МГТУ им. Н.Э. Баумана

Краткое описание языка программирования Julia и примеры его использования

Белов Глеб Витальевич

Версия от 07.08.2024

Аннотация

Целью настоящей работы является демонстрация возможностей языка программирования Julia для решения научных и технических задач. Представлены краткие сведения о языке, приведены примеры его использования.

Автор надеется, что предлагаемый текст поможет понять философию языка Julia, послужит справочником по его стандартным функциям, позволит читателю ознакомиться с некоторыми универсальными и прикладными библиотеками, такими, как DataFrames, CSV, Plots, LinearAlgebra, LsqFit, Dates, JuMP, NLopt, Optim, Поскольку Julia развивается довольно быстро, в работе приведены ссылки на первичные ресурсы сети Интернет, с помощью которых можно получить наиболее актуальную информацию о тех или иных возможностях этого языка и связанных с ним прикладных библиотеках.

В работе рассмотрены далеко не все возможности Julia, в частности, ничего не говорится о метапрограммировании, параллельных вычислениях, работе с базами данных, разработке Web-приложений. Однако заинтересованный читатель легко может найти эту информацию, используя приведенные список литературы и ссылки на ресурсы сети Интернет.

Автор выражает благодарность И. Лысанову за ценные замечания по тексту работы.

Оглавление

1. Сведения о языке

Введение

Язык программирования Julia разрабатывается с 2009 года в Массачусетском технологическом институте (MIT). Распространяется бесплатно по лицензии MIT.

Официальный сайт проекта: https://julialang.org/.

Все исходные тексты размещены в интернете на GitHub:

https://github.com/JuliaLang/julia.

Ключевые идеи языка изложены его авторами в статьях [1, 2].

Достоинства. Наличие компилятора позволяет создавать программы, быстродействие которых сопоставимо с быстродействием программ, написанных на C, Fortran. Исходный текст общедоступен и распространяется бесплатно. Язык кросс-платформенный. Большая часть Julia написана на Julia. Язык очень гибкий, что облегчает реализацию алгоритмов. Синтаксис Julia похож на синтаксис Matlab и Python, что облегчает перенос программ с одного языка на другой. Обеспечивается поддержка параллельных вычислений. В языке присутствуют широкие возможности метапрограммирования, благодаря чему можно написать программу, которая сгенерирует программу, которая будет выполняться в среде Julia. Язык очень удобен для реализации численных методов с использованием готовых библиотек (линейная алгебра, линейная и нелинейная оптимизация, с ограничениями и без них). Обеспечивается возможность использования прикладных библиотек, созданных для Python. Наконец, язык Julia достаточно прост для изучения.

Недостатки. Язык относительно молодой, поэтому возможны изменения, число прикладных библиотек не так велико, как для Python, учебников немного, на русском языке почти нет вообще. Время компиляции может быть ощутимым. Сторонние прикладные библиотеки не всегда до конца отлажены. Прикладные библиотеки время от времени изменяются, и тексты программ, которые используют эти библиотеки иногда перестают работать, поскольку, например, изменилось название одной из функций. Ситуация с быстродействием программ на языке Julia не столь однозначна. На сайте https://docs.julialang.org/en/v1/manual/performance-tips/index.html приводится довольно большой текст, посвященный тому, как повысить эффективность работы программы. Из текста можно понять, что быстродействие — это возможность, которой нужно еще суметь воспользоваться.

О начале работы с языком Julia см. ссылки

https://docs.julialang.org/en/v1/manual/getting-started/

https://en.wikibooks.org/wiki/Introducing_Julia/Getting_started

Для первичного ознакомления с языком Julia можно рекомендовать сайт

https://techytok.com/from-zero-to-julia/

В качестве справочника удобно использовать материалы сайта

https://en.wikibooks.org/wiki/Introducing_Julia

Наконец, вероятно самая сжатая информация по синтаксису языка Julia приводится здесь

https://juliadocs.github.io/Julia-Cheat-Sheet/

В качестве учебников по языку Julia можно использовать [3-4, 14]. В учебниках по линейной алгебре и оптимизации [5-7] приведены многочисленные примеры решения задач соответствующих предметных областей с использованием этого языка. Учебники по искусственному интеллекту [8-10] также содержат тексты программ на языке Julia. Более полный список литературы см. здесь
https://julialang.org/learning/books/

Небольшой обзор областей применения языка Julia приводится в статье [11].

Диалоговое окно (REPL)

Есть несколько способов работы со средой Julia. Простейший из них предполагает использование черно-белого диалогового окна REPL (Read Evaluate Print Loop). Окно REPL имеет три режима работы: основной, режим установки библиотек (packages) и режим справки.

Для перехода в режим установки библиотек нужно ввести символ "**]**". Соответствующая библиотека устанавливается командой **add** PackageName (вместо PackageName нужно ввести имя соответствующей библиотеки). Чтобы вернуться в основной режим, нужно нажать клавишу **BackSpace**.

Для перехода в режим справки нужно ввести символ вопроса **?**. Далее вводится имя функции, команды или оператора сведения о которых нужно получить. Справка (если она есть по данной теме) выводится на экран после нажатия клавиши **Enter**. Чтобы вернуться в основной режим, нужно нажать клавишу **BackSpace**.

Текст программы можно загружать непосредственно в диалоговое окно, используя функцию копирования. Более удобно загружать программу из файла, который в этом случае должен быть расположен в текущем каталоге.

Получить результат последнего действия - **`ans`**

Завершение работы - **`exit()`** или **`[Ctrl] + [D]`**

Очистить экран - **[Ctrl] + [L]**

Прервать работу программы (процесс вычислений) - **[Ctrl] + [C]**

Загрузить программу из файла — **include("filename.jl")**

Вывести текущий каталог: **pwd()**

Изменить текущий каталог: **cd("newdir")**, например, **cd("d:\\Julia")**

Более подробно о работе с диалоговым окном можно прочитать здесь:
https://en.wikibooks.org/wiki/Introducing_Julia/The_REPL

В качестве продвинутого текстового редактора можно использовать Julia for VSCode
https://www.julia-vscode.org/
Подробности установки и настройки приводятся здесь
https://www.julia-vscode.org/docs/dev/gettingstarted/#Installation-and-Configuration-1

Типы данных

Иерархия типов данных представлена на рисунке 1. Самый верхний (общий) тип (супертип) — `Any`. Все остальные типы — подтипы (субтипы). Тип `Number` является подтипом `Any`. В свою очередь, `Number` является супертипом для типов `Complex` и `Real`. Типы данных бывают абстрактные и конкретные. Переменная может иметь только конкретный тип.

В языке Julia есть несколько целочисленный типов: со знаком `+/-` `Int8`, `Int16`, `Int32`, `Int64`, `Int128` и без знака `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128` (8, 16, …128 — число бит, которое отводится для хранения данных). Наибольшее значение, которое может принять переменная можно получить с помощью функции `typemax()`: `typemax(Int8)`=127, функция `typemin()` выводит значение наименьшего числа данного типа: `typemin(Int8)`=-128. Тип `Int` соответствует `Int64`.

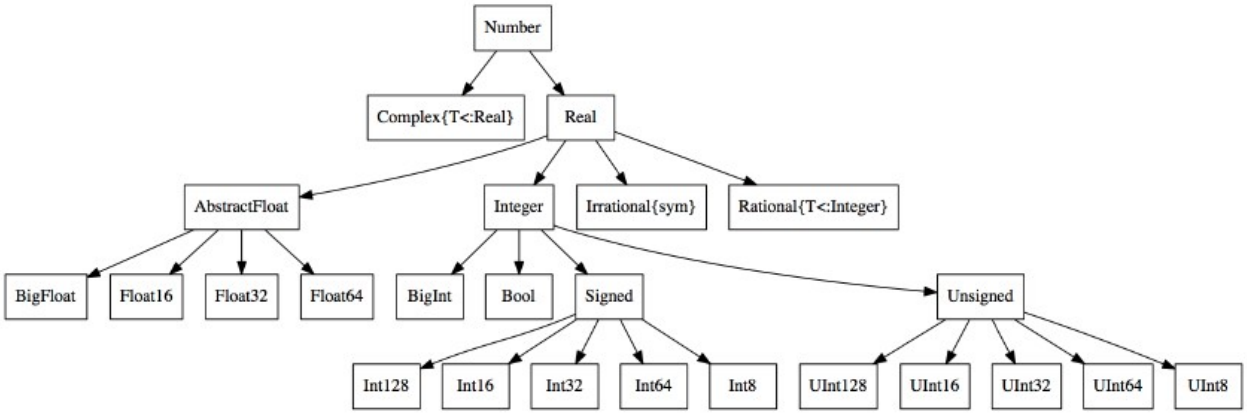

Рис. 1 Иерархия типов данных

Наибольшие и наименьшие значения чисел данного типа можно распечатать в среде Julia с использованием следующего фрагмента текста

```
for T in
[Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
end
```

Вывести результат в файл с именем "typ_max" можно так

```
open("typ_max","w") do f
for T in
[Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
println(f,"$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
end
end
```

Логические переменные `true` и `false` типа `Bool` являются также 8-битными целыми числами, `0` соответствует `false`, `1` — `true`. Отрицание производится оператором `!`: `!true`=>false. `Bool(1)`=> true, `Bool(0)`=>false.

Числа с плавающей точкой соответствуют стандарту IEEE 754 и могут иметь один из типов `Float16`, `Float32`, `Float64`, `BigFloat`. `Float32` соответствует одинарной точности (`eps(Float32)`=>1.1920929f-7), `Float64` – двойной точности (`eps(Float64)`=>2.220446049250313e-16). Переменные типа `Float32` записываются в виде 3.14f0, 5.6f2=>560; типа `Float64` – в виде 3.14e0, 6.7e2=>670.

Операции с плавающей точкой выполняются с машинной точностью (eps). Например,

0.2+0.4=0.6000000000000001, соответственно, результат операции сравнения 0.2+0.4==0.6 =>false

Для вычислений с повышенной точностью можно использовать типы `BigFloat` и `BigInt`. Точность расчетов при этом можно изменять с использованием функции

`setprecision(precision::Int)`, здесь `precision` - число бит, которые отводятся на хранение числа. Например,

```
setprecision(1000)
bigpi=BigFloat(pi)
sin(bigpi/BigFloat(6))-1/BigFloat(2)
i=typemax(Int128) # = 170141183460469231731687303715884105727
i*i=1  # следствие переполнения разрядной сетки
BigInt(i)*BigInt(i)=28948022309329048855892746252171976962977213799489202546401021394546514198529
```

Сложить два числа (0.2 и 0.4) с высокой точностью можно так: `big"0.2"+big"0.4"`.

Задать очень большое или очень маленькое значение, которое нельзя присвоить обычной переменной (например, $1.5*10^{500}$) можно так: `x=big"1.5e500"`, или так: `big(1.5)*big(10)^500`.

Кроме того, можно использовать библиотеку для расчетов с четверной точностью (`Float128`) ***Quadmath***.

В языке Julia реализована интересная возможность — дробные вычисления для рациональных чисел, например, можно рассчитать сумму 1/273+1/91 = 4/273. Запись выглядит так: 1//273+1//91. Иными словами, рациональные числа в Julia записываются в виде дроби вида a//b, где a и b – целые числа. Рациональное число можно преобразовать в число с плавающей точкой с использованием функции `float()`.

Комплексные числа записываются в виде `a+b`*im*, где a и b целые или вещественные числа, *im* — квадратный корень из -1, например, 3+2.15im. Базовые математические функции можно использовать с комплексными числами.

Переменные символьного типа `Char` записываются в виде `'A'`, значения переменных типа `Char` варьируются в пределах от `'\0'` до `'\Uffffffff'`. Значение переменной можно привести к целочисленному типу: Int, в частности `Int('A')`==65, а `Int('α')`== 945. И наоборот, `Char(65)== 'A', Char(945)`==Unicode U+03b1 (3b1 – число 945 в шестнадцатеричной кодировке.

`convert``(T,N)` → преобразовать тип числа N в тип T:

```
convert(Int64,7.0) = 7
```

Тип данных может задаваться явно или определяться во время работы программы. С точки зрения быстродействия, для массивов данных, словарей и составных типов лучше определять тип переменных заранее. Тип переменной указывается с использованием комбинации символов ::, например так

```
x::Integer, y::Float64, x::String.
```

Тип переменной во время работы программы можно узнать при помощи функций `typeof(), eltype()`: `typeof(ans), eltype(ans)`.

В Julia есть возможность определять составные типы, например

```
struct Pixel
     x::Int64
     y::Int64
     color::Int64
end
```

Составной тип можно определить и без указания типов переменных

```
struct Pixel
```

```
    x
    y
    color
end
```

В этом случае переменные будут иметь тип Any, что затруднит работу компилятора и может существенно снизить скорость выполнения расчетов.

С точки зрения быстродействия и универсальности для данного примера оптимальным является такое определение составного типа

```
struct Pixel{T <: AbstractFloat}
    x::T
    y::T
    color::T
end
```

См. также https://en.wikibooks.org/wiki/Introducing_Julia/Types.

В языке Julia используется динамическая типизация переменных. Это означает, что тип переменной определяется в момент присваивания. Например, переменная `a` может принимать любое значение

```
a=1 #целое число
a=1.1 #вещественное число
a=1.2345+10im #комплексное число
a='A' #тип Char
a="abc" #строка
```

Для реализации динамической типизации в Julia не значение присваивается переменной, а переменная привязывается к ячейке памяти с соответствующим значением. Эта концепция называется связыванием (binding). В приведенном примере значения переменной `a` хранятся в различных ячейках памяти.

Задачи

1. Сравнить результаты вычислений

`1/3` и `big(1)/big(3)`

`sin(pi)` и `sin(big(pi))`

выполнить команду

`setprecision(1024)` и сравнить результаты ещё раз.

2. Вывести на печать число π (pi), выполнить команды

`convert(Float32,pi), convert(Float64,pi), convert(Int64,pi).`

3. Сложить две дроби 34/35 и 998/999

4. Выполнить команды `typeof(pi), typeof(1), typeof(1.0),`

`eltype(1//3), eltype(1.0/3.0), typeof(2+3.0im), typeof(2+3im).`

5. Выполнить команды `Char(66), Char(100), Char(1000).`

6. Выполнить команды `eps(Float16), eps(Float(32), eps(Float64).`

7. Выполнить команды `sqrt(-1)` и `sqrt(-1+0im)`.

Общие сведения

Текст в Julia вводится с использованием кодировки UTF-8. UTF-8-это кодировка переменной ширины, в которой символы могут быть представлены разным количеством байтов. Поэтому при выборе текстового редактора важно убедиться в том, что он поддерживает эту кодировку. Разработчики языка Julia рекомендуют использовать в качестве текстового редактора vsCode (https://www.julia-vscode.org/). Поддержка UTF-8 важна в том случае, если либо в строковых литералах, либо в именах переменных, либо в комментариях используются символы не из кодировки ASCII.

Текст программы состоит из одной или нескольких команд, которые представляют собой выражения, записанные на языке программирования. Выполнение команды завершается возвратом результата (или сообщением об ошибке, если выражение записано неверно). Полученный результат можно присвоить переменной. Присваивание осуществляется с использованием знака равенства.

Длинные выражения можно записывать в несколько строк без знаков переноса. В одной строке можно записать несколько операторов, если использовать разделитель ";" (точка с запятой): `a=0; b=3`.

Комментарии

Можно использовать однострочные и многострочные (блочные) комментарии в тексте программы. В первом случае (одна строка) текст после символа # воспринимается компилятором как комментарий. Если комментарий достаточно велик и занимает несколько строк, его можно обозначить так: #= текст комментария =# или так """ текст комментария """. При вводе комментариев также следует использовать редактор с поддержкой UTF-8 и сохранять текст в этой кодировке, если используются не ASCII символы, поскольку компилятор анализирует и текст после символа #, чтобы обнаружить конец строки.

Чувствительность к регистру и использование кодировок символов

Язык Julia чувствителен к регистру, т. е. переменные с именами a и A являются разными, функции с названиями Func и func являются разными.

Julia позволяет использовать символы кодировки Unicode (в формате UTF-8) для обозначения переменных, поэтому допустимы такие выражения

`Джулия="Язык программирования"; синус=sin(α); синус/pi`. Ввести символ греческого алфавита в окне редактирования можно так:
`\delta + TAB`,

т.е. набираем `\delta` и нажимаем клавишу TAB.

Арифметические операции

`+ - * / ^` - базовые арифметические операции (сложение, вычитание, умножение, деление и возведение в степень);

`+. -. *. /. ^.` - поэлементные базовые арифметические операции (для векторов и матриц);

`//` - деление для рациональных чисел, результатом является рациональное число

`-a` - отрицание a;

`a+=1` - эквивалентная запись выражения `a = a+1`,

`a-=1` эквивалентно `a = a-1`,

`a*=b` эквивалентно `a = a*b`,

`a/=2` эквивалентно `a = a/2;`

`a\b` - эквивалент b/a.

Базовые математические функции

`div(a,b)` - a/b, округленное до целого

`cld(a,b)` - деление a/b с округлением до наибольшего целого

`fld(a,b)` - деление a/b с округлением до наименьшего целого

`rem(a,b)` или `a%b`- остаток деления a/b

`mod(a,b)` — остаток деления a/b

`gcd(a,b)` — наибольший положительный общий делитель чисел a,b

`lcm(a,b)` — наименьшее общее кратное чисел a,b

`min(a,b)` — минимальное значение из списка (для произвольного числа чисел в списке)

`max(a,b)` - максимальное значение из списка (для произвольного числа чисел в списке)

`minmax(a,b)` - минимальное и максимальное значения из для двух чисел a, b, результат в форме кортежа

`muladd(a,b,c)`- вычисляет значение a*b+c

Абсолютные значения и корни

`abs(a)` — абсолютное значение числа a

`abs2(a)` — квадрат числа a

`sqrt(a)` — квадратный корень числа a

`isqrt(a)` — целочисленный квадратный корень целого числа a

`cbrt(a)` — кубический корень числа a

Операции возведения в степень и логарифмы

`exp(a)` - экспонента числа a

`exp2(a)` - 2 в степени a
`exp10(a)` - 10 в степени a
`expm1(a)` - экспонента e^a-1 (точно)
`ldexp(a,n)` - a*(2^n) (a должно быть типа Float)
`log(a)` - натуральный логарифм числа a
`log2(a)` — логарифм a по основанию 2
`log10(a)` — десятичный логарифм a
`log(n,a)` - логарифм числа a по основанию n
`log1p(a)` - логарифм 1+a (точно)
Тригонометрические функции
если x в радианах, то `sin(x), cos(x), tan(x), cot(x), asin(x), acos(x), atan(x), acot(x), sec(x),`
если x в градусах, то `sind(x), cosd(x), tand(x), cotd(x), asind(x), acosd(x), atand(x), acotd(x), secd(x).`
`rad2deg(a)` — преобразовать угол a из радиан в градусы,
`deg2rad(a)` — преобразовать угол a из градусов в радианы.
гиперболические функции: `sinh(x),cosh(x),tanh(x),coth(x),`
Гипотенуза
`hypot(a,b)` - гипотенуза a и b
Комбинаторные функции
`factorial(a)` — факториал числа a
`binomial(a,b)` - число сочетаний из a по b (binomial(3,2)==3)

Некоторые полезные функции

`eps()` — точность представления вещественного типа (машинный эпсилон):

```
eps()
eps(Float16)
```

`eval(a)` — вычислить значение выражения a
`real(a)` — вещественная часть числа a
`imag(a)` — мнимая часть числа a
`reim(a)` — возвращает вещественную и мнимую части a (в виде кортежа)
`conj(a)` — комплексное сопряженное число a
`sign(a)` — знак числа a
`round(a)` — округлить до ближайшего числа
`ceil(a)` — округлить до ближайшего большего числа
`floor(a)` — округлить до ближайшего меньшего числа
`trunc(a)` — отбросить дробную часть

`modf(a)` — кортеж, содержащий дробную и целую части числа a

`digits(a)` - массив десятичных цифр, образующих целое число

`isapprox()` - позволяет сравнивать два числа с заданным уровнем погрешности atol:

```
isapprox(1.0, 1.05; atol = 0.1) (true)
isapprox(1.0, 1.1; atol = 0.05) (false)
```

О функции eval()

Остановимся чуть подробнее на функции eval(). Иногда возникает необходимость рассчитать значение функции, хранящейся в базе данных в текстовом виде, например, таком:

```
-156735+284.1689*T-46.34274*T*LN(T)-0.0029287*T**2+563374*T**(-1)
-.02788144E+09*T**(-3)
```

Запишем эту функцию с использованием синтаксиса Julia и переменной g:

```
g=:(-156735+284.1689*T-46.34274*T*log(T)-0.0029287*T^2+563374*T^(-1)
-.02788144E+09*T^(-3))
```

Задать значение параметра T (например, T=500) можно либо в среде REPL, либо непосредственно в тексте для функции eval():

```
g=:(T=500; -156735+284.1689*T-46.34274*T*log(T)-0.0029287*T^2+563374*T^(-
1)-.02788144E+09*T^(-3))
```

Теперь значение функции можно рассчитать путем вызова функции `eval(g)`, результат равен -158257.1837.

Логические операторы

`!` - не

`&&` - и

`||` - или

`==` проверка эквивалентности объектов, `2==2.0` верно

`!==` - не равно?

`===` проверка идентичности объектов, `2===2.0` ложно, поскольку тип величин разный

`>` - больше?

`>=` - больше или равно?

`<` - меньше?

`<=` - меньше или равно?

Битовые операторы

`~` - битовое отрицание not

`&` - битовое and

`|` - битовое or

`xor` - битовое xor

`>>` - оператор побитового сдвига вправо
`<<` - оператор побитового сдвига влево
`>>>` - оператор побитового сдвига без знака

Функции подтверждения

`isa(a,Float64)` - тип числа a Float64?
`isnumeric(a)` — является ли символ числом?
`iseven(a)` — является ли целое число четным?
`isodd(a)` — является ли целое число нечетным?
`ispow2(a)` — является ли целое число степенью 2?
`isfinite(a)` — является ли число конечным?
`isinf(a)` — является ли число бесконечно большим?
`isnan(a)` — является ли число «не числом» NaN?

Сообщения об ошибках

`error``("text")` - эта функция генерирует сообщение об ошибке, выводя заданный текст.

Задачи

Попробуйте освоить перечисленные выше операции и функции и понять их назначение.

Коллекции

В описании языка Julia используется термин «коллекция». В широком смысле коллекция — это объект, предназначенный для хранения и упорядочивания других объектов. Примерами коллекций являются массивы, множества, кортежи, интервалы.

Интервалы (ranges)

Интервал обозначается начальным значением, шагом и конечным значением, синтаксис выглядит следующим образом `(start:step:stop)` или `(start:stop)`, если шаг (step) равен 1. Кроме того, интервал можно задать при помощи функции `range()`, используя начальное значение (start), конечное значение (stop), шаг (step) и число точек (length):

```
range(start, stop, length)
range(start, stop; length, step)
range(start; length, stop, step)
range(;start, length, stop, step)
```

Значение параметра step по умолчанию равно 1.

Примеры

```
1:10
10:-1.5:0
'a':'z'
range(1, length=100)
range(50, stop=100)
range(1, step=5, length=100)
range(1, step=5, stop=100)
range(1, 10, length=11)
range(1, 100, step=5)
range(stop=10, length=5)
range(stop=10, step=1, length=5)
range(start=1, step=1, stop=10)
range(1, 3.5, step=2)
```

Интервал задает коллекцию неявно. Доступ к элементам неявной коллекции можно получить несколькими способами. Например, к ним можно обратиться как к элементам массива:

```
x=1:5
println(x[1], "; ", x[end])
```

Доступ к элементам интервала можно обеспечить также путем организации цикла вида

```
for x in range(1, 10, length=11)
println(x)
end
```

Превратить интервал в явную коллекцию можно с использованием функции `collect()`. Например, результатом вызова функции `collect(1:5)` будет массив `[1,2,3,4,5]`.

Строки

Строка в Julia – это набор символов, заключенных между двойными кавычками `" "` или `""" """`. Символы вводятся в кодировке UTF-8. Строки могут содержать специальные символы, например, символ табуляции `'\t'` или символ перевода на новую строку `'\n'`: `b = "строка 1\nстрока 2\n"; println(b)`.

Строку можно рассматривать как одномерный массив (вектор). Например, если строка `s="abc"`, то `s[2]=='b'`, а `s[end]=='c'`. Однако изменять элементы строки присваиванием нельзя, т. е. оператор `s[3]='d'` является ошибочным с точки зрения языка Julia. В этом случае используется функция `replace()`.

`replace``(st, toSearch => toReplace)` – в строке `st` заменить везде подстроку `toSearch` на подстроку `toReplace`.

`length()` - возвращает число символов в строке, которое в общем случае не совпадает с числом ее элементов (байтов).

sizof() - возвращает число байт в строке (элементов массива).

Поскольку символы в разных кодировках имеют разную длину, число символов строки равно или меньше числа байт. Например, `st = "Köln"`, `length(st)==4, sizof(st)==5`, поскольку символ `'ö'` занимает два байта.

