Manual de Julia

Orientado al cálculo científico y análisis de datos

Alfredo Sánchez Alberca

2022-01-06

Tabla de contenidos

Prefacio

¡Bienvenido al Manual de Julia!

Este libro presenta una introducción al lenguaje de programación Julia con un enfoque orientado al cálculo científico y el análisis de datos.

Licencia

1 Introducción

1.1 ¿Por qué Julia?

Julia es otro lenguaje de programación más, orientado a cálculo científico el análisis de datos similar a Python, R o Matlab.

¿De veras necesitamos aprender otro lenguaje más?

1.2 ¿Qué pretende ser Julia?

De los creadores de Julia:

> *We want a language that is:*
>
> - *Open source.*
> - *With the speed of C.*
> - *Obvious, familiar mathematical notation like Matlab.*
> - *As usable for general programming as Python.*
> - *As easy for statistics as R.*
> - *As natural for string processing as Perl.*
> - *As powerful for linear algebra as Matlab.*
> - *As good at gluing programs together as the shell.*
> - *Dirt simple to learn, yet keeps the most serious hackers happy.*

1.3 ¿Qué es Julia?

- Julia es un lenguaje de alto nivel con una sintaxis fácil de aprender (similar a Python, R o Matlab) que permite escribir símbolos matemáticos en las expresiones (UTF-8).
- Julia es un lenguaje muy veloz (equiparable a C en muchas tareas.)
- Lenguaje dinámico (tipado dinámico y despacho múltiple).
- De propósito general, pero orientado a la computación científica y el análisis de grandes volúmenes de datos.
- Creado en 2019 en el MIT por el equipo del profesor Edelman.
- Última versión: 1.7 (bastante maduro).
- Desarrollado por una gran comunidad científica.
- Repositorio de paquetes de código abierto con más de 3000 paquetes en dominios muy diversos.

1.3.1 Comparativa de rapidez de Julia con otros lenguajes

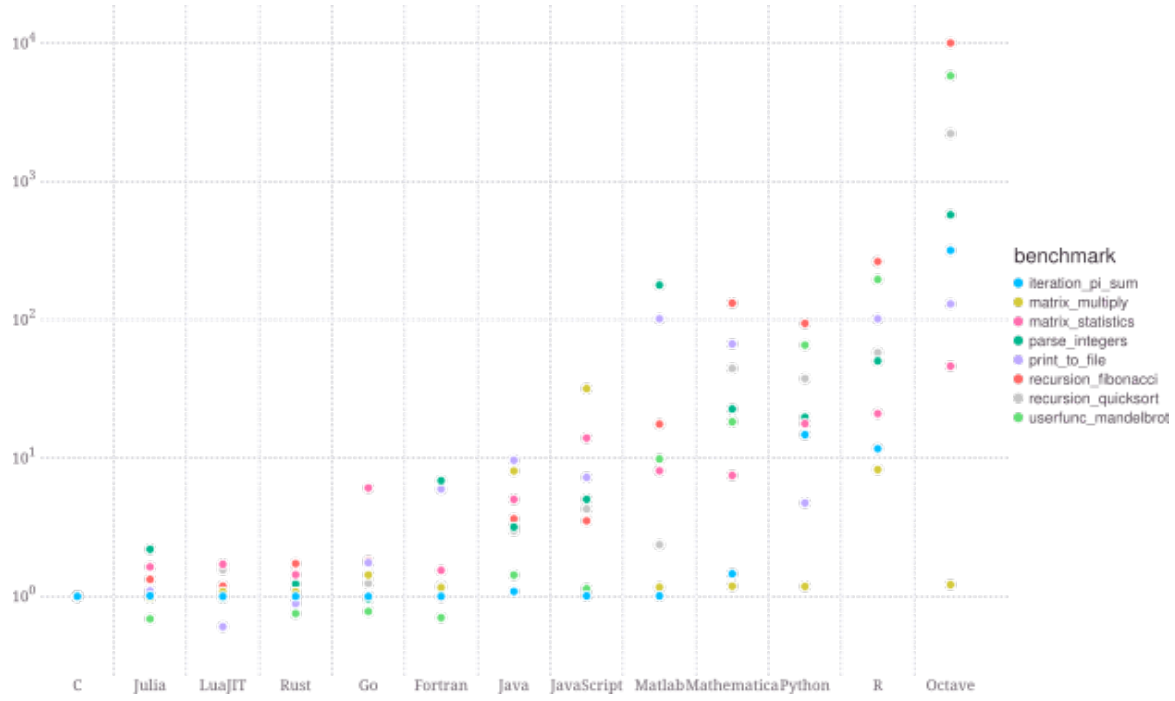


Figura 1.1: Comparativa de Julia con otros lenguajes

1.4 Algunas razones más para pensárselo

- Evita el problema de la doble reescritura de código (prototipo + versión operativa).
- Permite la programación paralela y distribuida.
- Puede ejecutar código realizado en otros lenguajes como C, Fortran, Python, R, Matlab, etc.

1.5 Instalación de Julia

1. Descargar e instalar Julia.
2. Añadir Julia al PATH del sistema.

Una vez instalado, para ejecutar el intérprete de Julia basta con abrir una terminal y teclear `julia`.

```
prompt> julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
```

```
   (_)     | (_) (_)    |
    _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
   | | | | | | |/ _` |  |
   | | |_| | | | (_| |  |  Version 1.7.3 (2022-05-06)
  _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
 |__/                   |

julia>
```

1.6 El REPL de Julia

El REPL[1] de Julia permite ejecutar código de Julia tecleándolo directamente en la terminal.

```
julia> 2 + 3
5
```

Tiene, además, varios modos:

- `;` para abrir el modo shell.
- `]` para abrir el modo de gestión de paquetes.
- `?` para abrir el modoe de ayuda.to open help mode
- `<backspace>` para volver al modo normal.

1.7 El gestor de paquetes de Julia

Como en otros lenguajes, es posible crear módulos o paquetes con código que puede ser reutilizado. Julia tiene un potente gestor de paquetes que facilita la búsqueda, instalación, actualización y eliminación de paquetes.

Por defecto el gestor de paquetes utiliza el repositorio de paquetes oficial pero se pueden instalar paquetes de otros repositorios.

Para entrar en el modo de gestión de paquetes hay que teclear `]`. Esto produce un cambio en el *prompt* del REPL de Julia.

Los comandos más habituales son:

- `add p`: Instala el paquete `p` en el entorno activo de Julia.
- `update`: Actualiza los paquetes del entorno activo de Julia.
- `status`: Muestra los paquetes instalados y sus versiones en el entorno activo de Julia.
- `remove p`: Elimina el paquete `p` del entorno activo de Julia.

[1] REPL es el acrónimo de Read, Evaluate, Print and Loop.

Ejemplo

```
(@v1.7) pkg> add CSV
   Resolving package versions...
   Installed CodecZlib       v0.7.0
   Installed SentinelArrays     v1.3.13
   Installed WeakRefStrings     v1.4.2
   Installed InlineStrings     v1.1.4
   Installed FilePathsBase     v0.9.18
   Installed TranscodingStreams   v0.9.6
   Installed CSV           v0.10.4
    Updating `~/.julia/environments/v1.7/Project.toml`
  [336ed68f] + CSV v0.10.4
    Updating `~/.julia/environments/v1.7/Manifest.toml`
  [336ed68f] + CSV v0.10.4
  [944b1d66] + CodecZlib v0.7.0
  [48062228] + FilePathsBase v0.9.18
  [842dd82b] + InlineStrings v1.1.4
  [91c51154] + SentinelArrays v1.3.13
  [3bb67fe8] + TranscodingStreams v0.9.6
  [ea10d353] + WeakRefStrings v1.4.2
Precompiling project...
```

1.8 Entornos de desarrollo

Cuando se trabaja en el desarrollo de varias aplicaciones, es habitual crear entornos de desarrollo individuales para instalar los paquetes de los que dependan y que no interfieran con los de otras aplicaciones.

Básicamente, un entorno de desarrollo es un directorio con código de Julia que contiene dos ficheros `Project.toml` y `Manifest.toml` que contienen los paquetes instalados en el entorno y sus dependencias.

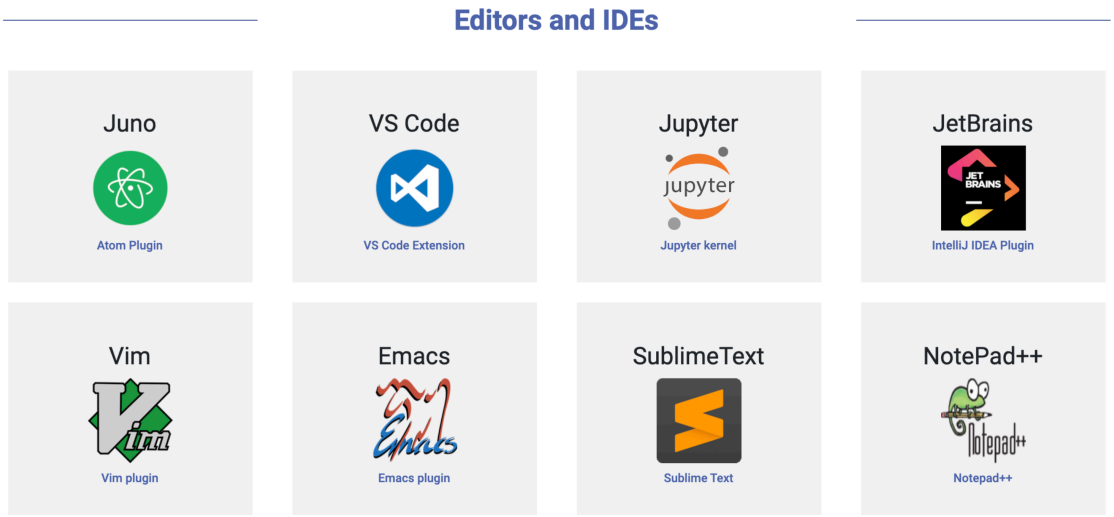
Para crear y activar un entorno de desarrollo se utilizan los siguiente comandos en el modo de gestión de paquetes:

- `generate dir`: Crea el directorio `dir` y lo convierte en un entorno de desarrollo de Julia.
- `activate dir`: Convierte en el entorno de desarrollo `dir` en el entorno activo dentro de la sesión de Julia. A partir de entonces, cualquier paquete que se instale o actualice será dentro de ese entorno.

Ejemplo

```
(@v1.7) pkg> generate app
  Generating  project app:
    app/Project.toml
    app/src/app.jl

(@v1.7) pkg> activate app
  Activating project at `~/app`

(app) pkg> add CSV # Instalar última versión del paquete CSV
...
(app) pkg> add DataFrames@v1.3.0;  # Instalar la versión 1.3.0 del paquete DataFrames
...
(app) pkg> status
     Project app v0.1.0
      Status `~/app/Project.toml`
  [336ed68f] CSV v0.10.4
  [a93c6f00] DataFrames v1.3.0
```

1.9 IDEs para Julia



https://julialang.org/

1.9.1 Visual studio code

- Descargar e instalar VSCode.
- Instalar la extensión de Julia.

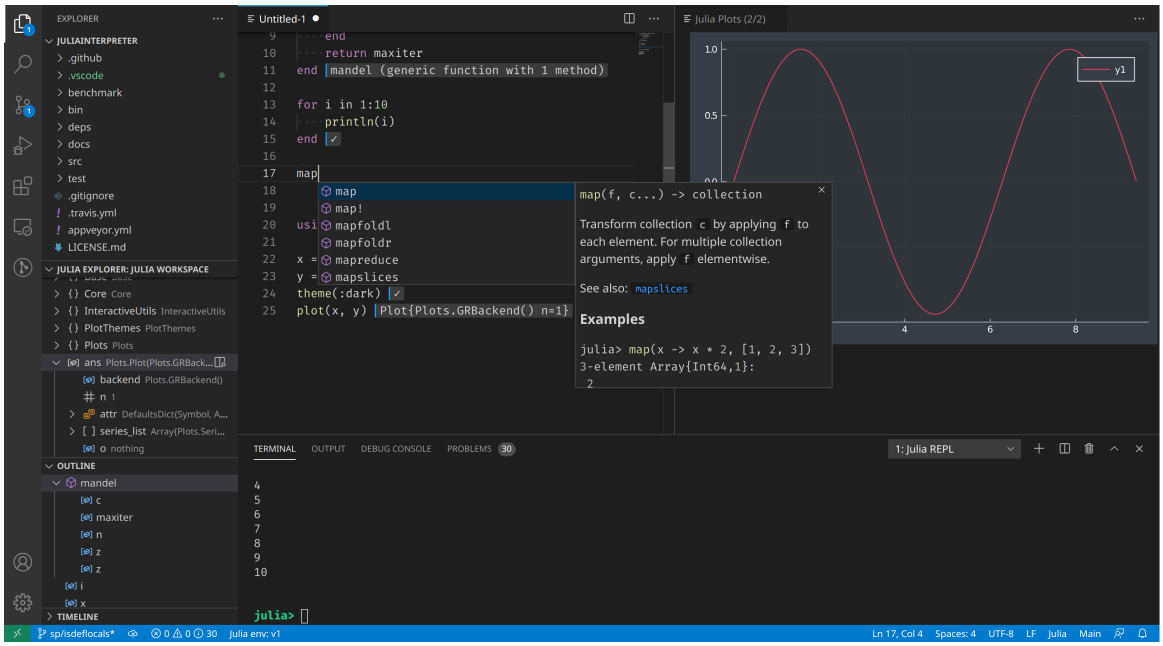


Figura 1.2: Visual Studio Code con la extensión para Julia

1.9.2 Jupyter

- Descargar e instalar Python
- Descargar e instalar Jupyter
- Instalar el paquete IJulia:

```
julia> using Pkg
julia> Pkg.add("IJulia")
```

1.9.3 Pluto

Pluto es entorno de desarrollo propio de Julia similar a Jupyter. Pluto permite crear *notebooks* reactivos cuyas celdas se actualizan cada vez que se produce un cambio en el estado del programa.

Para usarlo basta instalar el paquete `Pluto.jl`.

```
julia> using Pkg

julia> Pkg.add("Pluto");

julia> using Pluto
```

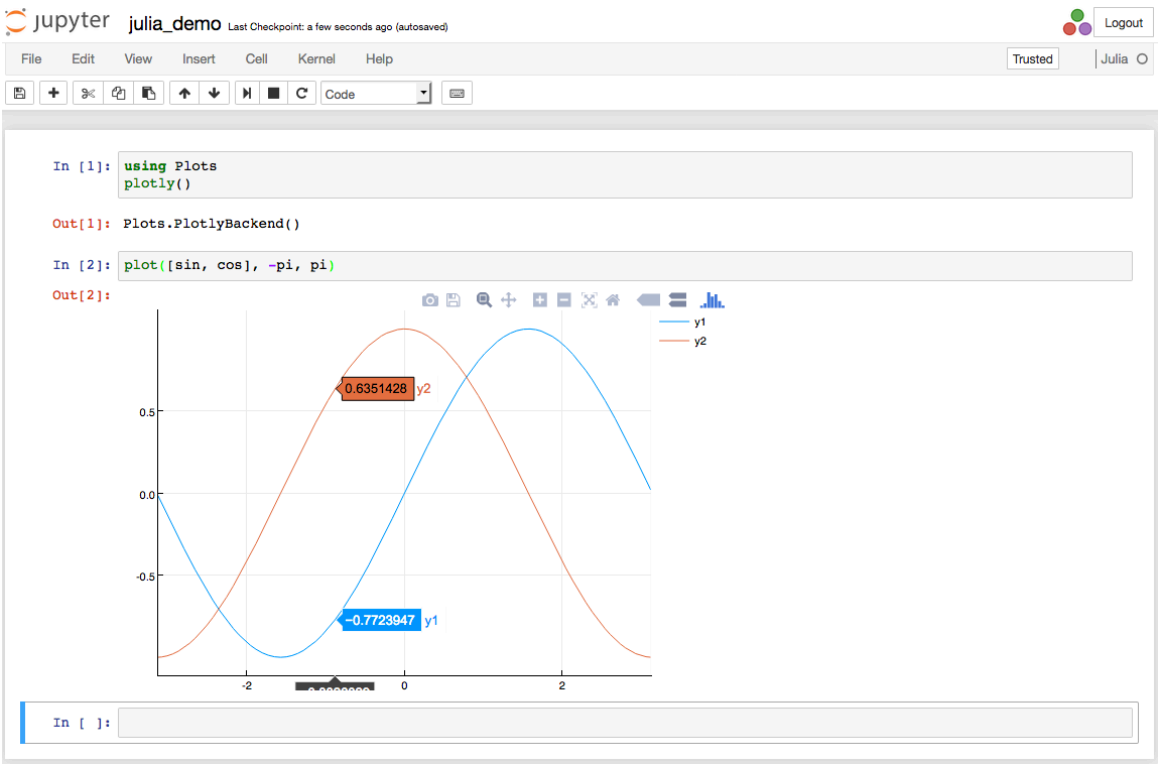


Figura 1.3: Jupyter con el kernel de Julia

```
julia> Pluto.run()
[ Info: Loading...
  Info:
  Opening http://localhost:1234/?secret=a63iBsIL in your default browser... ~ have fun!
  Info:
  Press Ctrl+C in this terminal to stop Pluto

Opening in existing browser session.
```

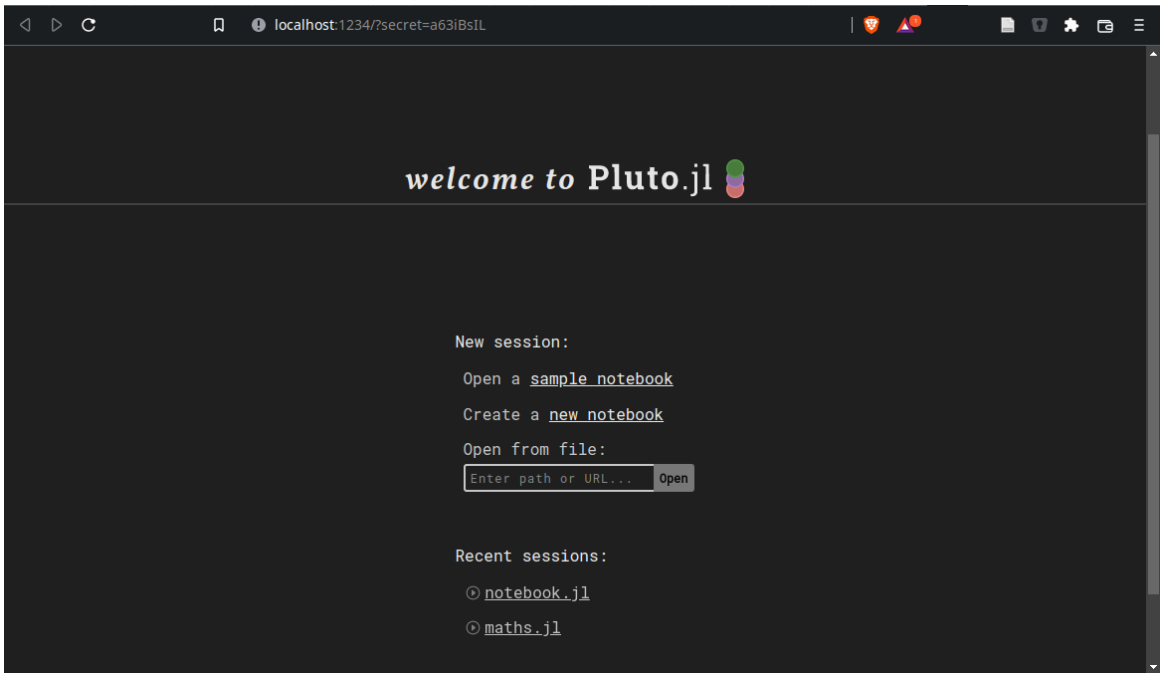


Figura 1.4: Página inicial de Pluto

2 Tipos de datos y variables

2.1 Tipos de datos numéricos

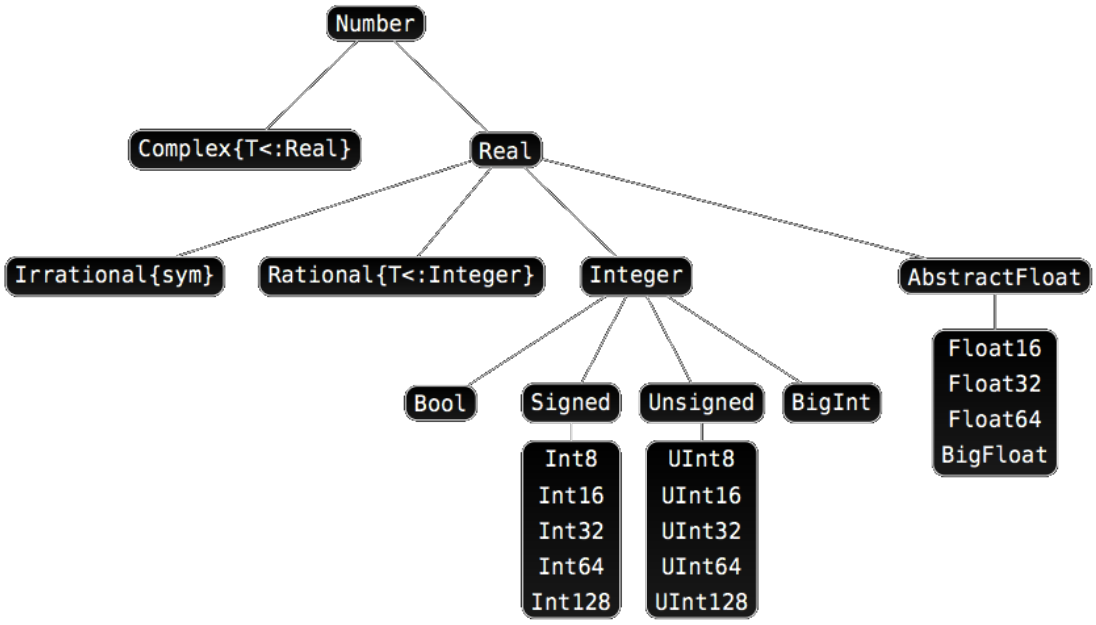
- Enteros: `Int64` (64 bits por defecto).
- Racionales: `Rational{Int64}`. Utilizando el operador `//`.
- Reales: `Float64` (64 bits por defecto).
- Complejos: `Complex{Int64}`. Utilizando `im` después de la parte imaginaria.

