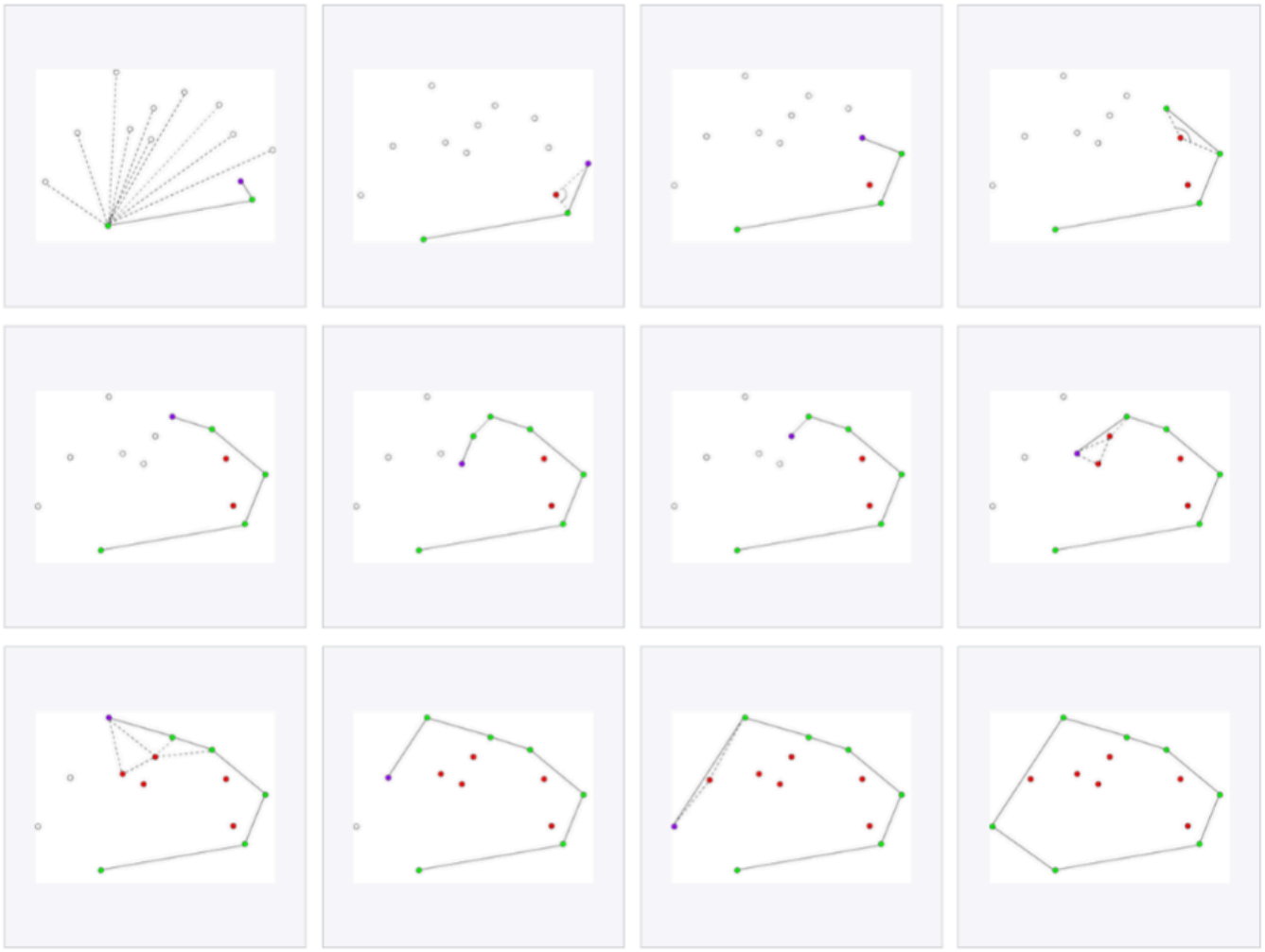
Алгоритм Грехома сводится к следующим этапам.

- Сначала надо найти базовую точку P_0, и выбираеть базовое направление $\vec{l}_0$ (точно так же, как это делалось в алгоритме Джарвиса).
- Все остальные точки сортируются по возрастанию угла между вектором $\vec{l}_0$ и вектором $\overrightarrow{P_0P_k}$ для $k = 1, 2, \ldots, N$.

