Julia in VS Code #3
Differential Equations in Julia

Chris Griffiths & Eva Delmas

January 5, 2021

This document follows on from "Julia in VS Code #1 and #2" and assumes that you're still working in your active project.

This document illistrates how to construct and solve differential equations in Julia using the `DifferentialEquations.jl` package. In particular, we are interested in modelling a two species Lotka-Volterra like (predator-prey/consumer-resource) system. Such systems are fundamental in ecology and form the building blocks of complex networks and the models that represent them.

First, import your package manager, activate and instantiate (only neccessary if you've closed VS Code between tutorials):

```
import Pkg
Pkg.activate(".")
Pkg.instantiate()
```

Then install the `DifferentialEquations.jl` package and let Julia know you want to use it:

```
# install
Pkg.add("DifferentialEquations")
# use
using DifferentialEquations
```

As with other packages, the `DifferentialEquations.jl` package will now be automatically added to your active project. Remember, you can check which packages (and their versions) are active using `Pkg.status()`.

1 Differential Equations

Differential equations are frequently used to model the change in variables of interest through time. These changes per unit time are often referred to as derivatives (or `du/dt`). In this case, we are interested in modelling changes to the abundance of a predator and its prey as a function of the systems key processes (growth, ingestion and mortality) and its parameters. In the following code chunk we create a function for our model using a set of differential equations:

```
# Our function accepts the following:
```

```
    # du (derivatives) - a vector of du/dt (changes in abundance) values for each
species
    # u (initial values) - a vector of abundances for each species
    # p (parameters) - a list of parameter values
    # t (time) - timespan
function LV_model(du,u,p,t)
   # growth of the prey species (modelled as a logistic growth function)
   GrowthPy = p.growthrate * u[1] * (1 - u[1]/p.K)
   # rate of prey ingestion by predator (modelled as a type 1 functional response)
   IngestPd = p.ingestrate * u[1] * u[2]
   # mortality of predator (modelled as density independent)
   MortPd = p.mortrate * u[2]
   # calculate and store changes in abundance (du/dt):
   # change in prey abundance
   du[1] = GrowthPy - IngestPd
   # change in predator abundance
   du[2] = p.assimeff * IngestPd - MortPd
end
```

Here, prey abundance is modelled as a function of its population growth rate (constrained by the systems carrying capacity K) and its mortality rate due to predation. In comparison, predator abundance is modelled as a function of its ingestion rate, a fixed assimilation efficiency (assimeff) and its density independent mortality rate.

To run the model we first have to define the system's initial values, the parameters and the timespan of the simulation:

```
# initial values:
u_init = [1.0;1.0]
# both species start with an initial abundance of 1

# parameters:
p = (
    growthrate = 1.0, # growth rate of prey (per day)
    ingestrate = 0.2, # rate of ingestion (per day)
    mortrate = 0.2,   # mortality rate of predator (per day)
    assimeff = 0.5,   # assimilation efficiency
    K = 10            # carrying capacity of the system (mmol/m3)
    )
# Here, we have chosen to define p as a named tuple (similar to a list in R). A vector
or dictionary would also work, however, named tuples are advantageous because they allow
us to use explicit names and are unmutable meaning that once it's created you can't
change it.

# time:
tspan = [0.0,200.0] # simulate for a timespan of 0.0-200.0
```

To solve the differential equations, we first define the problem using `ODEProblem` function:

```
# define the problem:
prob = ODEProblem(LV_model, u_init, tspan, p)
# when defining the problem, we provide the model function (LV_model), the initial
abundance values (u_init), the timespan (tspan) and the parameters (p)
```

And then solve the problem using `solve`:

```
# solve the problem:
sol = solve(prob)
# this might take a few mintues to compile
```

Here we have chosen to use the default algorithm because it's a simple problem, however there are several available - see here for more information. These two final steps (define and solve the problem) are analogous to using the `deSolve` package in R.

2 Plot

Once the problem has been solved, the results can be explored and plotted. In fact, the `DifferentialEquations.jl` package has its own built in plotting recipe that provides a very fast and conventient way of visualing the abundance of the two species through time:

```
# explore the data:
sol.u
# abundance through time - prey abundance is stored in column 1 and predator abundance
in column 2

# plot
plot(sol, ylabel = "Abundance", xlabel = "Time", title = "Lotka-Volterra", label =
["prey" "predator"], linestyle = [:dash :dot], lw = 2)
# the plotting recipe is defined explicity for differential equation solutions, so you
can directly pass the solution to the plot
```

You could also plot the data manually using `Plots.jl` or `Gadfly.jl`, manipulate it or store it in your project directory. For a recap on plotting, manipulation and visualation head back to "Julia in VS Code #2".