Para averiguar el tipo de un dato se utiliza la función `typeof()`.

2.2 Ejemplos de tipos de datos numéricos

```
julia> typeof(3)
Int64

julia> typeof(3/2)
Float64

julia> typeof(3//2)
Rational{Int64}

julia> typeof( )
Irrational{: }

julia> typeof(3+2im)
Complex{Int64}
```

2.3 Jerarquía de tipos de datos numéricos



2.4 Constantes predefinidas

Símbolo	Descripción
`pi` o	Número irracional π
(\euler + TAB)	Número irracional e
`Inf`	Infinito
`NaN`	Valor no numérico
`Missing`	Valor desconocido

2.5 Ejemplo de constantes predefinidas

```
julia>
  = 3.1415926535897...

julia> 1 / 0
Inf

julia> 0 / 0
```

```
NaN

julia> Inf + Inf
Inf

julia> Inf - Inf
NaN

julia> 0 * Inf
NaN
```

2.6 Tipos de datos alfanuméricos

- Caracteres: `Char`. Se representan entre comillas simples.
- Cadenas: `String`. Se representan entre comillas dobles.

```
julia> typeof('a')
Char

julia> typeof("julia")
String

julia> typeof("a")
String
```

2.7 Tipo de datos booleanos

- Booleanos: `bool`. Son un subtipo de los enteros `Integer`.

Solo contienen dos posibles valores: `true` (1) y `false` (0).

```
julia> typeof(true)
Bool

julia> typeof(false)
Bool

julia> typeof(1 < 2)
Bool

julia> true + true
2
```

2.8 Variables

Como lenguaje de tipado dinámico, no es necesario declarar una variable antes de usarla. Su tipo se infiere directamente del valor asociado.

```
julia> x = 1
1

julia> typeof(x)
Int64

julia> x = "julia"
"julia"

julia> typeof(x)
String
```

No obstante, para variables de ámbito local, por ejemplo en funciones, es posible fijar el tipo de una variable indicándolo detrás de su nombre con el operador `::`.

```
x::Int64
```

2.9 Nombres de variables

Julia reconoce la codificación Unicode (UTF-8), lo que permite utilizar caracteres con tildes, letras griegas, símbolos matemáticos y hasta emoticonos en los nombres de variables o funciones. Para ello se utilizan códigos especiales (en muchos casos son los mismos que en LaTeX), pulsando después la tecla de tabulación.

```
julia>   = 1
1

julia>   = 2
2

julia>   +
3

julia>   = "julia"
"julia"
```

Distingue entre mayúsculas y minúsculas.

2.9.1 Caracteres Unicode

La siguiente tabla contiene algunos caracteres Unicode habituales

Código	Símbolo	Uso notes
`\euler`		Constante de Euler `e`
`\pi`		Constante *pi*
`\alpha`		
`\beta`		
`\delta`		
`\Delta`	`Δ`	Variación
`\gamma`		
`\phi`		
`\Phi`	`Φ`	
`x\_1`	`x`	Subíndices
`x\^2`	x^2	Superíndices
`r\vec`	$\vec{r}$	Notación para vectores
`T\hat`	$\hat{T}$	Notación para vectores unitarios
`\partial`		Notación para derivadas parciales
`\nabla`		Notación para el gradiente
`\circ`		Operador de composición
`\cdot`		Operador de producto escalar
`\times`	`×`	Operador de producto vectorial

Los operadores pueden necesitar paréntesis, como por ejemplo `(f  g)(x)` para la composición de `g` con `f`.

2.10 Operadores aritméticos

Operador	Descripción
`x + y`	Suma
`x - y`	Resta
`x * y`	Producto
`x / y`	División
`x ÷ y`	Cociente división entera
`x % y`	Resto división entera
`x ^ y`	Potencia

2.11 Operadores de comparación

Operador	Descripción
`==`	Igualdad
`!=`,	Desigualdad
`<`	Menor que
`<=`,	Menor o igual que
`>`	Mayor que
`>=`,	Mayor o igual que

2.12 Operadores booleanos

Operador	Descripción
`!x`	Negación
`x && y`	Conjunción (y)
`x || y`	Disyunción (o)

2.13 Funciones numéricas predefinidas

2.13.1 Funciones de redondeo

Función	Descripción
`round(x)`	Devuelve el entero más próximo a `x`
`round(x, digits = n)`	Devuelve al valor más próximo a `x` con `n` decimales
`floor(x)`	Redondea `x` al próximo entero menor
`ceil(x)`	Redondea `x` al próximo entero mayor
`trunc(x)`	Devuelve la parte entera de `x`

2.13.2 Ejemplo de funciones de redondeo

```
julia> round(2.7)
3.0

julia> floor(2.7)
2.0

julia> floor(-2.7)
-3.0

julia> ceil(2.7)
```

```
3.0

julia> ceil(-2.7)
-2.0

julia> trunc(2.7)
2.0

julia> trunc(-2.7)
-2.0

julia> round(2.5)
2.0

julia> round(2.786, digits = 2)
2.79
```

2.13.3 Funciones de división

Función	Descripción
`div(x,y)`, `x÷y`	Cociente de la división entera
`fld(x,y)`	Cociente de la división entera redondeado hacia abajo
`cld(x,y)`	Cociente de la división entera redondeado hacia arriba
`rem(x,y)`, `x%y`	Resto de la división entera. Se cumple `x == div(x,y)*y + rem(x,y)`
`mod(x,y)`	Módulo con respecto a `y`. Se cumple `x == fld(x,y)*y + mod(x,y)`
`gcd(x,y...)`	Máximo común divisor positivo de `x`, `y`,...
`lcm(x,y...)`	Mínimo común múltiplo positivo de `x`, `y`,...

2.13.4 Ejemplo de funciones de división

```
julia> div(5,3)
1

julia> cld(5,3)
2

julia> 5%3
2

julia> -5%3
-2
```

```
julia> mod(5,3)
2

julia> mod(-5,3)
1

julia> gcd(12,18)
6

julia> lcm(12,18)
36
```

2.13.5 Funciones para el signo y el valor absoluto

Función	Descripción
`abs(x)`	Valor absoluto de `x`
`sign(x)`	Devuelve 1 si `x` es positivo, -1 si es negativo y 0 si es 0.

```
julia> abs(2.5)
2.5

julia> abs(-2.5)
2.5

julia> sign(-2.5)
-1.0

julia> sign(0)
0

julia> sign(2.5)
1.0
```

2.13.6 Raíces, exponenciales y logaritmos

Función	Descripción
`sqrt(x)`, √x	Raíz cuadrada de `x`
`cbrt(x)`, ∛x	Raíz cúbica de `x`
`exp(x)`	Exponencial de `x`
`log(x)`	Logaritmo neperiano de `x`

Función	Descripción
`log(b,x)`	Logaritmo en base `b` de `x`
`log2(x)`	Logaritmo en base 2 de `x`
`log10(x)`	Logaritmo en base 10 de `x`

2.13.7 Ejemplo de raíces, exponenciales y logaritmos

```
julia> sqrt(4)
2.0

julia> cbrt(27)
3.0

julia> exp(1)
2.718281828459045

julia> exp(-Inf)
0.0

julia> log(1)
0.0

julia> log(0)
-Inf

julia> log(-1)
ERROR: DomainError with -1.0:
log will only return a complex result if called with a complex argument.
...

julia> log(-1+0im)
0.0 + 3.141592653589793im

julia> log2(2^3)
3.0
```

2.13.8 Funciones trigonométricas

Función	Descripción
`hypot(x,y)`	Hipotenusa del triángulo rectángulo con catetos `x` e `y`
`sin(x)`	Seno del ángulo `x` en radianes

Función	Descripción
`sind(x)`	Seno del ángulo `x` en grados
`cos(x)`	Coseno del ángulo `x` en radianes
`cosd(x)`	Coseno del ángulo `x` en grados
`tan(x)`	Tangente del ángulo `x` en radianes
`tand(x)`	Tangente del ángulo `x` en grados
`sec(x)`	Secante del ángulo `x` en radianes
`csc(x)`	Cosecante del ángulo `x` en radianes
`cot(x)`	Cotangente del ángulo `x` en radianes

2.13.9 Ejemplo de funciones trigonométricas

```
julia> sin( /2)
1.0

julia> cos( /2)
6.123233995736766e-17

julia> cosd(90)
0.0

julia> tan( /4)
0.9999999999999999

julia> tand(45)
1.0

julia> tan( /2)
1.633123935319537e16

julia> tand(90)
Inf

julia> sin( /4)^2 + cos( /4)^2
1.0
```

2.13.10 Funciones trigonométricas inversas

Función	Descripción
`asin(x)`	Arcoseno (inversa del seno) de `x` en radianes
`asind(x)`	Arcoseno (inversa del seno) de `x` en grados

Función	Descripción
`acos(x)`	Arcocoseno (inversa del coseno) de `x` en radianes
`acosd(x)`	Arcocoseno (inversa del coseno) de `x` en grados
`atan(x)`	Arcotangente (inversa de la tangente) de `x` en radianes
`atand(x)`	Arcotangente (inversa de la tangente) de `x` en grados
`asec(x)`	Arcosecante (inversa de la secante) de `x` en radianes
`acsc(x)`	Arcocosecante (inversa de la cosecante) de `x` en radianes
`acot(x)`	Arcocotangente (inversa de la cotangente) de `x` en radianes

2.13.11 Ejemplo de funciones trigonométricas inversas

```
julia> asin(1)
1.5707963267948966

julia> asind(1)
90.0

julia> acos(-1)
3.141592653589793

julia> atan(1)
0.7853981633974483

julia> atand(tan( /4))
45.0
```

2.14 Precedencia de operadores

De mayor a menor prioridad.

Categoría	Operadores	Asociatividad
Exponenciación	`^`	Derecha
Unarios	`+ - √`	Derecha
Fracciones	`//`	Izquierda
Multiplicación	`* / % & \ ÷`	Izquierda
Adición	`+ - |`	Izquierda
Comparaciones	`> < >= <= == != !==`	
Asignaciones	`= += -= *= /= //= ^= ÷= %= |= &=`	Derecha

2.15 Operaciones con cadenas

Las cadenas son secuencias de caracteres alfanuméricos del tipo `char` entre dobles comillas.

Cada carácter tiene asociado un índice entero. El primer carácter de la cadena tiene índice 1.

Índice	1	2	3	4	5
Cadena	j	u	l	i	a

Podemos acceder a cada carácter usando su índice entre corchetes a continuación de la cadena:

- `s[i]`: Devuelve el carácter con índice `i` en la cadena `s`.

```
julia> c = "julia"
"julia"

julia> c[2]
'u': ASCII/Unicode U+0075 (category Ll: Letter, lowercase)
```

2.15.1 Acceso a caracteres Unicode

Sin embargo, como Julia permite caracteres Unicode, el índice de un carácter en una cadena, no siempre se corresponde con su posición en la cadena. Ello es debido a que la codificación UTF-8 no utiliza el mismo número de bytes para representar los caracteres Unicode. Mientras que los caracteres habituales del código ASCII (letras romanas y números árabes) solo necesitan un byte, otros caracteres como los símbolos matemáticos requieren más.

Índice	1	4	5	6	9
Cadena		x			y

2.15.2 Ejemplo de acceso a caracteres Unicode

```
julia> c = " x  y"
" x  y"

julia> c[1]
' ': Unicode U+2200 (category Sm: Symbol, math)

julia> c[2]
ERROR: StringIndexError: invalid index [2],
valid nearby indices [1]=>' ', [4]=>'x'
```

```
Stacktrace:
 [1] string_index_err(s::String, i::Int64)
   @ Base ./strings/string.jl:12
 [2] getindex_continued(s::String, i::Int64, u::UInt32)
   @ Base ./strings/string.jl:233
 [3] getindex(s::String, i::Int64)
   @ Base ./strings/string.jl:226
 [4] top-level scope
   @ REPL[128]:1
```

2.15.3 Acceso a índices en cadenas

Las siguientes funciones permiten acceder a los índices de una cadena:

- `firstindex(c)`: Devuelve el índice del primer carácter de la cadena `c`.
- `lastindex(c)`: Devuelve el índice del primer carácter de la cadena `c`.
- `nextind(c, i)`: Devuelve el índice del carácter de la cadena `c` que sigue al carácter con índice `i`.
- `prevind(c, i)`: Devuelve el índice del carácter de la cadena `c` que sigue al carácter con índice `i`.

2.15.4 Ejemplo de acceso a índices en cadenas

```
julia> firstindex(c)
1

julia> lastindex(c)
9

julia> c[9]
'y': ASCII/Unicode U+0079 (category Ll: Letter, lowercase)

julia> nextind(c,1)
4

julia> prevind(c, lastindex(c))
6
```

2.15.5 Subcadenas

Para obtener subcadenas se usan también los corchetes indicando los índices de inicio y fin separados por `:`.

- `s[i:j]`: Devuelve la subcadena que va desde el índice `i` al índice `j`, ambos incluidos.

También se pueden obtener subcadenas con la siguiente función:

- `SubString(s, i, j)`: Devuelve la subcadena que va desde el índice `i` al índice `j`, ambos incluidos.

```
julia> c = "julia"
"julia"

julia> c[2:4]
"uli"

julia> SubString(c, 2, 4)
"uli"
```

2.16 Concatenación de cadenas

- `a * b`: Devuelve la cadena que resulta de concatenar las cadenas `a` y `b`.
- `a ^ i`: Devuelve la cadena que resulta de repetir la cadena `a` el número de veces `i`.
- `repeat(a, i)`: Devuelve la cadena que resulta de repetir la cadena `a` el número de veces `i`.

```
julia> a = "Hola"
"Hola"

julia> b = "Julia"
"Julia"

julia> a * b
"HolaJulia"

julia> b ^ 3
"JuliaJuliaJulia"
```

2.16.1 Interpolación de cadenas

En una cadena se pueden introducir variables o expresiones precedidas del símbolo `$`, de manera que al evaluarlas julia sustituye la variable o expresión por su valor. Esto es muy util para formatear salidas.

```
julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

```
julia> s = "Julia"
"Julia"

julia> "Hola $s"
"Hola Julia"
```

2.16.2 Otras operaciones comunes con cadenas

- `length(c)`: Devuelve el número de caracteres de la cadena `c`.
- `findfirst(a, c)`: Devuelve el índice de la primera ocurrencia de la cadena `a` en la cadena `c`. Si `a` no es una subcadena de `c` devuelve nada (tipo `Nothing`).
- `findlast(a, c)`: Devuelve el índice de la última ocurrencia de la cadena `a` en la cadena `c`. Si `a` no es una subcadena de `c` devuelve nada (tipo `Nothing`).
- `findnext(a, c, i)`: Devuelve el índice de la primera ocurrencia de la cadena `a` en la cadena `c` posterior al índice `i`.
- `findprev(a, c, i)`: Devuelve el índice de la última ocurrencia de la cadena `a` en la cadena `c` anterior al índice `i`.

2.16.3 Otras operaciones comunes con cadenas

- `occursin(a, c)`: Devuelve `true` si la cadena `a` es una subcadena de `c`, y `false` en caso contrario.
- `contains(c, a)`: Devuelve `true` si la cadena `a` es una subcadena de `c`, y `false` en caso contrario.
- `replace(c, a => b)`: Devuelve la cadena que resulta de sustituir la cadena `a` por la `b` en la cadena `c`.
- `lowercase(c)`: Devuelve la cadena `c` en minúsculas.
- `uppercase(c)`: Devuelve la cadena `c` en mayúsculas.
- `prefix(c, a)`: Devuelve `true` si la cadena `a` es un prefijo de la cadena `c`.
- `suffix(c, a)`: Devuelve `true` si la cadena `a` es un sufijo de la cadena `c`.
- `split(c, a)`: Devuelve una lista con las cadenas que resulpan de partir la cadena `c` por el delimitador `a`.

2.16.4 Ejemplo de otras operaciones con cadenas

```
julia> c = "Hola Julia"
"Hola Julia"

julia> length(c)
10
```

```
julia> findfirst("a", c)
4:4

julia> findlast("Ju", c)
6:7

julia> findlast("x", c)

julia> occursin("Julia", c)
true

julia> occursin("julia", c)
false

julia> replace(c, "a" => "o")
"Holo Julio"

julia> uppercase(c)
"HOLA JULIA"

julia> split(c, " ")
2-element Vector{SubString{String}}:
 "Hola"
 "Julia"
```

2.17 Entrada y salida por terminal

Las siguientes funciones muestran una cadena en la terminal:

- `print(c)`: Muestra por la terminal la cadena `c` sin cambiar de línea.
- `println(c)`: Muestra por la terminal la cadena `c` y cambia de línea.

La siguiente función permite leer una línea de texto desde la terminal:

- `readline()`: Devuelve en una cadena una línea de texto introducida por el usuario en la terminal (hasta el carácter de cambio de línea `\n`)

```
julia> print("¿Cómo te llamas?")
¿Cómo te llamas?
julia> nombre = readline()
Alf
"Alf"

julia> println("Hola $nombre")
```

```
Hola Alf
```

2.17.1 Conversión de cadenas en números

La función `readline()` siempre devuelve una cadena aún cuando se pregunte al usuario por un valor numérico. Para convertir una cadena en un dato numérico se utiliza la siguiente función:

- `parse(tipo, c)`: Covierte la cadena `c` a un número del tipo numérico `tipo`, siempre que puedad realizarse la conversión.

```
julia> print("Introduce tu edad")
Introduce tu edad
julia> edad = parse(Int, readline())
18
18

julia> println("Vas a cumplir $(edad + 1) años")
Vas a cumplir 19 años

julia> typeof(edad)
Int64
```

3 Estructuras de control

3.1 Condicionales

`if` *condición 1*
  bloque código 1
`elseif` *condición 2*
  bloque código 2
…
`else`
  bloque código n
`end`

La indentación de los bloques de código no es necesaria, pero es una buena práctica.

3.1.1 Ejemplo de condicional

```
julia> x = -1
-1

julia> if x > 0
         signo = "positivo"
       elseif x < 0
         signo = "negativo"
       else
         signo = "nulo"
       end
"negativo"
```

3.1.2 Operador condicional

Una forma abreviada de la estructura condicional es el operador condicional.

condición `?` *bloque true* `:` *bloque false*

Este operador ejecuta el primer bloque de código si la condición es `true` y el segundo en caso contrario.

```
julia> x > 0 ? signo = "positivo" : signo = "negativo"
"negativo"
```

3.2 Bucles

3.3 Bucles iterativos

```
for iterador in secuencia
  bloque código
end
```

Ejecuta el bloque de código tantas veces como elementos tenga la *secuencia*. En cada iteración el *iterador* toma como valor el siguiente elemento de la *secuencia*.

```
julia> c = "Julia"
"Julia"

julia> for i in c
         println(i)
       end
J
u
l
i
a
```

3.3.1 Bucles iterativos con rangos

En muchas ocasiones la secuencia que se recorre en un bucle iterativo se genera mediante un rango, que es una secuencia de números igualmente espaciados. Existen distintas funciones para generar rangos:

- `i:j`: Genera la secuenciad de números desde `i` hasta `j`.
- `i:j:k`: Genera la secuencia de números desde `i` hasta `k` dando saltos de `j`.
- `StepRange(i, j, k)`: Genera la secuencia de números desde `i` hasta `k` dando saltos de `j`.
- `range(i, j, n)`: Genera una secuencia de `n` números desde `i` hasta `j`.

3.3.2 Ejemplo de bucles iterativos con rangos

```
julia> for i in 1:2:10
         println(i)
       end
1
3
5
7
9

julia> for i = range(0, 10, 5)
         println(i)
       end
0.0
2.5
5.0
7.5
10.0
```

3.3.3 Bucles iterativos anidados

En muchas ocasiones es habitual incluir un bucle iterativo en el bloque de código de otro bucle iterativo, lo que se conoce como *bucles anidados*.

Julia permite simplificar estas estructuras indicando los iteradores en la cabecera de un único bucle.

