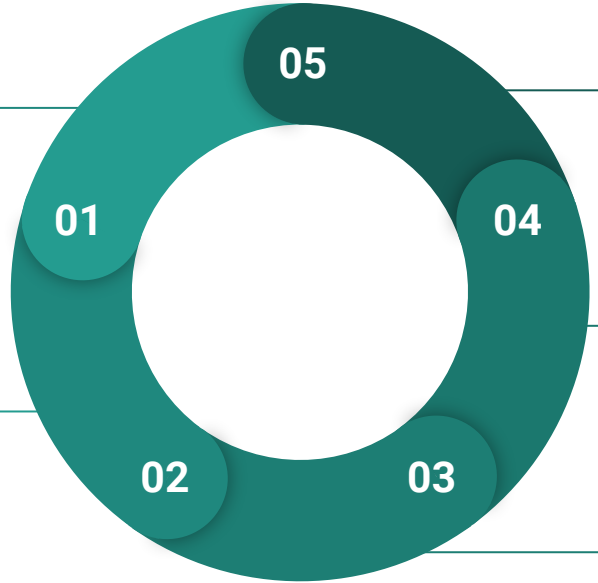
Introduction to the Julia Programming Language

NSF ACES

Jian Tao and Wes Brashear

20 July 2024

Introduction to Julia: Outline



Part I. A brief overview of Julia (~10 mins)

Part II. Getting Started with Julia on ACES (~10 mins)

Part III. Mathematical Operations in Julia (~10 mins)

Part IV. Variables, Data Types and Structures, Functions and Flow Control (~35 mins)

Part V. Plotting with Julia (~10 mins)

Part I: A brief overview of Julia

Julia is a high-level general-purpose dynamic programming language primarily designed for **high-performance numerical analysis and computational science**.

- Born in MIT's Computer Science and Artificial Intelligence Lab in 2009
- Combined the best features of Ruby, MatLab, C, Python, R, and others
- First release in 2012
- Latest stable release v1.10.2 as of Mar 31, 2024
- https://julialang.org/
- customized for "greedy, unreasonable, demanding programmers".
- Julia Computing established in 2015 to provide commercial support.

Major features of **Julia:**

- **Fast**: designed for high performance
- **General**: supporting different programming patterns
- **Dynamic**: dynamically-typed with good support for interactive use
- **Technical**: efficient numerical computing with a math-friendly syntax
- **Optionally typed**: a rich language of descriptive data types
- **Composable**: Julia's packages naturally work well together

*"Julia is as programmable as Python while it is as fast as Fortran for number crunching. It is like **Python on steroids**."*

--an anonymous Julia user on the first impression of Julia.

Where to Run Julia

- Juno IDE - developed for the Julia language (no longer under development)
- VSCode - extensions for Julia are actively being managed
- Jupyter Notebook
- Julia REPL
  - Run, Evaluate, Print, Loop
  - Interactive
  - Searchable history, tab-completion, keybindings, dedicated help and shell modes

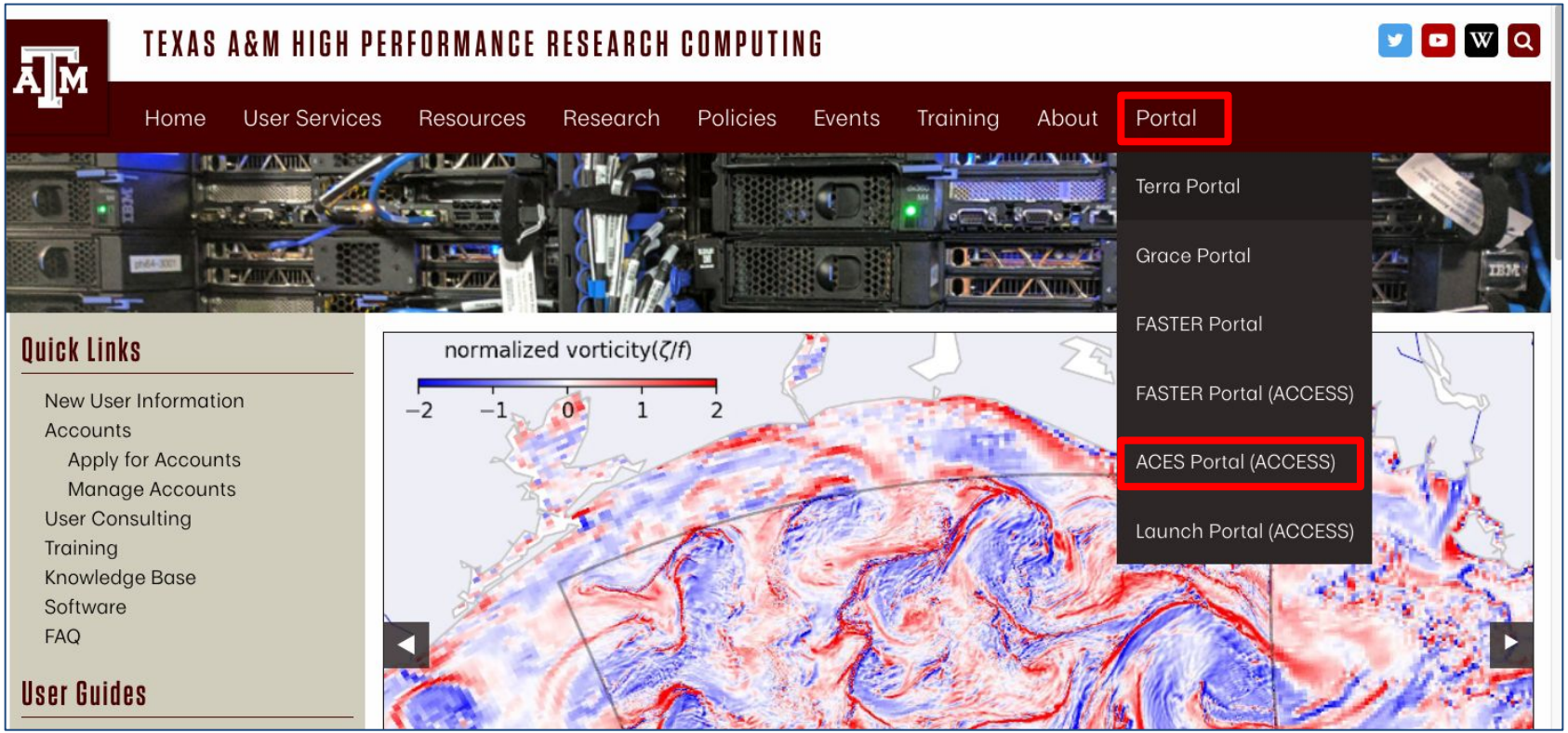
Part II: Getting started with Julia on ACES