Выделить подстроку из строки можно так: `ss=st[i1:i2]`, где st - строка, ss – подстрока, i1 – индекс начала подстроки в строке, i2 – индекс последнего элемента.

Для выделения подстроки строки `st` можно использовать также функцию

`SubString(st,i,len)`

`i` – начало выделения, `len` – число выделяемых байт.

Учитывая, что символы кодировок могут занимать разное число байт, «байтовые» способы работы со строками не всегда приводит к успеху, например, `st[1] = 'K'`, `st[4] = 'l'`, а `st[3]` вызывает сообщение об ошибке.

Фрагмент

```
for i in 1:sizeof(st) println(st[i]) end
```

выдаст сообщение об ошибке.

А эти фрагменты выведут символы строки `st`.

Вариант 1

```
for letter in st println(letter) end
```

Вариант 2

```
index = firstindex(st)
while index <= sizeof(st)
letter = st[index]
println(letter)
global index = nextind(st, index)
end
```

Еще один нюанс, строка и символ — существенно разные понятия языка Julia, поэтому равенство `"A" == 'A'` является ложным.

Проверить наличие символа s в строке st можно с использованием конструкции `s in st`, которая возвращает true или false, например, результатом `'b' in "abc"` является true.

Проверка того, что ss входит в st осуществляется с использованием функции **occursin**`(ss,st)`. Пример: `occursin("io","function")` → true.

findfirst`(ss,st)` → найти первое вхождение подстроки или символа ss в строке st. Если ss - строка, то результатом является первый и последний индексы подстроки ss в строке st. Если ss – символ, результатом будет индекс, который соответствует но-

меру символа в строке. Если подстрока или символ не найдены, функция возвращает `nothing`.

`findlast``(ss,st)` → найти последнее вхождение подстроки или символа ss в строке st. Если ss - строка, то результатом является первый и последний индексы подстроки ss в строке st. Если ss – символ, результатом будет индекс, который соответствует номеру символа в строке. Если подстрока или символ не найдены, функция возвращает `nothing`.

Пример: `st="comment";`

`findfirst("m",st)` →3:3

`findfirst('m',st)` →3

`findlast("m",st)` →4:4

`findnext``(ss,st,i)` → найти первое, начиная с индекса i, вхождение подстроки ss в строке st, результатом является первый и последний индексы подстроки ss в строке st.

`findnext("l","teller",3)` → 3:3

`findnext("l","teller",4)` → 4:4

Если заменить подстроку на символ, функция вернет либо номер индекса, если символ будет найден, либо `nothing`.

Конкатенация (объединение) производится с использованием символа звездочка `'*'`.

Пример.

`c="Hello,";d="world!";e=c*d` → `"Hello, world!"`.

Изменить на обратный порядок элементов строки st, в которой каждому символу соответствует один байт, можно так

`st[end:-1:1]`

`randstring``(n)` → строка случайных символов длиной n, можно использовать для генерации пароля. Перед использованием выполнить команду `using Random`.

`strip``(st)`, аналог trim в Pascal-Delphi, удаляет пробелы вначале и в конце строки, если строка состоит из пробелов, то **`strip`**`(st)` возвращает пустую строку.

`isempty``(st)` – проверяет, есть ли символы в строке, если нет, возвращает true, иначе — false.

`split``(st,'R')` → разобрать строку на элементы, если в качестве разделителя используется символ R.

Пример: `split(st,',')` - разделителем является запятая.

join(a,'R') → преобразовать элементы массива a в строку, используя в качестве разделителя символ R. join -оператор, обратный split.

Пример. a=[1.0, 2.0, 3.0], разделитель — запятая, join(a,','),

результат "1.0, 2.0, 3.0".

parse(T,st) – превращает строку st в число типа T.

Пример: parse(Float64, st).

Если содержимое строки st неизвестно, можно использовать функцию

tryparse(T,st) – которая также превращает строку st в число типа T, но если st нельзя превратить в число, эта функция возвращает nothing, т.е., если tryparse(T,st)==nothing, st нельзя превратить в число типа T.

Если нужно превратить строку чисел в массив, то вначале выполняется оператор split, а затем parse:

```
a="1.1 2.2"
b=split(a,' ')
c=parse.(Float64,b)
```

Две последних операции можно объединить одним из способов

```
c=map(x -> parse(Float64,x),split(a))
c=parse.(Float64,split(a))
```

Функция map() применяет функцию parse(Float64,x) к списку аргументов split(a).

uppercase(st) - преобразовать строку в верхний регистр,

lowercase(st) - преобразовать строку в нижний регистр,

titlecase(st) – преобразовать в верхний регистр первый символ каждого слова строки.

string(x) – превратить число x в строку

```
string(100)
> "100"
string(100.1)
> "100.1"
```

В Julia используется такое понятие, как *интерполяция*. Смысл его заключается в следующем. Если есть строка st="123.456", или число a=3.14, то их значения можно «внедрять» (интерполировать) в строку с использованием знака доллара $: "строка содержит $st, число a=$a". Подстановка значений произойдет при выводе строки на печать, поэтому вывод на печать строки "1 + 2 = $(1 + 2)" будет таким

```
"1 + 2 = 3"
```

Превратить строку в массив строк-символов можно так

```
ast=split(st,"")
```

Если `st=="qq"`, то после выполнения этой команды `ast` превратится в массив с двумя элементами `"q"` (не `'q'` !). При необходимости элементы массива можно менять (напомним, что элементы строки менять нельзя), например, `ast[2]="a"`. Используя функцию `join`, `ast` можно снова преобразовать в строку.

Превратить строку в массив символов можно так

```
ast=collect(st)
```

В этом случае замена элементов массива выполняется таким образом

```
ast[1]='a'
```

Полезно сравнить размеры массива `ast` для двух приведенных способов преобразования строки в массив с использованием функции `sizeof()`.

Задачи

Попробуйте выполнить следующие упражнения. Постарайтесь понять логику происходящего.

```
1.
st="Test 123"
st1=replace(st, "123" => "999")
st2=replace(st1, "9" => "0")
st3=replace(st, "1" => "111")
2.
st="abc"
length(st)
sizeof(st)
st[1:3]
for letter in st println(letter) end
st="абв"
length(st)
sizeof(st)
st[1:3]
for letter in st println(letter) end
3.
st="123456"
occursin("34",st)
occursin("43",st)
```

```
4.
st="АБВГД"
findfirst("ГД",st)
st="ABCDE"
findfirst("DE",st)
5.
st="121212"
findlast("12",st)
findlast("0",st)
findlast('1',st)
6.
using Random
randstring(10)
randstring(10)
7.
st="  123  "
sizeof(st)
st=strip(st)
sizeof(st)
8.
st="1,2,3,4"
x=split(st,',')
st=join(x,';')
for i in 1:length(x) z=parse(Int32,x[i]); println(z) end
y=parse.(Float64,x)
9.
st="a b c"
x=split(st)
for i in 1:length(x) z=tryparse(Int32,x[i]); println(z) end
10.
a="1.23 4.56"
c=map(x -> parse(Float64,x),split(a))
11.
st="abc"
uppercase(st)
st="абв"
uppercase(st)
```

```
12.
st="ABCYZ"
lowercase(st)
st="АБВЮЯ"
lowercase(st)
13.
string(123.456)
14.
x=0.1
println("sin($x)=$(sin(x))")
15.
st="abc"
x=split(st,"")
sizeof(x)
y=collect(st)
sizeof(y)
```

Массивы

Массив это структура данных. Различают одномерные (вектор), двумерные (матрица), многомерные массивы.

Элементы массива хранятся по столбцам, т. е. массив `a=[1 3 5;2 4 6]` вида

```
1 3 5
2 4 6
```

хранится в форме 1,2,3,4,5,6, проверка

```
for i in eachindex(a) println(a[i]) end.
```

Нумерация элементов массива начинается с 1, поэтому первый элемент массива `a[1] или a[begin].` Доступ к последнему элементу можно получить так: `a[end].`

Одномерный массив хранится в памяти как двумерный с одним столбцом, поэтому доступ к элементам одномерного массива a можно осуществлять так: `a[1,1], a[2,1]...`

Способы объявления массива

В языке Julia существует довольно много способов объявления массива. Рассмотрим некоторые из них.

Простейший способ объявления пустого одномерного массива: `T[]`, где `T` - тип данных: `a=String[]; b=Float64[]; c=Int32[]; a,b,c` - пустые массивы типа, `String, Float64, Int32` соответственно.

`a=Array{Float64,1}(undef,3)` – одномерный массив с тремя элементами типа `Float64`, элементы массива не определены (мусор),

`a=Array{Float64,2}(undef,3,5)` – двумерный массив 3*5 — три строки, пять столбцов типа Float64, элементы массива не определены (мусор),

`a=Array{Float64,n}(undef,i1,i2,…,in)` – *n*-мерный массив $i_1*i_2*...*i_n$ типа `Float64`, элементы массива не определены (мусор),

`a=Vector{Float64}(undef,10)` - одномерный массив с десятью элементами типа `Float64`, элементы массива не определены (мусор),

`a=Matrix{Int64}(undef,2,5)` – двумерный массив 2*5 — две строки, пять столбцов типа `Int64`, элементы массива не определены (мусор).

`a=[1 2 3 4]` – вектор-строка (элементы через пробел). При таком объявлении Julia создает двумерный массив с одной строкой и *n* столбцами, доступ к элементам в этом случае можно реализовать двумя способами:

`1) a[1], a[2], a[3]…`

`2) a[1,1], a[1,2], a[1,3]…`

С этой особенностью можно столкнуться при использовании функции сортировки `sort()`, которая потребует указать номер строки или столбца .

`a=[1, 2, 3, 4]` – вектор-столбец (элементы через запятую или точку с запятой, но при определении можно использовать разделители одного типа),

`a=[1 2; 3 4]` – двумерная матрица

`1 2`

`3 4`

`a=zeros(3)` – одномерный массив с тремя элементами типа `Float64`, с нулевыми значениями элементов,

`a=zeros(3,4)` – двумерный массив 3*4 типа `Float64`, с нулевыми значениями элементов,

`a=ones(3)` – одномерный массив с тремя элементами типа `Float64`, с единичными значениями элементов,

`a=ones(3,4)` – двумерный массив 3*4 типа `Float64`, с единичными значениями элементов,

`a=zeros(Int32,0)` – одномерный массив типа `Int32`, без элементов,

`a=Int.(zeros(3))` – одномерный массив с тремя элементами типа `Int64`, с нулевыми значениями элементов,

`a=convert(Vector{Int}, zeros(3))`– одномерный массив с тремя элементами типа `Int64`, с нулевыми значениями элементов,

`a=convert(Matrix{Int}, zeros(3,4))` – двумерный массив 3*4 типа `Int64`, с нулевыми значениями элементов,

`a=convert(Array{Int,3}, zeros(3,4,5))`– трехмерный массив 3*4*5 типа `Int64`, с нулевыми значениями элементов,

`a=fill(0,10)` – одномерный массив типа `Int64` длины 10, с нулевыми значениями элементов,

`a=fill(0.0,10)` – одномерный массив типа `Float64` длины 10, с нулевыми значениями элементов,

`a=fill(1,3,3)` – двумерный массив 3*3 типа `Int64`, с единичными значениями элементов,

`a=fill(Float64[],0)` - одномерный массив векторов типа `Float64` нулевой длины,

`a=fill("",5)` – одномерный массив, заполненный пятью пустыми строками,

`a=rand(5)` – одномерный массив из 5 элементов, заполненный случайными числами от 0 до 1, распределение равномерное.

`a=rand(1:5,5)` – одномерный массив из 5 элементов, заполненный случайными целыми числами от 1 до 5, распределение однородное.

`a=rand(3,5)` – двумерный массив 3*5, заполненный случайными числами от 0 до 1.

`Vector{Int64}()` – одномерный массив нулевой длины типа `Int64`,

`Vector{String}()` – одномерный массив нулевой длины типа `String`,

Пустой массив типа `Any` можно создать с использованием оператора `a=[]`.

`a=Array{Float64,1}()` - одномерный массив типа `Float64` нулевой длины.

`x=collect(a:b:c)` – создать одномерный массив, первый элемент которого равен `a`, второй `a+b`, `a+2b`,..., последний `c`.

Если есть массив `a`, можно создать массив `b` того же типа и той же размерности с использованием функции **`similar()`**:

```
b=similar(a)
```

Примеры:

`x=collect(1:1:10.0)` – массив типа `Float64: 1.0, 2.0,...10.0`.

`y=collect('a':1:'z')` - массив символов алфавита `'a'...'z'`.

`x=10.0.^(-3:3)` - массив чисел: 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0.

`a=[x + y for x in [1,2,3], y in [0.1,0.2,0.3]]` – двумерный массив 3*3 типа `Float64`. Такой способ создания массива с одновременным заполнением его ячеек называется **comprehension**.

Работа с элементами массива

Добавить элемент `v` в конец массива `a` можно с использованием функции **`push!(a,v)`**:

```
push!(a,1.23).
```

Добавить элемент `v` в начало массива `a` можно с использованием функции **`pushfirst!(a,v)`**:

```
pushfirst!(a,1.23).
```

Наличие символа "!" в конце имени функции обычно служит указанием на то, что первая переменная в списке параметров будет изменена.

Добавить строку или столбец в двумерный массив можно с использованием функций **`hcat()`** и **`vcat()`** (см. далее Объединение двух массивов).

Для массива a типа `Vector` определена операция вставки элемента x в произвольную позицию n: **`insert!(a,n,x)`**. Пример: `insert!([1,2,3],2,8)`

Последний элемент массива `a` можно удалить так: **`pop!(a)`**.

Первый элемент массива `a` можно удалить так: **`popfirst!(a)`**.

Удалить элемент массива в позиции pos можно так: **`deleteat!(a, pos)`**.

Поменять порядок элементов массива на обратный: **`a=a[end:-1:1]`**.

Проверить, есть данный элемент x в массиве: **`in(x, a)`**.

Определить длину массива: **`length(a)`**.

Определить число измерений массива: **`ndims(a)`**.

Определить размерность массива: **`size(a)`**.

Определить размер массива в байтах: **`sizeof(a)`**.

Найти максимальное число массива: **`maximum(a)`**.

Найти минимальное число массива: **`minimum(a)`**.

Преобразовать вектор-строку в вектор-столбец: **`b=vec(a)`**.

Операция транспонирования осуществляется с использованием функции **`transpose()`** или символа `'` после имени массива: `a'`.

Умножение матриц: `a*b`.

Суммирование по строкам и столбцам

Суммирование элементов коллекции выполняется с использованием функции **`sum()`**. Пусть есть одномерный массив `a=[1,2,3,4,5]`, тогда сумму значений элементов этого массива можно найти так: `sum(a)`. Если массив `a` двумерный:

```
1 2 3
4 5 6
```

то сумму значений элементов каждого столбца j $\sum_i a_{ij}$ можно найти так: `sum(a, dims=1)` , а сумму значений элементов каждой строки i $\sum_j a_{ij}$ `sum(a, dims=2)`, здесь `dims` указывает тип суммирования: 1 — по первому индексу, 2 — по второму индексу.

Выборки (*slice*)

Сделать копию вектора `a`: `b=a[:]`.

Выбрать элементы вектора `a` с i_1 по i_3 с шагом i_2: `a[i1:i2:i3]`.

Выбрать элементы вектора `a` с 3 по 5: `a[3:5]`.

Можно сделать выборку с шагом: `a[1:3:10]`. - выбираем элементы 1, 4, 7 и 10.

Для матрицы `b` выборку можно делать так

`b[2:3,2:3]` – выбрать элементы матрицы из строк 2, 3 и столбцов 2, 3.

`b[1:2,:]` - выбрать строки 1 и 2.

Двоеточие на месте одного из индексов означает «каждый элемент этого измерения»:

`b[:,2:3]` - выбрать столбцы 2 и 3,

`b[1,:]` - выбрать элементы первой строки.

Сделать копию двумерного массива `a`: `b=a[:,:]`.

Применительно к элементам массива можно использовать операции с точкой (векторизация вычислений)

`a.^2` – вычислить квадрат всех элементов массива,

`log.(a)` – вычислить натуральный логарифм всех элементов массива.

Если a и `b` – два массива с одинаковым числом элементов, то `a.+b` выдаст массив с тем же числом элементов, значения которых равны сумме соответствующих элементов `a` и `b`.

Интересной является возможность замены элементов массива с использованием выражений типа

`a[a.<0].=0` – обнулить все отрицательные элементы массива `a`. В данном случае используется то обстоятельство, что результатом операции `a.<0` является массив типа Bool.

Использование операций с точкой в сочетании с числами имеет одну особенность. Десятичная точка не должна восприниматься как признак векторизации. Например, если нужно добавить 1 ко всем элементам массива `a`, то после 1 необходим пробел:

```
1 .+a
1. .+a
```

Элементы массива можно заполнить с использованием функциональной зависимости (comprehension)

`a=[0 for i in 1:10]` – одномерный массив с десятью элементами типа Int64, заполненный нулями.

`a=[x=exp(x) for x = 0.0:0.1:1.0]` – одномерный массив типа `Float64` с одиннадцатью элементами exp(0), exp(0.1),..., exp(1).

`a=[z=x + y for x = 1:5, y = 11:15]` – двумерный массив целочисленного типа 5*5, элементы заполнены суммой значений x и y, x=1,2,...,5; y=11,12,...,15.

`a=[x*y for x in 1:10 for y in 1:2]` -одномерный массив типа `Int64`.

`a=collect(1:5)` – одномерный массив типа `Int64`, элементы которого содержат числа 1, 2,...5.

`a=collect(1.0:5.0)` – одномерный массив типа `Float64`, элементы которого содержат числа 1.0, 2.0,...5.0.

`a=collect(2.:2.:8.)` – одномерный массив типа `Float64`, элементы которого содержат числа 2.0, 4.0,...8.0.

Команда **`unique()`** позволяет выбрать уникальные элементы массива (например, `unique(a)`), при этом возвращается массив. Для этой цели можно использовать также команду Set (например, `Set(a)`), возвращается множество.

Объединение двух массивов

Если массивы x и y имеют одинаковое число столбцов, то объединить их (по вертикали) можно командой **`vcat(x,y)`** или `[x;y]`,

Если массивы x и y имеют одинаковое число строк, то объединить их (по горизонтали) можно командой **`hcat(x,y)`** или `[x y]`.

Команду `vcat` можно использовать для добавления строк в массив вместо команды `push!`, которая применима лишь к векторам (одномерным массивам).

Пример. Добавить к матрице [1 1] строку [2 2]:

```
z=[1 1]
z=vcat(z,[2 2])
```

В языке Julia нет команд для удаления строк или столбцов, однако удалить, например, вторую строку из массива `a`

```
1 2 3
4 5 6
7 8 9
```

можно так

```
a=vcat(a[1,:]',a[3,:]')
```

аналогичным образом, для удаления столбца, например, 3 можно так

```
a=hcat(a[:,1],a[:,2])
```

Работа с несколькими массивами

Команда **`union()`** позволяет выбрать уникальные элементы двух или более массивов , например, в данном случае

```
a=[1, 1]
b=[2, 3]
c=[1, 3]
union(a,b,c)
```

результатом будет массив чисел 1,2,3.

Команда **`intersect()`** позволяет выбрать из двух или более массивов совпадающие элементы. Для приведенных выше массивов результатом выполнения команды

`intersect(b,c)` будет 3.

Команда **`setdiff()`** позволяет выбрать те элементы, которые есть в первом массиве, но которых нет в остальных. Для приведенных выше массивов результатом выполнения команды

`setdiff(b,c)` будет 2.

Сортировка элементов одномерного массива (вектора)

`sort!(x)` – в порядке возрастания,

`sort!(x, rev=true)` – в порядке убывания.

Аналогичная функция `sort` (без восклицательного знака) не меняет элементы вектора `x`, а возвращает новый вектор, который содержит отсортированные элементы.

Часто бывает удобно использовать матрицу перестановок, которая содержит индексы элементов массива:

`mp=sortperm(x)` - в порядке возрастания,

`mp=sortperm(x,rev=true)` - в порядке убывания,

`xs=x[mp]` - отсортированный массив.

Печать отсортированного массива можно организовать таким образом:

```
for i in mp println(a[i]) end
```

Если массив двумерный `y[m,n]`, нужно указать номер размерности (1 или 2), по которой проводится сортировка

```
y=[4 3; 1 2]
```

```
 4  3
 1  2
1) sort!(y, dims=1)

 1  2
 4  3
2) sort!(y, dims=2)

 1  2
 3  4
```

Поиск номеров элементов массива, содержащих заданные значения

Найти индекс первого элемента массива, содержащего данное значение `value`:

`findfirst(x -> x == value, a)`

или так

`findfirst(isequal(value), a)`

или так

`findfirst(==(value), a)`

Найти индексы всех элементов массива, содержащих данное значение `value`:
`findall(x -> x == value, a)`

или так

`findall(isequal(value), a)`

или так

`findall(==(value), a)`

Удалить все элементы массива, содержащие значение `b`:

```
deleteat!(a, findall(x -> x == b, a)).
```

При необходимости найти номер элемента массива по его значению можно использовать функцию `indexin()`. При наличии нескольких значений в массива функция возвращает номер первого подходящего элемента массива.

Примеры.

```
a=[10,20,30,10,20,30]
```

`indexin(20,a)` – возвращает массив, первый элемент которого равен 2 .

```
s=["a","b","c"]
```

```
b="c"
```

`indexin([b],s)` - возвращает массив, первый элемент которого равен 3.

Можно использовать функцию `indexin()` и для поиска номеров группы элементов.

`indexin([20,10],a)` возвращает массив с элементами которого 2, 1.

Для работы с массивом структур используется техника, смысл которой ясен из примера

```
struct Sub
fml::String
idx :: Int64
end
subarr=Array{Sub,1}()
push!(subarr,Sub("H2",1));
push!(subarr,Sub("O2",2));
push!(subarr,Sub("Ar",1));
```

Выбрать все элементы массива `subarr` с параметром `idx`, равным 1:

```
filter(p->p.idx==1,subarr)
```

Найти порядковые номера всех элементов массива `subarr`, у которых значение параметра `idx` равно 1:

```
findall(p->p.idx==1,subarr)
```

Массив с произвольной индексацией

Как сказано выше, массивы в Julia индексируется с 1. Если есть необходимость использовать индексацию другого типа (например с 0), можно использовать библиотеку OffsetArrays. В этом случае массив вначале создается одним из перечисленных выше способов, а затем его границы изменяются:

```
A=zeros(11) # создан массив с индексацией от 1 до 11
OA=OffsetArray(A,0:10)  # индексация элементов массива изменена
```

Для использования функции OffsetArray необходимо установить библиотеку OffsetArrays.

Изменение размерности массива

При необходимости можно изменить размерность массива с использованием функции **`reshape()`**. Допустим, нужно преобразовать одномерный массив

`a=[1,2,3,4,5,6]` в двумерный с двумя строками и тремя столбцами. Это можно сделать следующим образом

```
a=reshape(a,2,3)
```

Результат выглядит так

```
1 3 5
2 4 6
```

Преобразовать двумерный массив `a` в одномерный можно аналогичным образом:

```
a=reshape(a,6)
```

Задачи

Попробуйте выполнить следующие упражнения. Постарайтесь понять логику происходящего.

```
1.
a=zeros(3)
push!(a,1)
pushfirst!(a,9)
insert!(a,2,8)
pop!(a)
a
popfirst!(a)
a
deleteat!(a, 1)
2.
a=[1,2,3]
b=a[end:-1:1]
3.
x=[1,2,3]
in(1, x)
in(10,x)
length(x)
sizeof(x)
4.
x=rand(10)
maximum(x)
minimum(x)
```

```
5.
a=[1,2,3]
a.^2
a.+1
6.
a=[x=x for x = 0.0:0.1:1.0]
b=collect(1:11)
a.+b
7.
x=[1,2,3]
y=[4,5,6]
z=vcat(x,y)
z=[x;y]
z=hcat(x,y)
z=[x y]
8.
a=[10, 11]
b=[20, 31]
c=[10, 31]
union(a,b,c)
intersect(b,c)
setdiff(b,c)
9.
x=rand(10)
sort(x)
sort(x, rev=true)
mp=sortperm(x)
for i in 1:length(x) println(x[mp[i]]) end
for i in mp println(x[i]) end
10.
x=collect(1:2:20)
findfirst(a -> a == 11, x)
findall(a-> a>11,x)
11.
a=[1,1,1,2,2,2]
findall(isequal(2), a)
deleteat!(a, findall(x -> x == 2, a))
```

```
a
12.
a=[1,2,3,1,2,3]
indexin(3,a)
indexin([1,2],a)
indexin([2,1],a)
indexin([20,10],a)
```

Кортежи (tuples)

Кортеж — группа значений некоторых величин фиксированной длины, , разделенных запятыми, иногда эту группу окружают круглыми скобками: `c=(1, 1.1, pi,'c', "Julia", 1//3)`. Кортеж — это контейнер гетерогенных данных, в отличие от массива, который является контейнером однородных (гомогенных) данных.

