Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

MCS 507 Lecture 10
Mathematical, Statistical and Scientific Software
Jan Verschelde, 13 September 2023

Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

Julia community

The page `https://julialang.org/community` points to the wide ecosystem of packages, organized in several different areas:

1. General
2. Computing
3. Mathematics
4. Scientific Domains
5. Astronomy/Space
6. Physics/Quantum mechanics
7. Data Science
8. Visualizations
9. Miscellaneous

The Julia Packages `https://juliapackages.com` is a website for browsing Julia packages.

`Pkg` is the builtin package manager

```
julia> Base.load_path()
2-element Array{String,1}:
```

shows the path names to two folders:

1. the location of the installed packages, and
2. the standard library of the Julia language.

Visit `https://docs.julialang.org`
for the extensive documentation.

Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

OSCAR = Open Source Computer Algebra Research

https://www.oscar-system.org

Written in Julia, it combines the well established systems

- Singular, computational algebraic geometry,
- GAP, computational discrete algebra,
- Polymake, polyhedral geometry,
- ANTIC (Hecke, Nemo and AbstractAlgebra), number theory,

into a comprehensive tool for computational algebra.

M. Belotti, M. Joswig, C. Meroni, V. Schleis, and J. Schmitt:
Algebraic and Geometric Computations in OSCAR.
SIAM News. pages 9-10, September 2023.

Development at `https://github.com/oscar-system`.

Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

calling code defined by Python modules

```
julia> using PyCall

julia> np = pyimport("numpy")
PyObject <module 'numpy' from
... shows the location of numpy if installed ...

julia> np.cos(1)
0.5403023058681398

julia> np.cos(1 + 2*pi)
0.5403023058681399

julia>
```

This will work if

1. `numpy` was installed in Python, and
2. `PyCall` uses that Python interpreter.

calling our own module

At the command prompt (Linux, Mac OS X, or PowerShell):

```
$ cat hellopython.py
def sayhello():
    print("Python says hello!")

if __name__ == "__main__":
    sayhello()

$ python hellopython.py
Python says hello!
```

inserting the current directory to the Python path

In a new interactive Julia session:

```
julia> using PyCall

julia> pushfirst!(PyVector(pyimport("sys")["path"]), @__DIR__)
PyObject [ ... path name of the current folder ... , etc...

julia> hello = pyimport("hellopython")
PyObject <module 'hellopython' from
... the current folder ...

julia> hello.sayhello()
Python says hello!

julia> pyimport("hellopython")[:sayhello]()
Python says hello!

julia>
```

Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

writing a value to a text file

Let us write a 32-digit approximation of π to file.

```
julia> import Printf

julia> strpi = Printf.@sprintf("%.32e", pi)
"3.14159265358979311599796346854419e+00"

julia> f = open("store32pi.txt", "w")
IOStream(<file store32pi.txt>)

julia> write(f, strpi)
38

julia> close(f)

julia> strpi = ""
""
```

reading a value from a text file

```
julia> f = open("store32pi.txt", "r")
IOStream(<file store32pi.txt>)

julia> r = readline(f)
"3.14159265358979311599796346854419e+00"

julia> setprecision(110)
110

julia> pi32 = parse(BigFloat, r)
3.14159265358979311599796346854419
```

calling external programs

The argument of `run()` must be within backticks.

```
julia> sayhello = `echo "Hello world!"`
`echo 'Hello world!'`

julia> run(sayhello)
Hello world!
Process(`echo 'Hello world!'`, ProcessExited(0))

julia> ret = run(sayhello);
Hello world!

julia> ret
Process(`echo 'Hello world!'`, ProcessExited(0))
```

running a pipeline

The Unix pipe `ls | grep ".jl"` lists all files
with the extension .jl in the current directory.

```
# Illustrates a pipeline to list the names of
# the Julia scripts in the current directory.

run(pipeline(`ls` ,
    pipeline(`grep ".jl"`, stdout="listing.txt")))
outputfile = open("listing.txt", "r")
lines = readlines(outputfile)
println("Julia scripts in current directory :")
for i=1:length(lines)
    println(lines[i])
end
```

Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

calling an external root finder

Suppose we have a good polynomial root finder
that we want to use from Julia code.