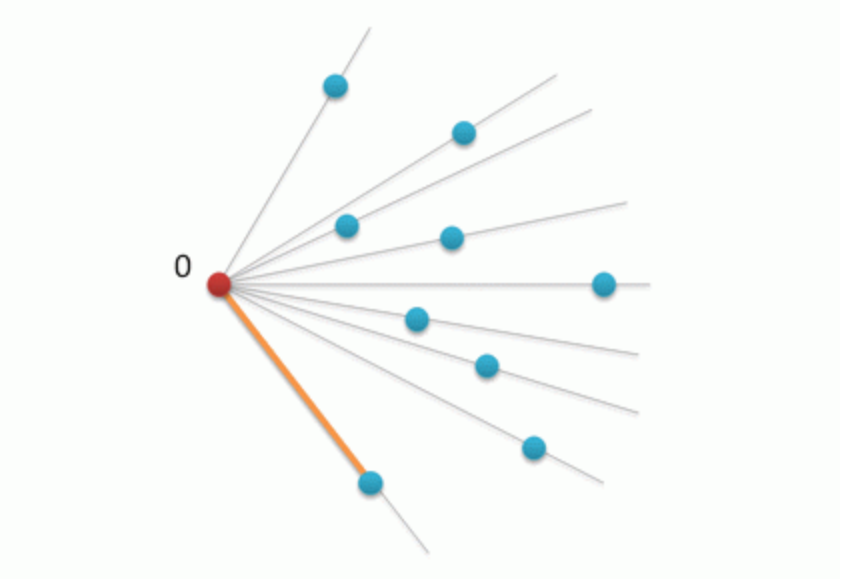
- Далее в выпуклую оболочку помещаются точки $P_1 u P_2$ (они гарантированно в неё входят).
- Каждая следующая в отсортированном порядке точка помещается в выпуклую оболочку, но пока временно. Т.е. эта точка помещается на вершину стека, в который в конце-концов должна быть помещена вся выпуклая оболочка, но на следующих шагах алгоритма некоторые точки с вершины этого стека могут быть сняты.

А именно, пусть в стеке уже имеется k точек. Тогда прежде чем на его вершину будут помещена очередная точка, из него сначала должны быть извлечены все точки, наличие которых делает "текущую оболочку" не выпуклой (под "текущей оболочкой" здесь понимается многоугольник, вершины которого находятся в данный момент в стеке и ещё одна его вершина - это новая точка из отсортированной последовательности точек).

Следующая серия рисунков поясняет логику алгоритма Грехома



Визуализацию можно посмотреть, например,



Оценка сложности. Оценка сложность алгоритма Грехома определяется оценкой сложности алгоритма сортировки, поэтому можно считать, что сложность оценивается как $N \log N$

Указание для программирования. Реализовать стек проще всего на базе динамического массива и использовать функции: isempty, push!, pop!.