Кортеж можно превратить в вектор типа `Any`: `v=[c…]`.

Пример. Создать вектор z типа `Tuple{Char, Char}`, содержащий пары строчных и прописных значений символов латинского алфавита

```
A = join('A':'Z')
a = join('a':'z')
z = collect(zip(a, A))
```

в котором

```
z[1]==('a','A')
z[1][1]=='a'
z[1][2]=='A'
```

Вектор можно превратить в кортеж: `c=(v…,)`.

Элементы кортежа изменять нельзя, в отличие от элементов массива, оператор `c[1]=2` вызовет сообщение об ошибке.

Можно использовать именованный кортеж вида `c=(a=1, b=2.2)`. Доступ к элементам кортежа либо через индекс, например, `c[1]`, либо по имени `c.a, c.b`.

Если список возвращаемых величин какой-либо функции содержит несколько значений, то возвращаются они в форме кортежа или массива (когда возвращается массив данных).

Множества

Множества (Set) используются для хранения коллекций неупорядоченных уникальных значений: `s=Set()` - пустое множество, `s=Set([1,3,5,7])`. Добавле-

ние элемента к множеству производится с использованием функции `push!`. Удаление элемента множества производится с использованием функции `pop!`:

`pop!(s,z)` – удаляет указанный элемент `z` множества.

Другие функции для работы с множествами: пересечение `intersect(set1,set2)`, объединение `union(set1,set2)`, разность `setdiff(set1,set2)`. Эти функции можно использовать и для массивов.

Превратить строку `st` в множество содержащихся в ней символов можно так:

```
x=Set(st).
```

Аналогичным образом в множество можно превратить массив `z` (одномерный или многомерный):

```
x=Set(z).
```

Используя функцию `collect()`, можно превратить множество `x` в массив `z`:

```
z=collect(x).
```

Множество можно задать также при помощи интервала: `x=Set(10:4:30).`

Проверить наличие элемента `s` в множестве `m`: `s in m.`

Итератор enumerate

Доступ к элементам последовательности (массива, кортежа) можно получить с помощью итератора enumerate, использование которого целесообразно в том случае, если требуется вывести элемент последовательности вместе с его порядковым номером.

Пример. Пусть задан массив `a = [10,20,30]`, нужно вывести на печать пронумерованные по порядку элементы этого массива

```
for (i,v) in enumerate(a)
println("$i $v")
end
```

Функция zip

`zip()` – это функция-итератор, предназначенная для объединения двух или более последовательностей в кортеж, при этом каждый элемент кортежа состоит из элементов объединяемых последовательностей. По смыслу `zip()` является итератором, который можно использовать,например, для вывода на печать упорядоченных элементов последовательностей:

```
st="ABC"
```

```
n=[10,20,30]
for x in zip(st, n)
println(x)
end
```

Функцию-итератор `zip()` можно превратить в массив кортежей:

```
collect(zip(st,n))
```

Если исходные последовательности имеют разную длину, то длина объединенной последовательности равна длине самой короткой из них.

Функцию `zip()` можно использовать для проверки того, содержит ли объединенная последовательность определенную группу значений.

Пример. Пусть нужно проверить, есть ли в объединении последовательностей A и B комбинация значений `(x,y)`. Условие проверки записывается так

```
if (x, y) in zip(A, B)
```

Наконец, функцию `zip()` можно использовать для создания словаря:

```
d=Dict(zip(A,B))
```

Функция reverse

Функция `reverse()` предназначена для того, чтобы изменить порядок расположения элементов последовательности (массив `a`, кортеж `c`) на обратный: `reverse(a)`, `reverse(c)`. Если последовательность изменяемая (mutable), то можно использовать оператор `reverse!`. В частности, допустимо выражение `reverse!(a)`, но выражение `reverse!(c)` не допустимо.

Функция clipboard

Функция `clipboard()` обеспечивает интерактивное взаимодействие в буфером обмена компьютера. Считать текстовую информацию из буфера обмена в переменную s можно так:

```
s=clipboard()
```

Перенести текстовую информацию из переменной s в буфер обмена можно так:

```
clipboard(s)
```

Словари (dictionaries)

Словарь — это ассоциативный массив, состоящий из пар ключ-значение. Пустой словарь создается так: **`d = Dict()`**, словарь со значениями:

```
d = Dict('a'=>1, 'b'=>2, 'c'=>3)
```

или так

```
d = Dict("a"=>1, "b"=>2, "c"=>3),
```

или так

```
d = Dict(10=>1, 20=>2, 30=>3),
```

или так

```
d = Dict("10"=>1, '2'=>"a", 4=>"abc"),
```

или так

```
d = Dict([("one", 1), ("two", 2)]).
```

Заполнить словарь символами алфавита и их номерами можно так

```
y=collect('a':'z')
d = Dict()
for i in 1:length(y) d[y[i]]=i end
```

Словарь можно создать с использованием функции zip:

```
d = Dict(zip("abc", [10,20,30])).
```

Информацию в словарях можно менять: `d['a']=100; d['a']+=1.`

Добавление пары ключ-значение в словарь: **`d[key] = value`**, например, `d["new"]=123.`

Удаление из словаря: **`delete!(d,key)`**, например,

```
delete!(d,"new")
```

`pop!(d,"new")` – возвращает значение удаляемой пары 123.

Поиск в словаре d значения по ключу (`'a'`): `d['a']`.

Альтернативный способ поиска в словаре d значения по ключу (`'a'`): `get(d,'a',defaultvalue)`, `defaultvalue` – значение, которое функция возвращает, если заданное значение не найдено.

Получить список ключей в словаре d : `keys(d)`.

Получить список значений в словаре d : `values(d)`.

Проверить наличие ключа в словаре d: `haskey(d, 'a')`.

Распечатать словарь (ключ-значение)

```
for (k,v) in d
    println("$k - $v")
end
```

Последовательность элементов словаря не фиксируется, т. е. может быть произвольной. Значения в словаре можно изменять.

В качестве ключа словаря можно использовать кортеж. Например, нужно сохранить в словаре `substance (substance=Dict())` следующую информацию о некоторых свойствах (`dfh298, cp298, s298, h298`) химического вещества (`formula`) в данном состоянии (`pstate`) :

```
substance[formula, pstate]=[dfh298, cp298, s298, h298]
```

Данные в словаре хранятся в неотсортированном виде. Вывести на печать отсортированный список символов и их номеров можно так

```
for c in sort(Char.(keys(d)))
println("$c-",d[c])
end
```

Пример. Создадим словарь, содержащий список символов и их количество в данном тексте s:

```
function histogram(s)
     d = Dict()
     for c in s
     if c in keys(d) # haskey(d, c)
          d[c] += 1
          else
          d[c] = 1
          end
     end
return d
end
```

Проверим работу функции

```
h = histogram("brontosaurus")
h = histogram("акула, карась")
```

Найти в словаре символ, который встречается заданное число (v) раз

```
function reverselookup(d, v)
     for k in keys(d)
     if d[k] == v
     return k
     end
  end
  error("LookupError")
end
```

Проверим работу функции

```
key = reverselookup(h, 2)
```

Упорядочим символы по их количеству в тексте. Создадим словарь, содержащий список букв их количество в данном словаре d:

```
function invertdict(d)
inverse = Dict()
for key in keys(d)
val = d[key]
if !(val in keys(inverse))
inverse[val] = [key]
else
```

```
push!(inverse[val], key)
end
end
inverse
end
```

Проверим работу функции

```
inverse = invertdict(h)
```

При большом числе элементов поиск значения по ключу в словаре происходит быстрее, чем последовательный поиск в массиве.

Структуры данных

Структуры данных являются составными типами, это совокупность именованных полей, сгруппированных вместе и рассматриваемых как единое целое. Пример структуры

```
struct MyStruct
           bar
           baz::Int
           qux::Float64
       end
```

Поле, тип которого не задан (`bar` в данном случае), по умолчанию имеет тип Any. Создать объект типа `MyStruct` можно так

```
ms = MyStruct("Hello, world.", 23, 1.5)
```

Обращаться к полям объекта можно так: `ms.bar, ms.baz,ms.qux.` Однако менять значения полей нельзя. `ms.bar=3` выдаст сообщение об ошибке.

Если требуется менять значения полей переменных типа структуры, ее следует объявить с ключевым словом `mutable`:

```
mutable struct MyStructM
bar
baz::Int
qux::Float64
end
```

Однако, если структура содержит массив, значения его полей можно менять независимо от наличия слова `mutable` в ее объявлении.

```
Tar=Array{Float64,1}
struct MyStructAr
bar::Int
ar::Tar
end

f=MyStructAr(1,[1,2,3])
f.ar[1]=0
```

```
push!(f.ar,4)
```

Можно создать структуру, которая содержит структуру:

```
struct TG
x::MyStructAr
end
```

Можно создать структуру, которая содержит параметризованные поля:

```
struct TP{T}
x::T
y::T
end
struct TP2{T1, T2}
x::T1
y::T2
end
struct TP3{T<:Union{Integer, Real}}
x::T
y::T
end
```

Тип полей переменной в этом случае будет определяться автоматически на основании численных значений, которые присваиваются полям:

`t1=TP(1,1)` - поля целого типа,

`t2=TP(1.0,1.0)` и `t3=TP{Float64}(1,1.0)` - поля вещественного типа.

`t4=TP2(1,1.0)` - одно поле типа Int64, другое - Float64.

`t5=TP3(1,1)` - поля целого типа, `t6=TP3(1.0,1.0)` - поля вещественного типа.

Несколько функций для работы с полями структуры.

`fieldnames(MyStructAr)` — получить список полей структуры `MyStructAr`.

`isdefined(MyStructAr, :ar)` — содержит ли структура `MyStructAr` поле `ar`?

Массив структур

Создание пустого массива для хранения структур типа `MyStructAr`:

```
af=MyStructAr[]
```

Добавление элемента к массиву `af`:

```
push!(af,MyStructAr(1,[pi,2pi,3pi]))
```

Missing, nothing and NaN

Язык Julia поддерживает еще некоторые переменные: missing, nothing и NaN. Переменная типа Nothing (nothing) возвращается функцией, которая не содержит возвращаемых значений, аналог NULL.

Переменная типа Missing (missing) соответствует значению, которое не задано (отсутствует), используется при обработке статистических данных. Как правило, любая операция (сложение, умножение,...) с переменной типа Missing приводит к результату того же типа, т. е. происходит распространение типа. В библиотеке Statistics есть функция `skipmissing()`, которая позволяет выполнять действия с данными, содержащими переменные типа Missing. Например,

```
a=[1,2,missing,3],
```

результатом вызова функции `sum(a)` будет missing. Результатом вызова функции `sum(skipmissing(a))` будет 6.

Переменная NaN (no a number) тип Float64 является следствием операции, результат которой не определен (например, 0/0).

Результатом деления -1/0 является -Inf, а при делении1/0 получаем Inf.

Константы

Поскольку тип переменных а Julia легко изменяется, это приводит к дополнительной нагрузке на ресурсы компьютера. Чтобы уменьшить эту нагрузку, можно использовать определение переменной с использованием ключевого слова const:

```
const amount = 10.00
const z=zeros(3)
```

Значения переменных, объявленных таким образом можно изменять, однако их тип менять нельзя, т. е. `amount = 4.0` допустимо, `amount = zeros(3)` – нет; `z[1]=3` допустимо, `z=3` – нет. Ключевое слово `const` позволяет зафиксировать тип переменной.

О присваивании значений и копировании в Julia

Присваивание осуществляется оператором `=`. Пример: `a=3`. Допускается множественное присваивание вида `a=b=3`. Если необходимо поменять значения переменных (a на b, b на a), допустимо использовать конструкцию `a,b=b,a`.

Для того чтобы сократить избыточное использование памяти, в Julia используется механизм копирования ссылок на объекты при присваивании. Например, если `a` и `b` два вектора,

```
a=[1,2,3]
```

то после присваивания

```
b=a
```

оба вектора указывают на одну область памяти. Поэтому изменение значения ячейки в одном из векторов приведет к одновременному ее изменению и в другом векторе.

Если такая связь массивов нежелательна, вместо знака равенства следует использовать функцию **`copy()`**, которая создает независимую копию: `b=copy(a)`. В этом случае создается независимый объект `b`, в который копируется содержимое `a`. При этом в новом объекте могут содержаться ссылки на область памяти объекта `a`, (например на область памяти, занимаемую массивом). Если содержание области памяти (элементов массива) исходного объекта a меняется, изменится и содержание соответствующего массива объекта `b`. Чтобы избежать этого, можно использовать функцию **`deepcopy()`**, которая создает копию исходного объекта и рекурсивно копирует в него содержимое исходного объекта.

Рассмотрим несколько примеров, чтобы увидеть разницу. Исходными являются массивы a, b, c, d

```
a = [[[1,2],3],4] # [[[1, 2], 3], 4]
b = a             # [[[1, 2], 3], 4]
c = copy(a)       # [[[1, 2], 3], 4]
d = deepcopy(a)   # [[[1, 2], 3], 4]
```

Изменим значение одной из ячеек массива a

```
a[2] = 40
b # [[[1, 2], 3], 40] — значение элемента a[2] изменилось
c # [[[1, 2], 3], 4]  - значения элементов массива неизменны
d # [[[1, 2], 3], 4]  - значения элементов массива неизменны
```

Изменим теперь значение вложенного массива

```
a[1][2] = 30
b # [[[1, 2], 30], 40] — значение элемента изменилось
c # [[[1, 2], 30], 4]  — значение элемента изменилось
d # [[[1, 2], 3], 4]   - значения элементов массива неизменны
```

Наконец, присвоим a некоторое число

```
a=1.23
b # [[[1, 2], 30], 40] — значения элементов массива неизменны
c # [[[1, 2], 30], 4]  — значения элементов массива неизменны
d # [[[1, 2], 3], 4]   - значения элементов массива неизменны
```

Тождественность объектов проверяется с использованием оператора ===. Если `a = [1, 2]; b = [1, 2]`; то условия `a == b` и `a === a` верны (true), а условие `a===b` ложно.

Рассмотрим теперь структуру для хранения координат точки на плоскости.

1. Сначала неизменяемая структура

```
struct Point
x
y
end
```

```
p1 = Point(1.0, 2.0)
p2 = deepcopy(p1)
```

p1===p2 (результат true).

p1==p2 (результат true).

2. Теперь изменяемая структура

```
mutable struct MPoint
x
y
end
p1 = MPoint(1.0, 2.0)
p2=deepcopy(p1)
```

p1===p2 (результат false) – этот результат ожидаем, поскольку p1 и p2 являются разными объектами.

p1==p2 (результат false) – несмотря на то, что координаты точек p1 и p2 одинаковы, в данном случае оператор == ведет себя так же, как оператор ===, т. е. проверяется не эквивалентность, а идентичность объектов. Для изменяемых структур данных Julia (версия 1.7) не знает, что считать эквивалентным.

Операторы сравнения и условные операторы

Операторы сравнения: > (больше), >=(больше или равно), < (меньше), <=(меньше или равно), == (равно), != (не равно).

Условные операторы рассмотрим на примере использования связки

if-elseif-else-end

Простейший случай

```
if x < y
     println("x меньше, чем y")
     <другие операторы, если нужно>
end
```

Цепочка сравнений

```
if x < y
     println("x меньше, чем y")
     <другие операторы, если нужно>
elseif x > y
     println("x больше, чем y")
     <другие операторы, если нужно>
else
     println("x равно y")
     <другие операторы, если нужно>
end
```

Допустимы выражения вида: if 0 <= x < 10

Условный (тройной) оператор вида **`a ? b : c`** (обязательны пробелы слева и справа от символов ? и :), a – условие, если оно верно, то выполняется b, иначе выполняется c.

Пример: `x > 0 ? println("x>0") : println("x<=0")`

Операторы &, &&, |, || (Short-Circuit Evaluation)

Проверка условий (Short-Circuit Evaluation) — это концепция языка программирования, означающая, что при наличии нескольких условий проверка производится до невыполнения первого из них. Например, условие A верно, если выполняются условия B, C, D. Проверка условий прекращается, как только выясняется, что одно из них неверно. Рассмотрим, как эта концепция реализована в языке Julia.

В выражении **a** && **b** условие **b** проверяется только в том случае, если **a** верно (true), в выражении **a** || **b** условие **b** проверяется только в том случае, если **a** ложно (false). Операторы (&&, ||) можно использовать в качестве условных в выражениях:

выражения `if` **`a`** `then` **`b`** и **`a`** `&&` **`b`** эквивалентны,

выражения `if !`**`a`** `then` **`b`** и **`a`** `||` **`b`** эквивалентны.

```
a=1; b=1;
```

`a>0 & b>0` (результат `false`)

`(a>0) & (b>0)` (результат `true`)

`a>0 && b>0` (результат `true`)

```
a=1; b=0;
```

`a>0 | b>0` (результат `false`)

`(a>0) | (b>0)` (результат `true`)

`a>0 || b>0` (результат `true`)

Условие вида `a < x && a > y` на языке Julia можно записать так `y < a < x`. Пример `if 0 < a < 10`.

Операторы && и || можно использовать в связке с проверкой условия. Пример

```
a=1
if a > 0 println("a>0") end
```

или так

```
a > 0 && println("a>0")
```

или так

```
a < 0 || println("a>0").
```

Иными словами, && соответствует «если утверждение верно, то выполнить», || соответствует «если утверждение ложно, то выполнить».

Оператор(ы), которые должны быть исполнены после проверки условия в некоторых случаях должны быть заключены в круглые скобки.

Пример. Следующий фрагмент кода удаляет пробелы из строки `st`:

```
source=""
for i in 1:length(st)
  st[i] > ' ' && (source *= st[i])
end
```

Генераторы случайных чисел

`rand()` - случайное число из диапазона [0:1],

`rand(a:b)` - случайное целое число из диапазона [a:b],

`rand(a:0.1:b)` - случайное число из диапазона [a:b] с точностью до первого знака.

См. также раздел "Случайные числа (библиотека Random)".

Циклы

Цикл можно организовать следующим образом (`i1` - первое значение, `i2` - последнее, `i3`- шаг цикла, по умолчанию `i3=1` и шаг можно не задавать.

```
for i in i1:i2
...
end
for i = i1:i2
...
end
for i in i1:i3:i2
...
end
for i = i1:i3:i2
...
end
```

Примеры вывода на печать

```
for i in 1:5 println(i) end
for i = 1:5 println(i) end
for i in 0:0.1:1 println(i) end  # step = 0.1
for i = 1:3:15 println(i) end   # step = 3
```

Если `a` – одномерный массив (vector), множество (set) или диапазон (range), то доступ к элементам соответствующей коллекции удобно организовать таким образом:

```
for x in a println(x) end
```

Если коллекция индексируемая, например, `a` – одномерный массив (vector), то

```
for i in eachindex(a) println(a[i]) end
for i = eachindex(a) println(a[i]) end
for i in 1:length(a) println(a[i]) end
for i in 1:size(a,1) println(a[i]) end
```

```
while <условие верно>
     ...выполняются операции
end
```

Выйти из цикла можно с помощью команды **`break`**:

```
if i > 10 break end
```

Команда **`continue`** переводит цикл на следующую итерацию:

```
if i == 10 continue end
```

Примеры двойного цикла (результаты вывода отличаются!)

```
for i = 1:2, j = 3:4
    println((i, j))
end
for i = 1:2, j = 3:4
    println(i, j)
end
```

Функции map, foreach, filter

В широком смысле функция **`map`** превращает один набор данных в другой. В частности, функция **`map(f,c…)`** применяет функцию `f` ко всем элементам коллекции `c` и возвращает результат. Иными словами, это **`map`** - функция высшего порядка.

Примеры.

```
map(x-> x*2, [1,2,3]) # результат: 2, 4, 6
map(+, [1,2,3], [10,20,30]) # результат: 11, 22, 33
```

В более понятной но менее компактной форме эти примеры можно записать так

```
map([1,2,3]) do x
  x*2
end
map([1,2,3], [10,20,30]) do x,y
  x+y
end
```

Функция **foreach(f,c…)** вызывает функцию `f` для всех элементов коллекции `c`, но не возвращает результат.

Примеры.

```
a=[1,2,3]
foreach(println,a)  # Печать всех элементов вектора a
foreach(x->println(x^2),a) # Печать квадрата элементов вектора a
```

Функция **filter(p,x)** возвращает подмножество элементов коллекции `x`, которые соответствуют предикату `p`. Предикат — это функция, на вход которой подается некоторое значение и которая всегда возвращает логическую переменную true или false.

Примеры

```
a=[1,2,3,4,5]
filter(iseven,a)# возвращает четные значения массива a
filter(x->x>1,a)# возвращает все элементы массива a, которые >1
ff(x::Integer)= x < 3 # определяем предикат для функции filter
filter(ff,a) # возвращает все элементы массива a, которые < 3
```

Чтение и запись данных

Вывод данных в файл или на экран осуществляется с использованием функций **print()**, **println()** и **write()**.

Примеры

`print("Test1"); println("Test 2"); write(stdout, "Julia")`. В последнем случае кроме текста на экран будет выведено число напечатанных символов (5).

Чтение данных осуществляется функцией **read()**: `read(stdin,Char)` – данная команда ожидает ввода одного символа с клавиатуры. Чтение данных из файла производится функциями **read()** и **readline()**. Команда `readline(stdin)` считывает данные до тех пор, пока не встретится символ перевода каретки `\n` (нажатие клавиши Enter). Команда `x=readline()` прочитает введенный с экрана текст, присвоит его переменной x и отобразит его на экране. Переменная x при этом содержит строку символов.

В некоторых случаях (массивы, структуры данных) для вывода данных на экран удобно использовать функцию **show()**.

Форматирование вывода данных

Для форматированного вывода данных удобно использовать макрокоманду **@printf** (предварительно нужно подключить соответствующую библиотеку: using Printf). Данная макрокоманда позволяет форматировать вывод так, как это делает функция printf() языка C.

Примеры

```
@printf("Characters: %c %c \n", 'a', 65)
> Characters: a A
@printf("Decimals: %d %ld\n", 1977, 650000)
> Decimals: 1977 650000
@printf("Preceding with blanks: %10d \n", 1977)
> Preceding with blanks:       1977
@printf("Preceding with zeros: %010d \n", 1977)
> Preceding with zeros: 0000001977
@printf("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100)
> Some different radices: 100 64 144 0x64 0144
@printf("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416)
> floats: 3.14 +3e+00 3.141600E+00
@printf("%s \n", "A string")
> A string
```

Для вывода чисел удобнее использовать формат `g`, который автоматически выбирает оптимальный способ представления данных (знак `"-"` в примере обеспечивает выравнивание текста влево):

```
@printf("%-12.6g \n", pi)
> 3.14159
@printf("%-12.6g \n", pi*1.e-10)
> 3.14159e-10
@printf("%-12.6g \n", 1)
> 1
```

Более подробно см. http://www.cplusplus.com/reference/cstdio/printf/

Работа с файлами

По умолчанию рабочим является каталог, в который установлена Julia! Изменить текущий каталог можно с использованием команды **`cd`**, например, **`cd("d:\\Julia")`**. Работа с файлом начинается с того, что указывается его имя (`fname="test.dat"`). Далее файл нужно открыть (`f1=open(fname, mode)`), `mode` характеризует возможности работы с файлом, см. Табл. 1.

Таблица 1. Значения параметра `mode`

Mode	Description	Keywords
`r`	`read`	`none`
`w`	`write, create, truncate`	`write = true`
`a`	`write, create, append`	`append = true`
`r+`	`read, write`	`read = true, write = true`
`w+`	`read, write, create, truncate`	`truncate = true, read = true`
`a+`	`read, write, create, append`	`append = true, read = true`

Очевидно, можно сразу выполнить команду `f1=open("test.dat","r+")`. При указании имени файла следует указать путь к нему. Для работы в среде Windows путь должен иметь такой вид:

`"c:\\mytest\\test.dat".`

Проверить наличие файла можно с использованием функции **`ispath()`**, которая возвращает true, если файл существует, или false, если такого файла нет.

Пример:

```
ispath("c:\\mytest\\test.dat").
```

Функция **`readdir()`** выводит на экран все файлы, которые есть в данном каталоге. Например, вывести список файлов в текущем каталоге можно так

`readdir(pwd())`.

Можно прочитать все данные из файла сразу командой

```
alldata = readlines(f1)
```

В этом случае alldata содержит массив строк `Array{String,1}`.

Для обработки информации в массиве `alldata` можно использовать цикл

```
for line in alldata
     println(line)
     ...прочие действия
end
close(f1)
```

Рекомендуется всегда закрывать файл после завершения работы с ним.

Наиболее простой способ работы с данными из файла с именем `fname` выглядит так

```
open(fname, mode) do f1
     … действия с файлом f1
end
```

В этом случае файл закрывать не нужно, он закрывается автоматически после выхода из блока.

```
open(fname) do f1
     for line in eachline(f1)
     … действия с файлом f1
     end
end
```

Как ясно из таблицы 1, если предполагается запись информации в файл, то его нужно открыть с ключом `"w"`:

```
fname = "example2.dat"
f2 = open(fname, "w")
write(f2, "Текст записан в файл\n")
```

```
# выводится число 38 (число записанных в файл байт)
println(f2, "в том числе и командой println!")
close(f2)
```

Однако, если с ключом "w" открывается существующий файл, он будет перезаписан. Ключ "a" позволяет открыть существующий файл с возможностью добавления информации.

