Differential Equations in Julia

by Chris Griffiths, Eva Delmas and Andrew Beckerman, Dec. 2020.

This tutorial is adapted from an R script provided by Andrew Beckerman.

This document follows on from 'Getting started' and 'Basic Julia commands' and assumes that you're still working from your active project.

This document illustrates 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.

Load packages

For this tutorial you'll need the following two packages:

- `DifferentialEquations.jl` to solve the differential equations (same 'engine' as the one used by the `BioEnergeticFoodWebs.jl` model)
- `Plots.jl` to visualise the results

```julia
using DifferentialEquations, Plots
```

Differential equations

Differential equations are frequently used to model the change in variables of interest through time. These changes are often referred to as derivatives (or `du/dt`). In this case, we are interested in modelling changes in the abundance of a consumer and its resource as a function of the system's key processes (growth, ingestion and mortality) and its parameters.

This type of model can be formalised as a simple Lotka-Volterra predator prey model, consisting of a set of differential equations:

- Resource dynamics: $\frac{dR}{dt} = rR(1 - \frac{R}{K}) - \alpha RC$
- Consumer dynamics: $\frac{dC}{dt} = e\alpha RC - mC$

where R and C are the abundances of the resource and consumer respectively, r is the resource's growth rate, K is the system's carrying capacity, α is the consumer's ingestion rate, e is the assimilation efficiency and m is the consumer's mortality rate.

There are 3 major steps involved in constructing and solving this model in Julia:

1. Define a function for your model (i.e., transform the above differential equations into a function that can be read by the solver). This function tells the solver how the variables of interest (here R and C) change over time.
2. Define the problem. Here, the problem is defined by the function, the parameters (r, α, e and m), the initial conditions and the timespan of the simulation. In this step you provide the solver with all the details it needs to find the solution.
3. Solve!

Step 1. Define the function

Here we construct a function for our model. The function needs to accept the following:

- `du` (derivatives) - a vector of changes in abundance for each species
- `u` (values) - a vector of abundance for each species
- `p` (parameters) - a list of parameter values
- `t` (time) - timespan

```
LV_model (generic function with 1 method)
```

```julia
function LV_model(du,u,p,t)
    # growth rate of the resource (modelled as a logistic growth function)
    GrowthR = p.growthrate * u[1] * (1 - u[1]/p.K)
    # rate of resource ingestion by consumer (modelled as a type I functional response)
    IngestC = p.ingestrate * u[1] * u[2]
    # mortality of consumer (modelled as density independent)
    MortC = p.mortrate * u[2]
    # calculate and store changes in abundance (du/dt):
    # change in resource abundance
    du[1] = GrowthR - IngestC
    # change in consumer abundance
    du[2] = p.assimeff * IngestC - MortC
end
```

You'll notice that in the above function (`LV_model`), we've specified specific parameters using the `p.name` notation. This is because we've opted to store our parameters in a named tuple called `p`. `p` is created below but it's worth noting that when this notation is used e.g., `p.growthrate`, we are telling Julia that we want to use the value of `growthrate` that is stored as a named part of our tuple `p`.

Step 2. Define the problem

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

- **Parameters**:

```
p =  (growthrate = 1.0, ingestrate = 0.2, mortrate = 0.2, assimeff = 0.5, K = 10)
```

```
p = (
    growthrate = 1.0, # growth rate of resource (per day)
    ingestrate = 0.2, # rate of ingestion (per day)
    mortrate = 0.2,   # mortality rate of consumer (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.

- **Initial values:**

For simplicity, we start with $R = C = 1$:

```
u0 =  Float64[1.0, 1.0]
```

```
u0 = [1.0; 1.0] # this needs to be an array
```

- **Timespan:**

```
tspan =  (0.0, 100.0)
```

```
tspan = (0.0,100.0) # you have to use a Pair (tuple with 2 values) of floating point numbers.
```

We then formally define the problem by passing the function (`LV_model`), the parameters (listed in our named tuple `p`), the initial values (`u0`) and the timespan (`tspan`) to `ODEProblem()`:

```
prob =
 [36mODEProblem [0m with uType  [36mArray{Float64,1} [0m and tType  [36mFloat64 [0m. In-plac
timespan: (0.0, 100.0)
u0: [1.0, 1.0]
```

