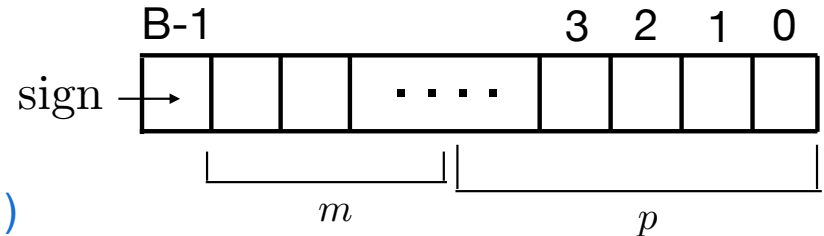


Bit representation of floating-point numbers

Arbitrary real-valued numbers cannot be represented by bits

- approximated by certain rational numbers; “floating-point numbers”
- p bits for “significand” (fraction, mantissa)
- m bits for exponent
- 1 sign bit



here bits $b(i)$ are counted
from left ($i=0$) to right ($i=p-1$)

$$R = \text{sign} \times 2^e \sum_{i=0}^{p-1} b(i)2^{-i} \rightarrow \text{sign} \times 2^e \left(1 + \sum_{i=0}^{p-1} b(i)2^{-(i+1)} \right)$$

$$1 \leq \text{significand} < 2$$

The exponent can be positive or negative

- exactly how the exponent is stored is a bit subtle (we don't need the details)

On most computers:

- single-precision (4 bytes); $p=23$, $m=8$ (precision about 7 decimals)
- double-precision (8 bytes); $p=53$, $m=10$ (precision about 16 decimals)
- some times 16-byte quadruple precision is available

Special values represented

+0, -0, +infinity, -infinity, “not a number” NaN

Example: floats, random numbers, arrays, multiple dispatch [\[randomarray.jl\]](#)

```
function makerandom(n::Int)
    r=Array{Float64}(undef,n)
    for i=1:n
        r[i]=rand()
    end
    return r
end
```

First method, Int argument

- array with n elements (undefined contents)

- one way to loop over values i

- i:th element assigned a random value in [0,1)

```
function makerandom(m::Float64)
    n=round(Int,m)
    r=Array{Float32}(undef,n)
    for i=1:n
        r[i]=rand()
    end
    return r
end
```

Second method, Float64 argument

- round to closest integer and convert to Int

In general, any number of methods can be used, as long as they can be uniquely identified by their arguments (more on functions later)


Two function declarations, same name, different argument types

- it's really one function with two methods

- the method that matches calling arguments is dispatched

Code calling this function:

```
n=5
m=convert(Float64,n)      - converts integer n to 64-bit float
a=makerandom(n)
for i=1:n
    println(i, " ",a[i])
end
a=makerandom(m)
for i=1:n
    println(i, " ",a[i])
end
```

Output 

Note, in second method:

Float64 value is assigned to a Float32 variable; OK but of course some precision is lost

Floating-point types in Julia

Float16, Float32, Float64

```
$ 1 0.768629462884634
$ 2 0.2031804749902122
$ 3 0.1664474670812679
$ 4 0.5501970241421752
$ 5 0.4978716671303165
$ 1 0.5057016
$ 2 0.65821403
$ 3 0.2276439
$ 4 0.83020467
$ 5 0.84432185
```

Examples of matrix operations [\[matrix.jl\]](#)

Function to make a random $n \times n$ matrix

```
function randmatrix(n::Int)
    mat=Array{Float64}(undef,n,n)
    for j=1:n
        for i=1:n
            mat[i,j]=rand()
        end
    end
    return mat
end
```

matrix = 2-dimensional array

```
a=randmatrix(n)
b=randmatrix(n)
c=a*b
```

here * means actual matrix multiplication

```
for i=1:n
    println(a[i,:], " ", b[i,:], " ", c[i,:])
end
```

: means all elements

```
c=a.*b
```

point . before operator means element-by-element

```
b=inv(a)
```

Base function for matrix inversion

Notes on variable/function names, non-ascii symbols

Names are case-sensitive; “Var” is different from “var”

- customary to use lower case for variables and function names
- use upper case first letter for module and type names
- functions that change arguments end in “!” (I violate this rule...)

