Практика 6

Пункты задания

1. Спроектировать типы `Vector2D` и `Segment2D` с соответсвующими функциями.
2. Написать функцию, проверяющую, лежат ли две заданные точки по одну сторону от заданной прямой (прямая задается некоторым содержащимся в ней отрезком).
3. Написать функцию, проверяющую, лежат ли две заданные точки по одну сторону от заданной кривой (кривая задается уравнением вида $F(x, y) = 0$).
4. Написать функцию, возвращающую точку пересечения (если она существует) двух заданных отрезков.
5. Написать функцию, проверяющую лежит ли заданная точка внутри заданного многоугольника.
6. Написать функцию, проверяющую, является ли заданный многоугольник выпуклым.
7. Написать функцию, реализующую алгоритм Джарвиса построения выпуклой оболочки заданных точек плоскости.
8. Написать функцию, реализующую алгоритм Грехома построения выпуклой оболочки заданных точек плоскости.
9. Написать функцию вычисляющую площадь (ориентированную) заданного многоугольника методом трапеций.
10. Написать функцию вычисляющую площадь (ориентированную) заданного многоугольника методом треугольников.

Все функции протестировать с использованием визуализации с помощью пакета `Plots`. При этом целесообразно использовать

ноутбуки Jupyter (непосредственно в VS Code). Целесообразно также основной программный код по-прежнему сохранять jl-файле, а затем "инклюдить" его в ячеки блокнота. Потом в блокноте формировать тестовые данные (с помощью функции `randn`, например) и строить соответствующие графики, используя для отображения отдельных точек функции `scatter` или `scatter!`, и для построения линий (соединяющих заданные точки) - функции `plot` или `plot!`.

Графический пакет Plots.jl

Для построения графиков в языке Julia имеется несколько специальных пакетов. Основные из них на сегодняшний день это `Makie` и `Plots`. Пакет `Makie` - это очень мощный пакет, но несколько тяжеловесный на текущий момент. Так в Julia 1.8.5 время компиляции после импортирования этого пакета очень большое (на довольно производительном ноутбуке может составлять около одной минуты), однако в только что вышедшей Julia 1.9.0 это время сокращено уже во многие десятки раз.

Здесь мы рассмотрим некоторые возможности пакета Plots, поскольку он является более "лёгким" (т.е. относительно быстро компилируется и загружается), и обладает следующими привлекательными возможностями.

После того как этот пакет будет установлен (`julia>] add Plots`) и импортирован (`using Plots`), имеется возможность выбрать ту или иную графическую библиотеку (`backend`). Для этого надо выполнить одну из следующих функций (из этого пакета): `pyplot()`, `gr()`, `plotly` и др. (или использовать библиотеку, заданную по умолчанию).

Так функция `pyplot()` актуализирует популярную питоновскую библиотеку `matplotlib`, функция `gr()` - другую популярную графическую библиотеку `GR` (используется по умолчанию), функция `plotly()` - одноимённую графическую библиотеку языка `Javascript`. Но не зависимо от того, какой `backend` подключен, пакет `Plots` обеспечивает унифицированный интерфейс к функциям всех перечисленных библиотек, так что на программный код `julia` факт использования той или иной графической библиотеки влияния не оказывает.

В принципе, в Julia имеется ещё возможность непосредственного использования питоновской библиотеки `matplotlib`, с привычным кому-то по опыту программирования в `Python` интерфейсом. Но для этого придётся уже использовать другой пакет - `PyPlot.jl`, документация к пакету имеется в pyplotjl.pdf.

Для построения ломаной линии в пакете `Plots` имеется функции `plot` и `plot!`. Первая из них используется для создания графического объекта (графика) - либо пустого, либо содержащего только одну ломаную линию. Например,

```
p=plot()
# создан пустой графический объект p, к которому можно будет
добавлять другие графики

xdata = 0:9, ydata = rand(10)
p=plot(xdata, ydata)
# p - графический объект, содержащий ломаную линию с 10
узловыми точками
```

Чтобы узнать тип переменной `p`, можно сделать следующее.

```
typeof(p)
Plots.Plot{Plots.GRBackend}
```

Теперь, чтобы к созданному тем или иным способом объекту `p` можно будет добавить еще одну или несколько ломаных линий. Для этого нужно уже будет воспользоваться функцией `plot!`.

Например, следующий код построит целое семейство кривых.

```
p=plot()
for _ in 1:5
    plot(p, 0:9, rand(10))
end
display(p)
```

Замечание 1. Для того, чтобы отобразился сформированный в объекие `p` график, необходимо вызвать функцию `display(p)`.

При выполнении команды типа `plot(x,y)` непосредственно в REPL, при условии что эта команда была последней (или единственной), функция `display` будет вызывана автоматически, получив в качестве аргумента ссылку на обект с графиком, возвращаемую в данном случае функцией `plot`.

Однако если имеется цикл, в котором вызывалась функция `plot`, то графики отображены не будут (автоматически), поскольку значение цикла есть `nothing`. Поэтому в этом слусае необходим явный вызов функции `display`.

Замечание 2. Если в программе иммется только одно окно с графиком, то это окно всегда будет **текшим** окном (графиком). И в этом случае создавать переменную `p` необходимости нет,

программа будет работать с текущим окном, например, прежний код мог бы быть записан еще и следующим образом.

```
plot()
for _ in 1:5
    plot(0:9, rand(10))
end
display()
```

Результат получится в точности тот же.