В некоторых случаях для чтения файла удобно использовать структуру **try...catch...finally**

```
try
open("myfile.txt","r") do f
…
end
catch ex
finally
close(f)
end
```

Функции

Шаблон определения функции в Julia имеет вид

```
function name(список параметров)
     тело функции
end
```

name – имя функции (не обязательно), список параметров тоже не обязателен. Возвращаемой величиной является последнее значение или список значений (кортеж). Перед списком возвращаемых значений можно использовать ключевое слово return. Функция может ничего не возвращать, в этом случае тип возвращаемой величины Nothing.

Пример

```
function test1(a)
     a+=5
     a/2
end
```

Если a равно 2, то test1(a) возвращает 3.5. При этом значение переменной a не меняется (по-прежнему равно 2). Однако, если в списке параметров функции есть массив или структура данных, элемент(ы) которых изменяются в теле функции, то эти изменения видны и в вызывающей функции, т. е. такие изменения глобальны.

Пример

```
function test_arr(a)
```

```
     a[1]+=1
     return nothing
end
a=zeros(1)
test_arr(a)
```

Теперь `a[1]` равно `1`

Пример

```
mutable struct SM
a::Int
b::Float64
end
s=SM(0,0)
function test_struct(s)
s.a=10
s.b=20
end
test_struct(s)
```

Проверка показывает, что поля a и b структуры SM изменились. Однако, если структуру объявить как неизменяемую (без ключевого слова mutable), при вызове функции будет сгенерирована ошибка `immutable struct cannot be changed`.

Следующий пример показывает, каким образом функция возвращает несколько значений.

```
function test2()
     sin(1), cos(1), "Julia test"
end
```

`test2()` возвращает два числа: `0.8414709848078965, 0.5403023058681398` и фразу `"Julia test"`.

Эти значения можно присвоить переменным при вызове функции

```
a,b,st = test(2)
```

Ключевое слово `return` обеспечивает выход из функции и может встречаться в тексте функции несколько раз.

```
function test3(n)
     if n < 0
     return "n < 0"
     elseif n==0
     return "n == 0"
     else
     return "n > 0"
     end
end
```

Функция может возвращать функцию.

Пример. Создадим функцию `power(n)`, которая возвращает функцию, возводящую число в определенную степень `n`. Показатель степени `n` указывается при вызове функции `power()`. Такой подход позволяет сократить список параметров, передаваемых в функцию:

```
function power(n)
  function test(a)
  a^n
  end
end
pf=power(2) # pf - функция, которая возводит число в степень 2
pf(3) # получаем 9
pf=power(3) # pf - функция, которая возводит число в степень 3
pf(3) # получаем 27
```

В приведенном примере функция `test()`, возвращаемая функцией `power()`, называется *closure* (замыкание). Она фиксирует некоторое внешнее состояние из доступной ей области. Можно сказать, что это функция с памятью.

В некоторых случаях бывает удобно использовать функции без имени (*anonymous functions* или *lambda expression*). Шаблон анонимной функции имеет вид

(список параметров) -> выражение

Примеры:

```
(x) -> x^3 # возведение в третью степень
(x,y,z) -> x+y+z # сумма трех слагаемых
```

Анонимную функцию можно присвоить переменной, а затем использовать:

```
fn=function (x,n) # возведение x в степень n.
  x^n
end
fn(2,3) # результат 8
```

Анонимные функции часто передаются в качестве параметра другой функции. Кроме того, их удобно использовать с функцией `map()`, см. выше.

Пример расчета производной функции `f`

```
f=(x)->x^3
function numerical_derivative(f, x, dx=1.e-6)
derivative = (f(x+dx) - f(x-dx))/(2*dx)
return derivative
end
```

`numerical_derivative(f, 2)` возвращает 12.000000000345068.

Заголовок функции может содержать переменное число аргументов. В этом случае в конце списка параметров помещается троеточие (оператор **`splat`**).

Пример функции, вычисляющей сумму списка чисел.

```
f=function(a...)
sum=0
for i in 1:length(a) sum+=a[i] end
return sum
end
```

`f(1,2,3)` возвращает `6`.

Заголовок функции может содержать обязательные и необязательные (опциональные) параметры. Вначале задаются обязательные, а затем опциональные параметры. Значения опциональных параметров по умолчанию задаются в заголовке функции.

Пример. Функция, которая печатает заданное число элементов массива

```
function printn(a,n=3)
for i in 1:n println(a[i]) end
end
function printarr(a::Array)
for i in 1:length(a) println(a[i]) end
end
```

Вызов

`printn([1,2,3,4,5])` – печатать три первых элемента массива,

`printn([1,2,3,4,5], 4)` – печатать четыре первых элемента массива.

Тип аргументов функции

Тип аргументов функции указывать необязательно. При необходимости можно в сигнатуре функции указать тип одной или нескольких переменных

```
fun(a::Integer, b, c::Float64) = a+b+c
```

В этом случае при вызове функции `fun()` переменная a должна иметь тип Integer, переменная c – тип Float64, переменная b должна быть числом.

В следующем примере при вызове функции производится печать элементов массива произвольной длины и типа:

```
function printarr(a::Array)
for x in a println(x) end
end
printarr([1,2,3,4,5])
printarr(collect(1:10))
printarr([1.1,2.2,3.3])
printarr(['a','b','c','d','e','f'])
printarr(["Hello,","World"])
printarr([1+2im,2+4im])
```

Рекурсия

Функция может вызывать себя (рекурсивный вызов). Пример расчета чисел Фибоначчи (каждое число последовательности F_n равно сумме двух предыдущих $F_n = F_{n-1} + F_{n-2}$, при этом $F_0 = 0$, $F_1 = 1$):

```
function fib(n)
     if n == 0 return 0 end
     if n == 1 return 1 end
     return fib(n-1) + fib(n-2)
end
```

Множественная диспетчеризация

В языке Julia тело функции является методом. Допустимо существование нескольких функций с одним названием, но разными списками и/или типами аргументов и разными методами. Выбор подходящего метода при вызове функции в этом случае осуществляется по списку и типу аргументов. Такой способ выбора метода функции называется множественная диспетчеризация (multiple dispatch).

Пример

```
fun(x::Integer) = "целочисленная переменная"
fun(x::String) = "строка"
fun(x) = "не строка и не целое число"
Вызов функции fun() с разными аргументами приводит к разным ре-
зультатам:
fun(1)
>"целочисленная переменная"
fun("1")
>"строка"
fun(1.0)
>"не строка и не целое число"
```

Если функцию с аргументом заданного типа найти не удается, Julia выдает сообщение об ошибке.

Выбор функции (диспетчеризация) происходит во время выполнения программы, а не во время компиляции.

Более подробно о множественной диспетчеризации можно прочитать здесь (статья "Непостижимая эффективность множественной диспетчеризации")

Векторизация

В языке Julia предусмотрена возможность векторизации (broadcasting) арифметических операций и функций, т. е. замена числа на вектор. Точка добавляется перед символами + , - , *, ^, но после имени функции.

Примеры.

Пусть `x` - массив чисел.

`x.^2` - возвести в квадрат все элементы массива x.

Все элементы массива можно умножить на число и без векторизации, например, `2x`. Однако, возможны некоторые тонкости. Если элементы массива нужно умножить на вещественное число (например, 2.0), то умножение на 2. приводит к неоднозначности, поскольку непонятно, как трактовать точку - как десятичный разделитель или как символ векторизации. `2 .*x` - умножить все элементы массива x на 2. Перед точкой в данном случае должен быть пробел, иначе возникает проблема, как трактовать эту точку - как десятичную для 2 или как символ векторизации. Однако если записать это выражение так
`2.0.*x`, то пробел перед точкой не обязателен.

Если `a,b` – векторы одинаковой длины, то допустимо выражение `minmax.(a,b)`, в этом случае результатом будет вектор-кортеж, содержащий минимальные и максимальные значения каждой пары элементов двух векторов.

Выражение типа `sin.(a)` позволяет вычислить синус каждого элемента массива `a`.

Чтобы применить функцию `f=(x)->x^3` к элементам массива `a`, можно использовать выражение `f.(a)`, результатом будет массив чисел.

Выражение `a==b` позволяет сравнить векторы `a` и `b`, результатом будет переменная типа Bool (true|false). Аналогичное выражение с точкой `a.==b` дает возможность сравнить содержимое векторов почленно, в этом случае результатом является массив типа Bool, поля которого содержат результаты сравнения соответствующих ячеек векторов (1 – true, 0 – false).

Области видимости переменных

Блок в Julia определен конструкциями `function`, `for`, `while`, `if/else`, `do`, `try/catch` и заканчивается оператором `end`. По умолчанию каждый блок определяет свою «область видимости» переменной. Область видимости определяет возможность модификации переменной. Иными словами, переменная `x` вне блока и внутри блока — две разные переменные. Переменная, определенная в теле функции не видна вне этой функции. Переменную внутри блока можно отождествить с внешней, если использовать ключевое слово **`global`**.

Пример

```
function sum_to(n)
s = 0 # новая локальная переменная
for i = 1:n
s = s + i # присвоить значение локальной переменной
end
return s # та же локальная переменная
```

```
end
sum_to(3)
>6
s
```

Последняя строка (s) вызовет сообщение об ошибке: `UndefVarrError: s not defined` (переменная s не определена). Действительно, переменная `s` определена только внутри функции `sum_to(n)`.

В следующей функции переменная s объявлена глобальной, поэтому ее «видно» и вне функции.

```
function sum_to(n)
global s = 0 # новая глобальная переменная
for i = 1:n
s = s + i # присвоить значение глобальной переменной
end
return s # та же глобальная переменная
end
sum_to(4)
>10
s
>10
```

И еще один пример, который иллюстрирует возможность передачи глобальных переменных из одной функции в другую

```
function test_2()
global n
println("n=$n")
end
function test_1(x)
global n=x
test_2()
end
test_1(1)
>n=1
```

При необходимости сделать переменную локальной используется ключевое слово **`local`**.

Два примера.

```
function f1(n)
     x = 0
     for i = 1:n
          x = i
     end
     x
```

```
end
```

`f1(10)` возвратит 10, поскольку в теле цикла используется переменная `x`, объявленная в теле функции.

```
function f2(n)
     x = 0
     for i = 1:n
          local x
          x = i
     end
x
end
```

В этом случае `f2(10)` возвратит 0, поскольку переменная `x` в теле цикла объявлена локальной.

Документация по этому разделу не является устоявшейся и иногда изменяется. Более подробно см.
https://docs.julialang.org/en/v1/manual/variables-and-scoping/

Дополнительная информация

В языке Julia допустима запись выражений вида +(a,b), данное выражение эквивалентно a+b. Данная форма допустима для всех стандартных операций, арифметических и логических.

Приоритет выполнения операций в Julia не отличается от общепринятого. Вначале выполняются операции возведения в степень, затем деление-умножение, затем сложение-вычитание, в последнюю очередь выполняются операции сравнения. Более подробно, см. (https://docs.julialang.org/en/v1/manual/mathematical-operations/#Operator-Precedence-and-Associativity-1).

В Julia допустимо опускать знак умножения в выражениях типа 10*a которое можно записать так: 10a или10.0a.

Допустимо использовать логические переменные как числовые (true это 1, false – 0)

```
a = true
b = false
c = 1.0
```

Результатом операции a+c является 2.0.

2. Использование библиотек (packages)

Целесообразность использования языка программирования во многом зависит от его назначения, возможностей и прикладных библиотек, которые образуют так называемую экосистему языка. В свою очередь, экосистема языка характеризуется количественными и качественными показателями. Причем, если получить ко-

личественную оценку довольно просто (достаточно посчитать число библиотек), то оценить их качество гораздо сложнее, поскольку качество прикладной библиотеки определяется целым рядом факторов, в частности, удобством использования, надежностью, быстродействием, кругом решаемых задач. С этой точки зрения библиотеки разделяют иногда на "зрелые" и "незрелые". Первые доведены до высокой степени готовности, вторые находятся в стадии разработки.

Система программирования Julia предоставляет возможность использования целого ряда универсальных и специализированных библиотек. Чтобы использовать функции из какой-либо библиотеки, нужно ввести команду `using PackageName` (вместо PackageName нужно ввести имя соответствующей библиотеки). Например, using LinearAlgebra. Если на экран выводится сообщение о том, что библиотека не найдена, следует попытаться установить ее командой **`add`** в режиме работы с библиотеками (`add PackageName`). Для перехода в режим работы с библиотеками нужно ввести символ **"]"** (см. Диалоговое окно).

Удаление библиотеки производится командой **`rm`** в режиме работы с библиотеками (`rm PackageName`).

Для просмотра списка установленных библиотек нужно выполнить команду **`status`** в режиме работы с библиотеками.

Работа с таблицами (библиотека DataFrames)

Переменная типа DataFrame (фрейм данных) предназначена для хранения и представления данных в форме таблицы. DataFrame похож на матрицу, но имеет ряд особенностей. Для работы с переменными этого типа необходимо подключить библиотеку DataFrames (`using DataFrames`).

DataFrame можно рассматривать как базу данных, находящуюся в оперативной памяти компьютера, с которой удобно работать. DataFrame содержит столбцы, каждый из которых имеет свой тип, причем обращение к данным можно осуществлять по имени столбца. Таким образом, в одном столбце могут содержаться формулы химических веществ (тип String), в другом — их молярные массы (тип Float32). Поля таблицы могут не содержать значений (значение `missing`).

Пример создания переменной типа DataFrame

```
using DataFrames
df = DataFrame()
df[!,:C1] = 10:10:40
df[!,:C2] = [log(10), pi, sqrt(5), 63]
df[!,:C3] = [true, false, true, false]
show(df)
```

Первая строка таблицы содержит заголовки столбцов и типы соответствующих переменных, первый столбец (вспомогательный) содержит номера строк.

Отметим, что данную таблицу можно создать и таким образом:

```
df = DataFrame(C1 = 10:10:40, C2 = [log(10), pi, sqrt(5), 63],
C3 = [true, false, true, false])
```

Обращение к данным столбца возможно по его номеру или по имени

```
show(df.C1)
show(df[!,3])
show(df[!,:C3])
```

Можно изменить значение ячейки таблицы (с учетом типа данных)

```
df.C1[3]=33
```

Обращение к данным строки возможно по её номеру: `df[2, :]`

Вывести данные строк 1 и 2: `df[1:2,:]`

Дальнейшая детализация (содержимое третьего столбца): `df[1:2,:C3]`

Вывести данные строк 3 и 4 колонок C1 и C3 таблицы df можно так

```
df[3:4,[:C1,:C3]]
```

Вывести первые n строк таблицы можно с использованием функции `first()`, например так: `first(df,3)`. Соответственно, чтобы вывести информацию n последних строк, можно использовать функцию `last(): last(df,2)`.

Имена колонок таблицы в виде массива можно получить при помощи функции `names(): names(df)`.

Тип колонки можно выяснить следующим образом: `eltype(df.C1)`. Типы всех колонок: `eltype.(eachcol(df))`.

Добавить строку (в конец таблицы) можно при помощи функции `push!()`: `push!(df,[1,1.1,false])`.

Функция `describe(df)` позволяет получить информацию статистического характера о тех колонках таблицы, для которых эта информация имеет смысл: минимальное, максимальное, медианное, среднее значения, число полей, не содержащих данные и т. д.

Выбрать одну или несколько колонок можно с использованием функции **`select()`**. Например, выбрать колонки C2 и C3 таблицы df можно так: `select(df,:C2,:C3)` или так: `select(df,"C2","C3")`.

Можно наоборот исключить одну или несколько колонок, например, колонки C2 и C3: `select(df,Not([:C2,:C3]))`.

Функция **`sort()`** позволяет упорядочить строки таблицы по содержимому заданной колонки. Например, чтобы отсортировать строки таблицы df по содержимому колонки C2, нужно выполнить команду `sort(df, :C2)`.

Загрузить данные из файла типа CSV можно с использованием библиотеки CSV.

Пример (в качестве разделителя использована точка с запятой):

```
using DataFrames, CSV
fname = "mydata.csv"
df=CSV.File(fname, delim = ';') |> DataFrame
```

или так (более быстрый способ)

```
df=CSV.read(fname, DataFrame, delim = ';')
```

или так

```
df=DataFrame(CSV.File(fname))
```

Записать данные таблицы в файл типа CSV можно так

```
CSV.write("mynewdata.csv", df, delim = ';')
```

Если файл `a.csv` содержит символы кириллицы, загружать его лучше таким образом

```
CSV.File(open(read, "a.csv", enc"WINDOWS-1251")) |> DataFrame
```

Библиотека DataFrames предоставляет возможность делать выборки из таблицы

```
df[df[!,:C3].== true,:]
```

Пример. Пусть дана такая таблица `a.csv`, в которой содержатся сведения о каких-то параметрах веществ из разных источников

```
"fml","value","source"
"O(g)",1, "source 1"
"O2(g)",20.0, "source 4"
"O2(g)",20.8, "source 5"
"O2(g)",20.2, "source 6"
"H(g)",15.0, "source 1"
"O(g)",1.1, "source 2"
"H2(g)",-12.0,"источник 1"
"OH(g)",115.0,"источник 1"
"H2O(c)",965.0,"источник 1"
"H2O(g)",340.0,"источник 1"
"O(g)",0.9,"источник 3"
"O2(g)",19.5,источник 1
```

Если текст таблицы сохранен в кодировке UTF-8, ее можно загрузить в память так:

```
d=DataFrame(CSV.File("a.csv")).
```

В противном случае (не UTF-8 - кириллица), для загрузки таблицы в память (работа в среде Windows) можно использовать одну из команд

```
d=DataFrame(CSV.File(read("a.csv", enc"windows-1251")))
d=DataFrame(CSV.File(open(read,"a.csv", enc"windows-1251")))
```

Выбрать индексы всех параметров для данного вещества (например, O_2(g)) можно так

```
idx=findall(x->x=="O2(g)", d.fml)
Ответ: 2, 3, 4, 12
```

Выбрать эти строки из таблицы можно так

```
d[idx,:]
```

Row	fml	value	source
	String	Float64	String
1	O2(g)	20.0	source 4
2	O2(g)	20.8	source 5
3	O2(g)	20.2	source 6
4	O2(g)	19.5	источник 1

или так

```
d[d.fml.=="O2(g)",:]
```

или так (использование функции **filter()**)

```
filter(:fml=>x->x=="O2(g)",d)
```

или так (использование функции subset**()**)

```
subset(d,:fml=>ByRow(x->x=="O2(g)"))
```

Более подробную информацию о работе с библиотекой DataFrames можно найти по адресу

https://juliadata.github.io/DataFrames.jl/stable/man/getting_started/

Задачи

1. Создайте текстовый файл, содержащий следующую информацию:

химическая формула вещества, список химических элементов, из которых образовано вещество, число атомов каждого химического элемента. Напишите четыре функции, которые позволяют осуществлять выборку информации из этого файла

- по заданному списку элементов, из которых образовано вещество, предполагается, что формула вещества может содержать только элементы из этого списка, все или часть;

- по списку элементов, все из которых одновременно присутствуют в формуле вещества;

- по списку элементов, которых нет в формуле вещества;

- по списку элементов, которые есть в формуле вещества, предполагается, что в формуле могут быть и другие элементы.

В функцию передается переменная типа DataFrame, функция возвращает массив формул веществ.

Пример списка веществ: C, H, O, N, H2, O2, N2, CO, NO, OH, CO2, H2O, CH4, C2H2.

Использование библиотек CSV и DelimitedFiles

Файл типа CSV (от англ. Comma-Separated Values) содержит упорядоченный набор данных, которые отделены друг от друга разделителем. В качестве разделителя можно использовать запятую, точку с запятой, табуляцию и др. Файлы имеют структуру таблицы, первая строка может содержать заголовки колонок. Например

```
Temperature;HeatCapacity
0;0
20;1.234
…
300;50.678
```

Данные, хранящиеся в таком виде можно прочитать с использованием функции `readdlm` библиотеки `DelimitedFiles`:

```
fname = "T_Cp.csv"
using DelimitedFiles
data = DelimitedFiles.readdlm(fname, ';')
```

Первый аргумент функции — имя файла данных, второй — символ разделитель полей. `data` будет содержать массив строк типа Any вида

```
"Temperature"  "HeatCapacity"
0.0            0.0
20.0           1.234
…
```

Иными словами, заголовок был отнесен к данным. Однако функция `readdlm` имеет несколько настраиваемых параметров, которые обеспечивают гибкость чтения данных из файла. В нашем случае можно использовать такой вариант

```
data = DelimitedFiles.readdlm(fname, ';', Float64, '\n', header=true)
```

Указаны тип данных (`Float64`), разделитель строк (`'\n'`) , наличие заголовка (`header=true`). Теперь data содержит данные отдельно в виде кортежа `([0.0 0.0; 20.0 1.234…], AbstractString["Temperature" "HeatCapacity"])`. `data[1]` содержит массив данных типа `Float64`, `data[2]` – заголовки колонок типа `AbstractString`. При этом `data[1][1,1]` содержит 0.0, `data[1][2,1]` содержит

20.0, `data[1][2,2]` содержит 1.234, `data[2][1,1]` содержит `"Temperature"`, `data[2][1,1]` содержит `"HeatCapacity"`.

Записать данные в файл типа CSV можно так

```
writedlm("my_data", data, ';')
```

С файлами типа CSV можно работать, используя библиотеки `CSV` и `DataFrames`. Пример

Пусть файл с именем `T_Cp.csv` содержит такой текст

```
T;Cp
10.0;1.0
20;2.0
30;3.0
```

Ввести информацию из этого файла в память компьютера можно так

```
fname = "T_Cp.csv"
```

1. using CSV:

```
data=CSV.File(fname)
```

`data` содержит массив вектор-столбцов, тип каждого из которых определяется автоматически.

Если в качестве разделителя полей использован один из символов `','`, `'\t'`, `' '`, `'|'`, `';'`, разделитель определяется автоматически. По умолчанию используется разделитель `','`.

При необходимости разделитель можно указать: `delim=X`, `X` - символ или строка. Поля первой строки воспринимаются как имена векторов. Если строка заголовка отсутствует, для чтения данных можно использовать такой способ:

```
data=CSV.File(fname, header=false),
```

в этом случае имена вектор-столбцов будут иметь вид `Column1, Column2,...`

2. using CSV, DataFrames:

```
data = CSV.File(fname, delim = ';') |> DataFrame
```

или так:

```
data=DataFrame(CSV.File(fname, delim = ';'))
```

Во втором случае `data` имеет тип `DataFrame`. Значения температур хранятся в столбце `data.Temperature`, столбец `data.HeatCapacity` содержит значения теплоемкостей типа `Float64`. `data.Temperature[2]` содержит `20.0`, `data.HeatCapacity[2]` содержит `1.234`.

Работать с CSV файлом с использованием библиотеки `CSV` удобнее, чем с использованием библиотеки `DelimitedFiles`.

Более подробная информация о библиотеке CSV приводится здесь
https://juliadata.github.io/CSV.jl/stable/#CSV.jl-Documentation-1

Построение графиков (библиотека Plots)

Подробная информация на сайте https://docs.juliaplots.org/latest/

Библиотека Plots обеспечивает доступ к специализированным библиотекам (**backends**), предназначенным для построения графиков, каждая из которых имеет свои особенности: GR, PyPlot, Plotly и т.д. (https://docs.juliaplots.org/latest/backends/) Переключение с одной библиотеки на другую осуществляется вызовом соответствующей функции: `gr(), plotly(), pyplot()`...

Подключение библиотеки: `using Plots`

Если какая-либо требуемая специализированная библиотека для построения графиков, например, `PyPlot`, не установлена на компьютере, ее нужно установить командой `add`.

Простейший способ построения графика имеет вид `plot(x,y)`. Пример

```
x=collect(1:1:100.0)
y=x.^0.5
plot(x,y)
```

Если нужно добавить на этот график еще один, используется вызов функции с восклицательным знаком: plot!

```
x = 1:10; y = rand(10);
plot!(x, y)
```

Если нужно добавить надписи на оси, то (выделено цветом)

```
plot(x,y, xlabel="X axis", ylabel="Y axis")
```

Выравнивание текста подписей. Горизонтальная ось

`xguidefonthalign=`

`:right` - вправо, `:left` - влево, `:center` - по центру.

Вертикальная ось

`yguidefontvalign=`

`:top` - вверх, `:bottom` - вниз, `:center` - по центру.