If the executable has a command line interface,
then we can wrap the program as follows:

1. Prepare input file as `input.txt`.
2. Call the program redirecting its input from the file.

   Redirect the output of the program to the file `output.txt`.
3. Parse the output in `output.txt` into Julia data.

roots and eigenvalues

To simulate the wrapping, we call a Julia program which has an interactive command line interface.

The `LinearAlgebra` module exports `eigen` which returns the eigenvalues and eigenvectors of a matrix.

The companion matrix of $p(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}$ is

$$C(p) = \begin{bmatrix} 0 & 0 & \cdots & 0 & -c_1/c_n \\ 1 & 0 & \cdots & 0 & -c_2/c_n \\ 0 & 1 & \cdots & 0 & -c_3/c_n \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -c_{n-1}/c_n \end{bmatrix}.$$

The eigenvalues of $C(p)$ are the solutions of $p(x) = 0$.

polynomials over any ring R

```
struct Polynomial{R}
   cff::Vector{R}
end

"""
    function ( p::Polynomial )(x)

Evaluates the polynomial at x.
"""
function ( p::Polynomial )(x)
    v = p.cff[end]
    for i = (length(p.cff)-1):-1:1
        v = v*x + p.cff[i]
    end
    return v
end
```

eigenvalues of the companion matrix

```
import LinearAlgebra

"""
    function roots(p::Polynomial)

Returns the roots as the eigenvalues of
the companion matrix of p.
"""
function roots(p::Polynomial)
    deg = length(p.cff)-1
    mat = zeros(deg, deg)
    for i = 2:deg
        mat[i,i-1] = 1.0
    end
    for i = 1:deg
        mat[i,deg] = -p.cff[i]/p.cff[deg+1]
    end
    eigvals, eigvecs = LinearAlgebra.eigen(mat)
    return eigvals
end
```

testing random inputs

```
"""
    function test(deg::Int)

Generates a polynomial of degree deg
with random coefficients and computes  its roots.
"""
function test(deg::Int)
    c = [rand() for i=1:(deg+1)]
    p = Polynomial(c)
    println("the coefficients : $(p.cff)")
    x = rand()
    y = p(x)
    println("value at a random point : $y")
    r = roots(p)
    for i=1:length(r)
        y = p(r[i])
        println("the root $i : $(r[i])")
        println("its residual : $y")
    end
end
```

reading the coefficient vector

```
"""
    function read(deg::Int,verbose::Bool=true)

Reads deg+1 complex coefficients of a polynomial.
Displays a prompt is verbose.
Returns the polynomial.
"""
function read(deg::Int,verbose::Bool=true)
    cffs = []
    for i=1:deg+1
        if verbose
            print("cff[$i] : ")
        end
        line = readline()
        cnbr = parse(Complex{Float64}, line)
        append!(cffs, cnbr)
    end
    return Polynomial(cffs)
end
```

the main interactive function

```
"""
    function interactive(degree::Int=0, verbose::Bool=true)

Prompts for the degree (if not provided)
and the coefficients of the polynomial.
Returns the roots of the polynomial.
"""
function interactive(degree::Int=0, verbose::Bool=true)
    if degree > 0
        deg = degree
    else
        if verbose
            print("Give the degree : ")
        end
        line = readline()
        deg = parse(Int, line)
    end
    if verbose
        println("degree : $deg")
        println("reading $(deg+1) coefficients ...")
    end
```

code continued

```
    p = read(deg,verbose)
    if verbose
        println("the coefficients : $(p.cff)")
    end
    r = roots(p)
    if verbose
        println("the roots : $r")
    end
    return r
end
```

The `verbose` flag is good for developing and testing.

When used by other code, the `verbose` is set to `false`.

the main function

```
"""
    function main()

Runs a random or interactive test.
"""
    # test(4)
    roots = interactive(0, false)
    for i=1:length(roots)
        println(roots[i])
    end
end

main()
```

If saved in `polyroots.jl`, run as `julia polyroots.jl`.

Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

make input

```
import Printf

"""
    function make_input(dim::Int,
                        name::String="input.txt")

Generates a random coefficient vector, of dimension dim,
writes the coefficients to file.
By default, the name of the file is input.txt.
"""
function make_input(dim::Int,
                    name::String="input.txt")
    c = rand(dim, 1)
    infile = open(name, "w")
    strn = Printf.@sprintf("%d", length(c)-1)
    write(infile, strn)
    write(infile, "\n")
    for i=1:length(c)
        strc = Printf.@sprintf("%.16e", c[i])
        write(infile, strc)
        write(infile, "\n")
    end
    close(infile)
end
```

parse output

```
"""
    function get_output(name::String="output.txt")

Opens the file with the given name
and returns the numbers on the file.
"""
function get_output(name::String="output.txt")
    result = []
    infile = open(name, "r")
    lines = readlines(infile)
    for i=1:length(lines)
        nbr = parse(Complex{Float64}, lines[i])
        append!(result, nbr)
    end
    return result
end
```

run

```
"""
    function main()

Generates a random coefficient vector,
writes the coefficients to file,
calls the root finder, and then
reads the roots from file.
"""
function main()
    make_input(5)
    infile = "input.txt"
    run(pipeline(`cat input.txt`,
        pipeline(`julia polyroots.jl`, stdout="output.txt")))
    r = get_output()
    println("the roots : ")
    for i=1:length(r)
        println(r[i])
    end
end

main()
```

Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

calling C code

With `ccall` we call C library functions.

Compile the C code with the `-shared` and `-fPIC` options.

The `clock` function of the standard C library:

```
julia> t = ccall((:clock, "libc"),Int32,())
1390571
```

The `rand` function of the standard C library (`libc` omitted):

```
julia> t = ccall(:rand, Int32, ())
2068623728
```

Both `clock` and `rand` take no arguments
and return an `Int32`.

receiving a C string

The `getenv` returns a pointer to the value of an environment variable.

We write `(Cstring,)` because we receive a 1-tuple which contains a `Cstring`.

```
julia> p = ccall((:getenv, "libc"),
                 Cstring, (Cstring,), "SHELL")
Cstring(0x00007ffee80ddc03)

julia> unsafe_string(p)
"/bin/bash"
```

allocation and conversion of C pointer

The C function `gethostname` returns the name of the computer.

The function below wraps the C function and returns a Julia string.

```
"""
    function gethostname()

Allocates memory for the hostname first
and the converts the pointer to a Julia string.
"""
function gethostname()
    hostname = Vector{UInt8}(undef, 128)
    ccall((:gethostname, "libc"), Int32,
          (Ptr{UInt8}, Csize_t),
          hostname, sizeof(hostname))
    hostname[end] = 0; # ensure null-termination
    return unsafe_string(pointer(hostname))
end
```

Julia Ecosystems

1. Connecting Julia Code
   - community
   - open source computer algebra research
   - calling code defined by Python modules
   - working with text files, calling external programs
2. Wrapping Executables
   - calling an external root finder
   - make input, run, parse output
3. Calling C Code
   - running `ccall`
   - calling `qsort`

calling `qsort`

We will wrap the C function qsort.

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void*, const void*));
```

The C `qsort` needs a callback function `compare`,
to define the order between the elements to be sorted.

defining a callback function in Julia for C

In Julia, we define the callback function as

```
function mycompare(a, b)::Cint
    return (a < b) ? -1 : ((a > b) ? +1 : 0)
end
```

Observe the definition of the return type.

To pass this Julia function to C, we use the macro `@function`.

```
mycompare_c = @cfunction(mycompare, Cint,
                         (Ref{Cdouble}, Ref{Cdouble}))
```

calling `qsort` on 16 random numbers

```
println("A random sequence of numbers :")
A = rand(16, 1)
println(A)

println("Calling qsort ...")

ccall(:qsort, Cvoid,
      (Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Cvoid}),
      A, length(A), sizeof(eltype(A)), mycompare_c)

println("The sorted sequence :")
println(A)
```

examining the parameters of `ccall`

The second argument in `ccall` is the return type.
The third argument of `ccall`

```
ccall(:qsort, Cvoid,
      (Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Cvoid}),
      A, length(A), sizeof(eltype(A)), mycompare_c)
```

is the tuple

```
(Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Cvoid})
```

which are the types of the input arguments of `qsort`:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void*, const void*));
```

The actual parameters are given as the last 4 parameters.

Summary and Exercises

Julia offers a large computational ecosystem.

Calling C code from Julia is straightforward.

Exercises:

1. What would be the best way to compute roots of polynomials in Julia? Explore the available packages in Julia's ecosystem.
2. Take a root finder in the SciPy stack and describe how to call this root finder from Julia.
3. Consider MPSolve of lecture 6 and develop a command line interface to call MPSolve from Julia code.
4. Can you call MPSolve with `ccall`?