```
julia> for i in "abc", j = 1:2
         println(i,j)
       end
a1
a2
b1
b2
c1
c2
```

3.4 Bucles condicionales

```
while condición
  bloque código
end
```

Repite la ejecución del bloque de código mientras que la *condición* sea cierta.

```
julia> x = 3
3

julia> while x >= 0
          println(x)
          x -= 1
       end
3
2
1
0
```

3.4.1 Interrupción de bucles

La instrucción `break` provoca inmediatamente la finalización de un bucle tanto iterativo como condicional.

```
julia> x=3
3

julia> while true
          if x < 0
             break
          end
          println(x)
          x -= 1
       end
3
2
1
0
```

3.4.2 Salto de bucles

La instrucción `continue` provoca la finalización del bloque de código de un bucle y pasa inmediatamente a la siguiente iteración.

```
julia> for i in 1:10
          if i % 2 == 0
             continue
          end
```

```
        println(i)
    end
```

```
1
3
5
7
9
```

4 Tipos de datos compuestos

4.1 Colecciones de datos

Colecciones de datos con distinta estructura y semántica.

- Arrays
  - Vectores
  - Matrices
- Tuplas
- Diccionarios
- Conjuntos

4.2 Arrays

Un **array** es una colección ordenada de datos de un mismo tipo.

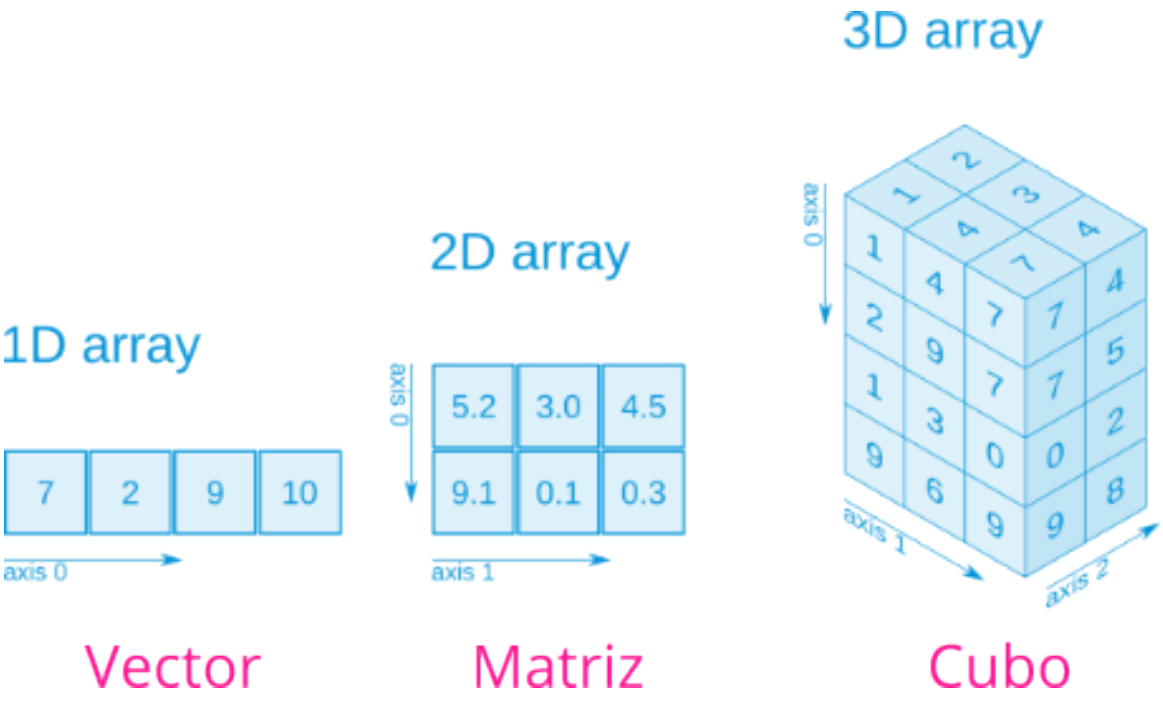
El tipo del array se infiere automáticamente a partir de los tipos de sus elementos. Si los elementos son de distintos tipos se convierten al tipo más específico de la jerarquía de tipos del que los tipos de los elementos son subtipos.

Se construyen escribiendo sus elementos separados por comas, puntos y comas o espacios entre corchetes.

```
julia> [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> [1.0, "julia", true]
3-element Vector{Any}:
    1.0
     "julia"
 true
```

4.3 Arrays multidimensionales

Los arrays pueden estructurar sus elementos en múltiples dimensiones. Dependiendo el número de dimensiones tenemos distintos tipos de arrays:



4.3.1 Funciones de arrays

- `length(A)`: Devuelve el número de elementos del array `A`.
- `eltype(A)`: Devuelve el tipo de los elementos del array `A`.
- `ndims(A)`: Devuelve el número de dimensiones del array `A`.
- `size(A)`: Devuelve una tupla con los tamaños de las dimensiones del array `A`.
- `size(A, n)`: Devuelve el tamaño de la dimensión `n` del array `A`.
- `axes(A)`: Devuelve una tupla con los índices válidos de cada dimensión del array `A`.
- `axes(A, n)`: Devuelve un rango con los índices válidos de la dimensión `n` del array `A`.
- `eachindex(A)`: Devuelve un iterador sobre los índices de los elementos del array `A`.

4.3.2 Constructores de arrays

- `zeros(dim)`: Devuelve un array de la dimensiones indicadas por la tupla `dim` con todos sus elementos ceros.
- `ones(dim)`: Devuelve un array de la dimensiones indicadas por la tupla `dim` con todos sus elementos unos.
- `fill(a, dim)`: Devuelve un array de la dimensiones indicadas por la tupla `dim` con todos sus elementos iguales `a`.

- `rand(dim)`: Devuelve un array de la dimensiones indicadas por la tupla `dim` con todos sus elementos números aleatorios entre 0 y 1.
- `trues(dim)`: Devuelve un array de la dimensiones indicadas por la tupla `dim` con todos sus elementos `true`.
- `falses(dim)`: Devuelve un array de la dimensiones indicadas por la tupla `dim` con todos sus elementos `false`.

4.3.3 Ejemplos de constructores de arrays

```
julia> zeros(3)  # Vector de tamaño 3
3-element Vector{Float64}:
 0.0
 0.0
 0.0

julia> rand(3,2)  # Matriz de tamaño 3 x 2
3×2 Matrix{Float64}:
 0.1469    0.891839
 0.953462  0.395681
 0.819468  0.720606

julia> fill( , 2, 2)
2×2 Matrix{Irrational{:}}:
```

4.3.4 Redimensionamiento de arrays

La siguientes funciones permiten cambiar las dimensiones de un array, reestructurando sus elementos:

- `reshape(A, dim)`: Devuelve el array que resulta de redimiensionar el array `A` con las dimensiones indicadas por la tupla `dim`.
- `permutedims(A)`: Devuelve el array de resulta de trasponer el array `A`.

> **Advertencia**
>
> El array resultante debe tener los mismos elementos que el array original, por lo que si las dimensiones no son compatibles se produce un error.

4.3.5 Ejemplo de redimensionamiento de arrays

```
julia> v = [1, 2, 3, 4, 5, 6]
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6

julia> reshape(v, 2, 3)
2×3 Matrix{Int64}:
 1  3  5
 2  4  6

julia> reshape(v, 3, 2)
3×2 Matrix{Int64}:
 1  4
 2  5
 3  6

julia> permutedims(reshape(v, 3, 2))
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

4.3.6 Comprensión de arrays

Una potente técnica de creación de arrays es la comprensión de arrays, que consiste en generar los elementos del array a partir de uno o varios iteradores.

- `[exp for i = ite]`: Devuelve el vector cuyos elementos resultan de evaluar a expresión `exp` para cada valor `i` del iterador `ite`.
- `[exp for i = ite if cond]`: Devuelve el vector cuyos elementos resultan de evaluar a expresión `exp` para cada valor `i` del iterador `ite` que cumpla la condición `cond`.

Se pueden utilizar varios iteradores para crear arrays de varias dimensiones.

```
julia> [i^2 for i = 1:4]
4-element Vector{Int64}:
  1
  4
  9
```

```
 16

julia> [i^2 for i = 1:4 if i % 2 == 0]
2-element Vector{Int64}:
  4
 16

julia> [i+j for i = 1:2, j = 3:4]
2×2 Matrix{Int64}:
 4  5
 5  6
```

4.4 Vectores

Los vectores son arrays de una dimensión.

Se construyen escribiendo sus elementos separados por comas o puntos y comas entre corchetes.

```
julia> v = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> length(v)
3

julia> eltype(v)
Int64

julia> ndims(v)
1

julia> size(v)
(3,)

julia> eachindex(v)
Base.OneTo(3)
```

4.4.1 Acceso a los elementos de un vector

El acceso a los elementos de un vector es mediante *índices*. Cada elemento del vector tiene asociado un índice entero que se corresponde con su posición desde 1 hasta el número de elementos.

- `v[i]`: Devuelve el elemento del vector `v` con índice `i`.

> **Advertencia**
>
> Si se proporciona un índice no válido se produce un error.

Las palabras reservadas `begin` y `end` se utilizan para referirse al primer y último índice de un vector.

4.4.2 Ejemplo de acceso a los elementos de un vector

```
julia> v = [2, 4, 6]
3-element Vector{Int64}:
 2
 4
 6

julia> v[2]
4

julia> v[end]
6

julia> v[4]
ERROR: BoundsError: attempt to access 3-element Vector{Int64} at index [4]
Stacktrace:
 [1] getindex(A::Vector{Int64}, i1::Int64)
   @ Base ./array.jl:861
 [2] top-level scope
   @ REPL[4]:1
```

4.4.3 Acceso a múltiples elementos de un vector

Es posible extraer varios elementos de un vector a la vez indicando los índices mediante un rango o un vector de enteros.

- `v[i:j]`: Devuelve un vector con los elementos del vector `v` desde el índice `i` al `j`.

- `v[u]`: Devuelve un vector con los elementos del vector `v` correspondientes a los índices del vector `u`.

```
julia> v = [2, 4, 6, 8];

julia> v[2:3]
2-element Vector{Int64}:
 4
 6

julia> v[[2,4,3]]
3-element Vector{Int64}:
 4
 8
 6
```

4.4.4 Modificación de los elementos de un vector

También es posible modificar un vector asignando nuevos elementos mediante los índices.

- `v[i] = a`: Añade el elemento `a` al vector `v` en el índice `i`.

```
julia> v = [2, 4, 6]
3-element Vector{Int64}:
 2
 4
 6

julia> v[2] = 0
0

julia> v
3-element Vector{Int64}:
 2
 0
 6
```

4.4.5 Añadir elementos a un vector

Las siguientes funciones permiten añadir elementos al final de un vector:

- `push!(v, a)`: Añade el elemento `a` al final del vector `v`.
- `append!(v, u)`: Añade los elementos del vector `u` al final del vector `v`.

```
julia> v = [];

julia> push!(v, 1)
1-element Vector{Any}:
 1

julia> append!(v, [2, 3])
3-element Vector{Any}:
 1
 2
 3

julia> v
3-element Vector{Any}:
 1
 2
 3
```

4.4.6 Recorrer un vector

Una operación habitual es recorrer los elementos de un vector para hacer cualquier operación con ellos. Existen dos posibilidades: recorrer el vector por índice o por valor.

```
julia> v = [2, 4, 6];

julia> for i in v  # Recorrido por valor
         println(i)
       end
2
4
6

julia> for i in eachindex(v)  # Recorrido por índice
         println(v[i])
       end
2
4
6
```

4.4.7 Operaciones con vectores numéricos

- `minimum(v)`: Devuelve el menor elemento del vector `v`.
- `maximum(v)`: Devuelve el mayor elemento del vector `v`.

- `argmin(v)`: Devuelve el índice del menor elemento del vector `v`.
- `argmax(v)`: Devuelve el índice del mayor elemento del vector `v`.
- `sum(v)`: Devuelve la suma de los elementos del vector `v`.
- `prod(v)`: Devuelve el producto de los elementos del vector `v`.
- `unique(v)`: Devuelve un vector con los elementos de `v` sin repetir.

```
julia> v = [4, 2, 3];

julia> maximum(v)
4

julia> argmax(v)
1

julia> sum(v)
9

julia> prod(v)
24
```

4.4.8 Ordenación de vectores

- `sort(v, rev=true)`: Devuelve el vector que resulta de ordenar en orden ascendente los elementos del vector `v`. Si se pasa `true` al parámetro `rev` el orden es descendente.
- `sort!(v, rev=true)`: Ordena el vector `v` en orden ascendente. Si se pasa `true` al parámetro `rev` el orden es descendente.
- `reverse(v)`: Devuelve el vector con los elementos del vector `v` en orden inverso.
- `reverse!(v)`: Modifica el vector `v` poniendo sus elementos en orden inverso.

4.4.9 Ejemplo de ordenación de vectores

```
julia> v = [4, 2, 3];

julia> sort(v)
3-element Vector{Int64}:
 2
 3
 4

julia> reverse(v)
3-element Vector{Int64}:
 3
 2
```

```
 4

julia> v
3-element Vector{Int64}:
 4
 2
 3

julia> reverse!(v)
3-element Vector{Int64}:
 3
 2
 4

julia> v
3-element Vector{Int64}:
 3
 2
 4
```

4.4.10 Extensión de funciones a vectores

Si una función recibe un parámetro del tipo de los elementos de un vector, se puede aplicar la función a cada uno de los elementos del vector, extendiendo la llamada de la función sobre los elementos del vector. Para ello basta con añadir un punto entre el nombre de la función y el paréntesis de los argumentos.

- `f.(v)`: Devuelve el vector que resulta de aplicar la función `f` a cada uno de los elementos del vector `v`.

> **Advertencia**
>
> En la llamada a la función hay que pasarle com argumentos tantos vectores como parámetros tenga la función. Si los vectores son de distinto tamaño, se reciclan los de menor tamaño.

> **Advertencia**
>
> Si la función no devuelve ningún valor el resultado es un vector de valores `nothing`.

La extensión de funciones también funciona con operadores, poniendo el punto delante del operador.

4.4.11 Ejemplo de extensión de funciones a vectores

```
julia> v = [1, 4, 9];

julia> sqrt.(v)
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> v .^ 2
3-element Vector{Int64}:
  1
 16
 81

julia> base = [2,  , 10];

julia> log.(base, v)
3-element Vector{Float64}:
 0.0
 1.3862943611198906
 0.9542425094393249
```

4.4.12 Filtrado de vectores

Otra operación bastante común son los filtros de vectores. Se puede filtrar un vector a partir de un vector de booleanos del mismo tamaño.

- `v[u]`: Devuelve el vector con los elementos que tienen el mismo índice que los valores `true` del vector booleano `u`.

Esto permite aplicar filtros a partir de condiciones que devuelvan un vector de booleanos.

4.4.13 Ejemplo de filtrado de vectores

```
julia> v = [1, 2, 3, 4];

julia> v[[true, false, true, false]]
2-element Vector{Int64}:
 1
 3
```

```
julia> v .% 2 .== 0  # Condición
4-element BitVector:
 0
 1
 0
 1

julia> v[v .% 2 .== 0]  # Filtro de números pares
2-element Vector{Int64}:
 2
 4
```

4.4.14 Álgebra lineal con vectores

- `u + v`: Devuelve el vector que resulta de la suma de los vetores `u` y `v`.
- `u - v`: Devuelve el vector que resulta de la resta de los vetores `u` y `v`.
- `a * v`: De vuelve el vector que resulta de multiplicar el vector `v` por el escalar `a`.
- `v'`: Devuelve el vector que resulta de trasponer el vector `v`. Si `v` es un vector fila, `v'` es un vector columna y viceversa.

Con el paquete `LinearAlgebra` también están disponibles las siguientes funciones:

- `dot(u, v)`: Devuelve el producto escalar de los vectores `u` y `v`.
- `norm(v)`: Devuelve la norma (módulo) del vector `v`.

4.4.15 Ejemplo de álgebra lineal con vectores

```
using LinearAlgebra

julia> u = [1, 2, 3]; v = [1, 0, 2];

julia> u + v
3-element Vector{Int64}:
 2
 2
 5

julia> 2u
3-element Vector{Int64}:
 2
 4
 6
```

```
julia> dot(u, v)  # Producto escalar
7

julia> u'v  # Producto escalar
7

julia> norm(v)  # Norma o módulo
2.23606797749979

julia> u / norm(u)  # Vector unitario
3-element Vector{Float64}:
 0.2672612419124244
 0.5345224838248488
 0.8017837257372732
```

4.5 Matrices

Las matrices son arrays de dos dimensiones (filas x columnas).

Se construyen escribiendo sus elementos entre corchetes, separando los elementos por espacio y las filas por punto y coma `;`.

```
julia> A = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> length(A)
6

julia> eltype(A)
Int64

julia> ndims(A)
2

julia> size(A)
(2, 3)
```

4.5.1 Acceso a los elementos de una matriz

El acceso a los elementos de una matriz es mediante *índices*. Cada elemento de la matriz tiene asociado un par de índices enteros que se corresponde la fila y la columna que ocupa.

- `A[i, j]`: Devuelve el elemento de la matriz `A` con índice de fila `i` e índice de columna `j`.

> **Advertencia**
>
> Si se proporciona algún índice no válido se produce un error.

También se puede acceder a los elementos de una matriz mediante un único índice. En ese caso se obtiene el elemento con ese índice en el vector que resulta de concatenar los elementos de la matriz por columnas.

4.5.2 Ejemplo de acceso a los elementos de una matriz

```
julia> A = reshape(1:6, 2, 3)
2×3 reshape(::UnitRange{Int64}, 2, 3) with eltype Int64:
 1  3  5
 2  4  6

julia> A[2, 1]
2

julia> A[4]
4
```

4.5.3 Acceso a múltiples elementos de una matriz

Es posible extraer varios elementos de una matriz a la vez indicando los índices de las filas y las columnas mediante un rango o un vector de enteros.

- `A[i:j, k:l]`: Devuelve una matriz con los elementos desde el índice de fila `i` al `j` y el índice de columna `k` al `l` de la matriz `A`.
- `A[u, w]`: Devuelve una matriz con los elementos correspondientes a los índices de fila del vector `u` y los índices de columna del vector `w` de la matriz `A`.

4.5.4 Ejemplo de acceso a múltiples elementos de una matriz

```
julia> A = reshape(1:9, 3, :)
3×3 reshape(::UnitRange{Int64}, 3, 3) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9
```

```
julia> A[1:2, 2:3]
2×2 Matrix{Int64}:
 4  7
 5  8

julia> A[[1, 3], [3, 1]]
2×2 Matrix{Int64}:
 7  1
 9  3

julia> A[2, :]  # Segundo vector fila
3-element Vector{Int64}:
 2
 5
 8
```

4.5.5 Modificación de los elementos de una matriz

También es posible modificar una matriz asignando nuevos elementos mediante los índices de fila y columna.

- `A[i, j] = a`: Añade el elemento `a` a la matriz `A` con el índice de fila `i` y el índice de columna `j`.

```
julia> A = zeros(2, 3)
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> A[2,3] = 1
1

julia> A
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  1.0
```

4.5.6 Concatenación de matrices

Dos o más matrices pueden concatenarse horizontal o verticalmente siempre que sus dimensiones sean compatibles.

- `[A B]`: Devuelve la matriz que resulta de concatenar horizontalmente las matrices `A` y `B`. Ambas matrices deben tener el mismo número de filas.

- `[A; B]`: Devuelve la matriz que resulta de concatenar verticalmente las matrices `A` y `B`. Ambas matrices deben tener el mismo número de columnas.

4.5.7 Ejemplo de concatenación de matrices

```
julia> A = zeros(2, 2)
2×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0

julia> B = ones(2, 1)
2×1 Matrix{Float64}:
 1.0
 1.0

julia> C = ones(1, 3)
1×3 Matrix{Float64}:
 1.0  1.0  1.0

julia> D = [A B]
2×3 Matrix{Float64}:
 0.0  0.0  1.0
 0.0  0.0  1.0

julia> [D ; C]
3×3 Matrix{Float64}:
 0.0  0.0  1.0
 0.0  0.0  1.0
 1.0  1.0  1.0
```

4.5.8 Concatenación de vectores

También es posible concatenar varios vectores horizontalmente o verticalmente para formar una matriz.