Если нужно добавить название серии точек (линии графика), то

```
plot(x,y,xlabel="X axis", ylabel="Y axis", label="Series name")
```

Добавить цвет

```
plot(x,y,xlabel="X axis", ylabel="Y axis", label="Series name",
color=:red)
```

Изменить толщину линии

```
plot(x,y,width=3)
```

Тип линии: `linestyle =` выбрать из списка `[:auto, :solid, :dash, :dot, :dashdot, :dashdotdot]`

Сгруппировать несколько графиков, рисунок 2:

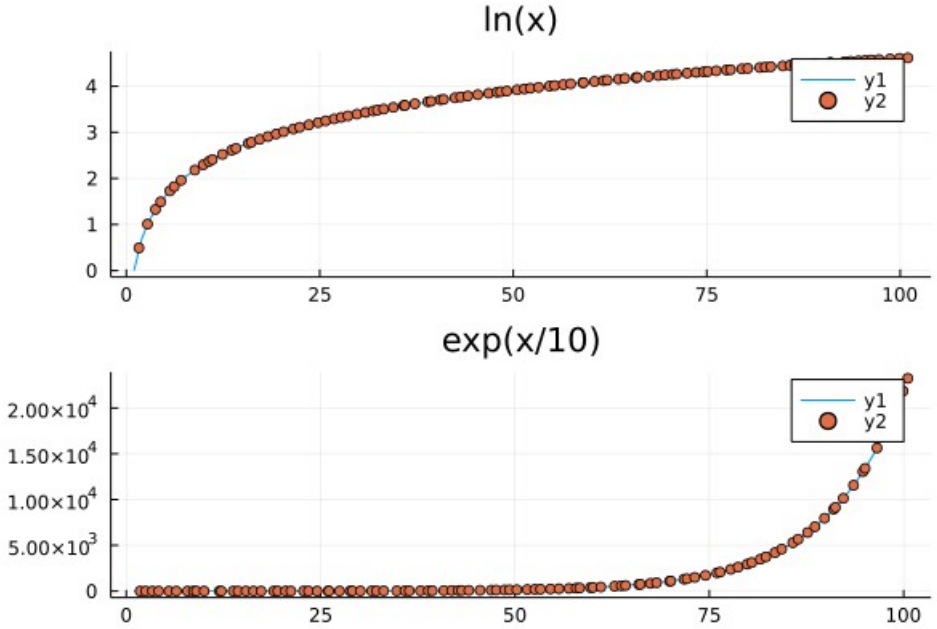

Рис.2. Группировка нескольких графиков

Ниже приводится текст программы к рисунку 2. Если убрать точку с запятой в квадратных скобках первой строки (`L = @layout [a b]`), то графики расположатся не вертикально, а горизонтально.

```
L = @layout [a ; b]
x=collect(1:100); y=log.(x);
p1 = plot(x,y, title="ln(x)");
x = x .+ rand(100); y=log.(x);
p1 = plot!(x,y,seriestype = :scatter);
x=collect(1:100); y=exp.(x./10);
p2 = plot(x,y, title="exp(x/10)");
x = x .+ rand(100); y=exp.(x./10);
p2 = plot!(x,y,seriestype = :scatter);
plot(p1, p2, layout = L)
```

Использовать точки вместо линий

```
plot(x, y, seriestype = :scatter)
```

или так

```
scatter(x, y, title = "My Scatter Plot")
```

или так, рисунок 3:

```
v=[[0.0,1.0],[1.0,1.0],[0.0,0.0],[1.0,0.0],[0.5,0.5]],
p = plot([Singleton(vi) for vi in v])
```

Размер точки задается параметром `markersize`, например, `markersize=2`.

Обозначения (legend) на графике можно отключить:

```
plot(x,y, legend=false)
```

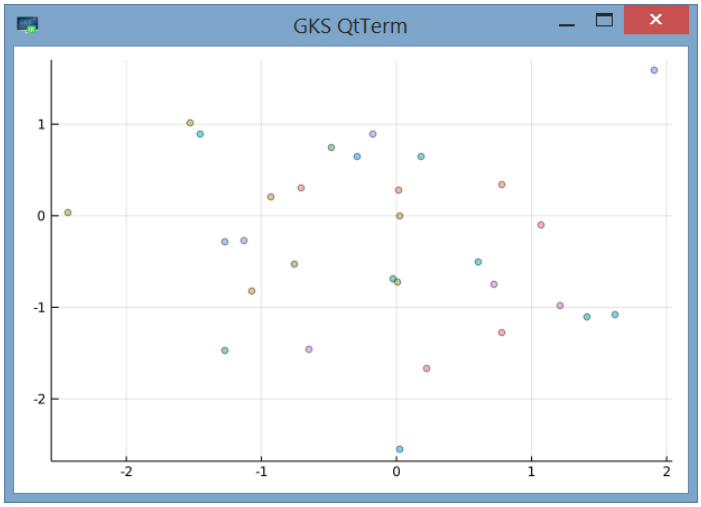

Рис. 3. Изображение отдельных точек

Положение legend можно изменить с помощью ключевого слова:

```
plot(x,y, legend=position)
```

`position` может принимать одно из следующих значений

```
:right, :left, :top, :bottom, :inside, :best, :legend, :topright,
:topleft, :bottomleft, :bottomright, :outerright, :outerleft, :out
ertop, :outerbottom, :outertopright, :outertopleft, :outerbottom-
left, :outerbottomrigh
```

```
plot(x,y,legend=:right)
```

Другие типы графиков (https://docs.juliaplots.org/latest/generated/gr)

```
bar(randn(10))
histogram(randn(1000), bins=:scott, weights=repeat(1:5,outer=200))
```

Сохранить график в файл в формате .png

```
savefig("myplot.png")
```

или

```
png("myplot.png")
```

Сохранить график в файл в формате .pdf

```
savefig("myplot.pdf")
```

или так

```
pdf("myplot.pdf")
```

Подробнее об атрибутах графиков

https://docs.juliaplots.org/latest/generated/attributes_series/

См. также статью "Tips to create beautiful, publication-quality plots in Julia"

https://nextjournal.com/leandromartinez98/tips-to-create-beautiful-publication-quality-plots-in-julia

Для построения графиков в полярных координатах нужно при вызове функции `plot()` указать `proj=:polar`.

Пример. Построить график следующей зависимости: $r(t)=\sin(6t)$, $t\in[0;2\pi]$ в полярных координатах.

```
t=collect(range(0,2pi, 100))
rt=sin.(6.0.*t)
plot(t,rt,proj=:polar, legend=false)
```

Если нужно построить несколько графиков в цикле, то рекомендуется вызвать функцию `gui()` после завершения цикла:

```
x=rand(10); y=rand(10); plot(x,y)
for i in 1:3
x=rand(10); y=rand(10); plot!(x,y)
end;
gui()
```

или так

```
x=rand(10); y=rand(10); plt=plot(x,y)
for i in 1:3
x=rand(10); y=rand(10); plt=plot!(x,y)
end;
gui(plt)
```

Задачи

1. Постройте графики функций (примеры заимствованы с сайта `grafikus.ru`)

2. Окружность: $x=\sin(t)$, $y=\cos(t)$, $t\in[0;2\pi]$.

3. Спираль: $x=t\sin(t)$, $y=t\cos(t)$, $t\in[0;5\pi]$.

4. Дельтоида: $x=2\cos(t)+\cos(2t)$, $y=2\sin(t)-\sin(2t)$, $t\in[0;2\pi]$.

5. Астроида: $x=2\sin^3(t)$, $y=2\cos^3(t)$, $t\in[0;2\pi]$.

6. Гипоциклоида: $x=20(\cos(t)+\cos(5t)/5)$, $y=20(\sin(t)-\sin(5t)/5)$, $t\in[0;2\pi]$.

7. Кардиоида: $x=(1+\cos(t))\cos(t)$, $y=(1+\cos(t))\sin(t)$, $t\in[0;2\pi]$.

8. Эпициклоида: $x=8(\cos(t)-\cos(4t)/4)$, $y=8(\sin(t)-\sin(4t)/4)$, $t\in[0;2\pi]$.

9. Полярная роза: $r(t)=\sin(74t)$,$t\in[0;8\pi]$.

10. Подготовьте текстовый файл, содержащий значения x и y. Напишите функцию, в которую передается имя файла данных, производится считывание значений x и y, строится график функции y(x).

Аппроксимация данных (библиотека LsqFit)

Подключение библиотеки: `using LsqFit`

Для того чтобы найти коэффициенты аппроксимирующей функции, нужно определить модель, т. е. аппроксимирующую функцию. Подробное описание библиотеки можно найти по адресу https://julianlsolvers.github.io/LsqFit.jl/latest/tutorial/

Для расчета коэффициентов используется метод нелинейных квадратов (Levenberg-Marquardt).

Пример. Пусть задан набор пар значений (x, y), который необходимо аппроксимировать соотношением вида $y = a \cdot e^{b \cdot x}$, где a и b неизвестные коэффициенты. Модель для библиотеки LsqFit в этом случае имеет вид

```
m(t, p) = p[1] * exp.(p[2] * t)
```

Для вызова функции аппроксимации требуется задать начальные значения неизвестных

```
p0 = [0.5, 0.5]
```

Вызов функции имеет вид

```
fit = curve_fit(m, x, y, p0)
```

Если решение найдено, значения коэффициентов содержатся в массиве `fit.param`, массив `fit.resid` содержит значения отклонений функции от исходных значений, `stderror(fit)` содержит стандартные погрешности каждого параметра, `fit.estimate_covar(fit)` вычисляет ковариационную матрицу.

Функция `margin_error()` вычисляет произведение стандартной погрешности параметров на коэффициент Стьюдента (`critical value`) с заданным уровнем значимости (по умолчанию, 5%). Величину `margin_error`, с уровнем значимости 10% можно найти так:

```
margin_of_error = margin_error(fit, 0.1)
```

Если требуется вычислить доверительный интервал с 10% уровнем значимости (alpha = 0.1), нужно выполнить команду `confidence_interval(fit, alpha)`, которая вычислит оценку `parameter value ± (standard error * critical value` с использованием t-распределения).

Пример модели для аппроксимации данных о теплоемкости с использованием функций Эйнштейна-Планка

```
function m(t,p)
r=3*8.3144626
global n
m = 0
for i in 1:n
  x=p[2*i-1]./t
  e=exp.(x)
  y=r .*p[2*i].*(e .* x.^2)./(e .- 1).^2
  m = m .+ y
end
return m
end
```

Значение числа функций `n` и начальные значения коэффициентов нужно задать перед вызовом функции аппроксимации. Например так

```
n=3
p0=ones(2*n)
fit = curve_fit(m, x, y, p0)
```

Эту же функцию можно записать в таком виде

```
function m(t,p)
r=3*8.3144626
global n
m = zeros(0)
for j in 1:length(t)
y=0.0
for i in 1:n
  x=p[2*i-1]/t[j]
  e=exp(x)
  y+=r*p[2*i]*(e*x^2)/(e- 1)^2
end
 push!(m,y)
end
return m
end
```

Рассчитать среднеквадратичное отклонение можно с использованием функции

`rms(x)=norm(x)/sqrt(length(x)),`

для работы которой требуется библиотека LinearAlgebra. Пример обращения к функции

`rms(y-ycalc)`, где `ycalc=m(x,fit.param)`.

Расчет с использованием весовых коэффициентов

В этом случае, нужно задать массив весовых коэффициентов `wt`, обращение к функции имеет вид

```
fit = curve_fit(m, tdata, ydata, wt, p0)
```

Задачи

1. Напишите программу, которая считывает из тестового файла координаты точек (*x*,*y*), аппроксимирует их

a) полиномом степени *n* (*n* задается);

b) с использованием функций Эйнштейна-Планка, число функций *n* задается;

строит графики зависимости *y*(*x*) - по точкам и с использованием аппроксимирующей функции.

2. Аппроксимация данных о теплоемкости с использованием функций Планка-Эйнштейна. Подготовить текстовый файл `Cp_T.txt вида`

```
T;Cp
4.81;0.009
6.14;0.08
7.97;0.011
9.4;0.019
...
```

В качестве разделителя использован символ ";".

Определить коэффициенты аппроксимирующей функции и построить график, на котором изображены исходные точки и результат аппроксимации.

Поиск корней уравнения

Один из простейших способов поиска корней уравнения предполагает использование библиотеки Roots, (подключение библиотеки: `using Roots`), в которой реализовано несколько алгоритмов поиска корня: с вычислением и без вычисления производных, для заданного интервала и для заданного начального значения. Можно найти один корень или все корни на заданном интервале.

Примеры

```
f(x) = exp(x) - x^4
## поиск корня на интервале
find_zero(f, (8,9), Bisection())    # решение 8.6131694564414
find_zero(f, (-10, 0))              # -0.8155534188089607
# поиск корня с заданным начальным значением
find_zero(f, 3)                     # 1.4296118247255556
```

```
# найти все корни
find_zeros(f, -10, 10)              # -0.815553, 1.42961 и 8.61317
```

Для минимизации числа вызовов функции рекомендуется использовать параметр `FalsePosition()`:

```
find_zero(f, (-10, 0), FalsePosition())
```

Задать погрешность расчета корней можно с использованием ключевого слова atol:

```
find_zero(f, (-10, 0), atol=1.e-8, FalsePosition())
find_zero(f, (-10, 0), atol=eps(), FalsePosition())
```

Более подробное описание можно найти по адресу https://github.com/JuliaMath/Roots.jl

Более универсальный способ поиска корней уравнения можно реализовать с использованием метода Ньютона, в соответствии с которым решение определяется итерационно по формуле

$x_{i+1}=x_i-f(x_i)/f'(x_i)$.

Метод Ньютона предполагает необходимость задания начального приближения x_0.

Пример. Найти корень уравнения `exp(x) - x^4 = 0`

```
f(x) = exp(x) - x^4
```

с использованием функции `findroot()`:

```
function findroot(f,x0,tol)
diff_complex(f, x; h=1e-20) = imag(f(x + h*im)) / h
x=x0
while true
  d=diff_complex(f, x)
  fv=f(x)
  abs(fv) < tol && break
  x=x-fv/d
  println("findroot:  x=$x,  f(x)=$fv")
end
return x
end
```

Результат обращения к функции `findroot()` будет зависеть от выбора начального приближения. Если x_0=1, будет найден корень 1.42961, если x_0=0, будет найден корень -0.815553, наконец, если x_0=10, будет найден корень 8.61317.

К сожалению, метод Ньютона не всегда применим. Например если попытаться найти с его помощью корень уравнения `log(x)-1=0 (f1(x)=log(x)-1)`, выбрав в качестве начального приближения 10, мы получим сообщение об ошибке DomainError при обращении к функции `findroot()`, поскольку на очередном шаге значение *x* станет отрицательным. Проблему можно решить, если в формулу Ньютона ввести параметр релаксации

$x_{i+1}=x_i-\alpha f(x_i)/f'(x_i), \alpha<1$:

```
function findroot1(f,x0,tol,alpha)
x=x0
while true
  d=diff_complex(f, x)
  fv=f(x)
  abs(fv) < tol && break
  x=x-alpha*fv/d
  println("findroot:  x=$x,  f(x)=$fv")
  sleep(1)
end
return x
end
findroot1(f1,10.0,1.e-8,0.1)
```

Однако при этом возрастет время вычислений (α=0.1). Если задать α=0.5,

```
findroot1(f1,10.0,1.e-8,0.5)
```

решение будет найдено быстрее.

Таким образом, при использовании метода Ньютона важно удачно выбрать начальное приближение и задать при необходимости разумное ограничение на величину итерационного шага.

Задачи

1. Найти корни функции $x^2+e^x-10=0$ на интервалах [0,10], [-10,0], [-10,10].

Автоматическое дифференцирование функций

Значение производной можно рассчитать аналитически или с использованием одной из функций численного дифференцирования:

```
diff_forward(f, x; h=sqrt(eps(Float64))) = (f(x+h) - f(x))/h
diff_central(f, x; h=cbrt(eps(Float64))) = (f(x+h/2) - f(x-h/2))/h
diff_backward(f, x; h=sqrt(eps(Float64))) = (f(x) - f(x-h))/h
diff_complex(f, x; h=1e-20) = imag(f(x + h*im)) / h
```

В состав экосистемы Julia входят несколько библиотек, предназначенных для автоматического дифференцирования функций (https://juliadiff.org/). Рассмотрим примеры использования одной из них — ForwardDiff.

1. Расчет производной функции одной переменной включает создание функции, указание значения величины, при которой производится вычисление производной и вызов функции ForwardDiff.derivative():

```
f(x)=exp(10x)
c=1
ForwardDiff.derivative(f,c)
```

Значение второй производной можно посчитать, добавив вторую функцию вида

```
function f2(x)
return ForwardDiff.derivative(f,x)
end
```

следующий вызов которой возвращает значение второй производной `ForwardDiff.derivative(f2,c)`

2. Расчет производной нескольких переменных (градиент функции) производится аналогично, но с использованием функции `ForwardDiff.gradient()`:

```
h(x) = sin(x[1]) + x[1] * x[2] + sinh(x[1] * x[2])
x = [1.4 2.2]
ForwardDiff.gradient(h,x)
```

3. Расчет гессиана функции производится с использованием вызова функции `ForwardDiff.hessian()`:

```
f(x)=log(10x[1])-sin(x[2])
x = [10.4 20.2]
ForwardDiff.hessian(f,x)
```

4. Расчет якобиана функции производится с использованием вызова функции `ForwardDiff.jacobian()`:

```
function ff(x)
#set array of functions
return [
x[1]*x[2]^2*x[3]^3*x[4]^4;
x[1]+2x[2]+3x[3]+4x[4];
x[1]*x[2]/x[3]/x[4];
]
end
x=[0.1; 1.0; 1.5; 2.0]
ForwardDiff.jacobian(ff,x)
```

Более интересный случай, когда в функцию дифференцирования нужно передать параметры:

```
lbs = [1.0, 2.0]
ubs = [2.0, 3.0]
t = [1.0; 2.0; 0.1]
function hfunc(t, lbs, ubs)
    n = div(length(t), 2)
    h = zero(t)
    h[end] = t[end]
    for i = 1:n
        if t[i] < 0.5
            h[i] = -lbs[i]
            h[i + n] = ubs[i]
        end
    end
    return h
end
j = ForwardDiff.jacobian(x -> hfunc(x, lbs, ubs), t)
```

Однако при таком обращении к функции дифференцирования выражение в ее сигнатуре будет компилироваться при каждом обращении, что приводит к увеличению времени вычислений. Поэтому, целесообразнее определить дополнительную функцию `f1(x)`:

```
f1(x)=hfunc(x, lbs, ubs)
j = ForwardDiff.jacobian(f1, t)
```

Интегрирование функций

Для численного интегрирования функций можно использовать несколько библиотек (например, `Cubature.jl`, `QuadGK.jl`).

Библиотека Cubature.jl позволяет интегрировать численно одномерные и многомерные интегралы. Подробное описание о реализованных методах вычислений приводится на сайте разработчика, ссылка на который приведена выше. Для ее использования нужно установить библиотеку Cubature.jl.

Обращение к функции интегрирования имеет вид

```
(val,err) = hquadrature(f::Function, xmin::Real, xmax::Real;
                        reltol=1e-8, abstol=0, maxevals=0)
```

где

`val` - значение интеграла,

`err` - погрешность вычислений,

`f` - подынтегральное выражение,

`xmin, xmax` - пределы интегрирования,

`reltol` - допустимая относительная погрешность,

`abstol` - допустимая абсолютная погрешность,

`maxevals` - допустимое число вычислений значения функции.

Пример. Рассчитать значение интеграла функции x^3 на интервале от 0 до 1 с выводом вычисляемых значений функции:

```
hquadrature(x -> begin println(x); x^3; end, 0,1)
```

Рассмотрим более сложный пример вычисления функции Дебая

$$\int_0^{\frac{\theta_D}{T}} \frac{x^4 e^x}{\left(e^x-1\right)^2} dx \quad .$$

```
function D(x)
ex=exp(x)
return x^4*ex/(ex-1)^2
end
x=1
hquadrature(D, 0,x, abstol=1e-8)
```

Дата и время (библиотека Dates)

Функции для работы с датами и временем содержатся в библиотеке Dates, подключить которую можно с использованием команды `using Dates`. Иерархия типов данных для хранения даты и времени приводится на рисунке 4.

Основные функции для работы с датой и временем

`today()` - возвращает текущую дату;

`now()` - возвращает текущие значения даты и времени;

`Time(now())` - возвращает текущее время.

Превратить текущую дату в строку можно так:

```
Dates.format(today(),"dd-mm-yyyy")
```

Более подробное описание библиотеки Dates можно найти по адресу https://en.wikibooks.org/wiki/Introducing_Julia/Working_with_dates_and_times

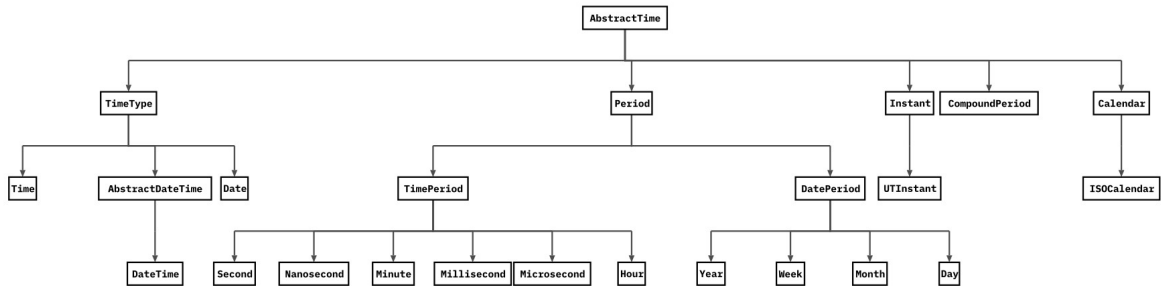

Рис. 4. Иерархия типов данных для хранения даты и времени

Если нужно определить время вычислений (разницу t2-t1), это можно сделать так:

```
t1=now() # начало интервала времени

..... # вычисления

t2=now()  # конец интервала времени
```

t2-t1 возвращает разницу в миллисекундах, причем эта разница имеет тип Millisecond!

Если нужно выполнить какие-то арифметические действия с этим временем, например, преобразовать его в секунды, разделив на 1000, нужно использовать функцию `Dates.value()`, которая возвращает число:

```
Dates.value(t2-t1)/1000
```

Случайные числа (библиотека Random)

Библиотека Random предназначена для генерации случайных чисел. Кратко рассмотрим три функции из этой библиотеки: `rand(), randn(), seed!()`.

`rand()` обеспечивает случайную выборку элемента из интервала [0,1) с использованием равномерного закона распределения. `rand(n)` обеспечивает случайную выборку n элементов (вектор значений). Интервал для выборки значений можно изменить: `rand(1.0:10.0)`. Можно задать интервал с шагом: `rand(1.0:0.1:10.0)`. Наконец, можно задать интервал как кортеж: `rand((1, 'z',"abc"))` или массив `rand([1,2,3])`.

`randn()` обеспечивает случайную выборку элемента с использованием нормального закона распределения (среднее 0, стандартное отклонение 1). Можно сгенерировать массив случайных величин с нормальным распределением: `randn(10)`.

Если требуется обеспечить воспроизводимость выборки, перед вызовом функций `rand()` и `randn()` нужно вызвать функцию `Random.seed!()` с одним и тем же числом, например, 100: `Random.seed!(100)`. После каждого вызова `Random.seed!(100)` функции `rand()` и `randn()` будут генерировать одну и ту же последовательность случайных величин.

Более подробно о библиотеке Random можно прочитать здесь: https://docs.julialang.org/en/v1/stdlib/Random/

Задача

1. Напишите функцию, которая вычисляет число π методом Монте-Карло с использованием генератора случайных чисел. На вход функции подается число пар случайных чисел.

2. Напишите две функции, которые вычисляют объем шара и цилиндра методом Монте-Карло.

Линейная алгебра (библиотека LinearAlgebra)

В состав дистрибутива Julia входит библиотека LinearAlgebra, которая содержит набор функций. Подключение библиотеки: `using LinearAlgebra`. Приведем обзор нескольких полезных функций из этого модуля и способах их применения, используя некоторые материалы учебника по линейной алгебре [6]

Норма вектора x может быть рассчитана одним из способов

```
1. norm(x)
2. sqrt(x'*x))
3. sqrt(sum(x.^2))
```

Среднеквадратичное отклонение можно посчитать так

```
rms = norm(x)/sqrt(length(x))
```

Расстояние между векторами x и y можно рассчитать так

```
d=norm(x-y)
```

Среднее значение набора чисел (вектора x) можно рассчитать с использованием функции

```
avg(x) = (ones(length(x)) / length(x))'*x
```

Расчет стандартного отклонения

```
stdev(x) = norm(x-avg(x))/sqrt(length(x))
```

Расчет угла между векторами x и y

```
ang(x,y) = acos(x'*y/(norm(x)*norm(y)))
```

Ранг матрицы a можно определить с использованием функции `rank()`:

```
r=rank(a)
```

qr-факторизация матрицы a осуществляется с использованием функции `qr()`, которая возвращает результат (матрицы ***Q*** и ***R***) в виде кортежа:

```
Q,R=qr(a)
```

Найти матрицу b, обратную к данной матрице a, можно с использованием функции `inv()`:

```
b=inv(a)
```

Другой способ обращения матриц предполагает использование результатов qr-факторизации

```
b=inv(R)*Q'
```

Решение системы линейных уравнений

Пусть дана система линейных уравнений вида ***Ax***=***b***, ***A*** – матрица n×n, ***b*** и ***x*** – векторы.