ACES

ACCELERATING COMPUTING
FOR EMERGING SCIENCES

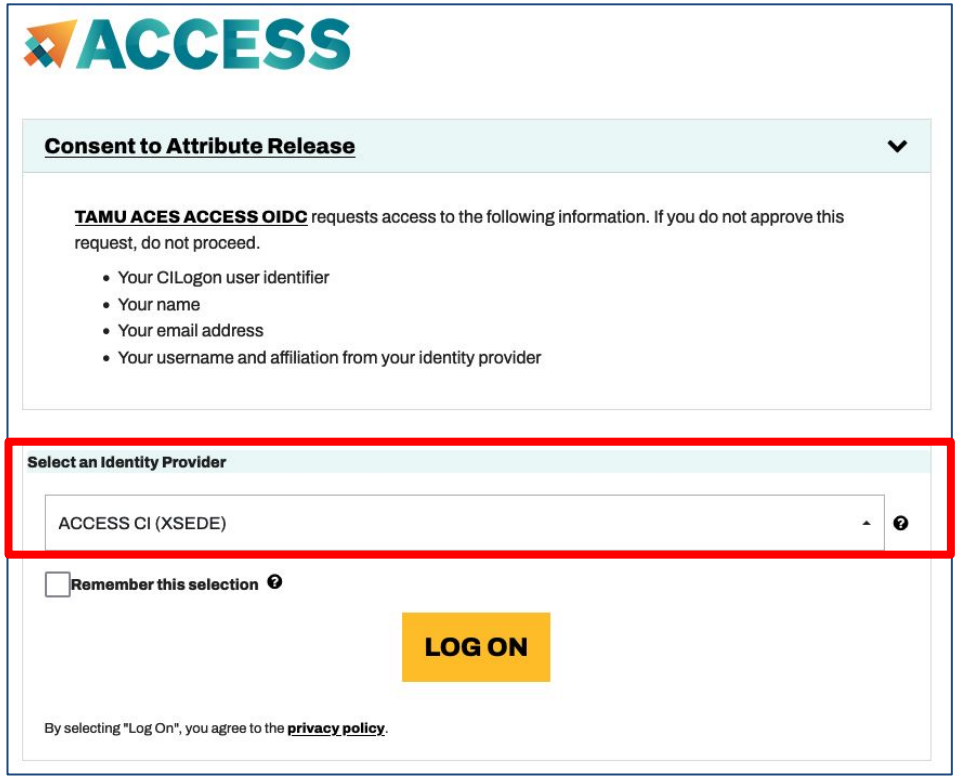
julia

Accessing the HPRC ACES Portal

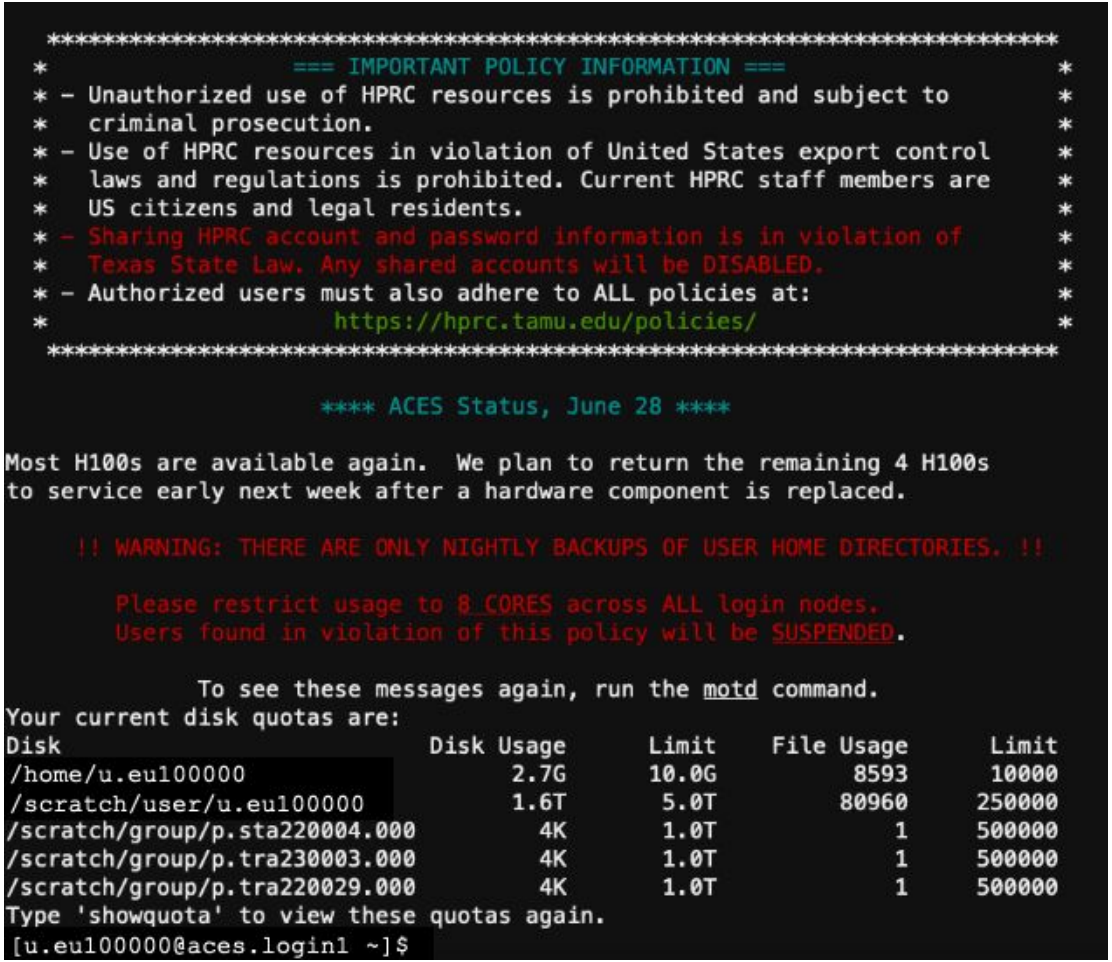


Accessing the HPRC ACES Portal

Select the Identity Provider appropriate for your account



Accessing the ACES Shell


****
* === IMPORTANT POLICY INFORMATION === *
* - Unauthorized use of HPRC resources is prohibited and subject to *
* criminal prosecution. *
* - Use of HPRC resources in violation of United States export control *
* laws and regulations is prohibited. Current HPRC staff members are *
* US citizens and legal residents. *
* - Sharing HPRC account and password information is in violation of *
* Texas State Law. Any shared accounts will be DISABLED. *
* - Authorized users must also adhere to ALL policies at: *
* https://hprc.tamu.edu/policies/ *
****
**** ACES Status, June 28 ****
Most H100s are available again. We plan to return the remaining 4 H100s
to service early next week after a hardware component is replaced.
!! WARNING: THERE ARE ONLY NIGHTLY BACKUPS OF USER HOME DIRECTORIES. !!
Please restrict usage to 8 CORES across ALL login nodes.
Users found in violation of this policy will be SUSPENDED.
To see these messages again, run the motd command.
Your current disk quotas are:
Disk Disk Usage Limit File Usage Limit
/home/u.eu100000 2.7G 10.0G 8593 10000
/scratch/user/u.eu100000 1.6T 5.0T 80960 250000
/scratch/group/p.sta220004.000 4K 1.0T 1 500000
/scratch/group/p.tra230003.000 4K 1.0T 1 500000
/scratch/group/p.tra220029.000 4K 1.0T 1 500000
Type 'showquota' to view these quotas again.
[u.eu100000@aces.login1 ~]$

Using Pre-installed Julia Modules

Step 1. Find the module to be loaded

$ module spider julia
...
Description:
Julia is a high-level, high-performance dynamic programming language for numerical computing

Versions:
Julia/1.8.5-linux-x86_64
Julia/1.9.3-linux-x86_64
Julia/1.10.0-musl-x86_64
Julia/1.10.2-linux-x86_64

...

Step 2. Load the module

$ module load Julia/1.10.2-linux-x86_64

Step 3. Start Julia REPL

$ julia

You can also use the web-based interface to find software modules available on HPRC systems.

Installing Your Own Copy of Julia

