12.010 Computational Methods of Scientific Programming

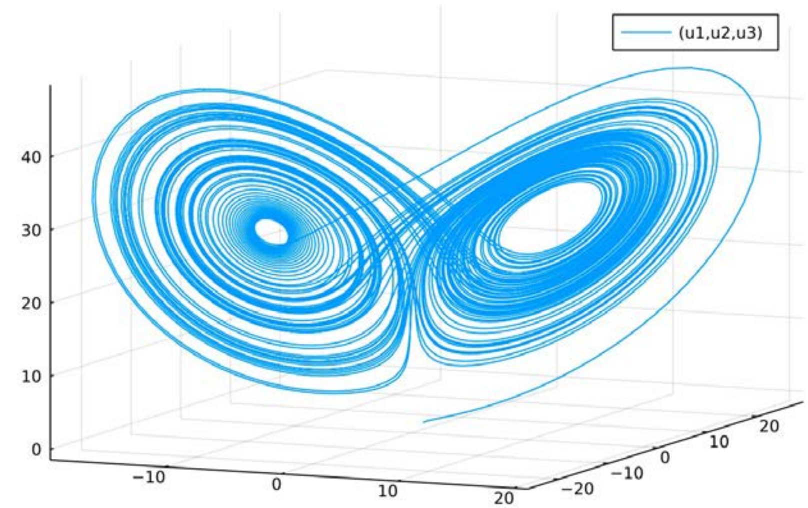
Lecture 20: Julia Objects, Multiple Dispatch

Summary

- Objects v Multiple Dispatch
  - Julia does not use “classes” exactly
    - instead functions can have multiple methods
    - which method/methods are used depends on the type of arguments
    - types are used extensively within Julia to direct which methods get invoked - "multiple dispatch"

Julia programming language cont…

ODE solve – look at Lorenz 63 again




```
l63!(du,u,p,t)
```

```
Steps forward the Lorenz63 equations for time t
elements 1, 2 and 3 of vector u and with parame
Elements of du are set to dxdt, dydt and dzdt r
```

```
julia> using DifferentialEquations, Plots
julia> u0=[1.,0.,0.]
julia> p=[10.,28.,8/3]
julia> tspan=90.,100.)
julia> prob=ODEProblem(l63!,u0,tspan,p)
julia> sol=solve(prob)
julia> plot(sol,vars=(1,2,3))
```

Julia programming language cont…

Defining l63.jl

- “”” … “”” is docstring
  - within docstring indent name by 4 characters, ``` creates literal block e.g.

```
"""
    l63!(du,u,p,t)

 Steps forward the Lorenz63 equations for time `t`.
 Prognostic variables x, y, z are defined as
 elements 1, 2 and 3 of vector `u` and with parameters
 σ, ρ and β defined as elements of vector `p`.
 Elements of du are set to dxdt, dydt and dzdt respectively.

```
 julia> using DifferentialEquations, Plots
 julia> u0=[1.,0.,0.]
 julia> p=[10.,28.,8/3]
 julia> tspan=(0.,100.)
 julia> prob=ODEProblem(l63!,u0,tspan,p)
 julia> sol=solve(prob)
 julia> plot(sol,vars=(1,2,3))
```

"""
function l63!(du,u,p,t)
        x, y, z=u
        σ, ρ, β=p

        dxdt=σ*(y-x)
        dydt=ρ*x-x*z-y
        dzdt=x*y-β*z

        du[1], du[2], du[3]=
        dxdt , dydt, dzdt
        return
end
```

```
help?> l63!
search: l63!

  l63!(du,u,p,t)

 Steps forward the Lorenz63 equations for time t. Prognostic variables x, y, z are defined as
 elements 1, 2 and 3 of vector u and with parameters σ, ρ and β defined as elements of vector p.
 Elements of du are set to dxdt, dydt and dzdt respectively.

  julia> using DifferentialEquations, Plots
  julia> u0=[1.,0.,0.]
  julia> p=[10.,28.,8/3]
  julia> tspan=(0.,100.)
  julia> prob=ODEProblem(l63!,u0,tspan,p)
  julia> sol=solve(prob)
  julia> plot(sol,vars=(1,2,3))

julia>
```

Julia programming language cont…

- Defining l63.jl
  - Function NAME! is a convention for indicating a function that alters/sets its arguments, by default first argument are the altered variables.
  - Here we use `l63!` the altered variable is `du`.
  - The arguments to l63! fit the standard layout for the Julia DifferentialEquations package ODE solver.

```
"""
     l63!(du,u,p,t)

 Steps forward the Lorenz63 equations for time `t`.
 Prognostic variables x, y, z are defined as
 elements 1, 2 and 3 of vector `u` and with parameters
 σ, ρ and β defined as elements of vector `p`.
 Elements of du are set to dxdt, dydt and dzdt respectively.