Простейшие способы решения системы линейных уравнений

```
x=A\b
x=inv(A)*b
```

Более сложный способ предполагает использование функции обратной подстановки

```
function back_subst(R,b)
  n = length(b)
  x = zeros(n)
  for i=n:-1:1
    x[i] = (b[i] - R[i,i+1:n]'*x[i+1:n]) / R[i,i]
  end
  return x
end;
```

Пример 1

```
A=rand(3,3)
b=rand(3)
x=A\b         # 1-й способ
x=inv(A)*b    # 2-й способ
```

```
Q,R=qr(A)     # 3-й способ
bb=Q'*b
back_subst(R,bb)
```

Пример 2

```
A=zeros(3,3);
A[1,:].=1; A[2,:].=2; A[3,:]=rand(3)
b=rand(3)
rank(A)
```

Ранг матрицы A равен 2, поэтому способы 1 и 2 использовать нельзя `(SingularException)`, однако можно использовать способ 3, если изменить процедуру `back_subst`:

```
function back_subst_m(R,b)
 n = length(b)
 x = zeros(n)
 for i=n:-1:1
   if abs(R[i,i]) > 1.e-15
     x[i] = (b[i] - R[i,i+1:n]'*x[i+1:n]) / R[i,i]
   else x[i] = 0
   end
   end
 return x
end;
Q,R=qr(A)        # 3-й способ
bb=Q'*b
back_subst_m(R,bb)
```

Решение примера 2 в этом случае найти можно.

Решение переопределенной системы линейных уравнений

Решить переопределенную систему линейных уравнений вида $\boldsymbol{Ax}=\boldsymbol{b}$, $\boldsymbol{A}$ – матрица m×n, $\boldsymbol{b}$ и $\boldsymbol{x}$ – векторы, с использованием метода наименьших квадратов можно несколькими способами.

Простейший способ

x=A\b

Более сложный способ предполагает выполнение qr-факторизации.

1. Выполнить qr-факторизацию матрицы A, найти Q и R, `(Q,R=qr(A))`
2. Рассчитать x по формуле $x=R^{-1}Q^{T}b$, `(x=inv(R)*Q'*b)`

Аппроксимация набора точек линейной комбинацией функций

Пусть имеется набор точек, определенных значениями координат (векторы $\boldsymbol{x}$ и $\boldsymbol{y}$). Требуется аппроксимировать зависимость y от x с использованием линейной

комбинации одной или нескольких функций одной переменной $f_j(x_i)$. Иными словами, нужно найти коэффициенты a_i зависимости (модели данных) вида

$$y=\sum_{j=1}^{m} a_j f_j(x) \quad .$$

Простейший способ предполагает использование для аппроксимации полиномиальную зависимость

$$y=a_0+a_1 x+a_2 x^2+...+a_n x^n \quad .$$

В этом случае целесообразно подключить библиотеку `Polynomials`, которая содержит функцию `fit(x,y,n), n` – степень полинома.

Пример 3. Сформируем массив точек, используя формулу

$y = 1 + 2(x+$*случайная погрешность*$)^2$

```
x=collect(1.0:10.0);
y=similar(x);
for i in 1:length(x)
y[i] = 1.0+2*(x[i]+rand())^2
end
```

Вычислим коэффициенты полинома степени 3 для данного набора точек:

```
p=fit(x,y,3)
> Polynomial(0.43279747887989134 + 5.915795408574915*x + 0.5537789693101738*x^2 + 0.12036222727678979*x^3)
```

Коэффициенты полученного полинома *p* можно извлечь так

```
c=coeffs(p)
>0.43279747887989134
 5.915795408574915
 0.5537789693101738
 0.12036222727678979
```

Массив значений полинома для данного вектора значений ***x*** можно рассчитать с использованием функции `polyval()`:

```
function polyval(c,x)
z=zeros(length(x))
for i in 1:length(c) z.=z.+c[i].*x.^(i-1); end
return z
end
polyval(c,x)
```

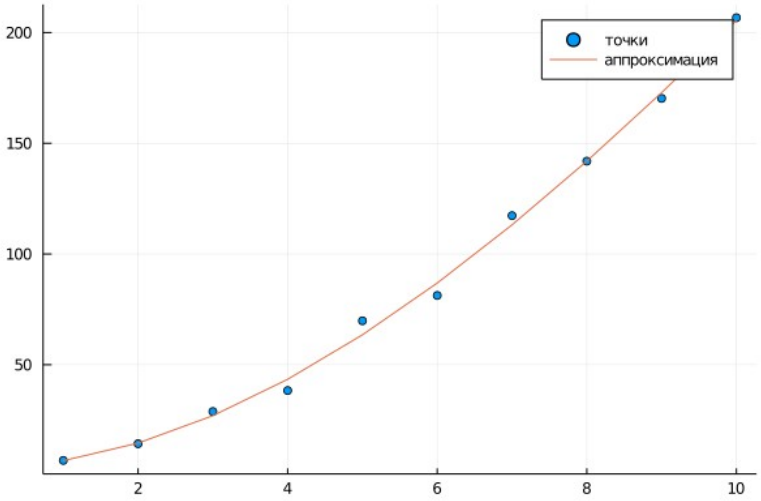

Рис.5. Исходные точки и результат аппроксимации

Изобразим заданные точки и результат аппроксимации на графике, рисунок 5.

```
scatter(x,y,label="точки")
plot!(x,polyval(c,x),label="аппроксимация")
```

Аппроксимирующую функцию можно собрать из произвольного набора функций одной переменной. Например, вышеприведенный набор точек можно попытаться аппроксимировать функцией вида

$$y=a_0+a_1x+a_2/x+a_3\ln(x) \quad .$$

Для расчета неизвестных значений a_i составим таблицу, первая колонка которой содержит значения x, последняя — значения y, в остальных колонках содержатся значения при коэффициентах a_i.

x_i	a_0	a_1	a_2	a_3	y_i
1	1	1	1	0	6.766986682...
2	1	2	0.5	0.69314718...	14.31765289...
3	1	3	0.333333333...	1.09861228...	28.85710713...
...	1	...	...	....	...
10	1	10	0.1	2.30258509	206.716760...

Выделенная цветом фона часть таблицы образует матрицу ***A***, последняя колонка — вектор **y**. Для расчета значений коэффициентов a_i нужно решить переопределенную систему линейных уравнений $\boldsymbol{Aa}=\mathbf{y}$:

```
a=A\y
```

Поскольку модель данных выбрана произвольно, чтобы проиллюстрировать идею, полученная погрешность аппроксимации в нашем примере будет довольно велика.

Если нужно учесть весовые коэффициенты w_i, строки матрицы ***A*** и соответствующие им значения вектора **y** нужно умножить на w_i.

Более сложный метод определения коэффициентов a_i основан на использовании метода неопределенных множителей Лагранжа. Если представить функцию Лагранжа в виде

$$\Lambda=\sum_{i=1}^{n}[a_1f_1(x_i)+a_2f_2(x_i)+a_3f_3(x_i)+...+a_mf_m(x_i)-y_i]^2 \ ,$$

то полагая, что $\Lambda \to min$, после несложных преобразований можно получить таблицу такого вида

№	a_1	a_2	a_3	...	a_m	b
1	$\sum (f_1(x_i))^2$	$\sum f_1f_2$	$\sum f_1f_3$	...	$\sum f_1f_m$	$\sum f_1(x_i)y_i$
2	$\sum f_2f_1$	$\sum (f_2(x_i))^2$	$\sum f_2f_3$	...	$\sum f_2f_m$	$\sum f_2(x_i)y_i$
3	$\sum f_3f_1$	$\sum f_2f_2$	$\sum (f_3(x_i))^2$	...	$\sum f_3f_m$	$\sum f_3(x_i)y_i$
...	...	...	...	...	...	...
m	$\sum f_mf_1$	$\sum f_mf_2$	$\sum f_mf_3$	...	$\sum (f_m(x_i))^2$	$\sum f_m(x_i)y_i$

Для краткости использовано обозначение $\sum f_kf_l$, означающее $\sum f_k(x_i)f_l(x_i)$.

Выделенная цветом часть таблицы образует матрицу ***A*** размером m×m, последняя колонка — вектор ***b***. Для расчета значений коэффициентов a_i нужно решить систему линейных уравнений ***Aa***=***b*** одним из рассмотренных выше способов, например так:

```
a=A\b
```

Пример 4. Сформируем массив точек, используя формулу $y = 1 + 2x^2$ и попытаемся описать ее полиномом третьей степени

```
x=collect(1.0:10.0);
y=similar(x);
for i in 1:length(x)
y[i] = 1.0+2*x[i]^2
end
f1(x)=1.0
f2(x)=x
f3(x)=x^2
f4(x)=x^3n=10; m=4;
A=zeros(m,m); b=zeros(m)
for i in 1:n
  A[1,1]+=f1(x[i]); A[1,2]+=f2(x[i]);     A[1,3]+=f3(x[i]);       A[1,4]+=f4(x[i]);          b[1]+=y[i]*f1(x[i]);
                A[2,2]+=(f2(x[i]))^2; A[2,3]+=f2(x[i])*f3(x[i]); A[2,4]+=f2(x[i])*f4(x[i]); b[2]+=y[i]*f2(x[i]);
                                      A[3,3]+=(f3(x[i]))^2;      A[3,4]+=f3(x[i])*f4(x[i]); b[3]+=y[i]*f3(x[i]);
A[4,4]+=(f4(x[i]))^2;      b[4]+=y[i]*f4(x[i]);
end
A[2,1]=A[1,2];
A[3,1]=A[1,3]; A[3,2]=A[2,3];
A[4,1]=A[1,4]; A[4,2]=A[2,4];          A[4,3]=A[3,4];
```

Результатом выполнения операции `A\b` будет массив [`1.0,0,2.0,0`], т. е. решение найдено точно.

Этот же результат можно получить без организации цикла суммирования:

```
A[1,1]=sum(f1.(x)); A[1,2]=sum(f2.(x));
A[1,3]=sum(f3.(x)); A[1,4]=sum(f4.(x)); b[1] = sum(y.*f1.(x))
A[2,2]=sum(f2.(x).^2); A[2,3]=sum(f2.(x).*f3.(x));
A[2,4]=sum(f2.(x).*f4.(x)); b[2]=sum(y.*f2.(x));
A[3,3]=sum(f3.(x).^2); A[3,4]=sum(f3.(x).*f4.(x));
b[3]=sum(y.*f3.(x));
A[4,4]=sum(f4.(x).^2); b[4]=sum(y.*f4.(x));
A[2,1]=A[1,2];
A[3,1]=A[1,3]; A[3,2]=A[2,3];
A[4,1]=A[1,4]; A[4,2]=A[2,4]; A[4,3]=A[3,4];
A\b
```

Линейная аппроксимация с ограничениями

В некоторых случаях возникает необходимость использования модели данных, включающей дополнительные линейные ограничения. Например, когда аппроксимируются данные о теплоемкости, важно, чтобы значение функции при стандартной температуре принимало определенное значение. Если весь температурный интервал, соответствующий определенному фазовому состояния вещества, невозможно описать одной функцией, приходится прибегать к кусочной аппроксимации всего набора данных, см. рисунок 6. При этом важно обеспечить отсутствие разрывов функции и ее производной на внутренних границах интервалов. Рабочая матрица ***A*** становится блочной, причем каждый блок соответствует определенному температурному интервалу. Коэффициенты линейных ограничений также должны входить в состав матрицы.

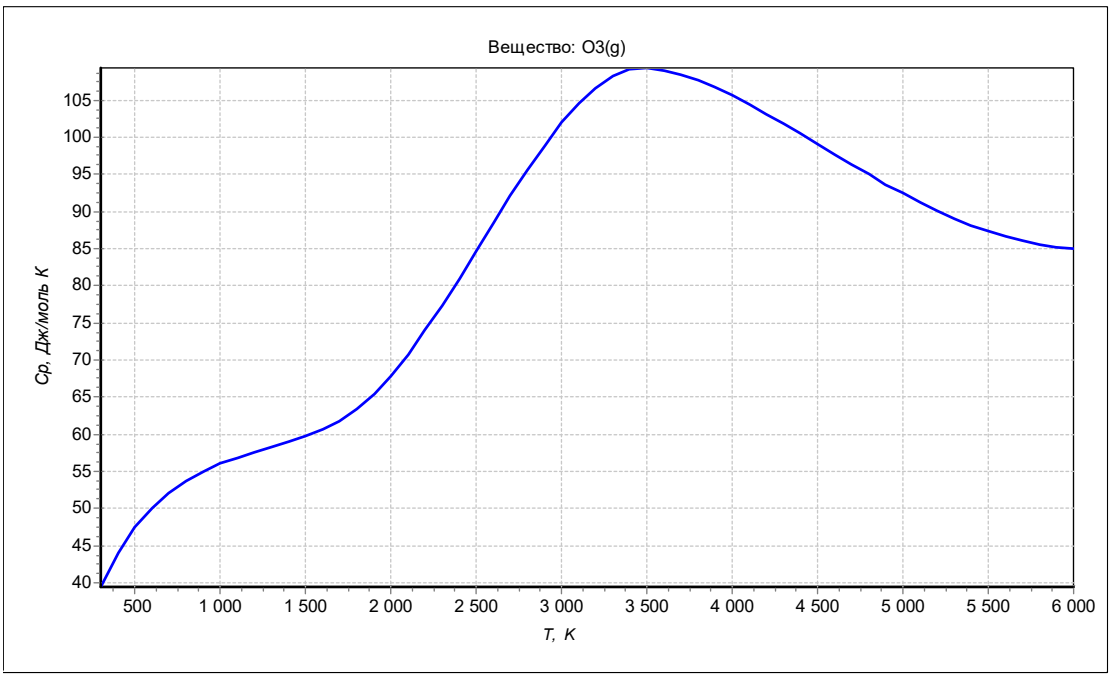

Рис. 6. Зависимость теплоёмкости озона от температуры

Для иллюстрации идеи рассмотрим следующий пример. Допустим, что нужно аппроксимировать зависимость теплоемкости от температуры на участках [T_0, T_1] и [T_1, T_2] функцией вида $y=a+bT+cT^2$ так, чтобы при температуре T_0 функция принимала значение y_0, а при температуре T_1 выполнялись условия $y_1(T_1)=y_2(T_1)$ и $y'_1(T_1)=y'_2(T_1)$. Для решения задачи в данном случае можно использовать метод неопределенных множителей Лагранжа. Пусть число известных значений теплоемкости на первом интервале равно n_1, а на втором - n_2, тогда функция Лагранжа имеет вид

$$\Lambda=\sum_{i=1}^{n_1}[a_1+b_1T_i+c_1T_i^2-y_i]^2+\sum_{i=n_1+1}^{n_1+n_2}[a_2+b_2T_i+c_2T_i^2-y_i]^2+\lambda_1Z_1+\lambda_2Z_2+\lambda_3Z_3 \ ,$$

где Z_i - дополнительные ограничения:

$Z_1=a_1+b_1T_1+c_1T_1^2-y_1$,

$Z_2=a_1-a_2+T_1(b_1-b_2)+T_1^2(c_1-c_2)$,

$Z_3=b_1-b_2+2T_1(c_1-c_2)$.

Неизвестными являются $a_1,b_1,c_1,a_2,b_2,c_2,\lambda_1,\lambda_2,\lambda_3$. Расчетную систему линейных уравнений получим путем дифференцирования функции Лагранжа по a_i,b_i,c_i,λ_i .

Эту же задачу можно решить и путем решения переопределенной системы линейных уравнений. Для этого в систему уравнений необходимо ввести линейные ограничения, умножив их на большие весовые коэффициенты.

Выпуклые оболочки

Для построения выпуклых оболочек множества точек можно использовать библиотеку LazySets (https://juliareach.github.io/LazySets.jl/stable/),
в состав которой входит функция convex_hull().

Пример использования

```
using LazySets, Plots
a=fill(Float64[],0)
for i in 1:100 push!(a,[rand(),rand()]) end
hull=convex_hull(a)
# hull - одномерный массив, содержащий координаты точек
p = plot([Singleton(vi) for vi in a])
plot!(p,VPolygon(hull), alpha=0.2)
```

Результат работы вышеприведенного фрагмента приведен на рисунке 7.

Определить индексы элементов массива `a`, которые находятся на выпуклой оболочке можно так:

`z=indexin(hull,a)` # `z` содержит искомый массив индексов,
`a[z]` содержит элементы массива `hull`.

При необходимости можно построить выпуклую оболочку большей размерности, однако в этом случае необходимо подключить библиотеку Polyhedra.

Пример

```
using LazySets,Polyhedra
v = [randn(3) for _ in 1:30]
hull = convex_hull(v)
```

Проверить, принадлежит ли точка *x* выпуклой оболочке, можно с использованием оператора `in`:

```
x = sum(hull)/length(hull)
x in P
```

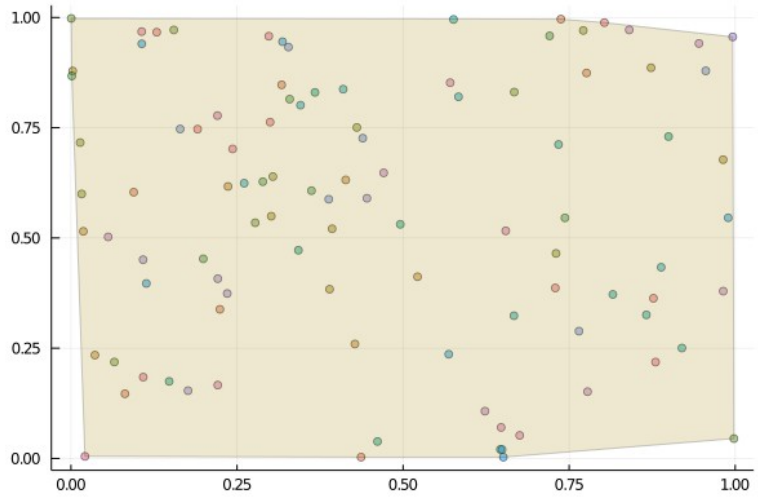

Рис. 7. Выпуклая оболочка множества точек на плоскости

Оптимизация

Библиотека Optim.jl

https://julianlsolvers.github.io/Optim.jl/stable/

Optim.jl - это библиотека для оптимизации функций без ограничений. Используемые решатели при определенных условиях будут сходиться к локальному минимуму. В случае, когда требуется глобальный минимум, предлагаются другие методы, такие как (ограниченный) simulated annealing и particle swarm.

Пример. Оптимизация функции Розенброка

```
using Optim
f(x) = (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
# Задать начальное приближение:
x0 = [0.0, 0.0]
res=optimize(f, x0)
# Получить решение и значение минимума функции:
Optim.minimizer(res)
```

```
Optim.minimum(res)
```

Можно использовать один из нескольких решателей:

NelderMead - не требуется градиент функции, используется по умолчанию:

```
res=optimize(f, x0, NelderMead())
```

L-BFGS - квазиньютоновский метод, требуется градиент функции:

```
res=optimize(f, x0, LBFGS())
```

SimulatedAnnealing() - не требуется градиент функции:

```
res=optimize(f, x0, SimulatedAnnealing())
```

Если функция для расчета градиента не задана, градиент вычисляется автоматически.

Функцию для расчета градиента можно задать так

```
function g!(G, x)
G[1] = -2.0 * (1.0 - x[1]) - 400.0 * (x[2] - x[1]^2) * x[1]
G[2] = 200.0 * (x[2] - x[1]^2)
end
```

Тогда вызов процедуры оптимизации осуществляется таким образом:

```
res=optimize(f, g!, x0, LBFGS())
```

Кроме функции для расчета градиента, можно задать функцию для расчета матрицы Гессе

```
function h!(H, x)
    H[1, 1] = 2.0 - 400.0 * x[2] + 1200.0 * x[1]^2
    H[1, 2] = -400.0 * x[1]
    H[2, 1] = -400.0 * x[1]
    H[2, 2] = 200.0
end
```

Тогда вызвать процедуру оптимизации можно так

```
res=optimize(f, g!, h!, x0, LBFGS())
```

Библиотека JuMP

JuMP - это библиотека с открытым исходным кодом, в которой реализован язык алгебраического моделирования, позволяющий пользователям формулировать широкий спектр задач оптимизации (линейные, нелинейные, с ограничениями и без ограничений). Сам язык моделирования не решает задач оптимизации, его цель — передать задачу в процедуру оптимизации (решатель или солвер). Подробную документацию можно найти на сайте https://jump.dev/, список библиотек, обеспечивающих доступ к решателям, приводится здесь: https://github.com/jump-dev.

Технические задачи, которые должен выполнять язык моделирования, можно условно разделить на две части: во-первых, загрузить в память проблему, введенную пользователем, а во-вторых, сгенерировать входные данные, требуемые процедурой оптимизации в соответствии с типом проблемы. Обе эти задачи решает библиотека JuMP, которая использует технику автоматического (или алгоритмического) диффе-

ренцирования для вычисления производных выражений, введенных пользователем. JuMP, как и другие языки алгебраического моделирования выполняет простую работу: превращает в стандартную форму сформулированную пользователем математическую проблему, передает ее в процедуру оптимизации, ожидает, когда процедура закончит работу, а затем передает решение из процедуры пользователю.

Кроме градиентов функций процедуры оптимизации часто используют матрицы вторых производных. Такого рода матрицы также могут вычислены библиотекой JuMP с использование техники автоматического дифференцирования.

Для того, чтобы решить задачу, нужно выполнить следующие действия:

1. подключить библиотеку JuMP и библиотеку для вызова решателя;
2. объявить переменную-модель с указанием решателя;
3. объявить переменные задачи и задать ограничения;
4. объявить целевую функцию и тип оптимизации (минимизация или максимизация);
5. вызвать решатель;
6. напечатать результат.

Пример 1. Задача линейного программирования

Найти min 12x + 20y

при условиях

$6x + 8y \geqslant 100$

$7x + 12y \geqslant 120$

$x \geqslant 0$

$y \geqslant 0$

Решение

```
using JuMP, GLPK
model = Model(GLPK.Optimizer)
@variable(model, x >= 0)
@variable(model, y >= 0)
@constraint(model, 6x + 8y >= 100)
@constraint(model, 7x + 12y >= 120)
@objective(model, Min, 12x + 20y)
optimize!(model)
@show value(x);
@show value(y);
@show objective_value(model);
```

Комментарии

GLPK – обеспечивает подключение к библиотеке линейной оптимизации оптимизации (GNU Linear Programming Kit library - http://www.gnu.org/software/glpk/).

```
# создаем переменную для обращения к библиотеке оптимизации
model = Model(GLPK.Optimizer)
```

```
# объявляем две переменные и накладываемые ограничения
@variable(model, x >= 0)
@variable(model, y >= 0)
@constraint(model, 6x + 8y >= 100)
@constraint(model, 7x + 12y >= 120)
# объявляем линейную целевую функцию и тип оптимизации.
@objective(model, Min, 12x + 20y)
# вызов процедуры оптимизации
optimize!(model)
# вывод результатов оптимизации
@show value(x);
@show value(y);
@show objective_value(model);
```

Пример 2. Функция Розенброка

```
using JuMP, Ipopt
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x, start=0.0)
@variable(model, y, start=0.0)
@NLobjective(model, Min, (1-x)^2 + 100 * (y-x^2)^2)
#@constraint(model, x + y == 10.0)
#@NLconstraint(model, x*y == 1.0)
JuMP.optimize!(model)
@show JuMP.termination_status(model)
@show JuMP.primal_status(model)
@show JuMP.objective_value(model)
@show JuMP.value(x)
@show JuMP.value(y)
```

Комментарии

`Ipopt (Interior Point OPTimizer) # библиотека для нелинейной оптимизации задач большой размерности` - https://github.com/coin-or/Ipopt.

`set_silent(model) # отключаем печать сообщений оптимизатора`

`@variable(model, x, start=0.0) # можно (опционально) задавать начальное приближение неизвестной (start=0.0)`

`@NLobjective(model, Min, (1-x)^2 + 100 * (y-x^2)^2) # объявляем НЕлинейную целевую функцию и тип оптимизации`

Добавить ограничение к модели довольно просто

```
@constraint(model, x + y == 10.0) # добавить линейное ограничение
@NLconstraint(model, x*y == 1.0)  # добавить нелинейное ограничение
```

Список переменных можно задать как массив одним оператором

`@variable(model, x[1:5])`.