Step 1. Find the version to be installed at Download Julia

Current stable release: v1.10.4 (June 4, 2024)

Checksums for this release are available in both SHA256 and MD5 formats.

Platform	64-bit	32-bit
Windows [help]	installer, portable	installer, portable
macOS x86 (Intel or Rosetta) [help]	.dmg, .tar.gz	
macOS (Apple Silicon) [help]	.dmg, .tar.gz	
Generic Linux on x86 [help]	glibc (GPG), musl[1] (GPG)	glibc (GPG)
Generic Linux on ARM [help]	AArch64 (GPG)	
Generic Linux on PowerPC [help]	little endian (GPG)	
Generic FreeBSD on x86 [help]	.tar.gz (GPG)	

* You can install the latest Julia version (v1.10.4 June 4, 2024) directly by running this in your terminal:

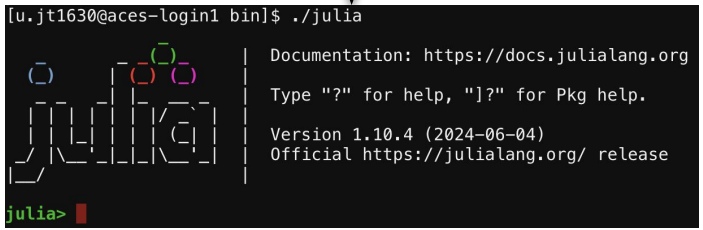
$curl -fsSL https://install.julialang.org | sh

Step 2. Download & Unzip

```
$ cd $SCRATCH
$ wget https://.../julia-1.10.4-linux-x86_64.tar.gz
$ tar -zxvf julia-1.10.4-linux-x86_64.tar.gz
```

Step 3. Start Julia REPL

```
$ module purge
$ cd $SCRATCH/julia-1.10.4/bin; ./julia
```



Installing Julia Packages

```
# export Julia Depot path (default to ~/.julia)
$export JULIA_DEPOT_PATH=$SCRATCH/.julia

# start Julia
$julia

# type ']' to open Pkg REPL
# press backspace or ^C to quit Pkg REPL.
julia>]
(@v1.9) pkg> add Plots UnicodePlots Plotly
```

Copying Course Examples

- Navigate to your personal scratch directory

```
$ cd $SCRATCH
```

- Files for this course are located at

  `/scratch/training/julia_examples.tgz`

  Make a copy in your personal scratch directory

```
$ cp /scratch/training/julia/julia_examples.tgz $SCRATCH/
```

- Extract the files

```
$ tar -zxvf julia_examples.tgz
```

- Enter this directory (your local copy)

```
$ cd julia_examples
```

```
$ julia helloworld.jl
```

Julia - Quickstart

- The julia program starts the interactive **REPL**.
- You can switch to the **shell mode** if you type a **semicolon**.
- A **question mark** will switch you to the **help mode**.
- The **<TAB>** key can help with autocompletion.
- To get version information:

```
julia> versioninfo()
```

- Special symbols can be typed with a backslash **and** **<TAB>**, but they might not show properly on the web-based terminal.

```
julia> \sqrt <TAB>
julia> for i ∈ 1:10 println(i) end  #\in <TAB>
```