Names of variables and functions can contain Unicode characters

- in addition to the conventional ASCII characters

Example: `function 笨蛋(γ)` [[specialnames.jl](#)]
 $\alpha=1$
 $\beta=1$
 $\delta=\alpha+\beta+\gamma$
 return δ
end
`println(笨蛋(2))`

Depending on your editor/environment, it may be painful to enter characters

- in the REPL, Latex commands can be used, e.g., `\delta<tab>` for δ
- probably better to avoid using special characters in code

Elementary Mathematical Operations from julia.org

Expression	Name	Description
$+x$	unary plus	the identity operation
$-x$	unary minus	maps values to their additive inverses
$x + y$	binary plus	performs addition
$x - y$	binary minus	performs subtraction
$x * y$	times	performs multiplication
x / y	divide	performs division
$x \div y$	integer divide	x/y , truncated to an integer
$x \setminus y$	inverse divide	equivalent to y / x
$x ^ y$	power	raises x to the y th power
$x \% y$	remainder	equivalent to $\text{rem}(x, y)$

$x \text{ op } y$ is really equivalent to $\text{op}(x,y)$, i.e., op is a function with two arguments. Try in the REPL:

```
|julia> +  
+ (generic function with 190 methods)
```

same as $\text{div}(x,y)$; \div is `\div<tab>` in the REPL

Updating ops: `+=` `-=` `*=` `/=` `\=` `÷=` `%=` `^=` `&=` `|=` `⊔=` `>>>=` `>>=` `<<=`

$x += y$ is equivalent to $x = x + y$, etc.

Rounding functions

Function	Description	Return type
<code>round(x)</code>	round x to the nearest integer	<code>typeof(x)</code>
<code>round(T, x)</code>	round x to the nearest integer	T
<code>floor(x)</code>	round x towards -Inf	<code>typeof(x)</code>
<code>floor(T, x)</code>	round x towards -Inf	T
<code>ceil(x)</code>	round x towards +Inf	<code>typeof(x)</code>
<code>ceil(T, x)</code>	round x towards +Inf	T
<code>trunc(x)</code>	round x towards zero	<code>typeof(x)</code>
<code>trunc(T, x)</code>	round x towards zero	T

Conversion function

`Convert(T,x)`

converts x to type T if possible

Functions related to division

Function	Description
<code>div(x,y)</code> , <code>x÷y</code>	truncated division; quotient rounded towards zero
<code>fld(x,y)</code>	floored division; quotient rounded towards -Inf
<code>cld(x,y)</code>	ceiling division; quotient rounded towards +Inf
<code>rem(x,y)</code>	remainder; satisfies $x == \text{div}(x,y)*y + \text{rem}(x,y)$; sign matches x
<code>mod(x,y)</code>	modulus; satisfies $x == \text{fld}(x,y)*y + \text{mod}(x,y)$; sign matches y

Sign related functions

Function	Description
<code>abs(x)</code>	a positive value with the magnitude of x
<code>abs2(x)</code>	the squared magnitude of x
<code>sign(x)</code>	indicates the sign of x, returning -1, 0, or +1
<code>signbit(x)</code>	indicates whether the sign bit is on (true) or off (false)
<code>copysign(x,y)</code>	a value with the magnitude of x and the sign of y
<code>flipsign(x,y)</code>	a value with the magnitude of x and the sign of x*y