При определении целевой функции или ограничений можно использовать функцию суммирования sum():

```
using JuMP, Ipopt
model = Model(Ipopt.Optimizer)
```

```
set_silent(model)
@variable(model, x[1:5])
@NLobjective(model, Min, sum((1.0-x[i])^2 for i = 1:5))
@constraint(model, sum(x[i] for i=1:3) == 1)
JuMP.optimize!(model)
@show JuMP.termination_status(model)
@show JuMP.primal_status(model)
@show JuMP.objective_value(model)
for i in 1:5 println("x[$i]=",JuMP.value(x[i])) end
```

Для того чтобы извлечь из решения значения неопределенных множителей Лагранжа задачи с ограничениями, необходимо дать наименование соответствующим ограничениям:

```
@constraint(model, con1, sum(x[i] for i=1:3) == 1)
```

Значение множителя Лагранжа можно получить с использованием функции shadow_price():

```
shadow_price(con1)
```

Если ограничений несколько, в некоторых случаях их можно объединить в массив

`@constraint(model, con, A*x .== b)`, здесь A, x, b – массивы,

или так:

```
@constraint(model, con[i=1:nc], g[i] >= sum(A[i,j]*Lam[j] for j in 1:m)),
```

здесь задаются `nc` ограничений вида $g_i \geq \sum_{1}^{m} A_{ji} * \lambda_j$.

В этих случаях `con` также будет массивом, и доступ к его элементам осуществляется так: `shadow_price(con[i])`.

Библиотека NLopt

https://nlopt.readthedocs.io/en/latest/

https://github.com/JuliaOpt/NLopt.jl

NLopt-это свободно распространяемая библиотека с открытым исходным кодом для нелинейной оптимизации, предоставляющая общий интерфейс для ряда различных бесплатных процедур оптимизации, доступных в Интернете, а также оригинальные реализации различных других алгоритмов. Ее ключевые особенности

- вызывается из языков программирования C, C++, Fortran, Matlab или GNU Octave, Python, GNU Guile, Julia, GNU R, Lua, OCaml и Rust;
- общий интерфейс для нескольких алгоритмов—выбрать алгоритм можно, изменив только один параметр;
- возможность оптимизации задач большой размерности (некоторые алгоритмы масштабируются до миллионов параметров и тысяч ограничений);
- реализованы алгоритмы локальной и глобальной оптимизации;

- реализованы алгоритмы, использующие только значения функций (без производных), а также алгоритмы, использующие заданные градиенты;
- реализованы алгоритмы для оптимизации без ограничений, оптимизации с ограничениями и общих нелинейных ограничений вида неравенства и/или равенства.

Библиотека NLopt реализует интерфейс MathOptInterface для нелинейной оптимизации, поэтому ее можно использовать взаимозаменяемо с другими библиотеками оптимизации из библиотек моделирования, таких как JuMP.
Параметрами решателя NLopt (NLopt.Optimizer) являются:

- `algorithm`
- `stopval`
- `ftol_rel`
- `ftol_abs`
- `xtol_rel`
- `xtol_abs`
- `constrtol_abs`
- `maxeval`
- `maxtime`
- `initial_step`
- `population`
- `seed`
- `vector_storage`

Параметр `algorithm` является обязательным, все остальные необязательны.
Пример решения задачи нелинейной оптимизации с ограничениями из NLopt Tutorial:

```
using NLopt
function myfunc(x::Vector, grad::Vector)
    if length(grad) > 0
        grad[1] = 0
        grad[2] = 0.5/sqrt(x[2])
    end
    return sqrt(x[2])
end

function myconstraint(x::Vector, grad::Vector, a, b)
    if length(grad) > 0
        grad[1] = 3a * (a*x[1] + b)^2
        grad[2] = -1
    end
    (a*x[1] + b)^3 - x[2]
end

opt = Opt(:LD_MMA, 2)
opt.lower_bounds = [-Inf, 0.]
opt.xtol_rel = 1e-4
```

```
opt.min_objective = myfunc
inequality_constraint!(opt, (x,g) -> myconstraint(x,g,2,0), 1e-8)
inequality_constraint!(opt, (x,g) -> myconstraint(x,g,-1,1), 1e-8)
(minf,minx,ret) = optimize(opt, [1.234, 5.678])
numevals = opt.numevals # the number of function evaluations
println("got $minf at $minx after $numevals iterations (returned $ret)")
```

Эту же задачу можно решить с использованием библиотеки JuMP, которая предоставляет интерфейс к NLopt.

```
using JuMP
using NLopt
model = Model(NLopt.Optimizer)
set_optimizer_attribute(model, "algorithm", :LD_MMA)
a1 = 2
b1 = 0
a2 = -1
b2 = 1
@variable(model, x1)
@variable(model, x2 >= 0)
@NLobjective(model, Min, sqrt(x2))
@NLconstraint(model, x2 >= (a1*x1+b1)^3)
@NLconstraint(model, x2 >= (a2*x1+b2)^3)
set_start_value(x1, 1.234)
set_start_value(x2, 5.678)
optimize!(model)
println("got ", objective_value(model), " at ", [value(x1), value(x2)])
```

Задачи

1. Найти минимум функции $x^2+e^x-1=0$.

Применение библиотеки GLPK для аппроксимации набора точек линейной комбинацией функций

Использованы материалы книг [12, 13].

Допустим нам известны координаты n точек (x_i, y_i). Требуется найти модель $\sum_{j=1}^{m} a_j f_j(x_i)$, которая является линейной комбинацией произвольного набора функций $f(x_i)$, описывающую с приемлемой точностью зависимость $\boldsymbol{y}(\boldsymbol{x})$. Значения коэффициентов a_i необходимо вычислить.

1. Норма L_1: минимизировать сумму абсолютных погрешностей

$$z_1=\sum_{i=1}^{n}|y_i-\sum_{j=1}^{m}a_j f_j(x_i)|\to min \quad .$$

Эту задачу можно сформулировать так.

$$z=\sum_{i=1}^{n}(r_i+s_i)\to min$$

при выполнении условий

$$r_i - s_i + \sum_{j=1}^{m} a_j f_j(x_i) = y_i \ ,$$

$$r_i \geq 0, s_i \geq 0, i = 1,2,\ldots n \ .$$

Проверить работу алгоритма можно с использованием следующего примера (аппроксимирующая функция имеет вид $y = a_1 + a_2 x + a_3 x^2 + a_4 x^3 + a_5 / x^2$)

```
n=10
x=collect(1.0:n);
y=similar(x);
for i in 1:length(x)
y[i] = 1.0+2*(x[i])^2;
end
m=5
model = Model(GLPK.Optimizer)
# объявляем переменные и ограничения
@variable(model, a[1:m]);
@variable(model, r[1:n]>=0);
@variable(model, s[1:n]>=0);
for i in 1:n
@constraint(model,
  r[i]-s[i] +a[1]+a[2]*x[i]+a[3]*x[i]^2+a[4]*x[i]^3+a[5]/
x[i]^2==y[i]);
 end
# объявляем линейную целевую функцию и тип оптимизации.
@objective(model, Min, sum(r[i]+s[i] for i = 1:n))
# вызов процедуры оптимизации
optimize!(model)
# вывод результатов оптимизации
for i in 1:m println("a[$i]=",JuMP.value(a[i])) end
# вывод значения целевой функции
@show objective_value(model);
```

2. Норма L_∞: минимизировать максимальную погрешность аппроксимации

$$z_\infty = \max_{1 \leq i \leq m} |y_i - \sum_{j=1}^{m} a_j f_j(x_i)| \rightarrow min \ .$$

Эту задачу можно сформулировать следующим образом.

$$z \rightarrow min$$

при выполнении условий

$$z - r_i - s_i \geq 0 \ ,$$

$$r_i - s_i + \sum_{j=1}^{m} a_j f_j(x_i) = y_i \ ,$$

$$r_i \geq 0, s_i \geq 0, i = 1,2,\ldots n \ .$$

Используем следующий пример для проверки работы алгоритма

```
n=10
x=collect(1.0:n);
y=similar(x);
for i in 1:length(x)
y[i] = 1.0+2*(x[i])^2;
end
m=5
model = Model(GLPK.Optimizer)
# объявляем переменные и ограничения
@variable(model, a[1:m]);
@variable(model, r[1:n]>=0);
@variable(model, s[1:n]>=0);
@variable(model, z);
for i in 1:n
@constraint(model,
  r[i]-s[i] +a[1]+a[2]*x[i]+a[3]*x[i]^2+a[4]*x[i]^3+a[5]/
x[i]^2==y[i]);
end
for i in 1:n
@constraint(model, z - r[i]-s[i] >= 0);
end
# объявляем линейную целевую функцию и тип оптимизации.
@objective(model, Min, z)
# вызов процедуры оптимизации
optimize!(model)
# вывод результатов оптимизации
for i in 1:m println("a[$i]=",JuMP.value(a[i])) end
# вывод значения целевой функции
@show objective_value(model);
```

3. Визуализация данных с Makie

Makie – экосистема визуализации данных, для языка программирования Julia. Для работы с Makie достаточно установить одну из библиотек GLMakie.jl (OpenGL), CairoMakie.jl (Cairo), или WGLMakie.jl (WebGL). Переключение между этими библиотеками производится вызовом функции `activate()`, например, `CairoMakie.activate!()`. К сожалению, REPL от Julia не позволяет отображать графики на экране компьютера, разработчики рекомендуют для этой цели IDE, которое поддерживает вывод графики в формате png или svg, например, VSCode, Atom/Juno, Jupyter, Pluto, либо использовать библиотеку ElectronDisplay.jl.

Рассмотрим основные принципы построения 2D графиков с использованием библиотеки Makie (https://makie.juliaplots.org/stable/tutorials/basic-tutorial/).

Для начала построим зависимость y = sin(x), см. текст программы и рис. 8.

```
using CairoMakie
x = range(0, 10, length=100);
y = sin.(x);
f=scatter(x, y)
save("figure.png", f)
```

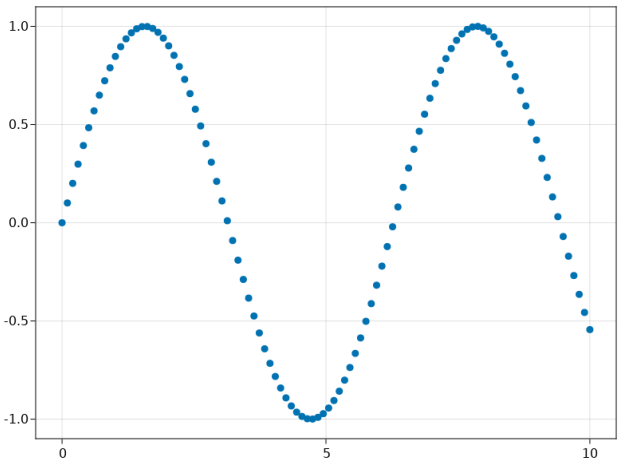

Рис. 8. Пример построения зависимости y = sin(x)

Если нужно построить на одном графике несколько функций, это можно сделать с использованием вызова функций с восклицательным знаком. Рассмотрим в качестве примера построение на одном графике зависимостей y = sin(x), y = cos(x), см. текст и рис. 9.

```
using CairoMakie
x = range(0, 10, length=100);
y1 = sin.(x);
y2 = cos.(x);
lines(x, y1)
lines!(x, y2)
f=current_figure()
```

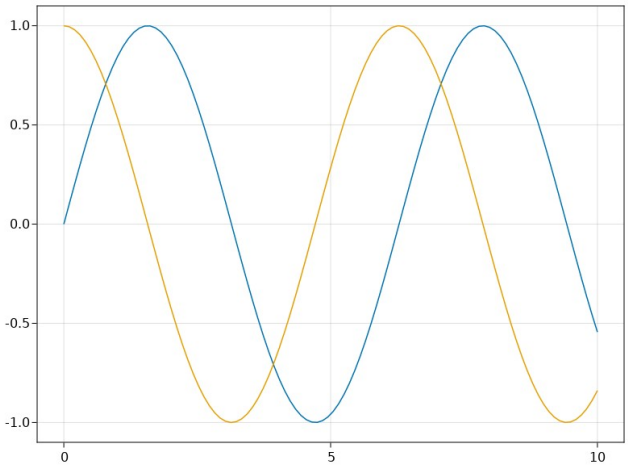

Рис. 9. Пример построения на зависимостей y = sin(x), y = cos(x)

Следующий пример иллюстрирует возможность работы с некоторыми атрибутами графиков. `color` позволяет выбрать цвет точек или линий. Для графиков типа `scatter`: `markersize` — размер точек. Для графиков типа `lines`: `linewidth` – толщина линии, `linestyle` - стиль линии, см. текст и рис. 10.

```
using CairoMakie
x = range(0, 10, length=100);
y1 = sin.(x);
y2 = cos.(x);
lines(x, y1, color = :red, linewidth = 5, linestyle = :dash)
scatter!(x, y2, color = :blue, markersize = 10)
f=current_figure()
```

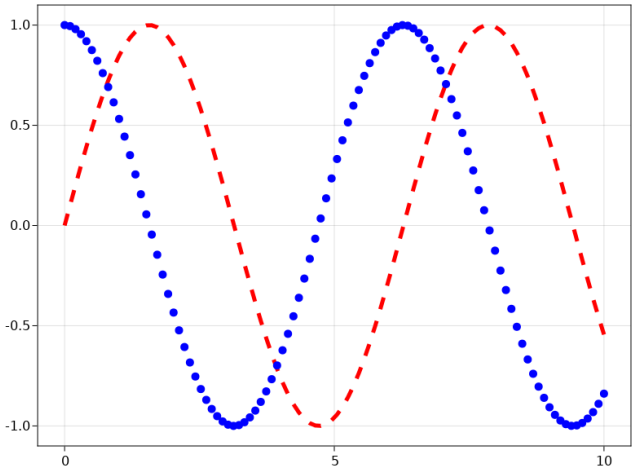

Рис. 10. Пример использования атрибутов

Добавить легенду на график очень просто с использованием ключевого слова `label` и команды `axislegend()`, см. текст и рис. 11.

```
using CairoMakie
x = range(0, 10, length=100);
y1 = sin.(x);
y2 = cos.(x);
lines(x, y1, color = :red, label = "sin")
lines!(x, y2, color = :blue, label = "cos")
axislegend()
f=current_figure()
```

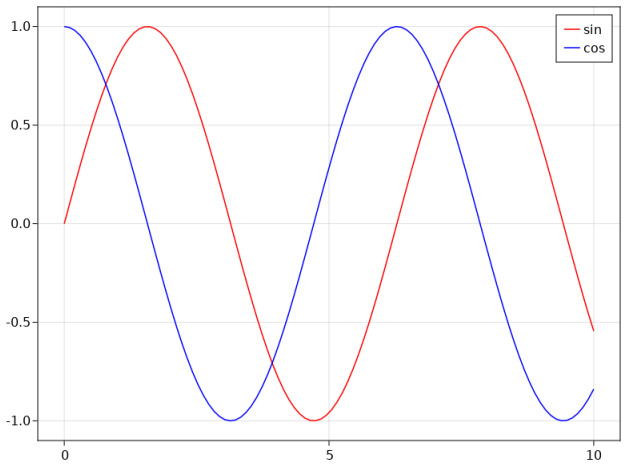

Рис. 11. Пример использования легенды

Иногда возникает необходимость компактного размещения нескольких графиков в отдельных окнах. Это можно сделать с использованием объекта `Figure()`. Размещение окон регулируется опцией `fig[row, col]`, смысл которой ясен из приведенного ниже примера, см. текст программы, рис.12 и раздел «Объект Figure» ниже.

```
using CairoMakie
x = LinRange(0, 10, 100);  # x values, 100 points in [0,10] interval
y = sin.(x);
fig = Figure();
lines(fig[1, 1], x, y, color = :red)
lines(fig[1, 2], x, cos.(x), color = :blue)
scatter(fig[2, 1:2], x, y.*cos.(x), color = :green)
fig
```

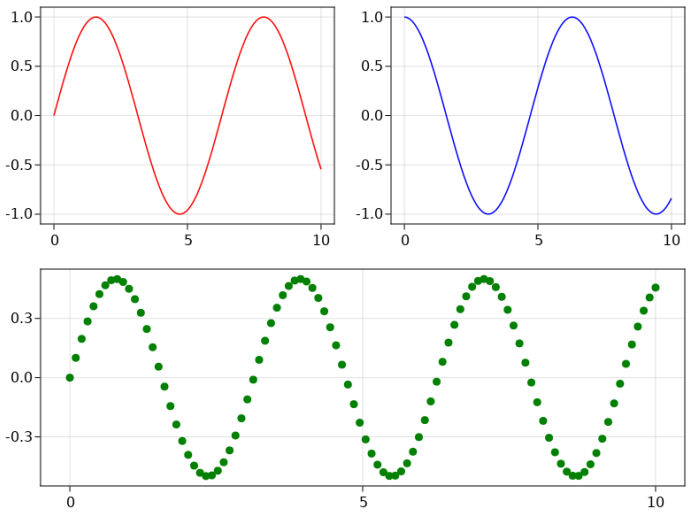

Рис. 12. Пример размещения нескольких графиков в отдельных окнах.

Оси окон можно создать отдельно с тем, чтобы затем построить в них графики, см. текст и рис. 13, 14.

```
using CairoMakie
fig = Figure()
```

```
ax1 = Axis(fig[1, 1])
ax2 = Axis(fig[1, 2])
ax3 = Axis(fig[2, 1:2])
fig
```

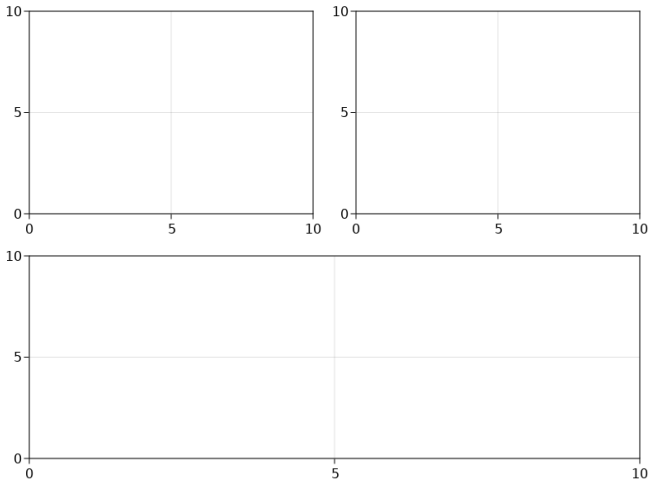

Рис. 13. Пример размещения нескольких осей в отдельных окнах

А теперь построим график на этих осях, масштабирование осей производится автоматически.

```
lines!(ax1, 0..20, sin)
lines!(ax2, 0..20, cos)
lines!(ax3, 0..20, sqrt)
fig
```

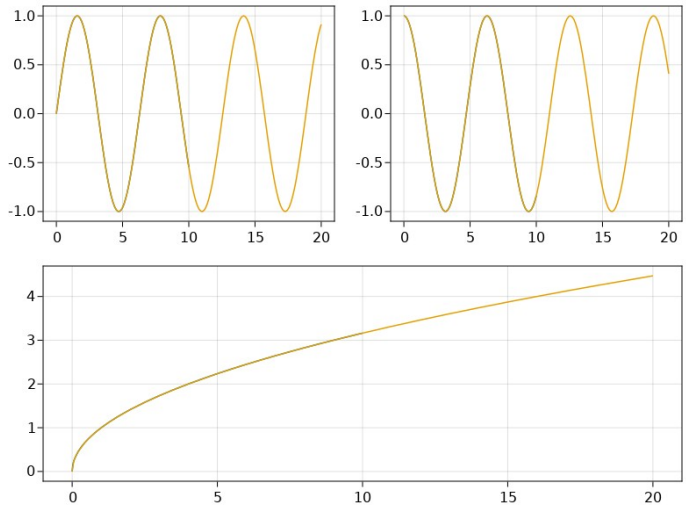

Рис. 14. Пример построения графиков на созданных осях.

Работа с осями

О работе с осями графиков библиотеки Makie можно ознакомиться здесь: https://makie.juliaplots.org/stable/examples/layoutables/axis/index.html

Следующий фрагмент текста предназначен для создания осей с автоматическим выбором интервалов, созданием титула и заголовков осей. На рис.15 показан результат его выполнения.

```
using CairoMakie
f = Figure()
ax = Axis(f[1, 1], xlabel = "x label", ylabel = "y label",
    title = "Title")
f
```

На существующих осях можно построить график функции. Результат работы нижележащего текста приводится на рис. 16. Если параметр `ax` не передается в функцию построения графика, он отображается в активном окне `current_axis()`, обычно, это последнее созданное окно.

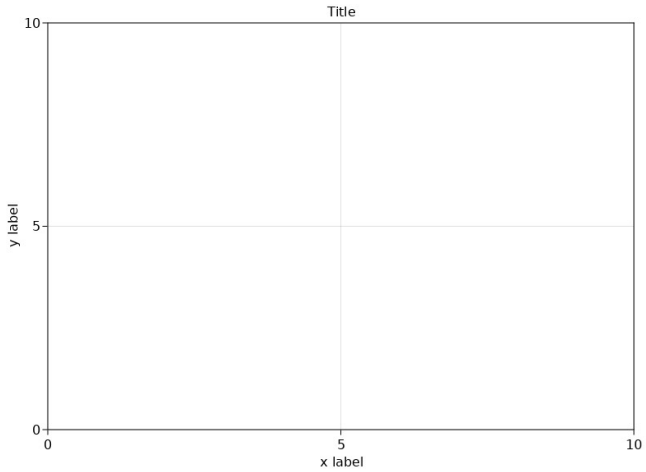

Рис. 15. Пример создания осей с титулом и заголовками осей

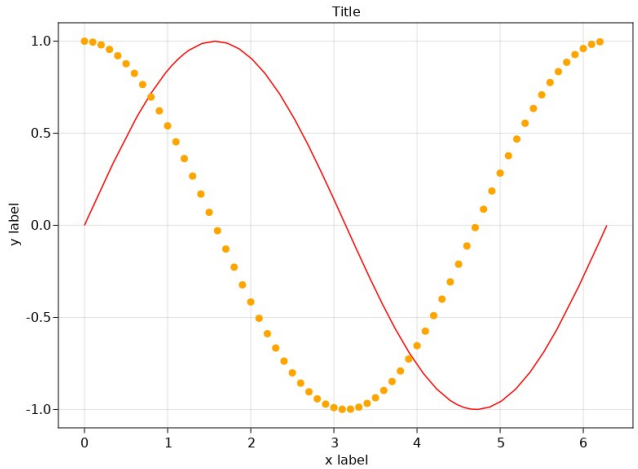

Рис. 16. Пример построения графиков на созданных осях

```
lineobject = lines!(ax, 0..2pi, sin, color = :red)
scatobject = scatter!(0:0.1:2pi, cos, color = :orange)
f
```

График на осях ax можно удалить с использованием команды `delete!(ax, plotobj)`. Для приведенного выше примера один из графиков можно удалить так:

```
delete!(ax, lineobject)
f
```

Результат показан на рис. 17.