Julia REPL Keybindings

Keybinding	Descrition
^d	Exit (when buffer is empty)
^c	Interrupt or cancel
^l	Clear console screen
Return/Enter, ^J	New line, executing if it is complete
? or ;	Enter help or shell mode (when at start of a line)
^R, ^S	Incremental history search
]	Enter Pkg REPL
Backspace or ^c	Quit Pkg REPL

Part III: Mathematical Operations in Julia

ACES

ACCELERATING COMPUTING
FOR EMERGING SCIENCES

Arithmetic Operators

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 ÷ y	integer divide	x / y, truncated to an integer
x ^ y	power	raises x to the yth power
x % y	remainder	equivalent to rem(x,y)

More about Arithmetic Operators

- The **order of operations** follows the math rules.
- The **updating version** of the operators is formed by placing a "**=**" immediately after the operator. For instance, **x+=3** is equivalent to **x=x+3**.
- **Unicode** could be defined as operator.
- **A "dot" operation** is automatically defined to perform the operation element-by-element on arrays in every binary operation.
- **Numeric Literal Coefficients**: Julia allows variables to be immediately preceded by a numeric literal, implying multiplication.

Arithmetic Expressions

Some examples:

```
julia> 10/5*2
julia> 5*2^3+4\2
julia> -2^4
julia> 8^1/3
julia> pi*ℯ #\euler <TAB>
julia> x=1; x+=3.1
julia> x=[1,2]; x = x.^(-2)
```

Relational Operators

==	True, if it is equal
!=,≠	True, if not equal to #\ne <TAB>
<	less than
>	greater than
<=,≤	less than or equal to #\le <TAB>
>=,≥	greater than or equal to #\ge <TAB>

** try ≠(4,5), what does this mean? How about !=(4,5)*

Boolean and Bitwise Operators

&&	Logical and
||	Logical or
!	Not
xor()	Exclusive OR
|	Bitwise OR
~	Negate
&	Bitwise And
>>	Right shift
<<	Left shift

NaN and Inf

NaN is a not-a-number value of type Float64.

Inf is positive infinity of type Float64.

-Inf is negative infinity of type Float64.

- **Inf** is equal to itself and greater than everything else except **NaN**.
- **-Inf** is equal to itself and less then everything else except **NaN**.
- **NaN** is not equal to, not less than, and not greater than anything, including itself.

```
julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false

julia> isequal(NaN, NaN)
true

julia> isnan(1/0)
false
```

Part IV: Variables, Data Types and Structures, Functions and Flow Control

ACES

ACCELERATING COMPUTING
FOR EMERGING SCIENCES

Variables

Examples:

```
julia> b = true; typeof(b)
julia> x = "Hi"
julia> y = 10
julia> z = complex(1, y)
julia> println(b, x, y, z)
julia> b = nothing; show(b)
julia> 🏈=2; 🏃=1     # \:football: <TAB> \:runner: <TAB>
```

Naming Rules for Variables

- Variable names must begin with a letter or underscore

  julia> 4c = 12

- Names can include any combinations of letters, numbers, underscores, and exclamation symbol. Some unicode characters could be used as well

  julia> c_4 = 12; δ = 2

- Maximum length for a variable name is not limited
- Julia is case sensitive. The variable name **A** is different than the variable name **a**.

Displaying Variables

- We can display a variable (i.e., show its value) by simply typing the name of the variable at the command prompt (leaving off the semicolon).

- We can also use **print** or **println** (print plus a new line) to display variables.

```
julia> print("The value of x is:"); print(x)
julia> println("The value of x is:"); print(x)
```

Exercise

Create two variables: **a = 4** and **b = 17.2**

Now use Julia to perform the following set of calculations:

$(b+5.4)^{1/3}$

$b^2-4b+5a$

```
a>b && a>1.0
```

```
a!=b
```

Data Types

- Data types in Julia are polymorphic, they can be
  - dynamic: determined at runtime (e.g. Python, R)
  - static: defined explicitly (e.g. Java, C++)

*more information on declaring types here: https://docs.julialang.org/en/v1/manual/types/

- Data types include:
  - char
  - string
  - float
  - int
  - bool

Chars and Strings

Char: represent a single character

- Denoted with single quotations ‘ ‘

String: can represent an object of one or more characters

- Denoted with double quotations “ “

```
julia> a = 'H'  #a is a character object
julia> b = "H"  #a is a string with length 1
```

Strings can be easily manipulated with built-in functions:

```
julia> c = string('s') * string('d')
julia> length(c); d = c^10*"4"; split(d,"s")
```

Working with Strings

1. The built-in type used for strings in Julia is **String**. This supports the full range of Unicode characters via the UTF-8 encoding.
2. Strings are **immutable.**
3. One can do comparisons and a limited amount of arithmetic with Char.
4. All indexing in Julia is **1-based**: the first element of any integer-indexed object is found at index 1.

```
julia> str = "Hello, world!"
julia> c = str[1]       #c = 'H'
julia> c = str[end]     #c = '!'
julia> c = str[2:8]     #c = "ello, w"
```

Working with Strings

Interpolation: Julia allows interpolation into string literals using **$**, as in Perl. To include a literal **$** in a string literal, escape it with a backslash:

```
julia> "1 + 2 = $(1 + 2)"  #"1 + 2 = 3"
julia> print("\$100 dollars!\n")
```