```
 julia> using DifferentialEquations, Plots
 julia> u0=[1.,0.,0.]
 julia> p=[10.,28.,8/3]
 julia> tspan=(0.,100.)
 julia> prob=ODEProblem(l63!,u0,tspan,p)
 julia> sol=solve(prob)
 julia> plot(sol,vars=(1,2,3))
```

"""
function l63!(du,u,p,t)
        x, y, z=u
        σ, ρ, β=p

        dxdt=σ*(y-x)
        dydt=ρ*x-x*z-y
        dzdt=x*y-β*z

        du[1], du[2], du[3]=
        dxdt , dydt, dzdt
        return
end
```

Julia programming language cont…

- Using l63.jl
  - To use a function directly we can use include in the REPL e.g.

```
julia> include("l63.jl")
l63!

julia> du=[0.,0.,0.];u0=[1.,0.,0.];p=[10.,28.,8/3]
3-element Vector{Float64}:
 10.0
 28.0
  2.6666666666666665

julia> l63!(du,u0,p,0)

julia> du
3-element Vector{Float64}:
 -10.0
  28.0
   0.0
```

```
"""
    l63!(du,u,p,t)

 Steps forward the Lorenz63 equations for time `t`.
 Prognostic variables x, y, z are defined as
 elements 1, 2 and 3 of vector `u` and with parameters
 σ, ρ and β defined as elements of vector `p`.
 Elements of du are set to dxdt, dydt and dzdt respectively.

```
 julia> using DifferentialEquations, Plots
 julia> u0=[1.,0.,0.]
 julia> p=[10.,28.,8/3]
 julia> tspan=(0.,100.)
 julia> prob=ODEProblem(l63!,u0,tspan,p)
        sol=solve(prob)
        plot(sol,vars=(1,2,3))
```

```
 l63!(du,u,p,t)
 x, y, z=u
 σ, ρ, β=p

 dxdt=σ*(y-x)
 dydt=ρ*x-x*z-y
 dzdt=x*y-β*z

 du[1], du[2], du[3]=
 dxdt , dydt,  dzdt
 return
end
```
```

Julia programming language cont…

- Using l63.jl
  - Can also include in a notebook (see runl63.ipynb).

```
[1]: include("l63.jl")

[1]: l63!

[2]: du=[0.,0.,0.];u0=[1.,0.,0.];p=[10.,28.,8/3];

[3]: l63!(du,u0,p,0);

[4]: du

[4]: 3-element Vector{Float64}:
      -10.0
       28.0
        0.0

[ ]:
```

````
"""
    l63!(du,u,p,t)

 Steps forward the Lorenz63 equations for time `t`.
 Prognostic variables x, y, z are defined as
 elements 1, 2 and 3 of vector `u` and with parameters
 σ, ρ and β defined as elements of vector `p`.
 Elements of du are set to dxdt, dydt and dzdt respectively.

```
 julia> using DifferentialEquations, Plots
 julia> u0=[1.,0.,0.]
 julia> p=[10.,28.,8/3]
 julia> tspan=(0.,100.)
 julia> prob=ODEProblem(l63!,u0,tspan,p)
 julia> sol=solve(prob)
 julia> plot(sol,vars=(1,2,3))
```

"""
function l63!(du,u,p,t)
        x, y, z=u
        σ, ρ, β=p

        dxdt=σ*(y-x)
        dydt=ρ*x-x*z-y
        dzdt=x*y-β*z

        du[1], du[2], du[3]=
        dxdt , dydt,  dzdt
        return
end
````

Julia packages

- Most Julia code will use other packages
  - By convention examples tend to include “using”, but not package install so may need to install packages for code to work.
  - Packages are installed once and then will be available for reuse

```
"""
    l63!(du,u,p,t)

Steps forward the Lorenz63 equations for time `t`.
Prognostic variables x, y, z are defined as
elements 1, 2 and 3 of vector `u` and with parameters
σ, ρ and β defined as elements of vector `p`.
Elements of du are set to dxdt, dydt and dzdt respectively.

```
julia> using DifferentialEquations, Plots
julia> u0=[1.,0.,0.]
julia> p=[10.,28.,8/3]
```

```
[2]: using DifferentialEquations
```

```
ArgumentError: Package DifferentialEquations not found in current path:
- Run `import Pkg; Pkg.add("DifferentialEquations")` to install the DifferentialEquations package.