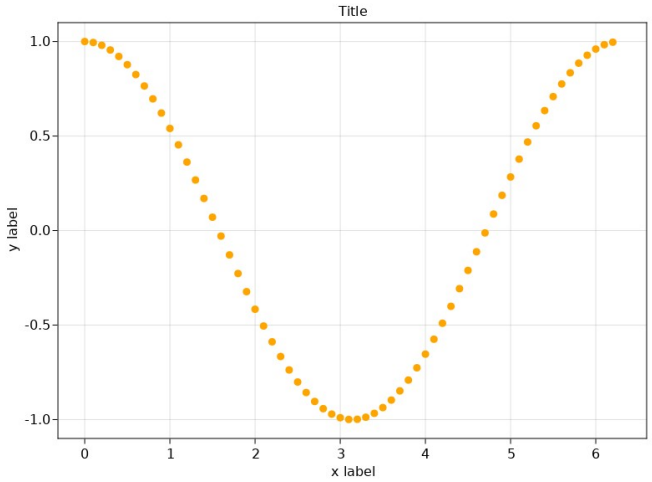

Рис. 17. Пример удаления графика

Границы интервалов на осях можно изменить с использованием функций `xlims!()`, `ylims!()`, `limits!()`. Следующие пример и рис.18 иллюстрируют идею

```
using CairoMakie
f = Figure()
axes = [Axis(f[i, j]) for j in 1:3, i in 1:2]
for (i, ax) in enumerate(axes)
    ax.title = "Axis $i"
    poly!(ax, Point2f[(9, 9), (3, 1), (1, 3)],
        color = cgrad(:inferno, 6, categorical = true)[i])
end
xlims!(axes[1], [0, 10]) # as vector
xlims!(axes[2], 10, 0)   # separate, reversed
ylims!(axes[3], 0, 10)   # separate
ylims!(axes[4], (10, 0)) # as tuple, reversed
limits!(axes[5], 0, 10, 0, 10) # x1, x2, y1, y2
limits!(axes[6], BBox(0, 10, 0, 10)) # as rectangle
f
```

Границы интервалов на осях можно задавать частично (аргумент nothing), либо ключевые слова low и high, см пример ниже и соответствующий рис. 19.

```
using CairoMakie
f = Figure()
data = rand(100, 2) .* 0.7 .+ 0.15
Axis(f[1, 1], title = "xlims!(nothing, 1)")
scatter!(data)
xlims!(nothing, 1)
```

```
Axis(f[1, 2], title = "xlims!(low = 0)")
scatter!(data)
xlims!(low = 0)
Axis(f[2, 1], title = "ylims!(0, nothing)")
scatter!(data)
ylims!(0, nothing)
Axis(f[2, 2], title = "ylims!(high = 1)")
scatter!(data)
ylims!(high = 1)
f
```

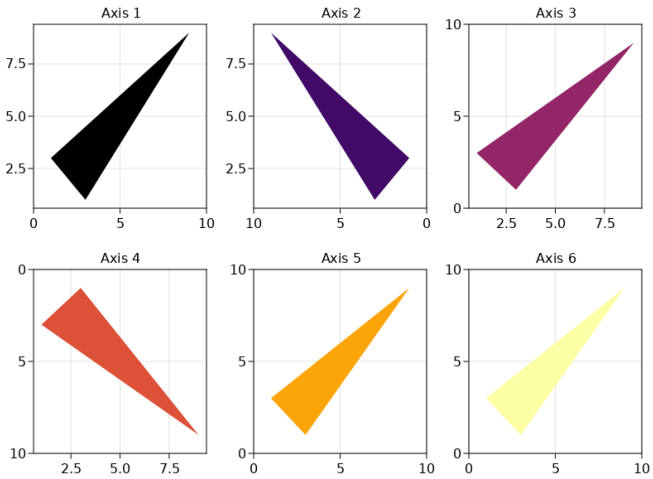

Рис. 18. Пример изменения границ интервалов на осях графика

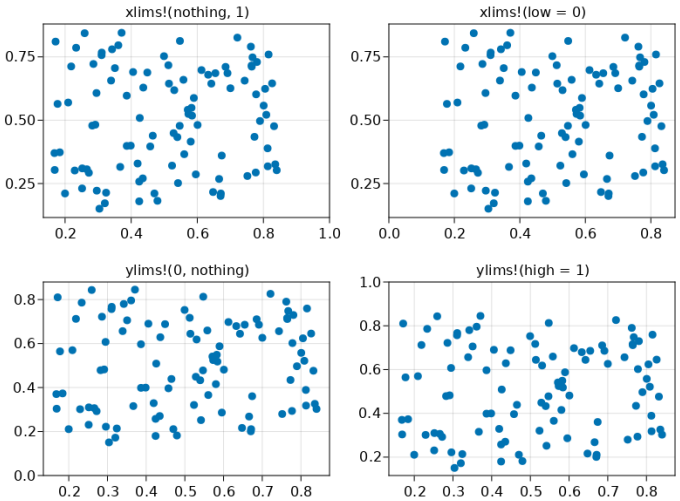

Рис. 19. Использование аргумента nothing и ключевых слов low, high

Изменение отметок на осях

Для изменения отметок на осях используются атрибуты `xticks/yticks` и `xtickformat/ytickformat`. В Makie используется несколько типов меток на осях,

тип по умолчанию - `LinearTicks(n)` , где `n` – число меток на оси. Работу с метками иллюстрирует нижеследующий фрагмент и рис. 20.

```
using CairoMakie
fig = Figure()
for (i, n) in enumerate([2, 5, 9])
    lines(fig[i, 1], 0..20, sin, axis = (xticks = LinearTicks(n),))
end
fig
```

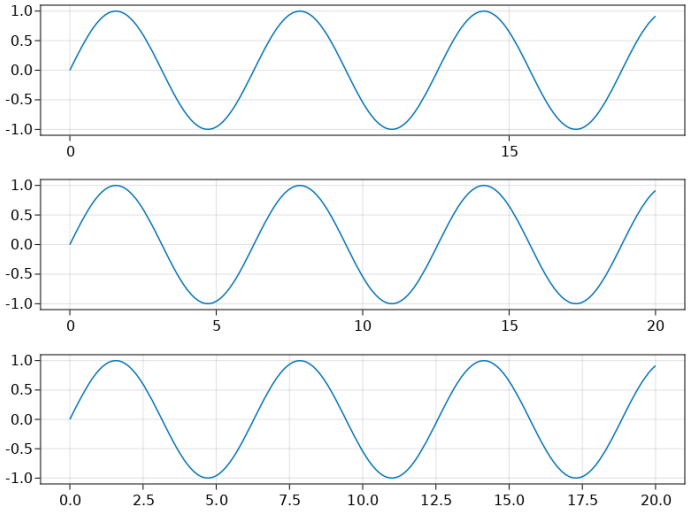

Рис. 20. Работа с метками на осях

Приведем еще один пример и рис.21 для демонстрации возможностей форматирования меток на осях

```
using CairoMakie
f = Figure()
axes = [Axis(f[i, j]) for i in 1:2, j in 1:2]
xs = LinRange(0, 2pi, 50)
for (i, ax) in enumerate(axes)
    ax.title = "Axis $i"
    lines!(ax, xs, sin.(xs))
end
axes[1].xticks = 0:6
axes[2].xticks = 0:pi:2pi
axes[2].xtickformat = xs -> ["$(x/pi)π" for x in xs]
axes[3].xticks = (0:pi:2pi, ["start", "middle", "end"])
axes[4].xticks = 0:pi:2pi
axes[4].xtickformat = "{:.2f}ms"
```

```
axes[4].xlabel = "Time"
f
```

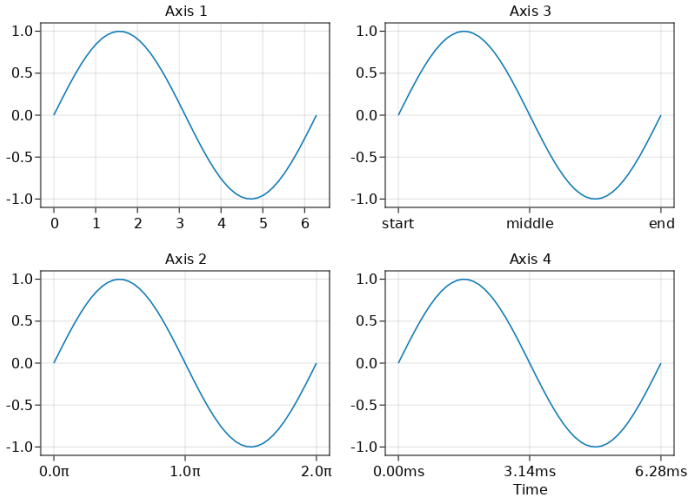

Рис. 21. Демонстрация возможностей форматирования меток на осях

Следующий пример (рис. 22) показывает, как можно изменить сетку на осях графика

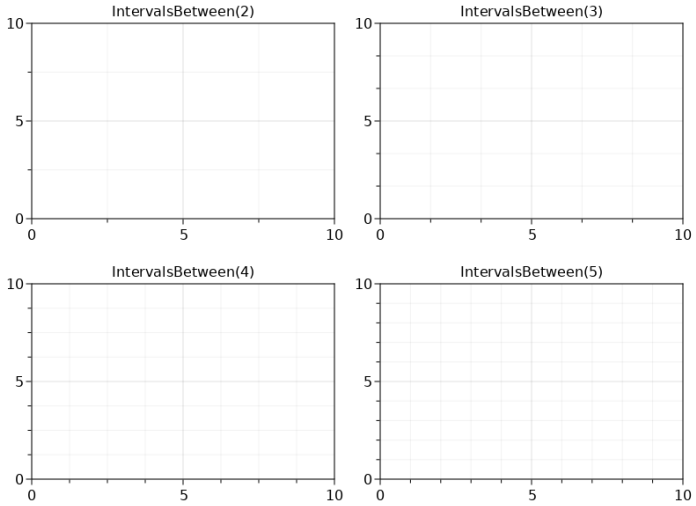

Рис. 22. К задаче изменения сетки на осях графика

```
using CairoMakie
theme = Attributes(
    Axis = (
        xminorticksvisible = true,
        yminorticksvisible = true,
        xminorgridvisible = true,
        yminorgridvisible = true,
    )
)
fig = with_theme(theme) do
    fig = Figure()
    axs = [Axis(fig[fldmod1(n, 2)...],
        title = "IntervalsBetween($(n+1))",
```

```
        xminorticks = IntervalsBetween(n+1),
        yminorticks = IntervalsBetween(n+1)) for n in 1:4]
    fig
end
fig
```

Логарифмический и другие масштабы осей

Библиотека Makie предоставляет достаточно широкие возможности использования нескольких типов осей. Следующий пример (рис. 23) демонстрирует некоторые виды осей графиков.

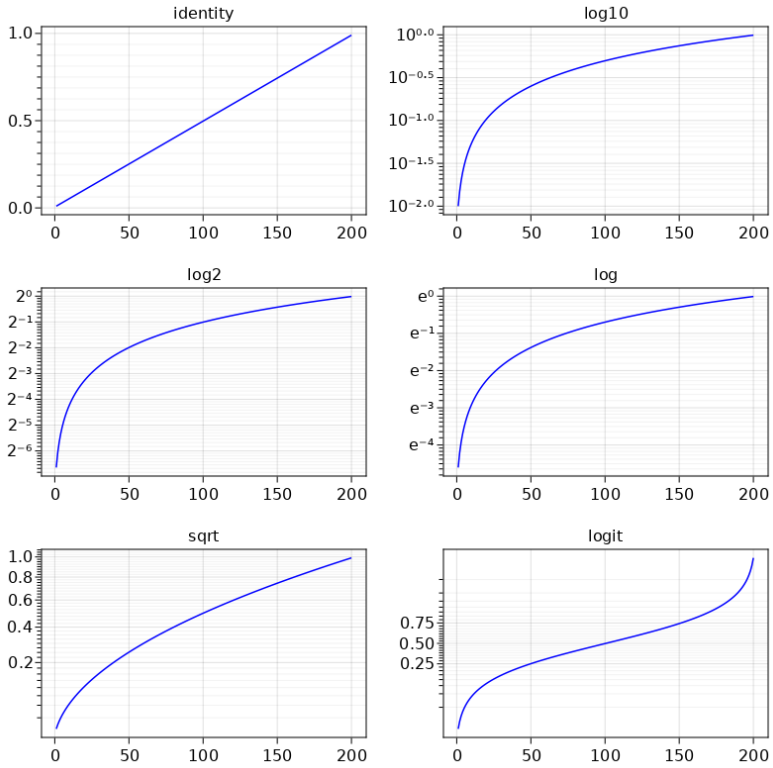

Рис. 23. Использование разных видов осей

```
using CairoMakie
data = LinRange(0.01, 0.99, 200)
f = Figure(resolution = (800, 800))
for (i, scale) in enumerate([identity, log10, log2, log, sqrt,
Makie.logit])
    row, col = fldmod1(i, 2)
    Axis(f[row, col], yscale = scale, title = string(scale),
        yminorticksvisible = true, yminorgridvisible = true,
        yminorticks = IntervalsBetween(8))
    lines!(data, color = :blue)
end
f
```

По умолчанию атрибуты `xscale` и `yscale` имеют значение `identity`. При использовании некоторых видов осей (например, логарифмических) нужно следить за границами интервалов (x > 0).

Легенда

В Makie есть несколько способов работы с легендой. Простейший способ предполагает передачу ее объекту типа axes: `Axis, LScene` или `Scene`. Все графики, у которых есть атрибут `label`, будут обозначены в легенде, рис. 24.

```
using CairoMakie
f = Figure()
ax = f[1, 1] = Axis(f, xlabel = "x label", ylabel = "y label",
    title = "Title")
lines!(0..15, sin, label = "sin",color = :blue)
lines!(0..15, cos, label = "cos", color = :red)
lines!(0..15, x -> -cos(x), label = "-cos", color = :green)
f[1, 2] = Legend(f, ax, "Trig Functions", framevisible = false)
f
```

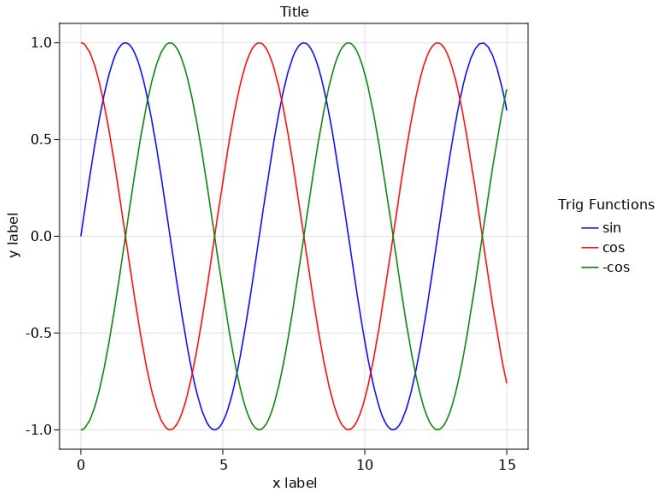

Рис. 24. График с легендой

Более универсальный способ предполагает передачу заголовков легенды и типов обозначений как массив. В приведенном ниже примере показана возможность обозначения линий (lin), точек (sca) , их комбинации ([lin, sca]) и прямоугольников, рис. 25.

```
f = Figure()
Axis(f[1, 1])
xs = 0:0.5:10
ys = sin.(xs)
lin = lines!(xs, ys, color = :blue)
sca = scatter!(xs, ys, color = :red)
sca2 = scatter!(xs, ys .+ 0.5, color = 1:length(xs), marker = :rect)
Legend(f[1, 2],
    [lin, sca, [lin, sca], sca2],
    ["a line", "some dots", "both together", "rect markers"])
```

```
f
```

Иногда на одном рисунке необходимо разместить много объектов, при этом возникают трудности с размещением легенды. Для подобного случая в Makie предусмотрена возможность создания нескольких колонок легенды (banks), рис. 26.

```
using CairoMakie
f = Figure()
Axis(f[1, 1])
xs = 0:0.1:10;
lins = [lines!(xs, sin.(xs .+ 3v), color = RGBf(v, 0, 1-v)) for v in 0:0.1:1]
Legend(f[1, 2], lins, string.(1:length(lins)), nbanks = 3)
f
```

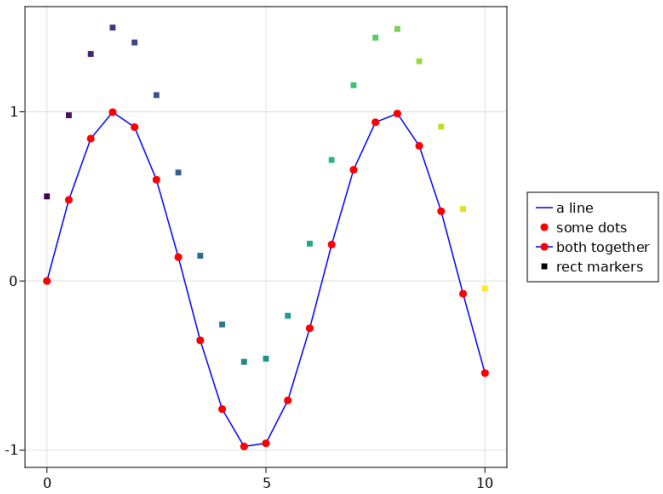

Рис. 25. Передача заголовков легенды как массив

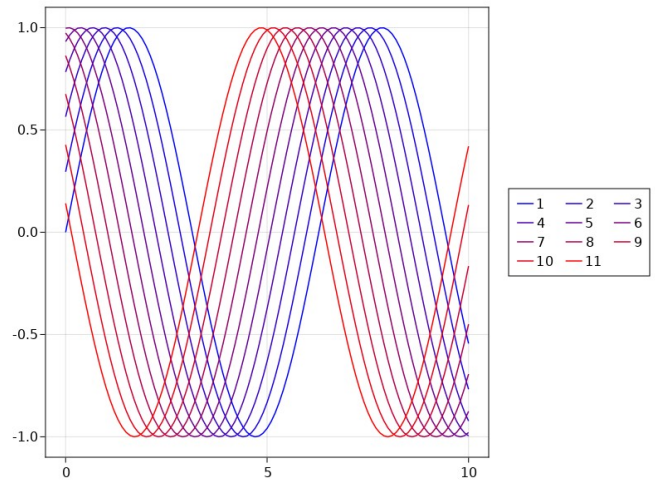

Рис. 26. Размещение обозначений на легенде в несколько столбцов

Легенду можно разместить вне графика и внутри графика. Во втором случае используется функция `axislegend()`. Кроме обозначений графиков в функцию можно передать заголовок легенды. Положение легенды (`position`) обозначается комбинацией двух букв: (l, r, c) — для горизонтального размещения (left, right, center) и (b, t, c) для вертикального (bottom, top, center), рис. 27.

```
using CairoMakie
```

```
f = Figure()
ax = Axis(f[1, 1])
sc1 = scatter!(randn(10, 2), color = :red, label = "Red Dots")
sc2 = scatter!(randn(10, 2), color = :blue, label = "Blue Dots")
scatter!(randn(10, 2), color = :orange, label = "Orange Dots")
scatter!(randn(10, 2), color = :cyan, label = "Cyan Dots")
axislegend()
axislegend("Titled Legend", position = :lb)
axislegend(ax, [sc1, sc2], ["One", "Two"], "Selected Dots", position = :rb,
    orientation = :horizontal)
f
```

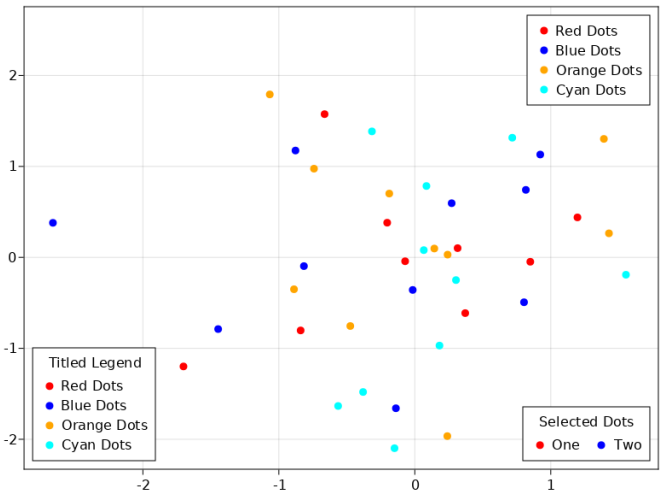

Рис. 27. Размещение легенды внутри графика

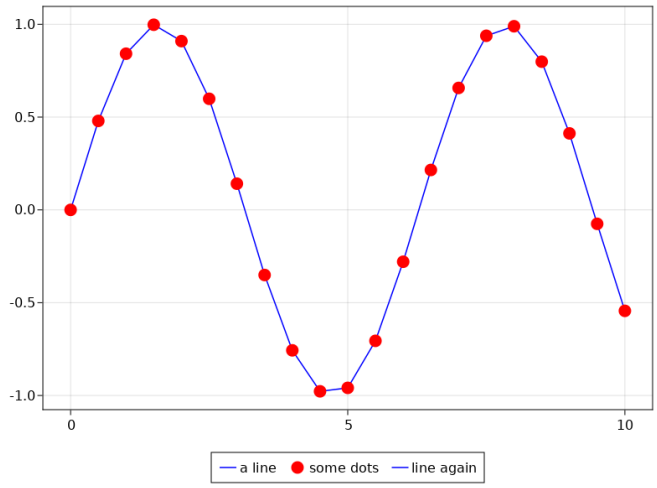

Рис. 28. Горизонтальное размещение легенды

Наконец, приведем пример с горизонтальным размещением легенды, рис. 28.

```
using CairoMakie
f = Figure()
Axis(f[1, 1])
xs = 0:0.5:10;
ys = sin.(xs)
lin = lines!(xs, ys, color = :blue)
sca = scatter!(xs, ys, color = :red, markersize = 15)
```

```
Legend(f[2, 1], [lin, sca, lin], ["a line", "some dots", "line again"],
    orientation = :horizontal, tellwidth = false, tellheight = true)
```

Объект Figure

Объект Figure используется для построения графиков. Этот объект может быть создан неявно при вызове функций, подобных `plot()`, `scatter()`, `lines()`, или явно:

```
f = Figure()
```

С помощью объекта Figure можно управлять внешним видом рисунка. Наиболее важным свойством является разрешение рисунка (resolution):

```
f = Figure(resolution = (600, 400))
```

Аргументы неявно создаваемого объекта Figure можно задавать с помощью ключевого слова figure:

```
scatter(rand(100, 2), figure = (resolution = (600, 400),))
```

На объекте Figure можно разместить оси

```
f = Figure()
ax = f[1, 1] = Axis(f)
```

Для размещения графиков на рис. 29 используется индексирование объекта Figure. Следующие пример и рисунок иллюстрируют идею.

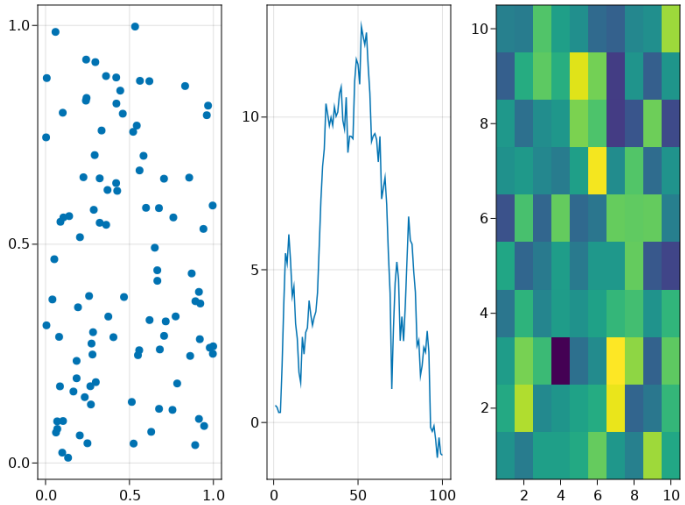

Рис. 29. Индексирование объекта Figure

```
using CairoMakie
f = Figure()
scatter(f[1, 1], rand(100, 2))
pos = f[1, 2]
lines(pos, cumsum(randn(100)))
heatmap(f[1, 3], randn(10, 10))
f
```

Атрибуты графика

Сведения об объектах Figure (`fig`), осях (`ax`), объекте построения (`pltobj`) можно получить так.

```
fig, ax, pltobj = scatterlines(1:10)
```

Атрибуты объекта построения `pltobj`:

```
pltobj.attributes
Attributes with 14 entries:
color => black
colormap => viridis
colorrange => Automatic()
inspectable => true
linestyle => nothing
linewidth => 1.5
marker => Circle{T} where T
markercolor => black
markercolormap => viridis
markercolorrange => Automatic()
markersize => 9
model => Float32[1.0 0.0 0.0 0.0; 0.0 1.0 0.0 0.0; 0.0 0.0 1.0
0.0; 0.0 0.0 0.0 1.0]
strokecolor => black
strokewidth => 0
```

Эти атрибуты можно менять интерактивно после построения графика. Например, так

```
pltobj.attributes.color=:blue
```

Большое количество атрибутов, которые можно менять интерактивно, есть и у осей (объекта ax)

```
ax.attributes
```

Литература

1. Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman. Julia: A Fast Dynamic Language for Technical Computing, arXiv, 2012 (https://arxiv.org/abs/1209.5145).

2. Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. SIAM REVIEW Vol. 59, No. 1, pp. 65–98, 2017 (https://dspace.mit.edu/handle/1721.1/110125).

3. Lauwens B., Downey A. Think Julia: how to think like a computer scientist. O'Reilly Media, 2019.

4. Lobianco, A. Julia Quick Syntax Reference. Apress, Berkeley, CA, 2019. 216 p.

5. Kwon C. Julia Programming for Operations Research, 2019-2021

6. Boyd S., Vandenberghe L. Introduction to Applied Linear Algebra – Vectors, Matrices, and Least Squares. Cambridge University Press. 2018, 474 p. (with a Julia Language Companion) https://web.stanford.edu/~boyd/vmls/

7. Kochenderfer M., Wheeler T. Algorithms for Optimization. MIT Press; 2019, 520 p.

8. Kochenderfer M., Wheeler T., Wray K. Algorithms for Decision Making, 2022, 694 p.

9. Klok H., Nazarathy Y. Statistics with Julia: Fundamentals for Data Science, Machine 8. Learning and Artificial Intelligence. https://github.com/h-Klok/StatsWithJuliaBook

10. Storopoli J., et al. Julia Data Science. https://juliadatascience.io/

11. Белов Г. В., Аристова Н. М. О возможностях использования языка программирования Julia для решения научных и технических задач //Вестник Московского государственного технического университета им. НЭ Баумана. Серия «Приборостроение». – 2020. – №. 2 (131). **DOI:** 10.18698/0236-3933-2020-2-27-43

12. Dantzig G. B., Thapa M. N. Linear programming 1: introduction. – Springer Science & Business Media, 2006.

13. Степанов Н.Ф., Ерлыкина М.Е., Филиппов Г.Г. Методы линейной алгебры в физической химии.-М.: Изд-во Московского ун-та, 1976. – 362 с.

14. Engheim E. Julia as a Second Language. – Manning Publications Co, 2023

15. Энгхейм Э. Julia в качестве второго языка– М.: ДМК Пресс, 2023. – 446 с.