Common math functions

Function	Description
<code>sqrt(x)</code> , \sqrt{x}	square root of x
<code>cbrt(x)</code> , $\sqrt[3]{x}$	cube root of x
<code>hypot(x,y)</code>	hypotenuse of right-angled triangle with other sides of length x and y
<code>exp(x)</code>	natural exponential function at x
<code>expm1(x)</code>	accurate $\exp(x) - 1$ for x near zero
<code>ldexp(x,n)</code>	$x \cdot 2^n$ computed efficiently for integer values of n
<code>log(x)</code>	natural logarithm of x
<code>log(b,x)</code>	base b logarithm of x
<code>log2(x)</code>	base 2 logarithm of x
<code>log10(x)</code>	base 10 logarithm of x
<code>log1p(x)</code>	accurate $\log(1+x)$ for x near zero
<code>exponent(x)</code>	binary exponent of x
<code>significand(x)</code>	binary significand (a.k.a. mantissa) of a floating-point number x

Trig functions (radian args)

`sin` `cos` `tan` `cot` `sec` `csc`
`sinh` `cosh` `tanh` `coth` `sech` `csch`
`asin` `acos` `atan` `acot` `asec` `acsc`
`asinh` `acosh` `atanh` `acoth` `asech` `acsch`
`sinc` `cosc`

Trig functions (degree args)

`sind` `cosd` `tand` `cotd` `secd` `cscd`
`asind` `acosd` `atand` `acotd` `asecd` `acscd`

Many special functions in package **SpecialFunctions**

Boolean Data Type and boolean operations

The type `Bool` is for variables with values `true` or `false`

- it uses 8 bits (even though 1 bit would be enough)
- `Bool` is a subset of `Int` (`true=1`, `false=0`)
- In most respects `Bool` is the same as `Int8`

Example: `function trueorfalse(b::Bool) [bool.jl]` Output:

```
println(b)           true
println(b*1)         1
println(b*2)         2
println(b>true)      true
end
trueorfalse(true)    false
println()            0
trueorfalse(false)  false
```

Boolean ops: `!x` - negation

`x and y are of type boolean (expressions)` `x && y` - and (short-circuit; only evaluates `y` if `x` is true)
`x || y` - or (short-circuit; only evaluates `y` if `x` is false)

Numerical comparisons from julia-lang.org

Operator	Name
<code>==</code>	equality
<code>!=, ≠</code>	inequality
<code><</code>	less than
<code><=, ≤</code>	less than or equal to
<code>></code>	greater than
<code>>=, ≥</code>	greater than or equal to

Used, e.g., in
“if-elseif-else” control structure
or “ternary operator”
- operator with 3 args

[Program ifelse.jl online](#)

These all result in
“true” or “false”
boolean values

```
if a
    dosomething
elseif b
    dosomethingelse
else
    doyetanotherthing
end
```

```
X ? Y : Z
```

- if X is true, evaluate Y
- if X is false, evaluate Z

Bitwise boolean Operations

from julia-lang.org

Performs boolean operations on

- individual bits of one argument
- same-index bits of two arguments

Expression	Name
$\sim x$	bitwise not
$x \& y$	bitwise and
$x y$	bitwise or
$x \underline{\vee} y$	bitwise xor (exclusive or)
$x \ggg y$	logical shift right
$x \gg y$	arithmetic shift right
$x \ll y$	logical/arithmetic shift left

Examples of these ops
in program 'bitwise.jl' on the web site

- same as `xor(x,y)`

- shifts all bits

- leaves sign bit (1s are shifted in if negative)

- does not preserve sign (0s shifted in on right)

Vectorized operators

All operators acting on single variables have vectorized “dot” versions

For an array x (any number of dimensions):

`.op x` performs “op” on each element

Example, for a vector x of length n

```
for i=1:n
    x[i] = x[i]^2
end
```

does the same as

```
x .= x.^2
```

`x = x.^2` also works, but allocates
a new x if x already exists (slower)

can also be expressed with the `@.` macro

```
@. x = x^2
```

Examples in [program timing.jl](#) online

- this program also introduces functionality for timing code for performance