Задача вычисления ориентированной площади плоского многоугольника

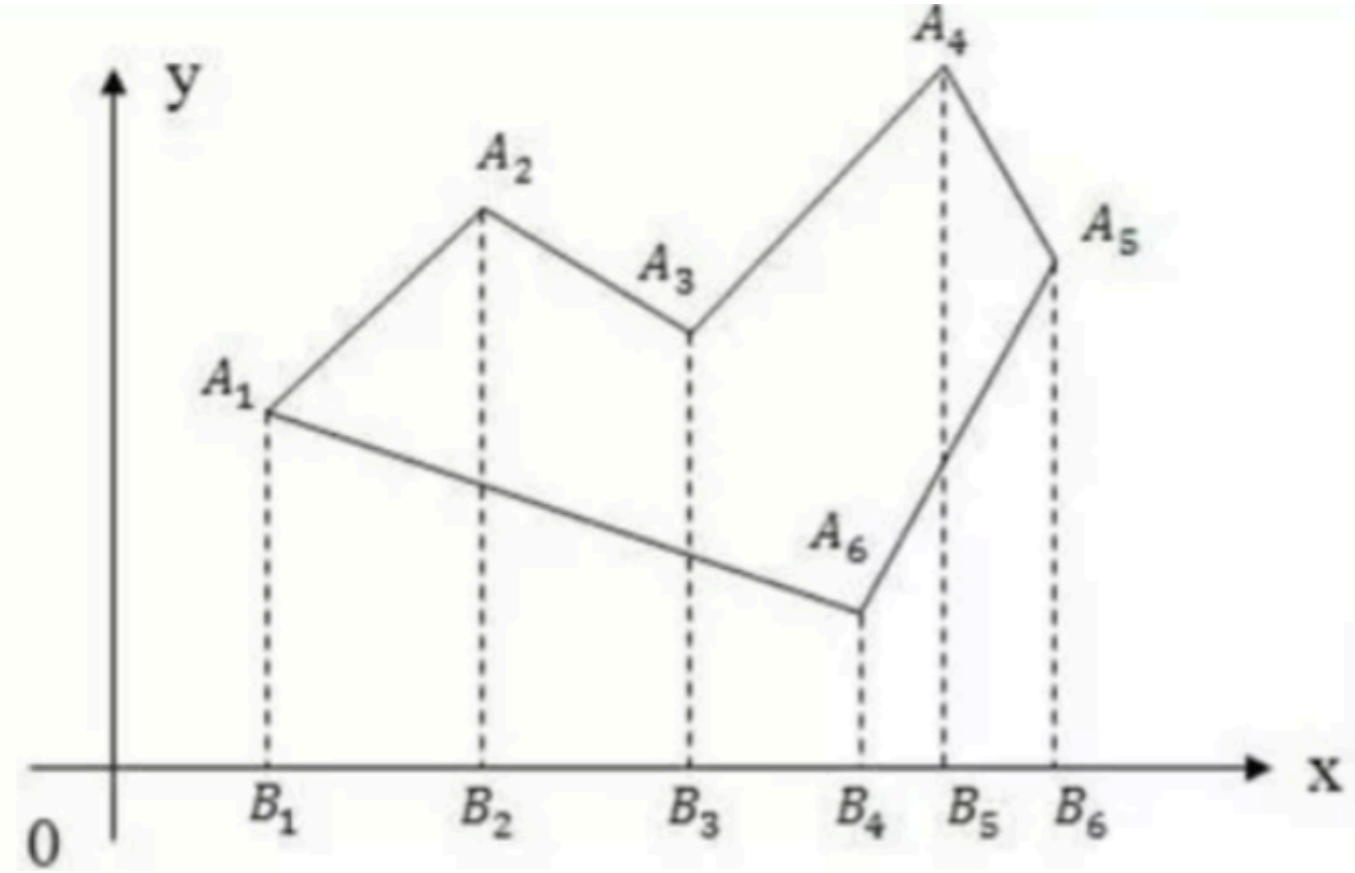
В данном случае речь идет о произвольном многоугольнике, не обязательно выпуклом.

Задача ставится так: дана последовательность точек - вершин произвольного многоугольника (допускаются даже пересечения его сторон), заданная в порядке обхода, например, против часовой стрелке (или по часовой стрелке - это не принципиально). Задание направление обхода означает, что граница многоугольника является ориентированной. Требуется вычислить площадь этого многоугольника. Причем, если ориентация границы положительная (направление обхода против часовой стрелки), то и площадь будет положительной, в противном случае площади приписывается знак минус.

Известно несколько простых методов решения этой задачи.

Метод трапеций

Следующий рисунок поясняет суть этого метода



Двигаясь по границе многоугольника от вершины к вершине вдоль заданной ориентации границы, надо вычислять площади трапеций одна из боковых сторон которых есть некоторое ребро многоугольникам(см. рисунок), причем периметры этих трапеций надо рассматривать как ориентированные границы, ориентация которых задаётся ориентацией самого многоугольника. Просуммировав площади трапеций с нужными знаками, найдём площадь многоугольника. Вычислять площади треугольников удобнее всего с помощью косого произведения, т.е. с помощью написанной нами функции `xdot`.

Пусть, например, $P_k = (x_k, y_k)$ и $P_{k+1} = (x_{k+1}, y_{k+1})$ - две соседние вершины многоугольника. При этом не важно в какой последовательности перечислены вершины многоугольника: по часовой стрелке (как на рисунке) или - против. От этого будет зависеть только знак результата. Если нас

интересует именно площадь, то можно просто брать его абсолютную величину.