Stacktrace:
 [1] require(into::Module, mod::Symbol)
   @ Base ./loading.jl:893
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1116
```
```

Julia packages

- Most Julia code will use other packages
  - To install
    - `using Pkg; Pkg.add(“NAME”)`
    - a package will often download and install other dependencies
    - can also use] in REPL to enter “Package mode”

```
(chrishill) pkg> add DifferentialEquations
```

    - use backspace to exit “Package mode”
    - once a package is installed it is added to “.julia” directory (or location of JULIA_DEPOT_PATH) and does not need to be downloaded again.

```
[2]: using DifferentialEquations

ArgumentError: Package DifferentialEquations not found in current path:
- Run `import Pkg; Pkg.add("DifferentialEquations")` to install the DifferentialEquations package.


Stacktrace:
 [1] require(into::Module, mod::Symbol)
   @ Base ./loading.jl:893
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1116

[*]: using Pkg; Pkg.add("DifferentialEquations")

  Installing known registries into `~/.julia`
       Added registry `General` to `~/.julia/registries/General`
   Resolving package versions...
   Installed Adapt ───────────────────────── v3.3.1
   Installed TimerOutputs ────────────────── v0.5.13
   Installed DifferentialEquations ───────── v6.20.0
   Installed ConstructionBase ────────────── v1.3.0
   Installed ExprTools ───────────────────── v0.1.6
   Installed Requires ────────────────────── v1.1.3
   Installed BandedMatrices ──────────────── v0.16.11
   Installed ForwardDiff ─────────────────── v0.10.23
   Installed DynamicPolynomials ──────────── v0.3.21
   Installed DEDataArrays ────────────────── v0.2.0
   Installed PolyesterWeave ──────────────── v0.1.2
   Installed LayoutPointers ──────────────── v0.1.4
   Installed StatsBase ───────────────────── v0.33.12
   Installed VectorizationBase ───────────── v0.21.21
   Installed StochasticDiffEq ────────────── v6.41.0
   Installed SortingAlgorithms ───────────── v1.0.1
   Installed DataValueInterfaces ─────────── v1.0.0
   Installed PDMats ──────────────────────── v0.11.3
   Installed NLSolversBase ───────────────── v7.8.2
   Installed CloseOpenIntervals ──────────── v0.1.4
   Installed RuntimeGeneratedFunctions ───── v0.5.3
   Installed OffsetArrays ────────────────── v1.10.8
   Installed OrdinaryDiffEq ──────────────── v5.67.0
   Installed PoissonRandom ───────────────── v0.4.0
   Installed DataAPI ─────────────────────── v1.9.0
   Installed FunctionWrappers ────────────── v1.1.2
   Installed FillArrays ──────────────────── v0.12.7
   Installed DiffEqNoiseProcess ──────────── v5.9.0
   Installed Preferences ─────────────────── v1.2.2
   Installed InverseFunctions ────────────── v0.1.2
   Installed JuliaFormatter ──────────────── v0.18.1
   Installed CEnum ───────────────────────── v0.4.1
   Installed ParameterizedFunctions ──────── v5.12.2
   Installed LaTeXStrings ────────────────── v1.3.0
   Installed CPUSummary ──────────────────── v0.1.6
   Installed SpecialFunctions ────────────── v1.8.1
```

Hands on

- Try `runl63jl.ipynb` (needs `l63.jl`)
- May need to `Pkg.import "DifferentialEquations"` and/or "`Plots`"
- Try and create a Julia equivalent of `Lec05-ode.ipynb` (try without looking at `l63_euler_julia_butterfly.ipynb` first!)

main fall-2021-12.010 / Lec05-ode.ipynb

ODE solvers

The Lorenz 63 "butterfly" plot

(https://en.wikipedia.org/wiki/Lorenz_system)

Julia programming language cont…

Polyarea package

- Using polyarea we can show how Julia uses types and multiple-dispatch to create the same sort of capabilities as object-oriented code. Julia does not have classes.

- Julia packages have a standard basic layout

```
MyPoly
MyPoly/Project.toml
MyPoly/src
MyPoly/src/MyPoly.jl
```

```
# Julia package manager can be used to create template for our
# own package.
# We will use polgon area as an example.
#
# Start by creating a template for Package called MyPoly
#
using Pkg
Pkg.generate("MyPoly")
cd("MyPoly")
Pkg.activate(".")
import MyPoly
MyPoly.greet()
~
```

```
$ cat MyPoly/src/MyPoly.jl
module MyPoly

greet() = print("Hello World!")

end # module
```