Working with Strings

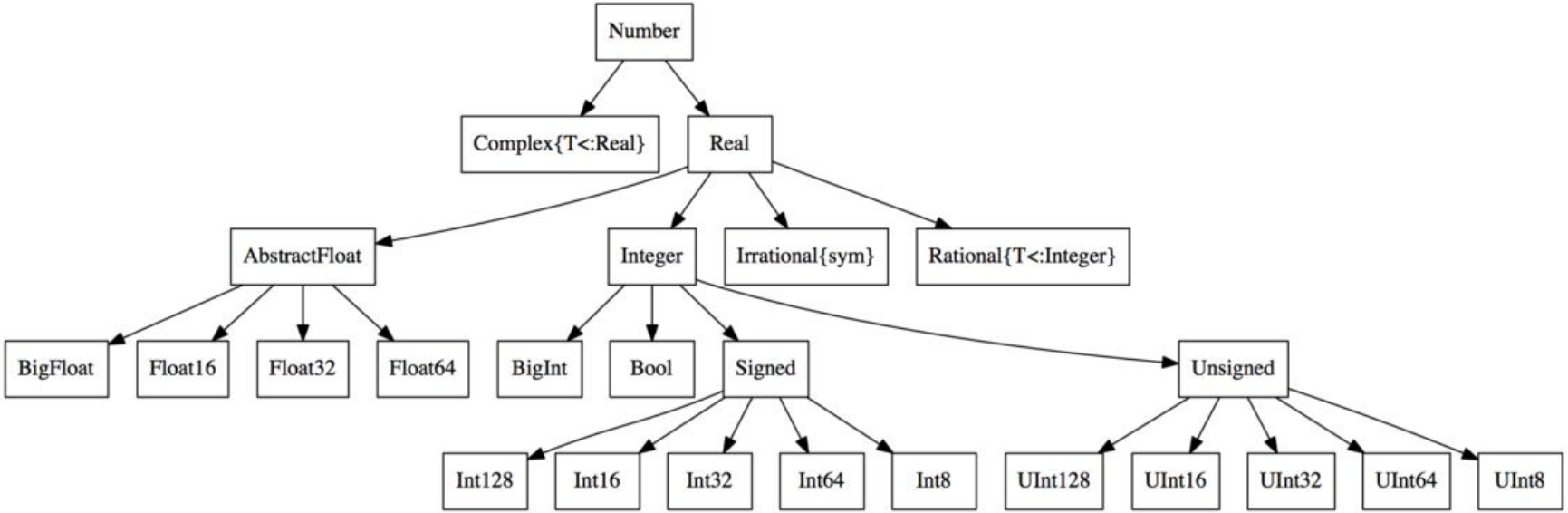
Julia comes with a collection of tools to handle strings.

```
julia> str="Julia"
julia> occursin("lia", str)
julia> z = repeat(str, 10)
julia> firstindex(str)
julia> lastindex(str)
julia> length(str)
```

Julia also supports Perl-compatible regular expressions.

```
julia> occursin(r"^\s*(?:#|$)", "# a comment")
```

Numerical Data Types



Integer Data Types

Type	Signed?	Number of bits	Smallest value	Largest value
Int8	✓	8	-2^7	2^7 - 1
UInt8		8	0	2^8 - 1
Int16	✓	16	-2^15	2^15 - 1
UInt16		16	0	2^16 - 1
Int32	✓	32	-2^31	2^31 - 1
UInt32		32	0	2^32 - 1
Int64	✓	64	-2^63	2^63 - 1
UInt64		64	0	2^64 - 1
Int128	✓	128	-2^127	2^127 - 1
UInt128		128	0	2^128 - 1
Bool	N/A	8	false (0)	true (1)

Handling Big Integers

An overflow happens when a number goes beyond the representable range of a given type. Julia provides **BigInt** type to handle big integers.

```
julia> x = typemax(Int64)
julia> x + 1
julia> x + 1 == typemin(Int64)
julia> x = big(typemax(Int64))^100
```

Floating Point Data Types

Type	Precision	Number of bits	Range
Float16	half	16	-65504 to -6.1035e-05 6.1035e-05 to 65504
Float32	single	32	−3.402823E38 to −1.401298E-45 1.401298E-45 to 3.402823E38
Float64	double	64	-1.79769313486232E308 to -4.94065645841247E-324 4.94065645841247E-324 to 1.79769313486232E308

- Additionally, full support for **Complex** and **Rational Numbers** is built on top of these primitive numeric types.
- All numeric types interoperate naturally without explicit casting thanks to a user-extensible **type promotion system**.

Working with Floating Points

Perform each of the following calculations in your head.

```
julia> a = 4/3
julia> b = a - 1
julia> c = 3*b
julia> e = 1 - c
```

What does Julia get?

Working with Floating Points

What does Julia get?

```
julia> a = 4/3   #1.3333333333333333

julia> b = a - 1 #0.3333333333333326

julia> c = 3*b   #0.9999999999999998

julia> e = 1 - c #2.220446049250313e-16
```

It is impossible to perfectly represent all real numbers using a finite string of 1's and 0's.

Working with Floating Points

Now try the following with BigFloat

```
julia> a = big(4)/3

julia> b = a - 1

julia> c = 3*b

julia> e = 1 - c #-1.7272337110188...e-77
```

Next, set the precision and repeat the above

```
julia> setprecision(4096)
```

BigFloat variables can store floating point data with arbitrary precision with a performance cost.

Complex and Rational Numbers

The global constant **im** is bound to the complex number **i**, representing the principal square root of **-1**.

```
julia> 2(1 - 1im)

julia> sqrt(complex(-1, 0))
```

Note that **3/4im == 3/(4*im) == -(3/4*im)**, since a literal coefficient binds more tightly than division. **3/(4*im)!=(3/4*im)**

Julia has a **rational number** type to represent exact ratios of integers.
Rationals are constructed using the **//** operator, e.g., **9//27**

Some Useful Math Functions

Rounding and division functions

Function	Description
round(x)	round x to the nearest integer
floor(x)	round x towards -Inf
ceil(x)	round x towards +Inf
trunc(x)	round x towards zero
div(x,y)	truncated division; quotient rounded towards zero
fld(x,y)	floored division; quotient rounded towards -Inf
cld(x,y)	ceiling division; quotient rounded towards +Inf
rem(x,y)	remainder; satisfies x == div(x,y)*y + rem(x,y); sign matches x
gcd(x,y...)	greatest positive common divisor of x, y,...
lcm(x,y...)	least positive common multiple of x, y,...