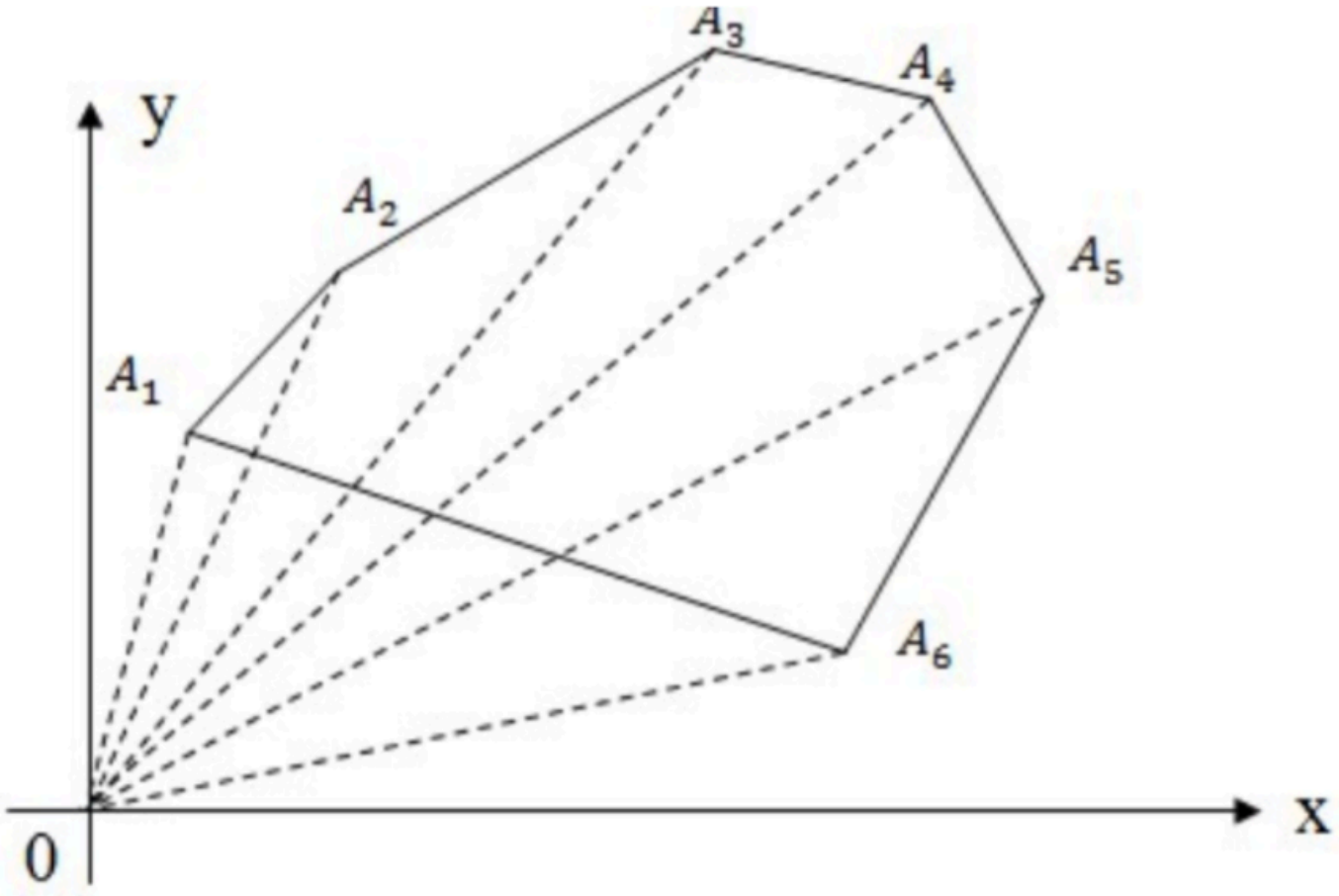
Тогда площадь соответствующей трапеции будет вычисляться по формуле

$$(y_k + y_{k+1})(x_{k+1} - xk)/2$$

(эта площадь будет иметь знак, игнорировать его нельзя).

Метод треугольников

Этот метод аналогичен методу трапеций, только вместо трапеций используются треугольники. Суть этого метода поясняется на следующем рисунке



Пусть, например, $P_0 = (x_0, y_0)$ - это координаты некоторой вершины многоугольника, выбранной (произвольным образом) в качестве базовой, и $P_k = (x_k, y_k)$ и $P_{k+1} = (x_{k+1}, y_{k+1})$ - какие-либо две другие соседние вершины многоугольника. При этом также как и в методе трапеций не важно в какой

последовательности перечислены вершины многоугольника: по часовой стрелке (как на рисунке) или - против. От этого также будет зависеть только знак результата.

Тогда площадь соответствующего треугольника может быть вычислена с помощью косого произведения векторов $P_k - P_0$ и $P_{k+1} - P_0$

(эта площадь также будет иметь знак, игнорировать который нельзя).

Замечание. Вид формул для вычисления площади трапеции в методе трапеций и - площади треугольника в методе треугольников не зависит от положения начала координат.

Формула Гаусса

Существует ещё формула Гаусса для площади многоугольника
формула Гаусса

Замечание. Для языка Julia имеется пакет GeometryBasics, реализующий многие функции вычислительной геометрии.