- `hcat(v...)`: Devuelve la matriz que resulta de concatenar horizontalmente los vectores del vector `v`.
- `vcat(v...)`: Devuelve la matriz que resulta de concatenar verticalmente los vectores del vector `v`.

4.5.9 Ejemplo de concatenación de vectores

```
julia> v = [[1, 2, 3], [4, 5, 6]]
2-element Vector{Vector{Int64}}:
 [1, 2, 3]
 [4, 5, 6]

julia> hcat(v...)
3×2 Matrix{Int64}:
 1  4
 2  5
 3  6

julia> v = [[1 2 3], [4 5 6]]
2-element Vector{Matrix{Int64}}:
 [1 2 3]
 [4 5 6]

julia> vcat(v...)
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

4.5.10 Recorrido de matrices

Una operación habitual es recorrer los elementos de una matriz para hacer una operación con ellos. El recorrido se suele hacer con dos bucles iterativos anidados.

```
julia> A = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> for i = 1:size(A, 1), j = 1:size(A, 2)  # Recorrido por filas
         println(A[i, j])
       end
1
2
3
4
5
6
```

```
julia> for j = 1:size(A, 2), i = 1:size(A, 1)  # Recorrido por columnas
         println(A[i, j])
       end
1
4
2
5
3
6
```

4.5.11 Operaciones con matrices numéricas

- `minimum(A)`: Devuelve el menor elemento de la matriz `A`.
- `maximum(A)`: Devuelve el mayor elemento de la matriz `A`.
- `argmin(A)`: Devuelve los índices de fila y columna del menor elemento de la matriz `A`.
- `argmax(A)`: Devuelve los índices de fila y columna del mayor elemento de la matriz `A`.
- `sum(A)`: Devuelve la suma de los elementos de la matriz `A`.
- `prod(A)`: Devuelve el producto de los elementos de la matriz `A`.

```
julia> A = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> minimum(A)
1

julia> argmax(A)
CartesianIndex(2, 3)

julia> sum(A)
21

julia> prod(A)
720
```

4.5.12 Extensión de funciones a matrices

Al igual que para vectores, se puede aplicar una una función a todos los elementos de una matriz. Para ello basta con añadir un punto entre el nombre de la función y el paréntesis de los argumentos.

- `f.(A)`: Devuelve la matriz que resulta de aplicar la función `f` a cada uno de los elementos de la matriz `A`.

> **Advertencia**
>
> En la llamada a la función hay que pasarle como argumentos tantos vectores como parámetros tenga la función. Si las matrices son de distinto tamaño, se reciclan las de menor tamaño.

La extensión de funciones también funciona con operadores, poniendo el punto delante del operador.

4.5.13 Ejemplo de extensión de funciones a matrices

```
julia> A = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> sqrt.(A)
2×3 Matrix{Float64}:
 1.0  1.41421  1.73205
 2.0  2.23607  2.44949

julia> A .+ 1
2×3 Matrix{Int64}:
 2  3  4
 5  6  7
```

4.5.14 Álgebra lineal con matrices

- `A + B`: Devuelve la matriz que resulta de la suma de las matrices `A` y `B`. Ambas matrices deben tener las mismas dimensiones.
- `A - B`: Devuelve la matriz que resulta de la resta de las matrices `A` y `B`. Ambas matrices deben tener las mismas dimensiones.
- `a * A`: Devuelve la matriz que resulta de multiplicar la matriz `A` por el escalar `a`.
- `A * B`: Devuelve la matriz producto de las matrices `A` y `B`. El número de columnas de `A` debe coincidir con el número de filas de `B`.
- `A'`: Devuelve la matriz traspuesta de la matriz `A`.
- `transpose(A)`: Devuelve la matriz traspuesta de la matriz `A`.

4.5.15 Ejemplo de álgebra lineal con matrices

```
julia> A = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> B = [1 1 1; 2 2 2]
2×3 Matrix{Int64}:
 1  1  1
 2  2  2

julia> A + B
2×3 Matrix{Int64}:
 2  3  4
 6  7  8

julia> C = A'
3×2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4
 2  5
 3  6

julia> A * C
2×2 Matrix{Int64}:
 14  32
 32  77

julia> C * A
3×3 Matrix{Int64}:
 17  22  27
 22  29  36
 27  36  45
```

4.5.16 Álgebra lineal con matrices

El paquete `LinearAlgebra` define las siguientes funciones:

- `Matrix(I, n, n)`. Devuelve la matriz identidad de dimensión `n`.
- `diag(A)`: Devuelve un vector con los elementos de la diagonal principal de la matriz `A`.
- `norm(A)`: Devuelve la norma de Frobenius de la matriz `A`.
- `tr(A)`: Devuelve la traza de la matriz cuadrada `A`.
- `det(A)`: Devuelve el determinante de la matriz cuadrada `A`.

- `inv(A)`: Devuelve la matriz inversa de la matriz cuadrada `A`.
- `A \ B`: Devuelve el vector `x` solución del sistema de ecuaciones $Ax = B$, donde `A` es una matriz cuadrada y `B` es un vector del mismo tamaño que el número de filas o columnas de `A`.

4.5.17 Ejemplo de álgebra lineal con matrices

```
using LinearAlgebra

julia> Matrix(I, 3, 3)
3×3 Matrix{Bool}:
 1  0  0
 0  1  0
 0  0  1

julia> A = [1 2 3; 0 1 0; 1 0 1]
3×3 Matrix{Int64}:
 1  2  3
 0  1  0
 1  0  1

julia> diag(A)
3-element Vector{Int64}:
 1
 1
 1

julia> norm(A)
4.123105625617661

julia> tr(A)
3

julia> det(A)
-2.0

julia> inv(A)
3×3 Matrix{Float64}:
 -0.5   1.0   1.5
  0.0   1.0   0.0
  0.5  -1.0  -0.5

julia> B = [10, 2, 4]
3-element Vector{Int64}:
```

```
 10
  2
  4

julia> A \ B  # Solución del sistema Ax = B
3-element Vector{Float64}:
 3.0
 2.0
 1.0
```

4.5.18 Álgebra lineal con matrices

Otras funciones más avanzadas del paquete `LinearAlgebra` son:

- `eigvals(A)`: Devuelve un vector con los autovalores de la matriz `A`.
- `eigvecs(A)`: Devuelve la matriz con los autovectores de la matriz `A`.
- `factorize(A)`: Devuelve las matrices resultantes de la factorización de la matriz `A`. La factorización dependerá de las propiedades de `A` (ver tipos de factorización)

4.5.19 Ejemplos de Álgebra lineal con matrices

```
using LinearAlgebra

julia> A = [1 2; 3 1]
2×2 Matrix{Int64}:
 1  2
 3  1

julia> eigvals(A)
2-element Vector{Float64}:
 -1.4494897427831779
  3.4494897427831783

julia> eigvecs(A)
2×2 Matrix{Float64}:
 -0.632456  0.632456
  0.774597  0.774597

julia> B = [1 2; 2 1]
2×2 Matrix{Int64}:
 1  2
 2  1
```

```
julia> factorize(B)
LU{Float64, Tridiagonal{Float64, Vector{Float64}}}
L factor:
2×2 Matrix{Float64}:
 1.0  0.0
 0.5  1.0
U factor:
2×2 Matrix{Float64}:
 2.0  1.0
 0.0  1.5
```

4.5.20 Copia de tipos de datos compuestas

En Julia cuando se asigna una variable de un tipo de datos compuesto a otra variable, no se hace una copia de la estructura de datos referenciada por la primera variable, sino que se la nueva variable apunta a la misma dirección de memoria de la estructura de datos (copia por referencia). El resultado son dos variables que apuntan a la misma estructura de datos y cualquier cambio en una de ellas se verá reflejado en la otra.

Para hacer copias por valor de un tipo de datos compuesto debe usarse explícitamente la siguiente función:

- `b = copy(a)`: Crea una copia de la estructura de datos referencia por `a` y asigna su referencia a `b`.

4.5.21 Ejemplo de copia de tipos de datos compuestos

```
julia> v = [1, 2, 3];

julia> u = copy(v);  # Copia por valor

julia> u[2] = 0;

julia> u
3-element Vector{Int64}:
 1
 0
 3

julia> v
3-element Vector{Int64}:
 1
 2
```

```
 3

julia> u = v;  # Copia por referencia

julia> u[2]=0;

julia> v
3-element Vector{Int64}:
 1
 0
 3
```

4.6 Tuplas

Una tupla es una colección ordenada de tamaño fijo que puede contener elementos de distintos tipos.

Generalmente se usan para pasar parámetros o devolver valores de funciones.

Se crean escribiendo sus elementos separados por comas entre paréntesis.

> **Advertencia**
>
> Las tuplas son inmutables, es decir, una vez creadas no pueden cambiarse sus elementos.

```
julia> ()  # Tupla vacía
()

julia> (1, "enero", 2020)
(1, "enero", 2020)

julia> t = (1, "enero", 2020)
(1, "enero", 2020)

julia> typeof(t)
Tuple{Int64, String, Int64}
```

4.6.1 Tuplas con nombres

Es posible asignar un nombre a cada uno de los elementos de la tupla. Para ello cada elemento de la tupla con nombre debe escribirse con la sintaxis `nombre = valor`.

```
julia> t = (día = 1, mes = "enero", año =  2020)
(día = 1, mes = "enero", año = 2020)

julia> typeof(t)
NamedTuple{(:día, :mes, :año), Tuple{Int64, String, Int64}}
```

> **Advertencia**
>
> No puede haber dos elementos con el mismo nombre en una tupla.

La ventaja de usar tuplas con nombres es que podemos acceder a sus elementos por nombre, además de por índice.

4.6.2 Acceso a los elementos de una tupla

Como las tuplas tienen orden, podemos acceder a sus elementos mediante índices, al igual que con los arrays de una dimensión.

- `t[i]`: Devuelve el elemento con íncide `i` de la tupla `t`.

Si la tupla tiene nombres también es posible acceder a sus elementos mediante los nombres.

- `t.x`: Devuelve el elemento con nombre `x` de la tupla `t`.

```
julia> t = (día = 1, mes = "enero", año =  2020)
(día = 1, mes = "enero", año = 2020)

julia> t[2]
"enero"

julia> t.año
2020

julia>
```

4.6.3 Asignación múltiple de tuplas

Es posible asignar los elementos de una tupla a distintas variables en una sola asignación.

`x, y, ... = t`: Asigna a las variables `x`, `y`, etc los elementos de la tupla `t` en orden. Si el número de variables es menor que el tamaño de la tupla, los últimos elementos quedan sin asignar.

`x, y... = t`: Asigna el primer elemento de la tupla `t` a la variable `x` y la tupla con los elementos restantes a la variable `y`.

4.6.4 Ejemplo de asignación múltipe de tuplas

```
julia> t = (1, "enero", 2020)
(1, "enero", 2020)

julia> d, m, a = t
(1, "enero", 2020)

julia> d
1

julia> m
"enero"

julia> a
2020

julia> d, ma... = t
(1, "enero", 2020)

julia> d
1

julia> ma
("enero", 2020)
```

4.7 Diccionarios

Un diccionario es una colección asociativa sin orden cuyos elementos son pares formados por una *clave* y un *valor* asociado a la clave.

Se parecen a las tuplas con nombre, pero, a diferencia de estas, son mutables, es decir, su contenido se puede alterar.

Se construyen con la siguiente constructor:

- `Dict(k1 => v1, ...)`: Crea un diccionario con los pares indicados en formato `clave => valor`.

> **Advertencia**
>
> En un diccionario no pueden existir dos pares con la misma clave, de modo que si se repite una clave se sobrescribe el par anterior.

4.7.1 Ejemplo de diccionarios

```
julia> Dict()  # Diccionario vacío
Dict{Any, Any}()

julia> d = Dict("ES" => "Euro", "US" => "Dollar", "CN" => "Yuan")
Dict{String, String} with 3 entries:
  "CN" => "Yuan"
  "ES" => "Euro"
  "US" => "Dollar"

julia> typeof(d)
Dict{String, String}
```

4.7.2 Comprensión de diccionarios

Al igual que para arrays se puede usar la técnica de compresión para generar diccionarios a partir de uno o varios iteradores.

- `Dict(kexp => vexp for i = ite)`: Devuelve el diccionario cuyos pares están formados por la claves y valores resultan de evaluar las expresiones `kexp` y `vexp` respectivamente, para cada valor `i` del iterador `ite`.
- `Dict(kexp => vexp for i = ite if cond)`: Devuelve el diccionario cuyos pares están formados por la claves y valores resultan de evaluar las expresiones `kexp` y `vexp` respectivamente, para cada valor `i` del iterador `ite` que cumpla condición `cond`.

Se pueden utilizar más de un iterador después de la palabra reservada `for`.

4.7.3 Ejemplo de comprensión de diccionarios

```
julia> Dict(i => i^2 for i = 1:4)
Dict{Int64, Int64} with 4 entries:
  4 => 16
  2 => 4
  3 => 9
  1 => 1

julia> Dict(i => i^2 for i = 1:4 if i % 2 == 0)
Dict{Int64, Int64} with 2 entries:
  4 => 16
  2 => 4
```

```
julia> Dict((i, j) => i + j for i = 1:2, j = 3:4)
Dict{Tuple{Int64, Int64}, Int64} with 4 entries:
  (2, 4) => 6
  (1, 3) => 4
  (1, 4) => 5
  (2, 3) => 5
```

4.7.4 Acceso a los elementos de un diccionario

Para acceder a los valores de un diccionario se utilizan sus claves asociadas entre corchetes.

- `d[k]`: Devuelve el valor asociado a la clave `k` en el diccionario `d`.

> **Advertencia**
>
> Si la clave no existe en el diccionario se produce un error.

Para evitar errores es conveniente usar alguna de las siguientes funciones:

- `haskey(d, k)`: Devuelve `true` la clave `k` está en diccionario `d` y `false` en caso contrario.
- `get(d, k, v)`: Devuelve el valor asociado a la clave `k` en el diccionario `d` o el valor `v` si la clave `k` no existe.
- `get!(d, k, v)`: Devuelve el valor asociado a la clave `k` en el diccionario `d`. Si la clave `k` no existe en el diccionario `d` añade el par con la clave `k` y el valor `v` y devuelve el valor `v`.

4.7.5 Ejemplo de acceso a los elementos de un diccionario

```
julia> d = Dict("ES" => "Euro", "US" => "Dollar", "CN" => "Yuan")
Dict{String, String} with 3 entries:
  "CN" => "Yuan"
  "ES" => "Euro"
  "US" => "Dollar"

julia> d["ES"]
"Euro"

julia> d["JP"]
ERROR: KeyError: key "JP" not found
Stacktrace:
 [1] getindex(h::Dict{String, String}, key::String)
   @ Base ./dict.jl:481
```

```
 [2] top-level scope
   @ REPL[22]:1

julia> get(d, "JP", "Dollar")
"Dollar"

julia> get!(d, "JP", "Yen")
"Yen"

julia> d
Dict{String, String} with 4 entries:
  "CN" => "Yuan"
  "ES" => "Euro"
  "JP" => "Yen"
  "US" => "Dollar"
```

4.7.6 Recorrido de las claves y valores de un diccionario

Las siguientes funciones permiten obtener todas las claves, valores y pares de un diccionario.

- `keys(d)`: Devuelve un iterador con las claves del diccionario `d`.
- `values(d)`: Devuelve un iterador con los valores del diccionario `d`.

Estos iteradores permiten recorrer fácilmente los pares de un diccionario.

4.7.7 Ejemplo de recorrido de las claves y valores de un diccionario

```
julia> d = Dict("ES" => "Euro", "US" => "Dollar", "CN" => "Yuan")
Dict{String, String} with 3 entries:
  "CN" => "Yuan"
  "ES" => "Euro"
  "US" => "Dollar"

julia> keys(d)
KeySet for a Dict{String, String} with 3 entries. Keys:
  "CN"
  "ES"
  "US"

julia> values(d)
ValueIterator for a Dict{String, String} with 3 entries. Values:
  "Yuan"
```

```
  "Euro"
  "Dollar"

julia> for k = keys(d)
         println("$k = $(d[k])")
       end
CN = Yuan
ES = Euro
US = Dollar

julia> for (k, v) = d
       println("$k = $v")
       end
CN = Yuan
ES = Euro
US = Dollar
```

4.7.8 Añadir elementos a un diccionario

Se pueden añadir pares nuevos a un diccionario de la siguiente manera:

- `d[k] = v`: Añade el par con clave `k` y valor `v` al diccionario `d`. Si la clave `k` ya existía en el diccionario `d`, cambia su valor asociado por `v`.
- `push!(d, k => v)`: Añade el par con clave `k` y valor asociado `v` al diccionario `d`.

```
julia> d = Dict("ES" => "Euro", "US" => "Dollar")
Dict{String, String} with 2 entries:
  "ES" => "Euro"
  "US" => "Dollar"

julia> d["CN"] = "Yuan"
"Yuan"

julia> push!(d, "JP" => "Yen")
Dict{String, String} with 4 entries:
  "CN" => "Yuan"
  "ES" => "Euro"
  "JP" => "Yen"
  "US" => "Dollar"
```

4.7.9 Eliminar elementos de un diccionario

Para eliminar un par de un diccionario se utiliza la siguiente función:

- `delete!(d, k)`: Elimina el par cuya clave es `k` del diccionario `d`.

```
julia> d = Dict("ES" => "Euro", "US" => "Dollar", "CN" => "Yuan")
Dict{String, String} with 3 entries:
  "CN" => "Yuan"
  "ES" => "Euro"
  "US" => "Dollar"

julia> delete!(d, "US")
Dict{String, String} with 2 entries:
  "CN" => "Yuan"
  "ES" => "Euro"
```

4.8 Conjuntos

Un conjunto es una colección de elementos del mismo tipo sin orden y sin repeticiones.

Se construyen con la siguiente constructor:

- `Set(a)`: Crea un conjunto con los elementos del array `a`.

Al igual que para arrays el tipo se infiere automáticamente a partir de los tipos de sus elementos. Si los elementos son de distintos tipos se convierten al tipo más específico de la jerarquía de tipos del que los tipos de los elementos son subtipos.

> ⚠ Advertencia
>
> Un conjunto no puede tener elementos repetidos, por lo que si el array contiene elementos repetidos solo se incluyen una vez.

4.8.1 Ejemplo de construcción de conjuntos

```
julia> Set()
Set{Any}()

julia> c = Set([1, "2", 3])
Set{Any} with 3 elements:
  "2"
  3
  1
```

```
julia> typeof(c)
Set{Any}
```

4.8.2 Añadir elementos a un conjunto

Para añadir elementos a un conjunto se utiliza la siguiente función:

- `push!(c, e)`: Añade el elemento `e` al conjunto `c`.

> **Advertencia**
>
> Si el elemento que se quiere añadir es de distinto tipo que los elemento del conjunto y no puede convertirse a este tipo, se produce un error.

4.8.3 Ejemplo de añadir elementos a un conjunto

```
julia> c = Set(1:3)
Set{Int64} with 3 elements:
  2
  3
  1

julia> push!(c, 4)
Set{Int64} with 4 elements:
  4
  2
  3
  1

julia> push!(c, "cinco")
ERROR: MethodError: Cannot `convert` an object of type String to an object of type Int64
Closest candidates are:
  convert(::Type{T}, ::T) where T<:Number at /usr/share/julia/base/number.jl:6
  convert(::Type{T}, ::Number) where T<:Number at /usr/share/julia/base/number.jl:7
  convert(::Type{T}, ::Base.TwicePrecision) where T<:Number at /usr/share/julia/base/twi
  ...
```

4.8.4 Eliminar elementos de un conjunto

Para eliminar elementos de un conjunto se utiliza la siguiente función:

- `delete!(c, e)`: Elimina el elemento `e` del conjunto `c`.

```
julia> c = Set(1:3)
Set{Int64} with 3 elements:
  2
  3
  1

julia> delete!(c, 2)
Set{Int64} with 2 elements:
  3
  1
```

4.8.5 Recorrido de los elementos de un conjunto

Un conjunto puede utilizarse también como un iterador para recorrer sus elementos.