Sign and absolute value functions

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

** The built-in math functions in Julia are implemented in C(openlibm).*

Getting Help with Functions

- For help on a specific function or macro, type **?** followed by its name, and press enter. This only works if you know the name of the function you want help with. With **^S** and **^R** you can also do historical search.

```
Julia> ?cos
```

- Type **?help** to get more information about help

```
Julia> ?help
```

Code Elements and Syntax

- Comments start with '#'
- Compound expressions
  - Can be created with ";" chains

```
julia>z = (x = 1, y = 2, x + y)
```

  - or with blocks: start with "begin" and finish with "end"

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
```

Data Structures

Tuples: ordered sequence of elements.

- Good for small fixed-length collections
- Immutable

```
julia> t = (1, 2, 3)
julia> t = ((1, 2), (3, 4))
julia> t[1][2]
```

Data Structures

Arrays: ordered collection of elements.

- In Julia, **arrays** are used for lists, vectors, tables, and matrices
- Mutable

```
julia> a = [1, 2, 3]          # column vector
julia> b = [1 2 3]            # row vector
julia> c = [1 2 3; 4 5 6]     # 2x3 vector
julia> d = [n^2 for n in 1:5]
julia> f = zeros(2,3); g = rand(2,3)
julia> h = ones(2,3); j = fill("A",9)
julia> k = reshape(rand(5,6),10,3)
julia> [a a]                          # hcat
julia> [b;b]                          # vcat
```

Array and Matrix Operations

Many Julia operators and functions can be used preceded with a dot. These versions are the same as their non-dotted versions, and work on the arrays element by element.

```
julia> b = [1 2 3; 4 5 7; 7 8 9]
julia> b .+ 10          # each element + 10
julia> sin.(b)          # sin function
julia> b'               # transpose (transpose(b))
julia> inv(b)           # inverse
julia> b * b            # matrix multiplication
julia> b .* b           # element-wise multiplication
julia> b .^ 2           # element-wise square
```

Data Structures

Sets: mainly used to eliminate repeated numbers in a sequence/list and are also used to perform some standard set operations.

```
julia> months=Set(["Nov","Dec","Dec"])
julia> typeof(months)
julia> push!(months,"Sept")
julia> pop!(months,"Sept")
julia> in("Dec", months)
julia> m=Set(["Dec","Mar","Feb"])
julia> union(m,months)
julia> intersect(m,months)
julia> setdiff(m,months)
```

Data Structures

Dictionaries: mappings between keys and items stored in the dictionaries.

- To define a dictionary, use **`Dict()`**

```
julia> m=Dict("Oct"=>"October",
              "Nov"=>"November",
              "Dec"=>"December")
julia> m["Oct"]
julia> get(m, "Jan", "N/A")
julia> haskey(m, "Jan")
julia> m["Jan"]="January"
julia> delete!(m, "Jan")
julia> keys(m)
julia> values(m)
julia> map(uppercase, collect(keys(m)))
```

Flow Control

Julia has the following controlling constructs:

- **ternary** expressions
- **boolean switching** expressions
- **if elseif else end** - conditional evaluation
- **for end** - iterative evaluation
- **while end** - iterative conditional evaluation
- **try catch error throw** exception handling

Conditional Statements

- Execute statements if condition is true.
- There is no "**switch"** and "**case"** statement in Julia.
- There is an "**ifelse"** statement.

```
julia> a = 8
julia> if a>10
           println("a > 10")
       elseif a<10
           println("a < 10")
       else
           println("a = 10")
       end
```

Loop Control Statements - *for*

for statements help repeatedly execute a block of code for a certain number of iterations. Loop variables are local.

```
julia> for i in 0:1:10
           if i % 3 == 0
               continue
           end
           println(i)
       end
julia> for l in "julia"
           print(l, "-^-")
       end
```

Loop Control Statements - *while*

while statements repeatedly execute a block of code as long as a condition is satisfied.

```
julia> n = 1
julia> s = 0
julia> while n <= 100
           s = s + n
           n = n + 1
       end
julia> println(s)
```

Exception Handling Blocks

try … catch construction checks for errors and handles them gracefully

```
julia> s = "test"
julia> try
           s[1] = "p"
       catch err
           println("caught an error: $err")
           println("continue with execution!")
       end
```

Functions: Building Blocks of Julia

Two equivalent ways to define a function

```
julia> function func(x,y)
           return x + y, x
       end

julia> Σ(x,y) = x + y, x
```

Operators are functions

```
julia> +(1,2); plusfunc=+
Julia> plusfunc(2,3)
```

Recommended style for function definition: **append ! to names of functions that modify their arguments**

Functions with Optional Arguments

You can define functions with optional arguments with default values.

```
julia> function point(x, y, z=0)
         println("$x, $y, $z")
       end
julia> point(1,2); point(1,2,3)
```

Keywords and Positional Arguments

Keywords can be used to label arguments. Use a **semicolon** after the function's unlabelled arguments, and follow it with one or more **keyword=value** pairs

```
julia> function func(a, b, c="one"; d="two")
         println("$a, $b, $c, $d")
       end
julia> func(1,2); func(d="four", 1, 2, "three")
```

Dotted Functions

Dot syntax can be used to vectorize functions, i.e., applying functions **elementwise** to arrays.