Julia programming language cont…

Polyarea packge

- Julia uses struct and type concepts
  - struct creates a type with same name as struct.
  - Here MyPoint is a type
  - The :: syntax can be used with program defined types
  - A constructor function with args matching struct members is created by default.

```
struct MyPoint
        x::Float64
        y::Float64
end
```

```
julia> p=MyPoint(1.,0.)
MyPoint(1.0, 0.0)

julia> typeof(p)
MyPoint

julia>
```

```
julia> p1=MyPoint(1.)
ERROR: MethodError: no method matching MyPoint(::Float64)
Closest candidates are:
  MyPoint(::Float64, ::Float64) at /Users/chrishill/projects/1
```

Julia programming language cont…

```
struct MyPoint
        x::Float64
        y::Float64
end
```

Polyarea package

- Julia uses struct and type concepts
  - struct creates a type with same name as struct.
  - Here `::MyPoint` is a custom type. Multiple-dispatch uses type matching in place of functions within classes.
  - Alternate constructor functions must be written

```
julia> p=MyPoint(1.,0.)
MyPoint(1.0, 0.0)

julia> p1=MyPoint(p)
MyPoint(1.0, 0.0)
```

```
function MyPoint(p::MyPoint)
        MyPoint(p.x, p.y)
end
```

```
julia> p1=MyPoint(1.)
ERROR: MethodError: no method matching MyPoint(::Float64)
Closest candidates are:
  MyPoint(::Float64, ::Float64) at /Users/chrishill/projects/1
```

Julia programming language cont…

Polyarea package

- Full "MyPoint" type and methods.
  - Overload "+" and "-"
  - A custom method

- Can now add this to the MyPoly package

```julia
"""
  Define a point type
"""
struct MyPoint
        x::Float64
        y::Float64
end

Base.:+(p1::MyPoint,p2::MyPoint) = MyPoint( p1.x+p2.x, p1.y+p2.y )

Base.:-(p1::MyPoint,p2::MyPoint) = MyPoint( p1.x-p2.x, p1.y-p2.y )

function MyPoint(p::MyPoint)
        MyPoint(p.x, p.y)
end
```

Julia programming language cont…

Polyarea package

- Adding to a package
- `include("xxxxx")`
- include starts search from directory where file with include lives
- typically, package component files are in same directory as the main package file

```
(base) chriss-MacBook-Pro:src chrishill$ ls
MyPoly.jl       point.jl        poly.jl
(base) chriss-MacBook-Pro:src chrishill$ cat MyPoly.jl
module MyPoly

include("point.jl")
include("poly.jl")


end # module
(base) chriss-MacBook-Pro:src chrishill$
```

Julia programming language cont…

Polyarea package

- `poly.jl` defines polynomial functions
  - Add points
  - Compute areas
- it is included after `point.jl` so that is can reference `MyPoint` types and methods.
  - e.g. Julia type hierarchy is "acyclic".

```
(base) chriss-MacBook-Pro:src chrishill$ ls
MyPoly.jl       point.jl        poly.jl
(base) chriss-MacBook-Pro:src chrishill$ cat MyPoly.jl
module MyPoly

include("point.jl")
include("poly.jl")


end # module
(base) chriss-MacBook-Pro:src chrishill$
```

Julia programming language cont…

Polyarea package

- poly.jl
  - needs
    - type
    - functions

```
struct MyPolyVar <: AbstractPolygon
        plist::Vector{MyPoint}
        nmax::Array{Int,1}
        ncur::Array{Int,1}
end
```

```
function area(poly::MyPolyVar)
        ta=0.
        p0=poly.plist[1]
        for i=3:poly.ncur[1]
                v1=poly.plist[i-1]-p0
                v2=poly.plist[i  ]-p0
                ta=ta+(v1.x*v2.y-v1.y*v2.x)*0.5
        end
        return ta
end
```

Julia programming language cont…

Using Polyarea package

- Need to tell Julia where to find package

```
[ ]: # Change this to where you have MyPoly
     # cd("/home/jpy_class/mit/12.010/cnh@mit.edu/poly-te

[ ]: using Pkg
     Pkg.activate(".")

[ ]: using MyPoly

[ ]: p1=MyPoint(0.,0.)
     p2=MyPoint(1.,0.)
     p3=MyPoint(1.,1.)
     p4=MyPoint(0.,1.)
     p5=MyPoint(0.,0.)
     poly1=MyPolyVar([p1,p2,p3,p4,p5],10)
     a=area(poly1)
```

```
using MyPoly
p1=MyPoint(0,0)
p2=MyPoint(1,0)
p3=MyPoint(1,1)
p4=MyPoint(0,1)
p5=MyPoint(0,0)
poly1=MyPolyVar([p1,p2,p3,p4,p5],10)
a=area(poly1)
println(a)
```

Hands on

- Try out [`mypoly-julia.ipynb`]
- `uses`