```
julia> c = Set(1:3)
Set{Int64} with 3 elements:
  2
  3
  1

julia> for i = c
         println(i)
       end
2
3
1
```

4.8.6 Pertenencia e inclusión de conjuntos

- `in(e, c)`, `e  c` : Devuelve `true` si el elemento `e` pertenece al conjunto `c` y `false` en caso contrario.
- `e  c`: Devuelve `true` si el elemento `e` no pertenece al conjunto `c` y `false` en caso contrario.
- `issubset(a, b)`, `a  b`: Devuelve `true` si todos los elementos de `a` pertenecen a `b` y `false` en caso contrario.
- `a  b`: Devuelve `true` si hay algún elemento de `a` que no pertenece a `b`.
- `a  b`: Devuelve `true` si el conjunto `a` está contenido estrictamente en el conjunto `b`, es decir, todos los elementos de `a` pertenecen a `b` pero `a` y `b` son distintos.
- `isdisjoint(a, b)`: Devuelve `true` si los conjuntos `a` y `b` no tienen elementos en común y `false` en caso contrario.

4.8.7 Ejemplos de pertenencia e inclusión de conjuntos

```
julia> a = Set(1:3);

julia> in(2, a)
true

julia> 3  a
false

julia> b = Set([3, 2, 1]);

julia> a  b
false

julia> a  b
true

julia> isdisjoint(a, b)
false
```

4.8.8 Álgebra de conjuntos

- `union(a, b)`, `a  b`: Devuelve el conjunto unión de los conjuntos `a` y `b`.
- `intersect(a, b)`, `a  b`: Devuelve el conjunto intersección de los conjuntos `a` y `b`.
- `setdiff(a, b)`: Devuelve el conjunto diferencia del conjunto `a` y `b`.
- `symdiff(a, b)`: Devuelve el conjunto diferencia simétrica de los conjuntos `a` y `b`.

Existen versiones de estas funciones acabadas en `!` que sobreescriben el conjunto dado como primer argumento con el resultado de la operación.

4.8.9 Ejemplo de álgebra de conjuntos

```
julia> a = Set(1:3)
Set{Int64} with 3 elements:
  2
  3
  1

julia> b = Set(2:2:6)
Set{Int64} with 3 elements:
  4
  6
```

```
  2

julia> union(a, b)
Set{Int64} with 5 elements:
  4
  6
  2
  3
  1

julia> intersect(a, b)
Set{Int64} with 1 element:
  2

julia> setdiff(a, b)
Set{Int64} with 2 elements:
  3
  1

julia> symdiff(a, b) == setdiff(a   b, a   b)
true
```

5 Funciones

5.1 Creación de funciones

Una función asocia un nombre a un bloque de código de manera que cada vez que se invoca a la función se ejecuta el bloque de código asociado.

Para crear una función se utiliza la siguiente sintaxis

```
function nombre(parámetros)
  bloque de código
end
```

> **Nota**
>
> La indentación del bloque de código no es necesaria pero es una buena práctica.

Para invocar una función basta con escribir su nombre y pasarle entre paréntesis los valores de los parámetros (*argumentos*) separados por comas.

5.1.1 Ejemplo de creación de funciones

```
julia> function saludo()  # Función sin parámetros
         println("¡Bienvenido!")
       end
saludo (generic function with 1 method)

julia> saludo()
¡Bienvenido!

julia> typeof(saludo)
typeof(saludo) (singleton type of function saludo, subtype of Function)
```

5.2 Parámetros y argumentos de una función

Una función puede recibir valores cuando se invoca a través de unas variables conocidas como *parámetros* que se definen entre paréntesis y separados por comas en la declaración

de la función. En el cuerpo de la función se pueden usar estos parámetros como si fuesen variables.

Los valores que se pasan a la función en una llamada o invocación concreta de ella se conocen como *argumentos* y se asocian a los parámetros de la declaración de la función.

```
julia> function calificacion(nota)  # Función con un parámetro
         if nota < 5
           println("Suspenso")
         else
           println("Aprobado")
         end
       end
calificacion (generic function with 1 method)

julia> calificacion(7)
Aprobado
```

5.2.1 Paso de argumentos a una función

Los argumentos se pueden pasar de dos formas:

- **Argumentos posicionales**: Se asocian a los parámetros de la función en el mismo orden que aparecen en la definición de la función.
- **Argumentos nominales**: Se indica explícitamente el nombre del parámetro al que se asocia un argumento de la forma `parametro = argumento`.

Cuando una función tiente parámetros posicionales y como nominales, los posicionales deben indicarse primero y los nominales después, separando ambos tipos de parámetros por punto y coma `;`.

5.2.2 Ejemplo de paso de argumentos a una función

```
julia> function saludo(nombre, apellidos; ciudad)
       println("¡Hola $nombre $apellidos, bienvenido a $(ciudad)!")
       end
saludo (generic function with 1 method)

julia> saludo("Alfredo", "Sánchez", ciudad = "Madrid")
¡Hola Alfredo Sánchez, bienvenido a Madrid!

julia> saludo("Alfredo", ciudad = "Madrid",  "Sánchez")
¡Hola Alfredo Sánchez, bienvenido a Madrid!
```

5.2.3 Argumentos por defecto

En la definición de una función se puede asignar a cada parámetro un argumento por defecto, de manera que si se invoca la función sin proporcionar ningún argumento para ese parámetro, se utiliza el argumento por defecto.

El valor por defecto de un parámetro se indica con la siguiente sintaxis `parámetro = valor`.

```
julia> function saludo(nombre, apellidos, ciudad = "Madrid")
         println("¡Hola $nombre $apellidos, bienvenido a $(ciudad)!")
       end
saludo (generic function with 2 methods)

julia> saludo("Alfredo", "Sánchez")
¡Hola Alfredo Sánchez, bienvenido a Madrid!

julia> saludo("Pepito", "Grillo", "Barcelona")
¡Hola Pepito Grillo, bienvenido a Barcelona!
```

5.2.4 Funciones con un número variable de argumentos

Julia permite definir funciones que pueden llamarse con un número variable de argumentos. Para que una función pueda recibir un número variable de argumentos hay que poner tres puntos suspensivos `...` al final de último parámetro posicional.

Cuando se llame a la función los argumentos se irán asociando a los parámetros posicionales en orden y el último parámetro se asociará a una tupla con el resto de argumentos en la llamada.

```
julia> function media(x...)
       println("Media de ", x)
       sum(x) / length(x)
       end
media (generic function with 1 method)

julia> media(1, 2, 3, 4)
Media de (1, 2, 3, 4)
2.5
```

5.2.5 Parámetros con tipo

Aunque Julia es un lenguaje de tipado dinámico también permite fijar el tipo de los parámetros de una una función. Esto permite definir diferentes variantes (*métodos*) de una misma

función dependiendo del tipo de los argumentos, así como detectar errores cuando se llama a la función con argumentos de distinto tipo.

Para indicar el tipo de los parámetros de una función se utiliza la sintaxis `parametro::tipo`.

> **Advertencia**
>
> Conviene no restringir demasiado el tipo de los parámetros de una función. Se debe elegir el tipo más general en la jerarquía de tipos para el que tiene sentido la función.

5.2.6 Ejemplo de parámetros con tipo

```
julia> function sumar(x::Number, y::Number)
       x + y
       end
sumar (generic function with 1 method)

julia> function sumar(x::String, y::String)
       x * y
       end
sumar (generic function with 2 methods)

julia> sumar(1, 2)
3

julia> sumar(1.5, 2.5)
4.0

julia> sumar("Hola", "Julia")
"HolaJulia"
```

5.2.7 Paso de argumentos por asignación

En Julia los argumentos se pasan a una función por asignación, es decir, se asignan a los parámetros de la función como si fuesen variables locales. De este modo, cuando los argumentos son objetos mutables (arrays, diccionarios, etc.) se pasa al parámetro una referencia al objeto, de manera que cualquier cambio que se haga en la función mediante el parámetro asociado afectará al objeto original y serán visibles fuera de ella.

```
julia> function matricular(curso, asignatura)
       push!(curso, asignatura)
       end
matricular (generic function with 1 method)
```

```
julia> primer_curso = [];

julia> matricular(primer_curso, "Álgebra Lineal");

julia> matricular(primer_curso, "Programación");

julia> primer_curso
2-element Vector{Any}:
 "Álgebra Lineal"
 "Programación"
```

5.2.8 Ámbito de los parámetros de una función

Los parámetros y las variables declaradas dentro de una función son de *ámbito local*, mientras que las variable definidas fuera de funciones son de ámbito *ámbito global*.

Tanto los parámetros como las variables del ámbito local de una función sólo están accesibles durante la ejecución de la función. Es decir, cuando termina la ejecución de la función estas variables desaparecen y no son accesibles desde fuera de la función.

Si en el ámbito local de una función existe una variable que también existe en el ámbito global, durante la ejecución de la función la variable global queda eclipsada por la variable local y no es accesible hasta que finaliza la ejecución de la función.

5.2.9 Ejemplo del ámbito de los parámetros de una función

```
julia> lenguaje = "Python";

julia> function saludo(nombre)
       lenguaje = "Julia"
       println("¡Hola $(nombre), bienvenido a $(lenguaje)!")
       end
saludo (generic function with 3 methods)

julia> saludo("Alf")
¡Hola Alf, bienvenido a Julia!

julia> lenguaje
"Python"

julia> nombre
ERROR: UndefVarError: nombre not defined
```

5.3 Retorno de una función

Una función devuelve siempre el valor de la última expresión evaluada en su cuerpo. Sin embargo, puede devolverse cualquier otro valor indicándolo detrás de la palabra reservada `return`. Cuando el flujo de ejecución de la función alcanza esta palabra, la ejecución de la función termina y se devuelve el valor que la acompaña.

Si una función no devuelve ningún valor se puede escribir la palabra `return` sin nada más.

Cuando se desea devolver más de un valor se puede pueden indicar separados por comas y la función devolverá la tupla formada por esos valores.

5.3.1 Ejemplo de retorno de una función

```
julia> function area_triangulo(base, altura)
         return base * altura / 2  # Devuelve un valor
       end
area_triangulo (generic function with 1 method)

julia> area_triangulo(3, 4)
6.0

julia> function area_perimetro_circulo(r)
         return   * r ^ 2, 2  * r  # Devuelve dos valores
       end
area_perimetro_circulo (generic function with 1 method)

julia> area_perimetro_circulo(1)
(3.141592653589793, 6.283185307179586)
```

5.4 Funciones compactas

Cuando el cuerpo de una función es una única expresión se puede definir la función de forma mucho más compacta de la siguiente manera:

`nombre(`*parametros*`) =` *expresión*

El valor que devuelve la función es el resultado de evaluar la expresión.

Esta forma de definir funciones es muy habitual para funciones matemáticas.

```
julia> area_triangulo(b, a) = b * a / 2
area_triangulo (generic function with 1 method)
```

```
julia> area_triangulo(3, 4)
6.0

julia> valor_absoluto(x) = x < 0 ? -x : x
valor_absoluto (generic function with 1 method)

julia> valor_absoluto(-1)
1
```

5.5 Funciones como objetos

En Julia las funciones son objetos como el resto de tipos de datos, de manera que es posible asignar una función a una variable y luego utilizar la variable para hacer la llamada a la función, pasar una función como argumento de otra función, o que una función devuelva otra función.

```
julia> suma(x, y) = x + y
suma (generic function with 1 method)

julia> adicion = suma
suma (generic function with 1 method)

julia> adicion(1, 2)
3

julia> calculadora(operador, x, y) = operador(x, y)
calculadora (generic function with 1 method)

julia> calculadora(suma, 1, 2)
3
```

5.6 Funciones anónimas

Julia permite también definir funciones sin nombre. Para ello se utiliza la siguiente sintaxis.

(*parametros*) -> *expresión*

El principal uso de las funciones anónimas es para pasarlas como argumentos de otras funciones.

```
ulia> calculadora(operador, x, y) = operador(x, y)
calculadora (generic function with 1 method)

julia> calculadora((x, y) -> x + y, 1, 2)
3

julia> calculadora((x, y) -> x - y, 1, 2)
-1
```

5.7 Funciones asociadas a operadores

En Julia los operadores tienen asociadas funciones que son llamadas por el intérprete cuando se evalúa una expresión con operadores.

```
julia> +(1, 2, 3)  # Equivalente a 1 + 2 + 3
6

julia>   = +
+ (generic function with 208 methods)

julia>  (1, 2, 3)
6
```

5.8 Funciones recursivas

Una función recursiva es una función que en su cuerpo contiene alguna llama a si misma.

La recursión es una práctica común en la mayoría de los lenguajes de programación ya que permite resolver las tareas recursivas de manera más natural.

Para garantizar el final de una función recursiva, las sucesivas llamadas tienen que reducir el grado de complejidad del problema, hasta que este pueda resolverse directamente sin necesidad de volver a llamar a la función.

> **Precaución**
>
> La recursión es una técnica que suele ser poco eficiente computacionalmente y conviene evitarla siempre que sea posible.

5.8.1 Ejemplo de funciones recursivas

```
julia> function factorial(n::Integer)
         if n <= 1
           return 1
         else
           return n * factorial(n-1)
         end
       end
factorial (generic function with 1 method)

julia> factorial(4)
24

julia> fib(n::Integer) = n  2 ? 1 : fib(n - 1) + fib(n - 2)
fib (generic function with 1 method)

julia> fib(10)
55
```

6 Gráficos

6.1 Paquetes gráficos

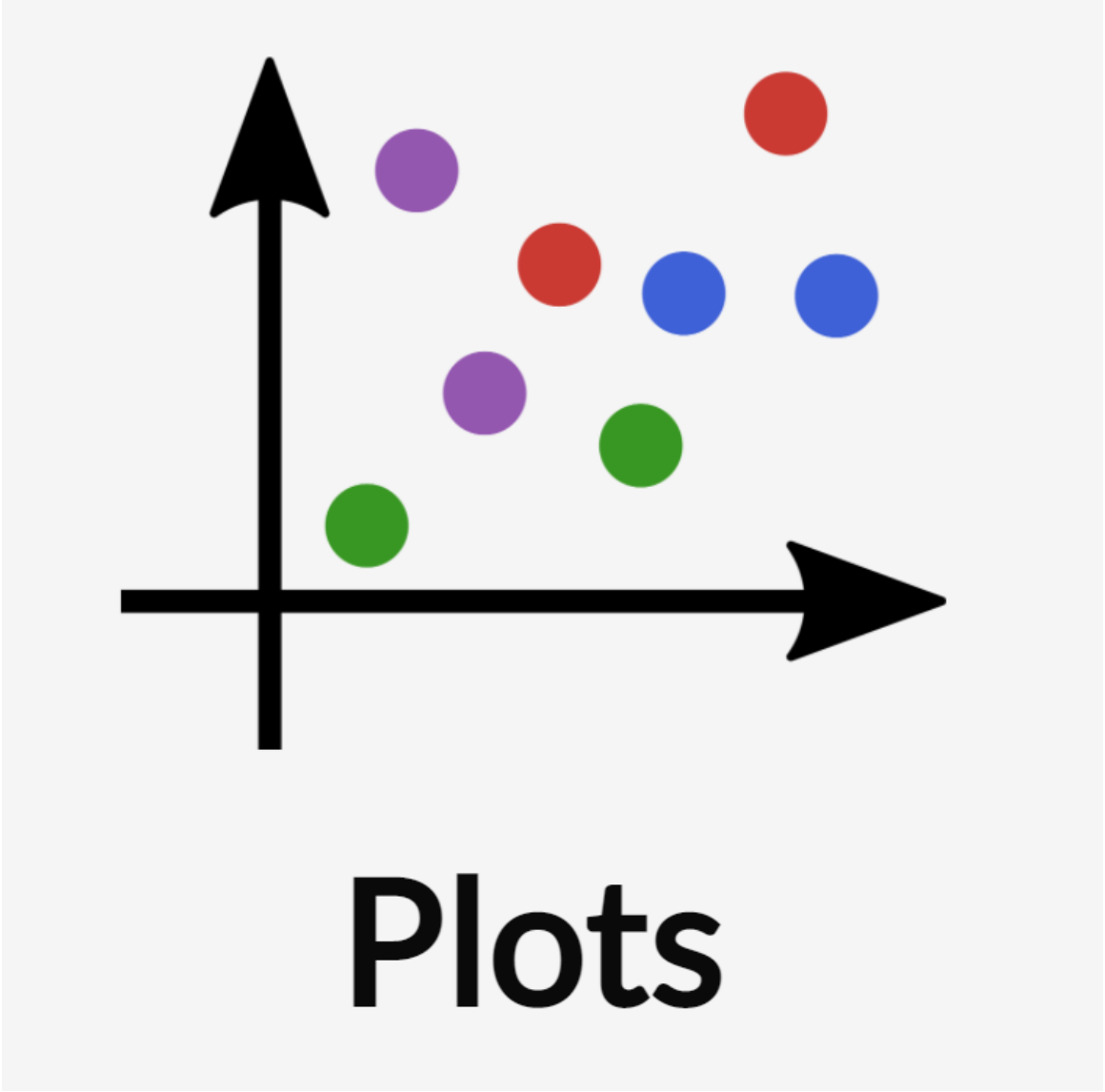
Existen muchos paquetes para la representación gráfica en Julia. Los más usados son:

- Plots.jl
- Makie.jl
- GadFly.jl.
- VegaLite.jl

6.2 Gráficos con el paquete Plots.jl

Plots.js es el paquete más usado por disponer de más posibilidades gráficas y ser bastante sencillo de usar.

Implementa una interfaz para otras librerías gráficas (backends), por lo que en algunas ocasiones puede ser bastante lento al tener que llamar a otras librerías.



6.2.1 Backends de Plot.jl

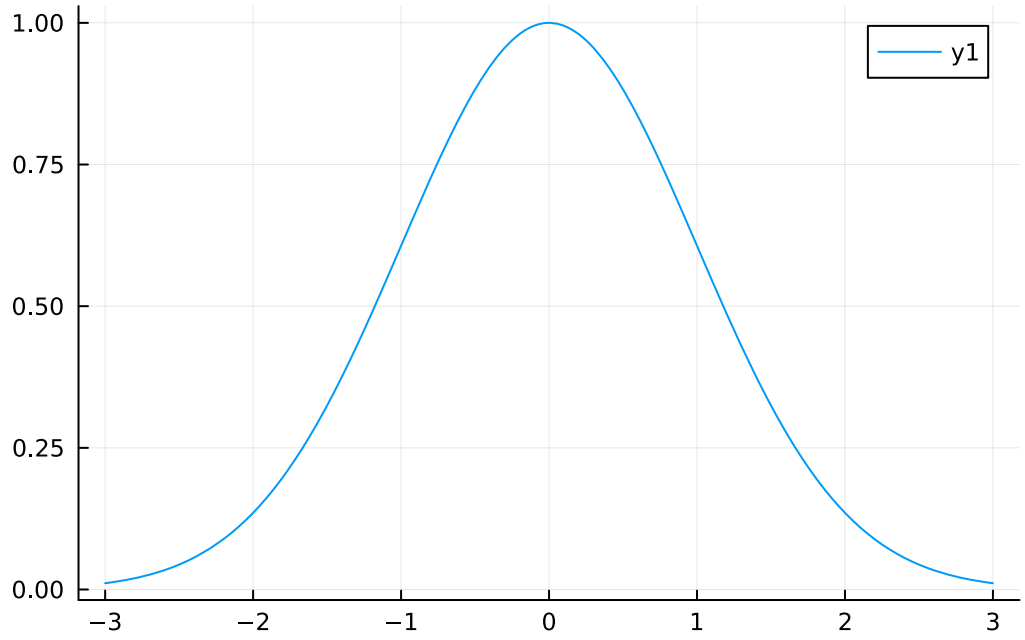
- GR. Es el backend pro defecto. Es bastante rápida y permite tanto gráficos 2D como 3D no interactivos. Se inicializa con la función `gr()`. (Ver ejemplos)
- PlotlyJS. Es más lenta pero permite gráficos 2D y 3D interactivos con un montón de funcionalidades. Se inicializa con la función `plotlyjs()`. (Ver ejemplos)
- PyPlot. Utiliza la librería gráfica Matplotlib de Python por lo que es bastante lenta. Sin embargo, tiene ofrece todas las posibilidades de Matplotlib que es bastante madura. Se inicializa con la función `pyplot()`. (Ver ejemplos)
- PGFPlotsX. Utiliza la librería PGF/TikZ de LaTeX por lo que genera gráficos de muy alta calidad tanto en 2D como 3D, especialmente para publicaciones.Se inicializa con la función `pgfplotsx()`. (Ver ejemplos)

- UnicodePlots. Permite dibujar gráficos en la terminal. Los gráficos son de poca calidad pero funciona con gran rapidez. Se inicializa con la función `unicodeplots()`. (Ver ejemplos)

6.2.2 Gráfica de una función de una variable

- `plot(f, min, max)`: Dibuja la gráfica de la función de una variable `f` para argumentos desde `xmin` a `xmax`.