```
julia> func(a, b) = a * b
julia> func(1, 2)
julia> func.([1,2], 3)
julia> sin.(func.([1,2],[3,4]))
```

Part V: Plotting with Julia

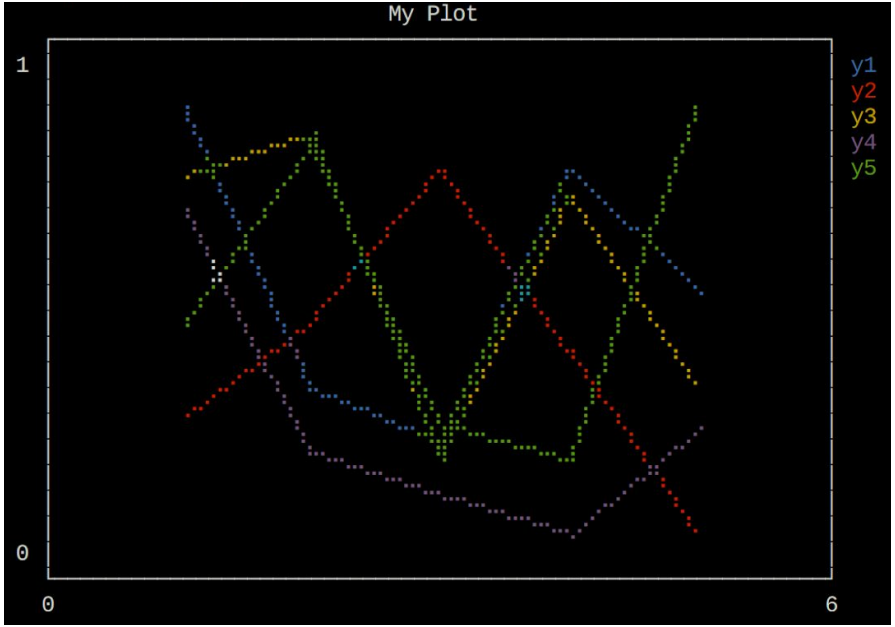
ACES

ACCELERATING COMPUTING
FOR EMERGING SCIENCES

UnicodePlots

UnicodePlots is simple and lightweight and it plots directly in your terminal (*might not work with web-based shell*).

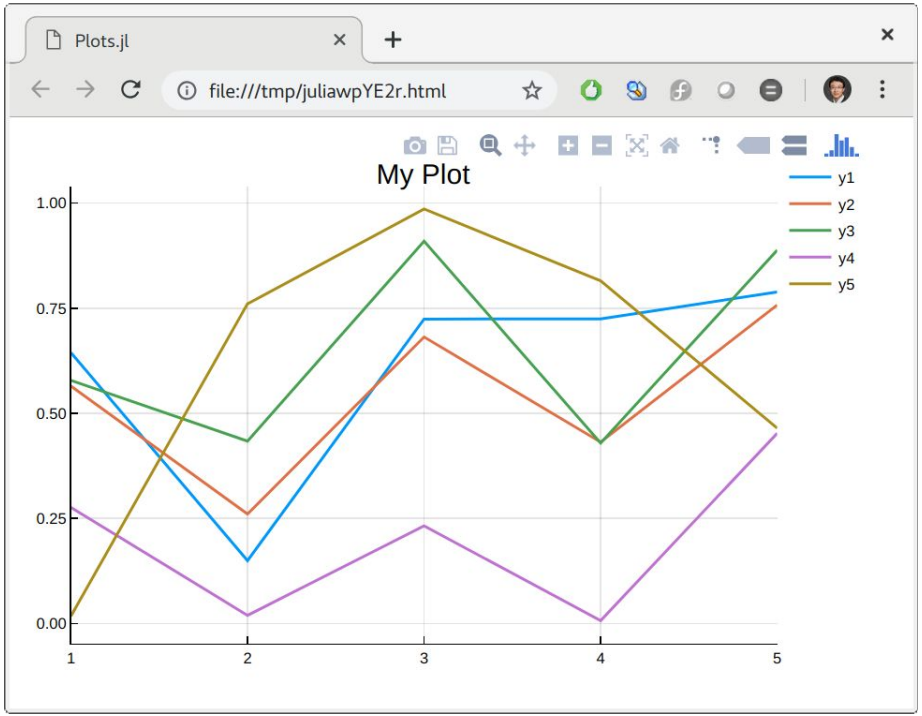
```
julia> using Plots
julia> unicodeplots()
julia> plot(rand(5,5), linewidth=2, title="My Plot")
```



Plotly Julia Library

Plotly creates leading open source software for Web-based data visualization and analytical apps. Plotly Julia Library makes interactive, publication-quality graphs online (not working with web-based shell).

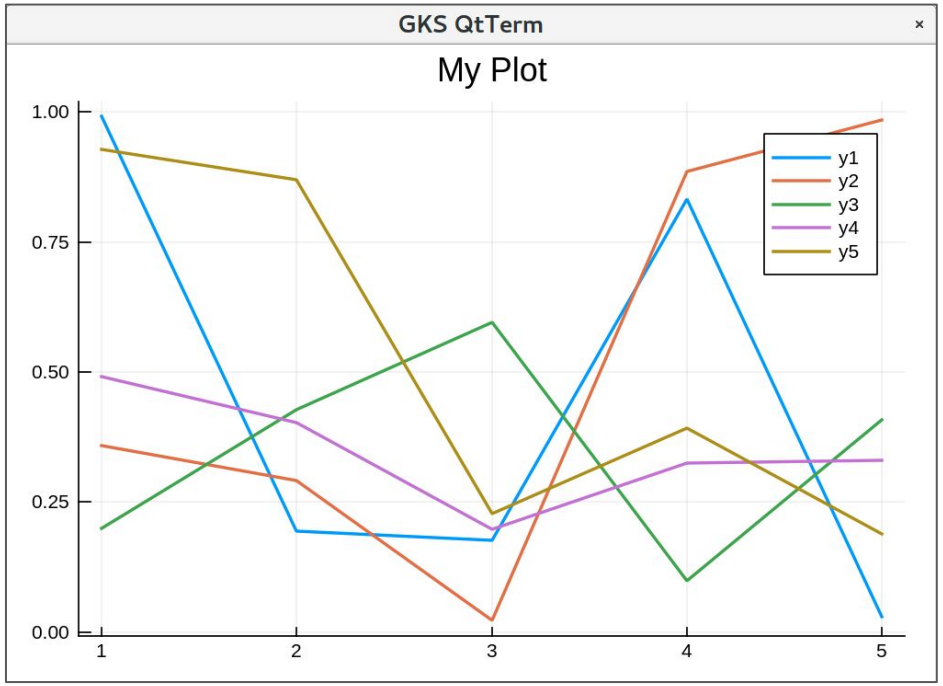
```
julia> using Plots
julia> plotly()
julia> plot(rand(5,5), linewidth=2, title="My Plot")
```



GR Framework

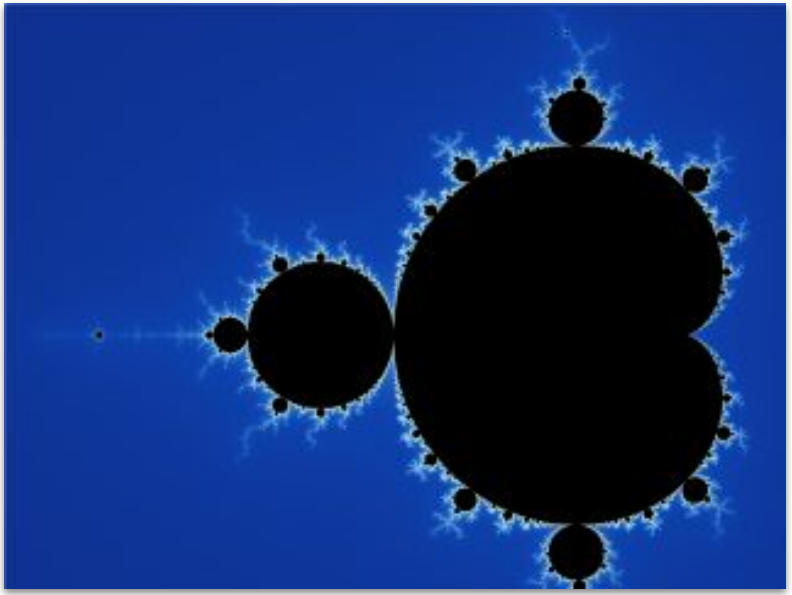
GR framework is a universal framework for cross-platform visualization applications (not working with web-based shell).