```
prob = ODEProblem(LV_model, u0, tspan, p)
```

Step 3. Solve

To solve the problem, we pass the `ODEProblem` object to the solver.

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.

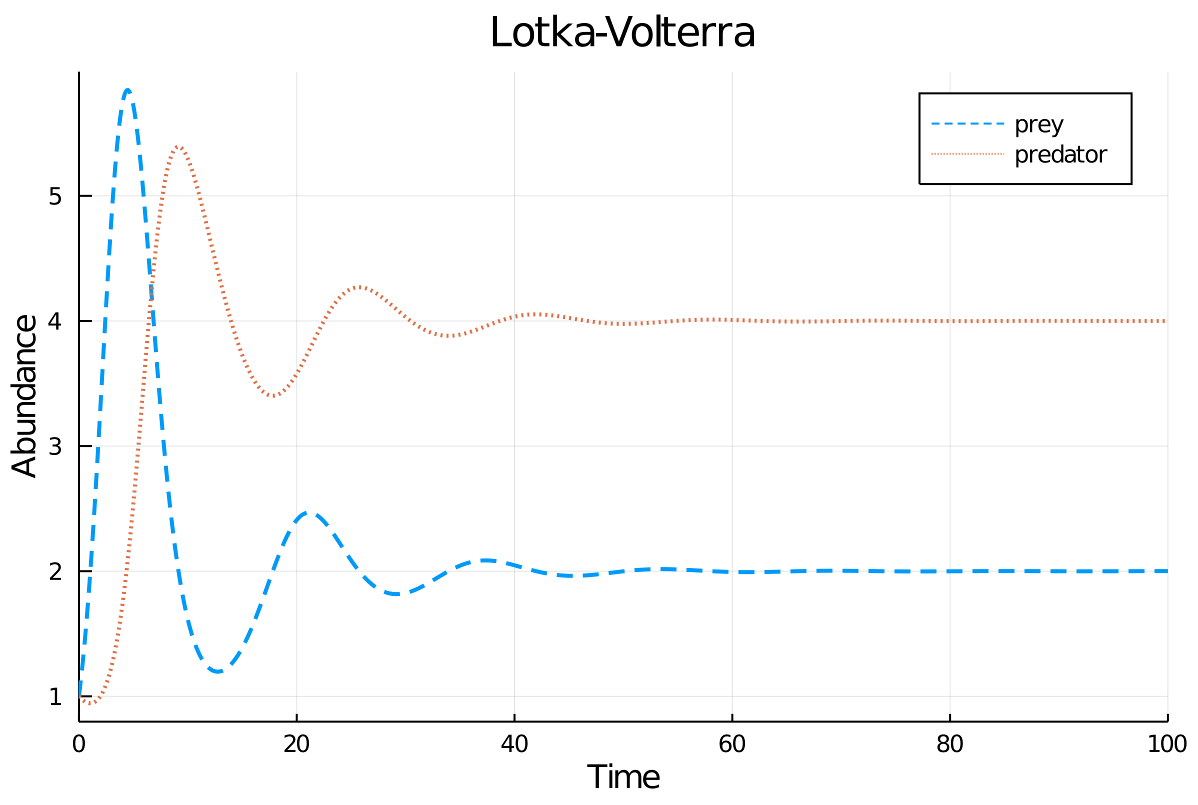
```
sol = retcode: Success
      Interpolation: Automatic order switching interpolation
      t: 34-element Array{Float64,1}:
         0.0
         0.11489280028272647
         0.47715886964074683
         1.0092730193307549
         1.666689525965059
         2.517906233438338
         3.500567862257175
         ⋮
        65.95385506069438
        72.13259957299113
        77.91907620021307
        84.66781244131667
        92.12234712150678
       100.0
      u: 34-element Array{Array{Float64,1},1}:
       [1.0, 1.0]
       [1.0833733276531547, 0.9890444361121592]
       [1.3870600469512069, 0.9618331065455541]
       [1.954002372525462, 0.9445018278455538]
       [2.8446262091296597, 0.9686331585805105]
       [4.156352860679624, 1.1003802884667164]
       [5.398892476255155, 1.452781526608154]
        ⋮
```

```
sol = solve(prob)
```

The solver produces 2 objects: `sol.t` and `sol.u` that respectivley store the time steps and the variables of interest through time. Let's have a look.

Visualise the outputs

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:



```julia
plot(sol,
    ylabel = "Abundance",
    xlabel = "Time",
    title = "Lotka-Volterra",
    label = ["prey" "predator"],
    linestyle = [:dash :dot],
    lw = 2)
```

One thing to note here, when plotting in Julia you don't need to seperate label names (`label = [prey predator]`) or linestyles (`linestyle = [:dash :dot]`) with a comma as you would in R. This will also be case for most plotting options in Julia.

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