```
using Plots

f(x) = exp(-x^2 / 2)
plot(f, -3, 3)
```

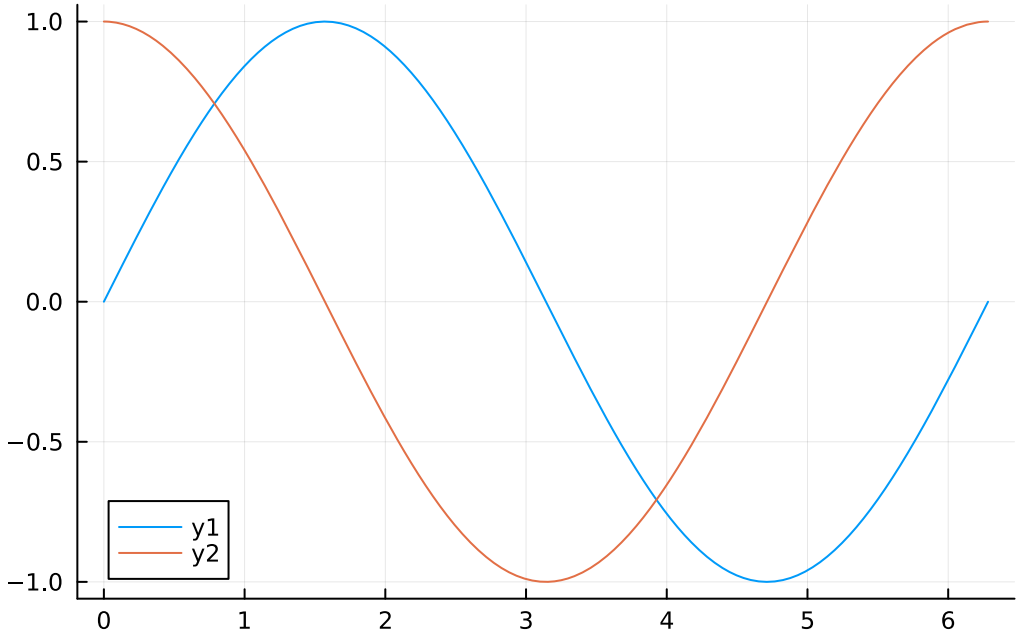


6.2.3 Gráficas de varias funciones

- `plot!(f, xmin, xmax)`: Añade la gráfica de la función de una variable `f` para argumentos desde `xmin` a `xmax` al último gráfico realizado.

```
using Plots

f(x) = sin(x)
g(x) = cos(x)
```

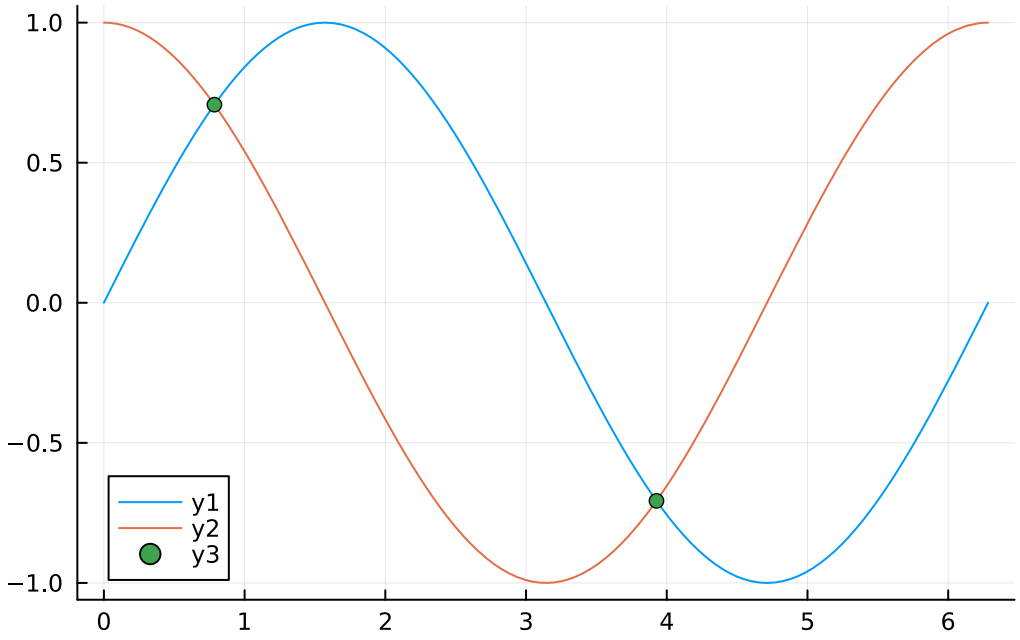
```
plot(f, -0, 2 )
plot!(g)
```



6.2.4 Añadir puntos a una gráfica

- `scatter(x, y)`: Dibuja los puntos con coordenadas x en el vector `x` y coordenadas y en el vector `y`.

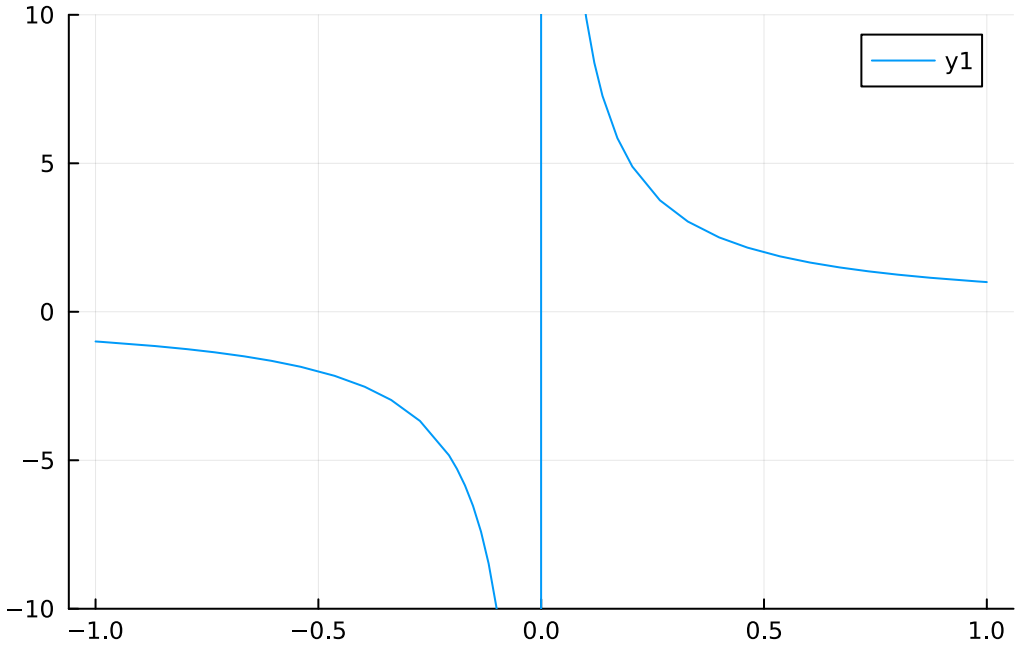
```
using Plots

f(x) = sin(x)
g(x) = cos(x)
plot(f, -0, 2 )
plot!(g)
x = [ /4, 5 /4]
y = sin.(x)
scatter!(x, y)
```



6.2.5 Ventana de graficación

Es posible restringir el área de graficación (rango de valores de los ejes) de una función añadiendo los parámetros `xlims =(xmin, xmax)` para establecer el rango del eje x o `ylims = (ymin, ymax)` para establecer el rango del eje y.

```
using Plots

f(x) = 1 / x
plot(f, -1, 1, ylims = (-10, 10))
```



6.2.6 Restringir la gráfica al dominio

Cuando una función no está definida para algún valor del rango de valores del eje x dado, la gráfica muestra una línea recta desde el punto de la gráfica anterior hasta el punto siguiente al punto donde la función no existe.

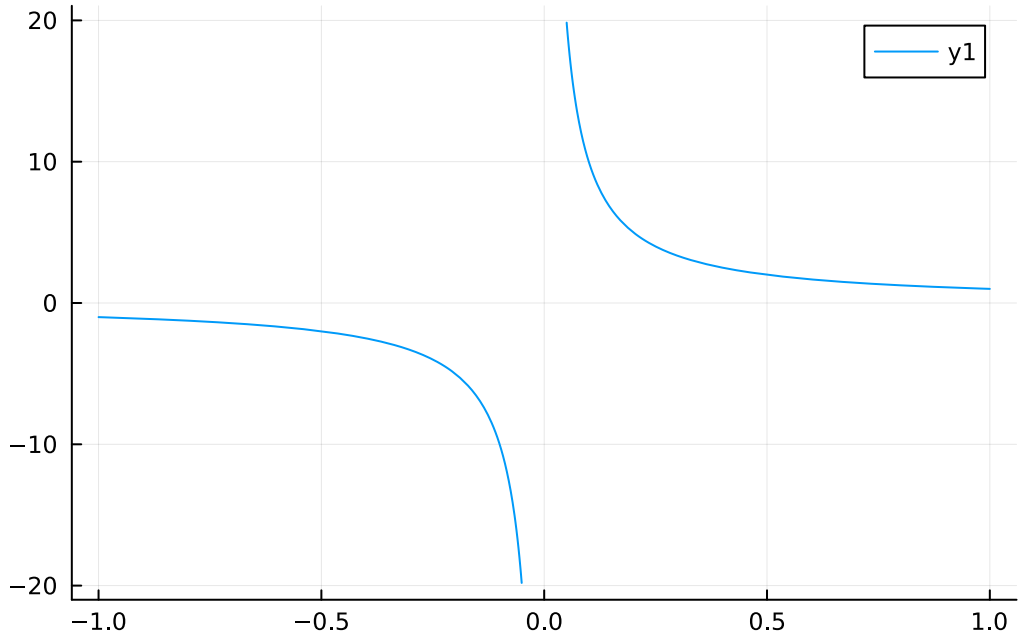
Este comportamiento no es deseable puesto que si la función no existe en un punto no debería existir gráfica para ese punto.

La siguiente función del paquete `MTH229` se encarga de evitar esto.

- `rangeclamp(f)`: Devuelve una función idéntica a la función `f` excepto para los puntos donde la función no existe o es infinito que devuelve `NaN`.

6.2.7 Ejemplo de restringir la gráfica al dominio

```
using Plots
using MTH229

f(x) = 1 / x
plot(rangeclamp(f), -1, 1)
```

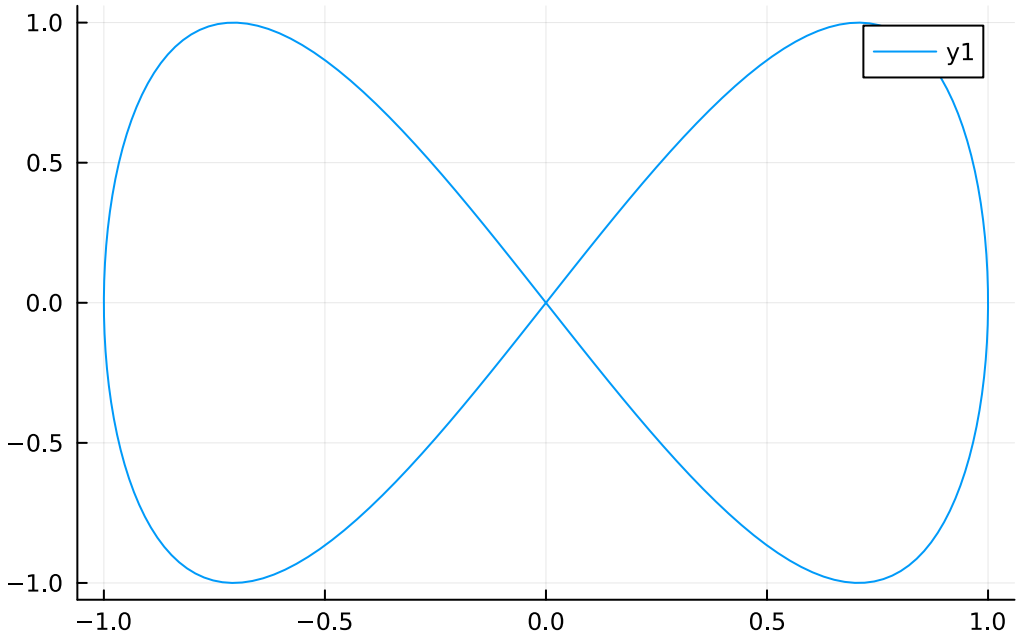


6.2.8 Gráficas paramétricas

La función `plot` también permite dibujar gráficas de funciones paramétricas pasándole las funciones de las coordenadas x e y.

- `plot(f, g, min, max)`: Dibuja la gráfica de la función paramétrica $(f(t), g(t))$ para valores del parámetro `t` entre `min` y `max`.

```
using Plots
f(x) = sin(x)
g(x) = sin(2x)
plot(f, g, 0, 2 )
```



6.2.9 Personalización de gráficos

Los siguientes parámetros pueden añadirse a la función `plot` para modificar el aspecto de los gráficos.

- `title`: Añade un título principal al gráfico.
- `xlab`: Añade un título al eje x.
- `ylab`: Añade un título al eje y.
- `color`: Establece el color de la gráfica.
- `linewidth`: Establece el grosor de la línea de la gráfica.
- `linestyle`: Establece el estilo de la línea de la gráfica.
- `aspect_ratio`: Establece la relación de aspecto entre la escala de los ejes.
- `legend`: Activa o desactiva la leyenda del gráfico.

6.2.10 Ejemplo de personalización de gráficos

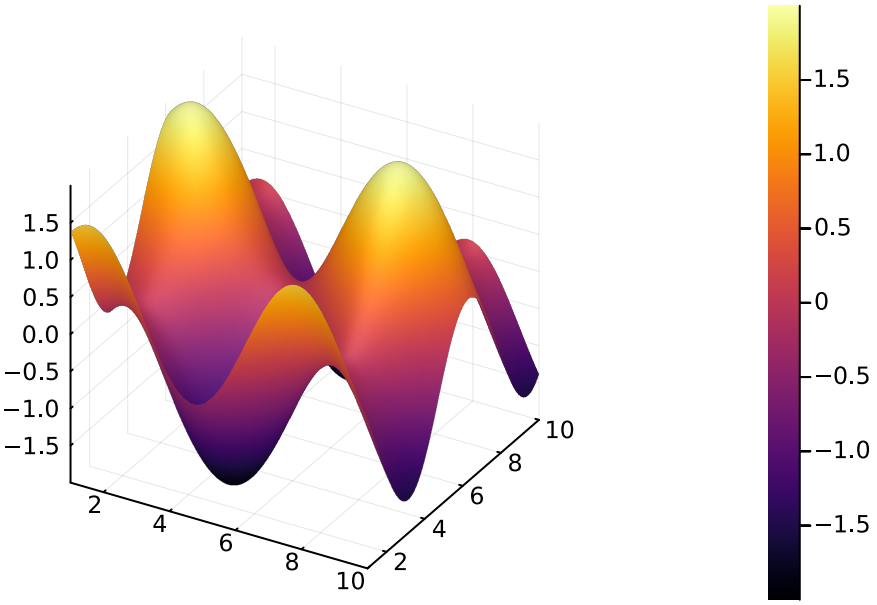
```
using Plots

f(x) = sin(x)
plot(f, - ,  , title = "Gráfica del seno",  xlab = "x", ylab = "f(x) = sen(x)",
  color = "green", linewidth = 3, linestyle = :dash, legend = false)
```

6.2.11 Gráficos en el espacio real

Para dibujar superficies en el espacio real se utiliza la función

`surface(x, y, f)`: Dibuja la superficie de la función $f(x, y)$ en el rango de valores `x` del eje x e `y` del eje y.

```
using Plots
xs = ys = range(1, stop=10, length=100)
f(x, y) = sin(x) + cos(y)
surface(xs, ys, f)
```



6.3 Gráficos con Makie

Makie es una colección de paquetes de visualización de datos eficiente y extensible que incorpora una gran variedad de tipos de gráficos tanto en 2D como en 3D.

6.3.1 Backends de `Makie`

Al igual que el paquete `Plots`, `Makie` utiliza distintos backends para construir el gráfico según la salida que se quiera.

- `CairoMakie.jl` se utiliza para crear gráficos no interactivos de alta calidad, especialmente para publicaciones impresas.
- `GLMakie.jl` se utiliza para crear gráficos interactivos tanto en 2D como en 3D en una ventana gráfica.
- `WGLMakie.jl` es similar a `GLMakie.jl` pero muestra el gráfico en un navegador web.

6.3.2 Figuras, ejes y objetos gráficos

Un gráfico con `Makie` es básicamente una figura (`Figure`) que contiene uno o varios ejes (`Axis`) que, a su vez, contienen objetos gráficos como puntos o lineas.

Para crear una figura se utiliza la función

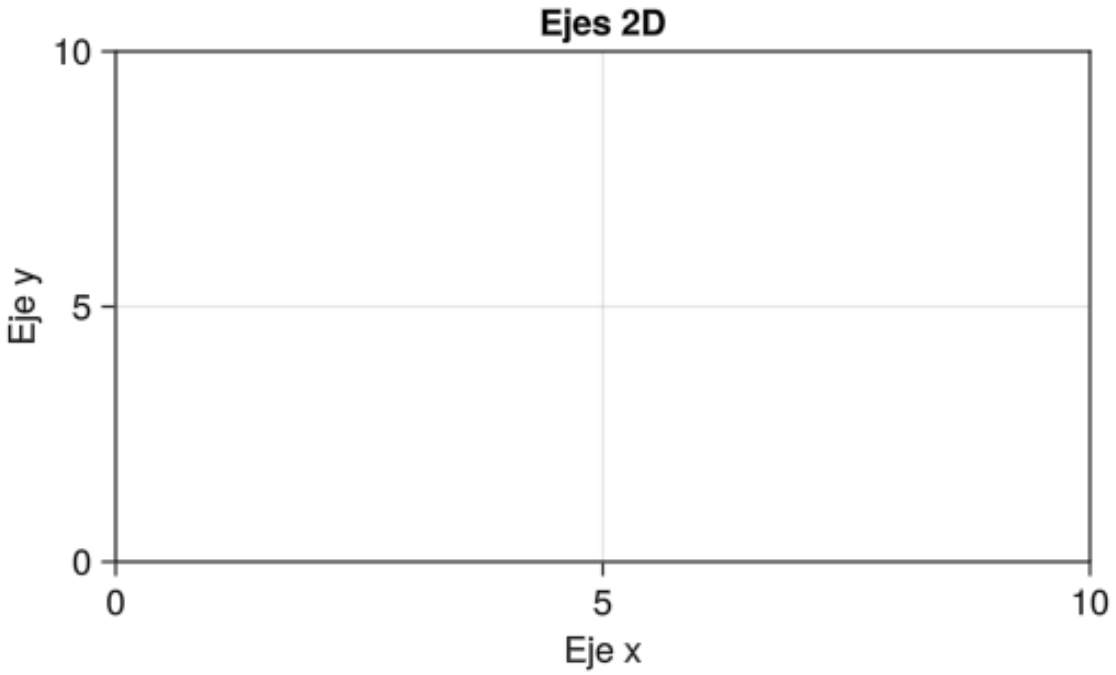
- `Figure(parámetros)`: Crea un área gráfica con la configuración indicada por los `parámetros`. Entre los parámetros se puede indicar la resolución (`resolution = (ancho, largo)`) o el color de fondo (`backgroundcolor = color`).