```
julia> using Plots
julia> gr()
julia> plot(rand(5,5), linewidth=4, title="My Plot", size=(1024,1024))
```



Fractals

- Fractals refer to geometric shapes containing detailed structures at arbitrarily small scales
- Fractals appear similar at various scales



Credit: Fractal - Wikipedia

Benoit Mandelbrot Set

$$z_{n+1} = z_n^2 + c$$

- z and c are complex numbers.
- Starting with z_0=0.
- Mandelbrot set is the set of values of c when z_n remains bounded for a relatively large n.

Mandelbrot - Julia Version

```
function mandelbrot(a)
  z = 0
  for i=1:50
      z = z^2 + a
  end
  return z
end

for y=1.0:-0.05:-1.0
  for x=-2.0:0.0315:0.5
      abs(mandelbrot(complex(x, y))) < 2
? print("*") : print(" ") # in one line
  end
  println()
end
```

The first published picture of the Mandelbrot set, by Robert W. Brooks and Peter Matelski in 1978, reproduced with the code to the left.

Online Resources

Official Julia Document

https://docs.julialang.org/en/v1/

Julia Online Tutorials

https://julialang.org/learning/

Introducing Julia (Wikibooks.org)

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

MATLAB–Python–Julia cheatsheet

https://cheatsheets.quantecon.org/

The Fast Track to Julia

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

Acknowledgments

- The slides are created based on the materials from Julia official website and the Wikibook Introducing Julia at wikibooks.org.
- Support from Texas A&M Engineering Experiment Station (TEES), Texas A&M Institute of Data Science (TAMIDS), and Texas A&M High Performance Research Computing (HPRC).
- Support from NSF OAC Award #2019129 - MRI: Acquisition of FASTER - Fostering Accelerated Sciences Transformation Education and Research
- Support from NSF OAC Award #2112356 - Category II: ACES - Accelerating Computing for Emerging Sciences

Texas A&M at PEARC24

Talk/Event	Date/Time	Room
Tutorial: Hands-on exercises on the Intel Data Center GPU Max 1100 (PVC-GPU) for AI/ML and Molecular Dynamics Workflows on the ACES Testbed	Mon, July 22, 2024 9:00 AM-12:30 PM ET	Room 553B
Seventh Workshop on Strategies for Enhancing HPC Education and Training (SEHET24)	Mon, July 22, 2024 9:00 AM-12:30 PM ET	Room 557
Workshop: Providing cutting-edge computing testbeds to the science and engineering community	Mon, July 22, 2024 1:30 PM-5:00 PM ET	Room 554A
Workshop: Engaging Secondary Students in Computing: K12 Outreach	Mon, July 22, 2024 1:30 PM-5:00 PM ET	Room 553A
Cultivating Cyberinfrastructure Careers through Student Engagement at Texas A&M University High Performance Research Computing	Tue, July 23, 2024 11:00 AM-11:25 AM ET	Junior Ballroom
Insight Gained from Migrating a Machine Learning Model to Intelligence Processing Units	Tue, July 23, 2024 11:00 AM-11:25 AM ET	Room 551 A&B
BOF 4: What's in it for me? How can we truly democratize the research computing and data community?	Tue, July 23, 2024 1:30 PM-2:30 PM ET	Room 551 A&B

Texas A&M at PEARC24

Talk/Event	Date/Time	Room
BRICCs: Building Pathways to Research Cyberinfrastructure at Under Resourced Institutions	Tue, July 23, 2024 3:25 PM-3:50 PM ET	Junior Ballroom
Memory Bandwidth Performance across Accelerators	Tue, July 23, 2024 3:25 PM-3:50 PM ET	Ballroom B
Container Adoption in Campus High Performance Computing	Wed, July 24, 2024 11:00 AM-11:25 AM ET	Ballroom B
Engaging Secondary Students in Computing and Cybersecurity	Wed, July 24, 2024 3:15 PM-3:30 PM ET	Room 557
Exploring the Viability of Composable Architectures to Overcome Memory Limitations in High Performance Computing Workflows	Wed, July 24, 2024 3:45 PM-4:00 PM ET	Room 553 A&B
Performance of Molecular Dynamics Acceleration Strategies on Composable Cyberinfrastructure	Wed, July 24, 2024 4:15 PM-4:30 PM ET	Room 551 A&B
BOF 17: Fantastic ACCESS Cyberinfrastructure Resources and Where to Find Them	Wed, July 24, 2024 4:45 PM-5:45 PM ET	Room 553 A&B
BOF 18: Recipes to build successful cross-institutional collaborative computing	Wed, July 24, 2024 4:45 PM-5:45 PM ET	Junior Ballroom

High Performance
Research Computing
DIVISION OF RESEARCH

Thank you

- We gratefully acknowledge support from National Science Foundation awards #2112356 (ACES), #2019129 (FASTER) and #19257614 (SWEETER)

- Please visit our talks and BoFs at PEARC24
- Helpdesk: help@hprc.tamu.edu