Аналогичная ситуация может возникнуть при помещении процедуры построения графика в функцию.

Например, если имеется следующая функция

```
function rand_curve(n)
    x = rand(n)
    y = rand(n)
    plot(x,y)
end
```

то в результате её вызова

```
julia> rand_curve(100)
```

график так же будет отображаться автоматически (потому что функция вернула значение, возвращаемое функцией plot).

Но если функцию определить чуть иначе следующим образом

```
function rand_curve(n)
    x = 0:n-1
```

```
    y = rand(n)
    plot(x,y)
    return y
end
```

то её вызов

```
julia> rand_curve(100)
```

уже не приведеь к отображению графика, потому что в этом случае будет отсутствовать ссылка на объект с графиком.

Ситуацию можно было бы поправить, например, так

```
function rand_curve(n)
    x = 0:n-1
    y = rand(n)
    p = plot(x,y)
    return p, y
end
```

Теперь, наша функция уже будет возвращать ссылку на объект с графиком, который можно будет отобразить следующим образом.

```
julia> p,y = rand_curve(100)
julia> display(p)
```

Или можно было бы сделать так

```
function rand_curve(n)
    x = 0:n-1
    y = rand(n)
    plot(x,y)
    display()
```

```
    return y
end
```

Но в общем случае такой вариант представляется не очень хорошим решением. Лучше когда функции только что-либо вычисляют, а соответствующие графики уже можно будет будет построить на самом верхнем уровне, т.е. непосредственно в REPL.

Во всех приведенных выше примерах свойства линий графика устанавливаются автоматически (значениями, принятыми по умолчанию), причем цвета графиков автоматически задаются так, чтобы разные линии различались по цвету.

Для того, чтобы свойства графика (`Series Attributes`): цвет линии (`linecolor`), толщина линии (`linewidth`), стиль линии (`linestyle`), размер узловых точек (`markersize`), их цвет (`markercolor`), их форма (`markershape`) и т.п. можно было задавать требуемым образом, у функций `plot`, `plot!`, `skatter`, `skatter!` предусмотрены соответствующие именованные аргументы.

Так, атрибуты линии графика `linecolor` и `markercolor` задается символьными значениями, такими как `:red` (красный), `:green` (зеленый), `:blue (голубой)`, `:violent (фиолетовый)`, `:yellow` (желтый) `:grey` (серый), `:white` (белый), `:black` (чёрный) и др., вот полный перечень наименований в системе цветов X11. Значение цвета также можно задавать с помощью конструктора цвета `RGB(r, g, b)`, где параметры `r`, `g`, `b` со значеями из отрезка $[0; 1]$ опередеяют интенсивность соответсвующих составляющих цвета.

Атрибут `linestyle` - определяет стиль линии, он задается символьным значением:

- `linestyle = :auto`,

- `linestyle = :solid` - сплошная линия,
- `linestyle = :dash` - пунктир,
- `linestyle = :dot` - точки,
- `linestyle = :dashdot` - штрих-пунктир,
- `linestyle = :dashdotdot` - двойной штрих-пунктир.

Атрибут `markershape` - определяет форму маркера, он задается символьным значением:

`markershape = :circle`, `markershape = :cross`, `markershape = :xcross` и др.

Атрибут `seriestype` - определяет тип графика, он задается символьным значением:

- `seriestype = :line` - строится обычная ломаная,
- `seriestype = :scatter` (строятся только узловые точки графика, не соединенные линиями),
- `seriestype = :shape` (строится многоугольник с заливкой цветом) и др. (предусмотрены ещё многие другие типы графиков).

Например, чтобы построить график многоугольника, залитого цветом, координаты вершин котрого содержатся в двух числовых массивах `xdata`, `ydata`, можно воспользоваться значением `:shape` фтрибута `seriestype`:

```
plot(xdata, ydata; seriestype = :shape, color = RGB(1,0,0)) # RGB(1,0,0) - это тоже самое, что и :red (красный)
```

Также иожно пользоваться конструктором специального типа `Shape`:

```
plot(Shape(xdata, ydata); color = :red)
```

Особенно удобно пользоватся этим конструктором, если координаты вершин многоугольника заданы не двумя отдельными числовыми массивами, а одним массивом кортежей пар вещественных чисел: `verdata::Vector{Tuple{Real, Real}}`:

```
plot(Shape(verdata); color = RGB(0,1,0))
```

Замечание 3. Если имеются числовые массивы `xdata`, `ydata`, то построить из них массив кортежей можно с помощью ыстроенной функции `zip` (возвращающей генератор соответствующих кортежей): `verdata=[p for p in zip(xdata, ydata)]`.

Атрибут `aspect_ratio` (или просто `ratio`) - определяет масштабы на координатных осях, он задается символьными значениями: `:auto` (масштабы выбираются автоматически), `:equal` (масштабы устанавливаются одинаковыми) и др.

Атрибуты `linewidth`, `markersize` - задаются числовыми значениям (нужные величины можно подобрать эксперментально).

Кроме функций `plot` и `plot!` в пакете имеются ещё функции `scatter` и, соответственно, `scatter!`. Вместо двух последних можно использовать также функции `plot` и `plot!`, вызываемые с нужным значением соответсвующего именованного параметра `seriestype = :scatter`, имеющегося у них.

Более подробную информацию о функциях пакета Plots можно найти здесь.