```
using GLMakie
fig = Figure(backgroundcolor = :gray, resolution = (400, 300))
```

Para añadir unos ejes en 2D a una figura se utiliza la función

- `Axis(fig[i,j], title = titulo, xlabel = etiqueta-x, ylabel = etiqueta-y)`: Crea unos ejes en la figura `fig` con el título principal dado en `titulo`, la etiqueta del eje x dada en `etiqueta-x` y la etiqueta del eje y dada en `etiqueta-y`.
  Cuando se quiere incluir varios ejes en una misma figura, el vector `[i,j]`, indica la posición de los ejes, donde la primera componente indica la fila y la segunda la columna en una disposición de los ejes en forma de tabla. Si la figura solo contiene unos ejes, se utiliza el vector `[1,1]`.

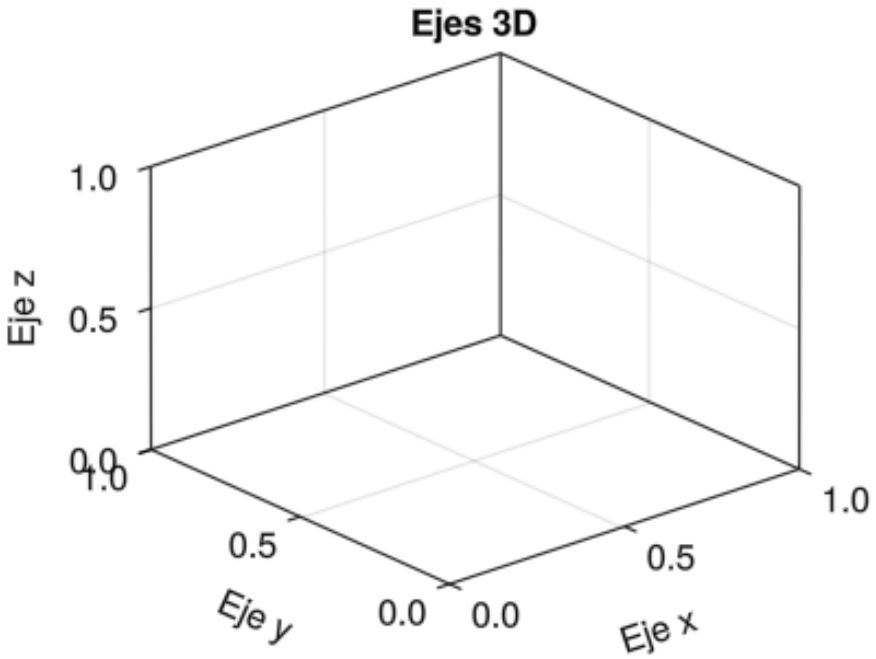
```
fig = Figure()
ax = Axis(fig[1,1], title = "Ejes 2D", xlabel = "Eje x", ylabel = "Eje y")
fig
```



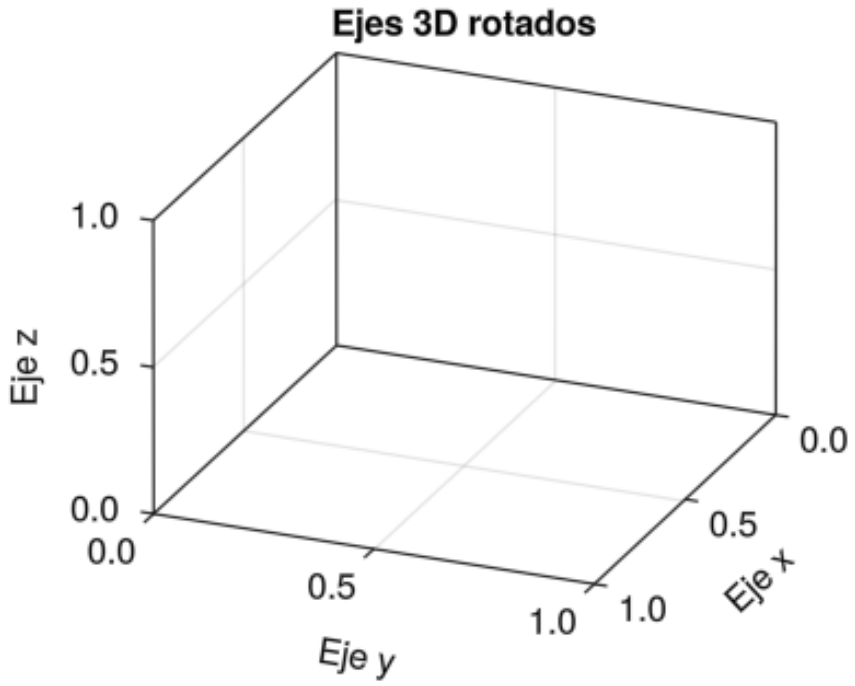
Y para añadir unos ejes en 3D a una figura se utiliza la función

- `Axis3(fig[i,j], title = titulo, xlabel = etiqueta-x, ylabel = etiqueta-y, zlabel = etiqueta-z, azimuth = ángulo-azimuth, elevation = ángulo-elevación)`: Crea unos ejes 3D en la figura `fig` con el título principal dado en `titulo`, la etiqueta del eje x dada en `etiqueta-x`, la etiqueta del eje y dada en `etiqueta-y`, la etiqueta del eje z dada en `etiqueta-z`, y un punto de visión dado por el ángulo de azimuth (izquierda-derecha) `angulo-azimuth` radianes (1.275π por defecto) y el ángulo de elevación (arriba-abajo) `ángulo-elevación` radianes ($\pi/8$ por defecto). Al igual que antes, cuando se quiere incluir varios ejes en una misma figura, el vector `[i,j]`, indica la posición de los ejes, donde la primera componente indica la fila y la segunda la columna en una disposición de los ejes en forma de tabla. Si la figura solo contiene unos ejes, se utiliza el vector `[1,1]`.

```
fig = Figure()
ax = Axis3(fig[1,1], title = "Ejes 3D", xlabel = "Eje x", ylabel = "Eje y", zlabel = "E
fig
```




```
fig = Figure()
ax = Axis3(fig[1,1], title = "Ejes 3D rotados", xlabel = "Eje x", ylabel = "Eje y", zla
fig
```

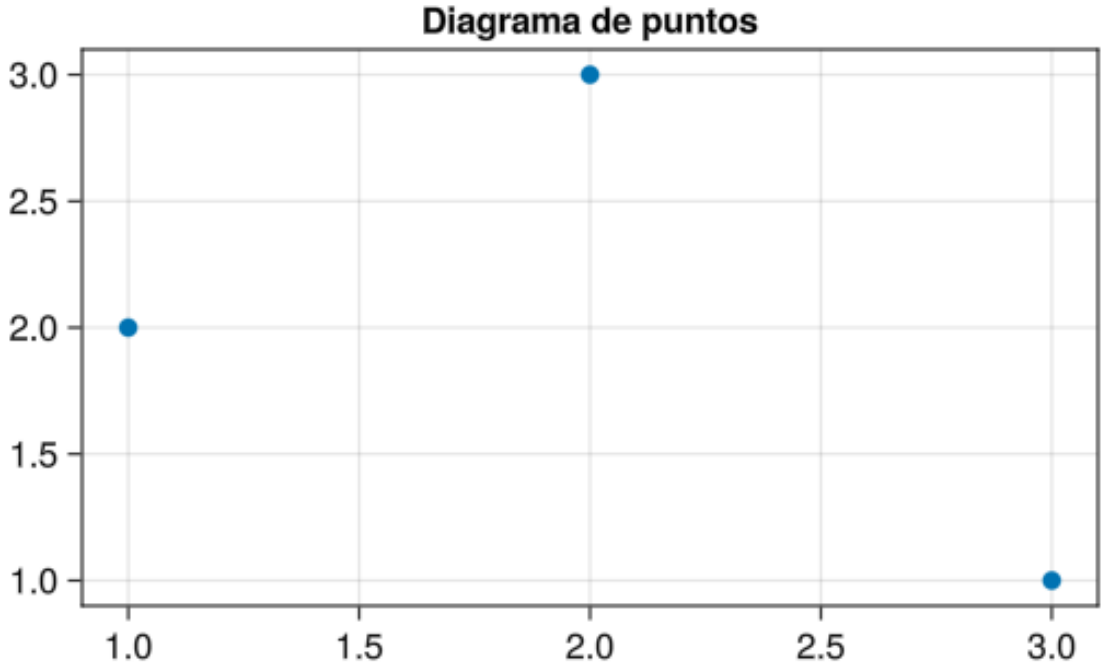


6.3.3 Diagrama de puntos

Para dibujar un diagrama de puntos se utiliza la función

- `scatter!(ax, xs, ys)`: Dibuja los ejes dados en `ax` los puntos con coordenadas dadas por los vectores `xs` e `ys`.

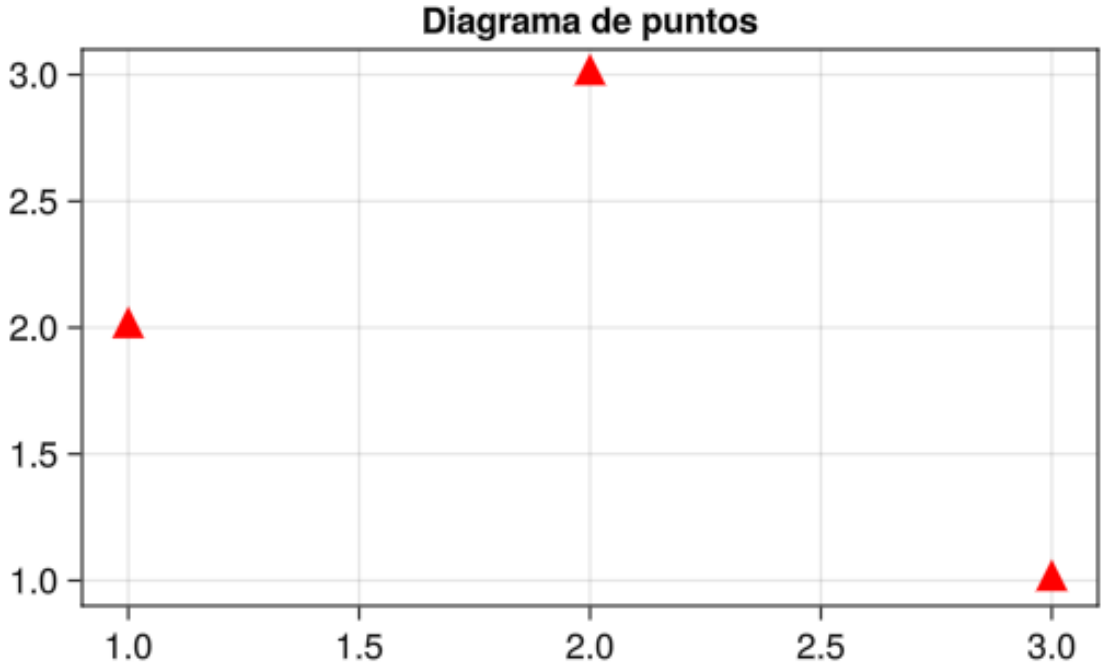
```
fig = Figure()
ax = Axis(fig[1, 1], title = "Diagrama de puntos")
xs = [1, 2, 3]
ys = [2, 3, 1]
Makie.scatter!(ax, xs, ys)
fig
```



Los siguientes parámetros pueden añadirse a la función anterior para modificar el aspecto del diagrama de puntos.

- `marker`: Establece el tipo de punto.
- `color`: Establece el color de los puntos.
- `markersize`: Establece el tamaño de los puntos.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Diagrama de puntos")
xs = [1, 2, 3]
ys = [2, 3, 1]
Makie.scatter!(ax, xs, ys, marker = :utriangle, color= :red, markersize = 20)
fig
```



6.3.4 Diagrama de líneas

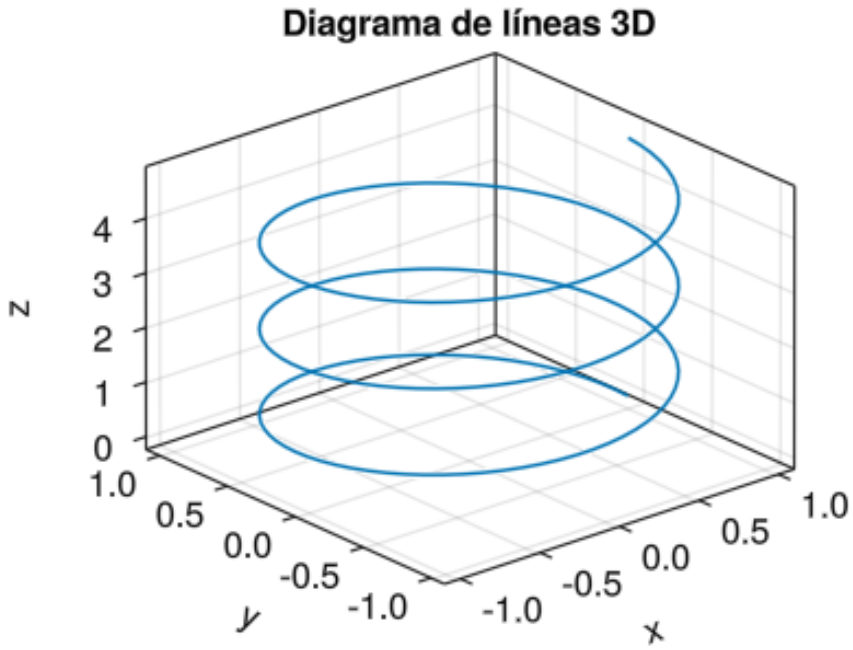
Para dibujar series de líneas se utiliza la función

- `lines!(ax, xs, ys)`: Dibuja en los ejes dados por `ax` una línea en el plano que une los puntos con coordenadas dadas por los vectores `xs` e `ys`.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Diagrama de líneas")
xs = range(0, 2pi, length = 100)
ys = sin.(xs)
lines!(ax, xs, ys)
fig
```

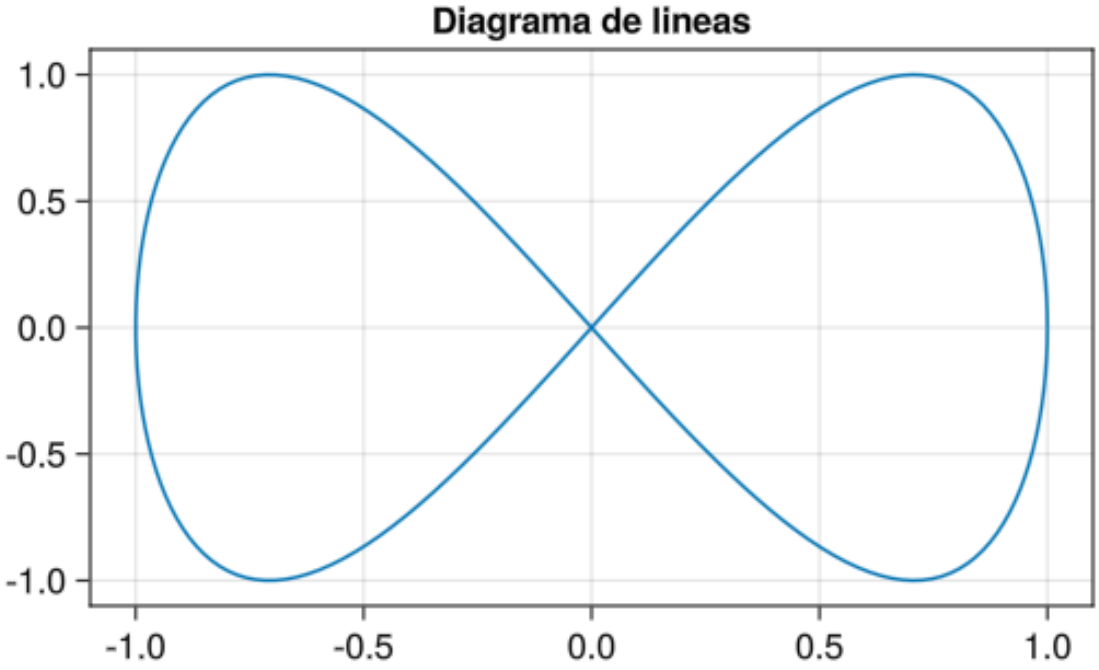
Si se añade un tercer vector de coordenadas `zs` sobre unos ejes 3D, se obtiene un diagrama de líneas en 3D. De esta forma se pueden representar, por ejemplo, la trayectorias de funciones vectoriales.

```
fig = Figure()
ax = Axis3(fig[1, 1], title = "Diagrama de líneas 3D")
ts = range(0, 6pi, length = 200)
xs = cos.(ts)
ys = sin.(ts)
zs = ts/4
lines!(ax, xs, ys, zs)
fig
```

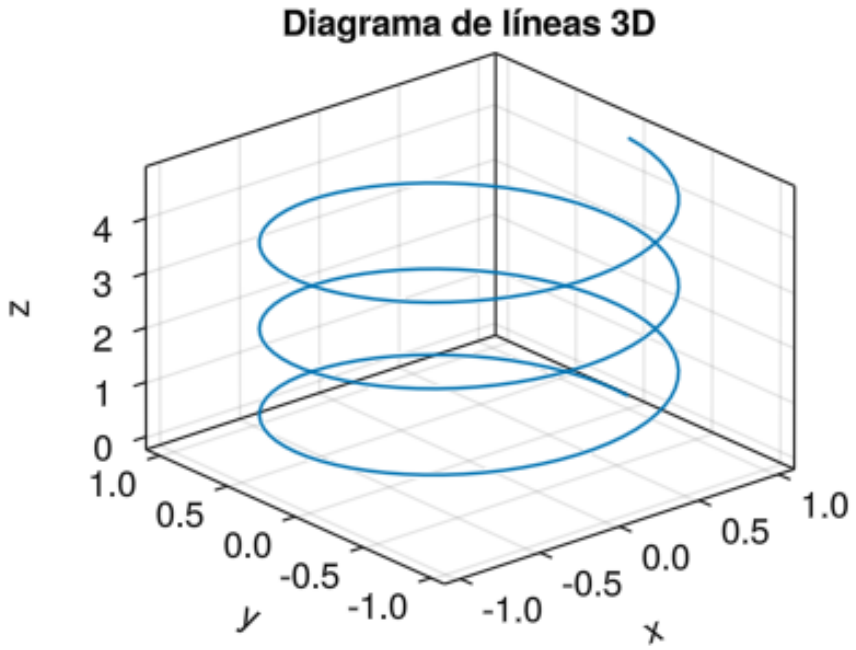


En lugar de pasar las coordenadas de los puntos en vectores separados, se pueden pasar empaquetadas en tuplas mediante las estructuras `Point(x,y)` para el plano real o `Point3(x,y,z)` para el espacio real, del paquete `GeometryBasics.jl`.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Diagrama de lineas")
ts = range(0, 2pi, length = 200)
points = Point.(cos.(ts), sin.(2ts))
lines!(ax, points)
fig
```

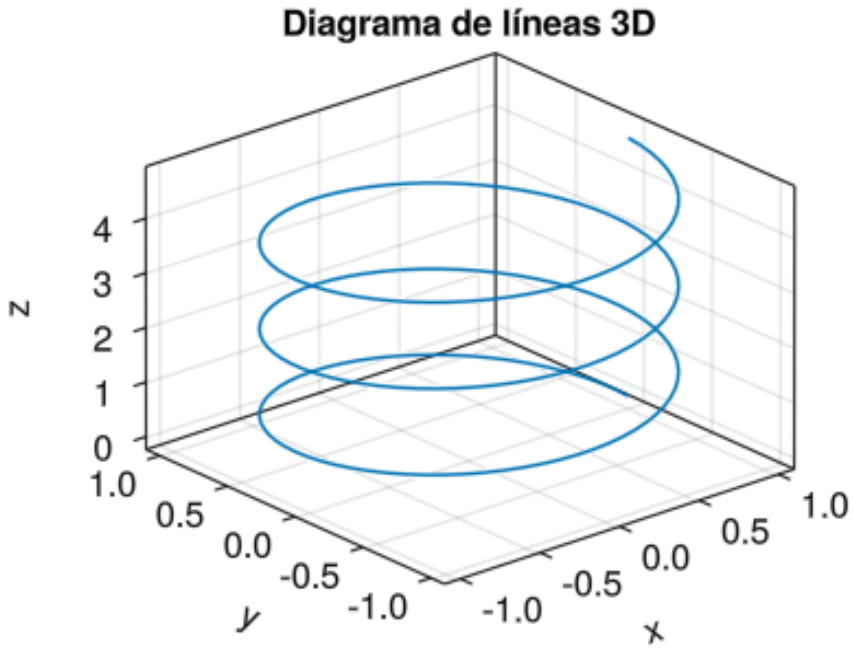



```
fig = Figure()
ax = Axis3(fig[1, 1], title = "Diagrama de líneas 3D")
ts = range(0, 6pi, length = 200)
f(t) = [cos(t), sin(t), t/4]
points = Point3.(f.(ts))
lines!(ax, points)
fig
```



En el caso de figuras en 3D, es preferible utilizar los backend `GLMakie.jl` o `WGLMakie.jl` ya que permiten interaccionar con el el gráfico para visualizarlo desde distintas perspectivas.

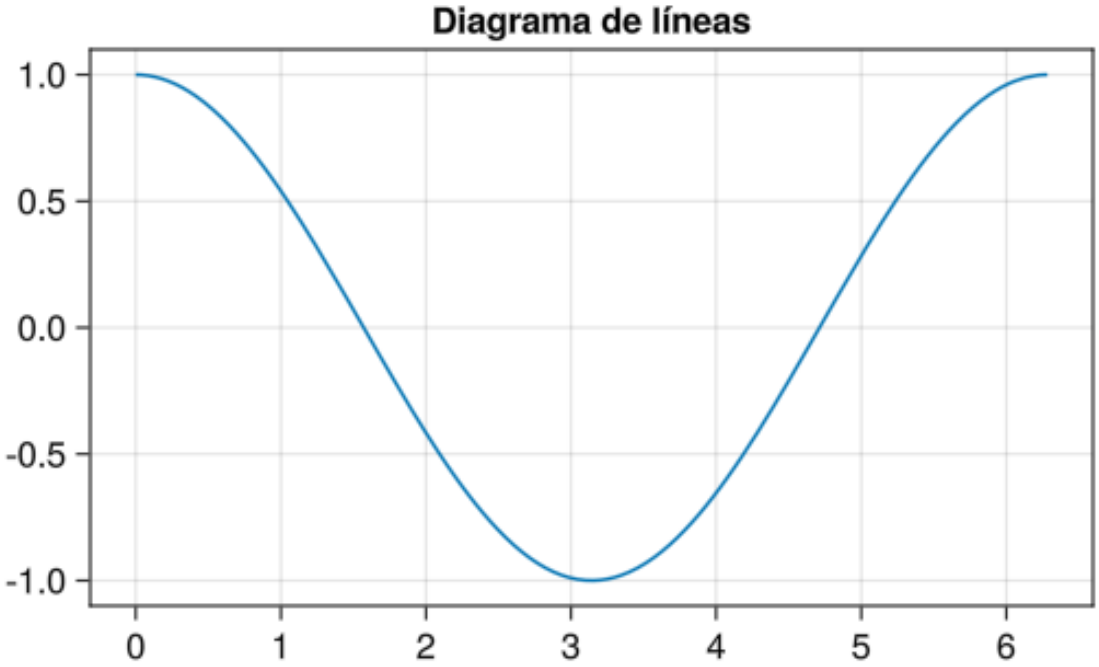
```
fig = Figure()
ax = Axis3(fig[1, 1], title = "Diagrama de líneas 3D")
ts = range(0, 6pi, length = 200)
f(t) = [cos(t), sin(t), t/4]
points = Point3.(f.(ts))
lines!(ax, points)
fig
```



También es posible pasar a la función `lines!` un rango de valores y una función a dibujar. Esto simplifica la creación de gráficas de funciones.

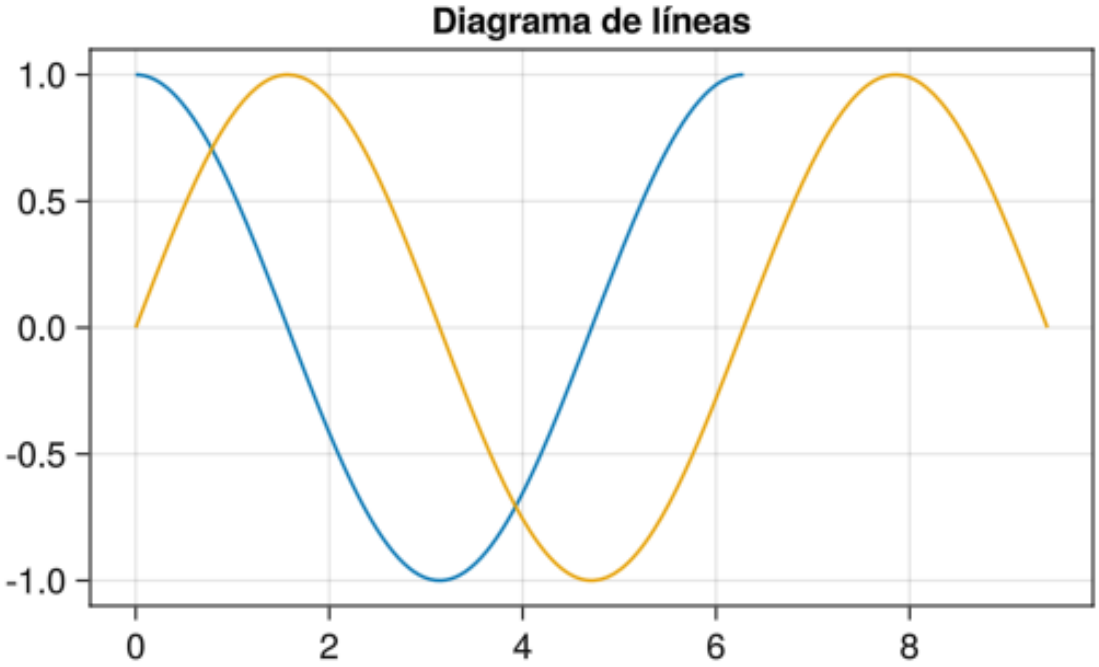
- `lines(ax, rango, función)`: Dibuja en los ejes dados por `ax` la gráfica de la función dada en `función` en el intervalo dado por `rango`. El intervalo del rango se especifica con la sintaxis `li..ls`, donde `li` es el límite inferior y `ls` el límite superior.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Diagrama de líneas")
lines!(ax, 0..2pi, cos)
fig
```



Para añadir nuevas líneas al los ejes, basta con volver a usar la función `lines!`.

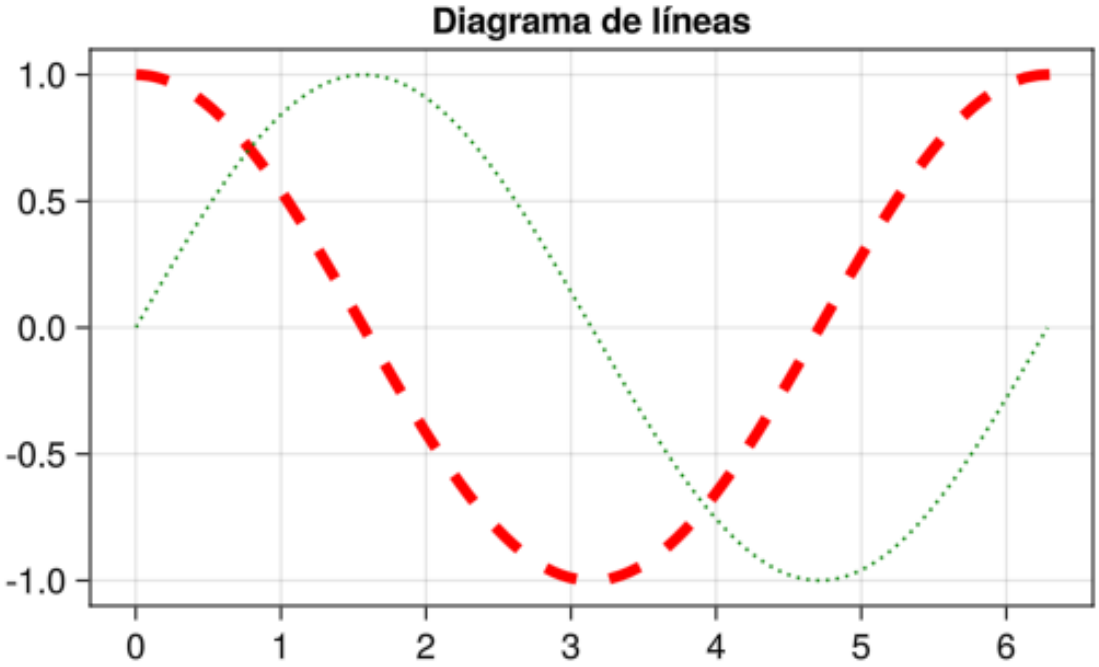
```
fig = Figure()
ax = Axis(fig[1, 1], title = "Diagrama de líneas")
lines!(ax, 0..2pi, cos)
lines!(ax, 0..3pi, sin)
fig
```



Los siguientes parámetros pueden añadirse a la función anterior para modificar el aspecto del diagrama de puntos.

- `linewidth`: Establece el grosor de la línea de la gráfica.
- `linestyle`: Establece el estilo de la línea de la gráfica.
- `color`: Establece el color de la linea.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Diagrama de líneas")
lines!(ax, 0..2pi, cos, linewidth = 5, linestyle = :dash, color = :red)
lines!(ax, 0..2pi, sin, linestyle = :dot, color = :green)
fig
```



6.3.5 Superficies

Para dibujar superficies en 3D se utiliza la función

- `surface!(ax, xs, ys, zs)`: Dibuja en los ejes 3D dados por `ax` la superficie que se obtiene al unir los puntos con coordenadas x dadas por el vector `xs`, coordenadas `y` dadas por el vector `ys` y coordenadas z dadas por el vector `zs`.

```
using WGLMakie
WGLMakie.activate!()
fig = Figure()
ax = Axis3(fig[1, 1], title = "Superficie")
xs = ys = LinRange(0, 10, 100)
zs = [cos(x) * sin(y) for x in xs, y in ys]
WGLMakie.surface!(ax, xs, ys, zs)
fig
```

```
Unable to display output for mime type(s): text/html
```

6.3.6 Leyenda

Si tenemos más de una serie de datos representada en un diagrama, conviene añadir una leyenda para identificar cada serie. Podemos añadir una leyenda con la función

- `axislegend(ax)`: Añade una leyenda a los ejes dados en `ax`, con las etiquetas de cada serie de datos incluida en los ejes. Para ello es necesario etiquetar cada serie de datos pasándole el parámetro `label = etiqueta` a la función que crea el objeto gráfico que representa la serie.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Diagrama de líneas con leyenda")
lines!(ax, 0..2pi, cos, label = "Coseno")
lines!(ax, 0..2pi, sin, label = "Seno")
axislegend()
current_figure()
```

```
Unable to display output for mime type(s): text/html
```

6.4 Gráficos con GadFly.jl

GadFly.js es un paquete nativo que genera gráficos interactivos 2D y 3D por medio de librerías de Javascript basadas en la gramática de gráficos (usada también por el paquete ggplot2 de R).

Al estar implementado en Julia es mucho más rápido que Plots.js pero ofrece menos posibilidades.

6.5 Gráficos con VegaLite.jl

VegaLite.jl es un paquete que genera gráficos estáticos por medio de las librerías de Javascript de la gramática de gráficos Vega.

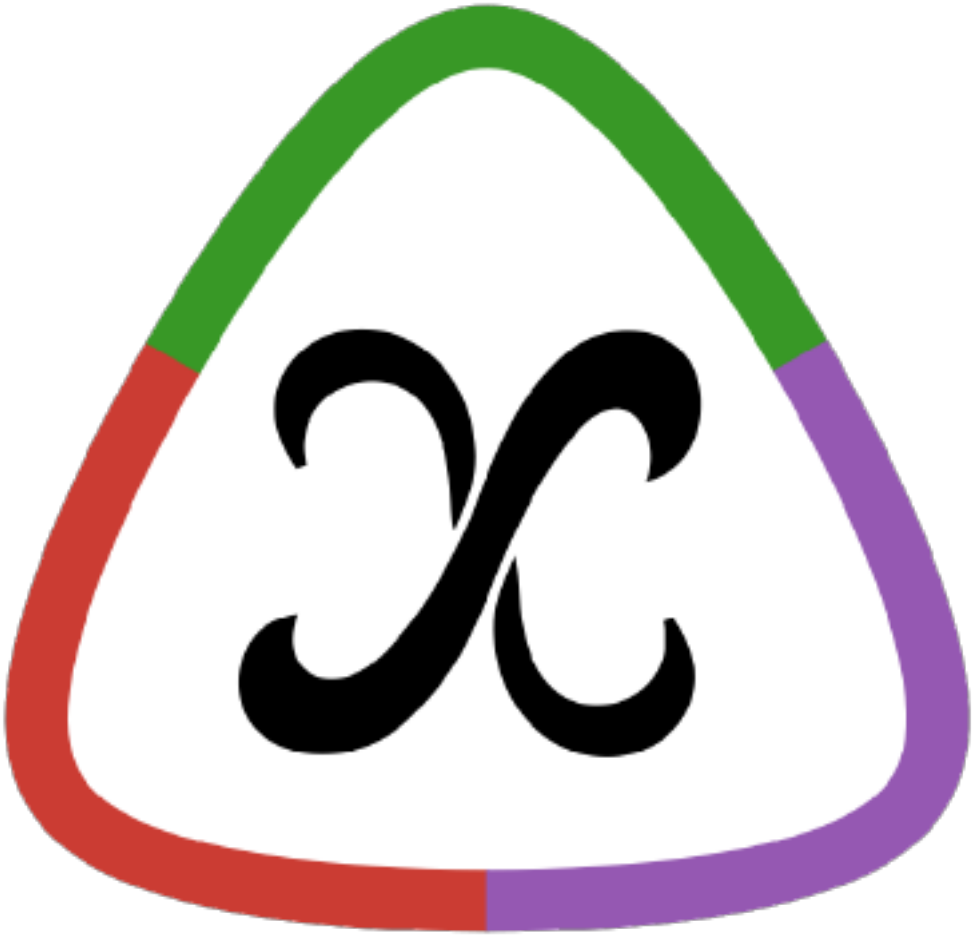
Dispone de muchas más opciones de personalización de gráficos que GadFly.jl.

7 Cálculo simbólico

7.1 Symbolics.jl

`Symbolics.jl` es un paquete que implementa un avanzado Sistema de Álgebra Computacional (CAS) basado en un lenguaje de modelado simbólico.

Las variables y las expresiones simbólicas pueden utilizarse con la mayoría de las funciones de Julia para cálculo numérico, por lo que se integran a la perfección en el ecosistema de Julia.



7.1.1 Variables y expresiones simbólicas

Para declarar variables simbólicas se utiliza la siguiente macro:

- `@variables x y ...`: Declara las variables `x`, `y`, etc. como variables simbólicas.

El tipo de las variables simbólicas es `Num`.

Cualquier expresión en la que interviene una variable simbólica se convierte automáticamente en una expresión simbólica.

7.1.2 Ejemplo de variables y expresiones simbólicas

```
using Symbolics

julia> @variables x y
2-element Vector{Num}:
 x
 y

julia> z = x^2 - y
x^2 - y

julia> typeof(z)
Num

julia> A = [x + y 2x; -y y - x]  # Matriz simbólica
2×2 Matrix{Num}:
 x + y     2x
    -y  y - x
```

7.1.3 Álgebra simbólica

Se pueden realizar operaciones algebraicas con expresiones simbólicas utilizando los mismos operadores del Álgebra numérica.

```
using Symbolics

julia> @variables x y;

julia> (x + 1) + (x + 2)
3 + 2x

julia> A = [x + y 2x; -y y - x]  # Matriz simbólica
```

```
2×2 Matrix{Num}:
 x + y     2x
    -y  y - x

julia> B = [x, y]  # Vector simbólico
2-element Vector{Num}:
 x
 y

julia> A * B  # Producto matricial
2-element Vector{Num}:
 x*(x + y) + 2x*y
  y*(y - x) - x*y
```

7.1.4 Simplificación de expresiones

Para simplificar expresiones simbólicas se utiliza la siguiente función:

- `simplify(e)`: Devuelve la expresión simbólica que resulta de simplificar la expresión simbólica `e`.

La simplificación utiliza el paquete `SymbolicUtils.jl` que implementa un potente sistema de reescritura de términos.

```
using Symbolics

@variables x y;

julia> simplify(2(x+y))
2x + 2y

julia> simplify(2(x+y))
2x + 2y

julia> simplify(sin(x)^2 + cos(x)^2)
1
```

7.1.5 Sustitución de variables en expresiones

Para sustituir una variable simbólica en una expresión se utiliza la siguiente función:

- `substitute(e, d)`: Realiza la sustitución de las claves por los valores del diccionario `d` en la expresión simbólica `e`.

```
using Symbolics

@variables x y;

julia> substitute(cos(2x), Dict([x => ]))
1.0

julia> substitute(x * y + 2x -y + 2, Dict([x => 1, y => 2]))
4
```

7.1.6 Resolución de ecuaciones

Para definir una ecuación se utiliza el símbolo `~` en lugar de la igualdad.

Para resolver una ecuación se utiliza la siguiente función:

- `Symbolics.solve_for(eq, var)`: Devuelve un vector con los valores de las variables del vector `var` que cumplen la ecuación o sistema de ecuaciones `eq`, siempre que la ecuación tenga solución.

> ⚠ Advertencia
>
> Actualmente solo funciona para ecuaciones lineales.

```
using Symbolics

@variables x y;

julia> Symbolics.solve_for(x + y ~ 0, x)
-y

julia> Symbolics.solve_for([x + y ~ 4, x - y ~ 2], [x, y])
2-element Vector{Float64}:
 3.0
 1.0
```

7.1.7 Cálculo de derivadas

Para calcular la derivada de una función se utiliza la siguiente función:

- `Symbolics.derivative(f, x)`: Devuelve la expresión simbólica de la derivada de la función `f` con respecto a la variable simbólica `x`.

```
using Symbolics

julia> @variables x y;

julia> Symbolics.derivative(exp(x*y), x)
y*exp(x*y)

julia> Symbolics.derivative(Symbolics.derivative(exp(x*y), x), y)
x*y*exp(x*y) + exp(x*y)
```

7.1.8 Cálculo de derivadas con operadores diferenciales

Para construir un operador diferencial ($\frac{d}{dx}$) se utiliza la siguiente función:

- `Differential(x)`: Crea el operador diferencial con respecto a la variable simbólica `x`.

Para obtener la función derivada, una vez aplicado el operador diferencial a una función, es necesario aplicar la siguiente función:

- `expand_derivatives(D(f))`: Devuelve la expresión simbólica que corresponde a la derivada de la función `f` con respecto a la variable del operador diferencial `D`.

7.1.9 Ejemplo de cálculo de derivadas con operadores diferenciales

```
using Symbolics

@variables x y;

julia> Dx = Differential(x)
(::Differential) (generic function with 2 methods)

julia> f(x) = sin(x^2)
f (generic function with 1 method)

julia> f1(x) = Dx(f(x))
f1 (generic function with 1 method)

julia> expand_derivatives(f1(x))
2x*cos(x^2)

julia> Dy = Differential(y)
(::Differential) (generic function with 2 methods)
```

```
julia> expand_derivatives(Dx(Dy(cos(x*y))))
-sin(x*y) - x*y*cos(x*y)
```

7.1.10 Gradiente y matriz Hessiana de una función de varias variables

Para calcular el vector gradiente de una función de varias variables se utiliza la siguiente función:

- `Symbolics.gradient(f, vars)`: Devuelve el vector gradiente de la función `f` con respecto a las variables del vector `vars`.

Y para calcular la matriz Hessiana se utiliza la siguiente función:

- `Symbolics.hessian(f, vars)`: Devuelve la matriz Hessiana de la función `f` con respecto a las variables del vector `vars`.

7.1.11 Ejemplo de gradiente y matriz Hessiana de una función de varias variables

```
using Symbolics

@variables x y;

julia> Symbolics.gradient(exp(x*y), [x, y])
2-element Vector{Num}:
 y*exp(x*y)
 x*exp(x*y)

julia> Symbolics.hessian(exp(x*y), [x, y])
2×2 Matrix{Num}:
 (y^2)*exp(x*y)               x*y*exp(x*y) + exp(x*y)
   x*y*exp(x*y) + exp(x*y)  (x^2)*exp(x*y)
```

8 Análisis de datos

8.1 El paquete DataFrames.jl

El principal paquete para análisis de datos es `DataFrames.jl` que proporciona herramientas para trabajar con conjuntos de datos en formato de tabla de forma similar a pandas en Python o data.frames y dplyr en R.

En conjunción con este paquete es frecuente utilizar también alguno de los siguientes paquetes:

- FreqTables.jl. Funciones para la construcción de tablas de frecuencias.
- Statistics.jl. Funciones para los principales estadísticos descriptivos.
- HypothesisTests.jl Funciones para los contrastes de hipótesis paramétricos y no paramétricos más comunes.
- GLM.jl. Funciones para modelos lineales generales.
- MultivariateStats.jl. Funciones para análisis multivariante.
- MLJ.jl. Funciones para los principales algoritmos de aprendizaje automático.

8.1.1 Creación de DataFrames

Para crear un DataFrame su utiliza la siguiente función:

- `DataFrame(x1=v1, x2=v2, ...)`: Devuelve el DataFrame que formado por las columnas de los vectores `v1`, `v2`, etc, con los nombres `x1`, `x2`, etc, respectivamente.

```
using DataFrames

julia> df = DataFrame(Nombre = ["María", "Luis", "Carmen"], Edad = [22, 18, 20])
3×2 DataFrame
 Row   Nombre  Edad
       String  Int64

   1   María      22
   2   Luis       18
   3   Carmen     20
```

8.1.2 Creación de DataFrames desde una url

Para crear una DataFrame a partir de un fichero csv en la nube, se utiliza la siguiente función del paquete `CSV.jl`:

- `CSV.read(download(url), DataFrame)`: Devuelve el DataFrame que resulta de importar el fichero csv con la url `url`.

```
using DataFrames, CSV

julia> df = CSV.read(download("https://raw.githubusercontent.com/asalber/manual-python/m
14×6 DataFrame
 Row   nombre                          edad   sexo     peso     altura   colesterol
       String                          Int64  String1  Int64?   String7  Int64?

   1   José Luis Martínez Izquierdo      18  H             85  1,79            182
   2   Rosa Díaz Díaz                    32  M             65  1,73            232
   3   Javier García Sánchez             24  H        missing  1,81            191
   4   Carmen López Pinzón               35  M             65  1,70            200
   5   Marisa López Collado              46  M             51  1,58            148
   6   Antonio Ruiz Cruz                 68  H             66  1,74            249
   7   Antonio Fernández Ocaña           51  H             62  1,72            276
   8   Pilar Martín González             22  M             60  1,66        missing
   9   Pedro Gálvez Tenorio              35  H             90  1,94            241
  10   Santiago Reillo Manzano           46  H             75  1,85            280
  11   Macarena Álvarez Luna             53  M             55  1,62            262
  12   José María de la Guía Sanz        58  H             78  1,87            198
  13   Miguel Angel Cuadrado Gutiérrez   27  H            109  1,98            210
  14   Carolina Rubio Moreno             20  M             61  1,77            194
```

9 Otras aplicaciones

9.1 Teoría de grafos

- JuliaGraphs. Análisis de grafos en Julia.
- Graphs. Creación y análisis de grafos.
- GraphPlot. Dibujo de grafos.

9.2 Cálculo simbólico

- SymPy. Sistema de Álgebra Computacional (CAS) basdado en la librería `SymPy` de Python.
- Symbolics Sistema de Álgebra Computacional (CAS) basado en Julia.

9.3 Aprendizaje automático

- MLJ.jl. Funciones para los principales algoritmos de aprendizaje automático.