Hands on Pluto : dynamical systems with reactive Julia notebooks

Ludovic Mailleret

Hands on `Pluto`

Dynamical systems with reactive `Julia` notebooks

Ludovic Mailleret, M2P2-ISA (INRAE, CNRS, UCA) & Biocore Inria; ludovic.mailleret@inrae.fr

What is `Pluto`?

Pluto is a notebook solution for Julia

- You get cells to put code or text/LaTeX notes

- Code is executed and rendered within the environment (data/plots exports are possible)

- Nice for exploring models, sandbox, code sharing, supplementary materials...
    - or making interactive presentations like the present one (we are actually in a `Pluto` notebook)

- `Pluto` was developed for the free MIT course Introduction to Computational Thinking (which is very nice!)

- `Pluto` is written in `Julia`, for `Julia` coding

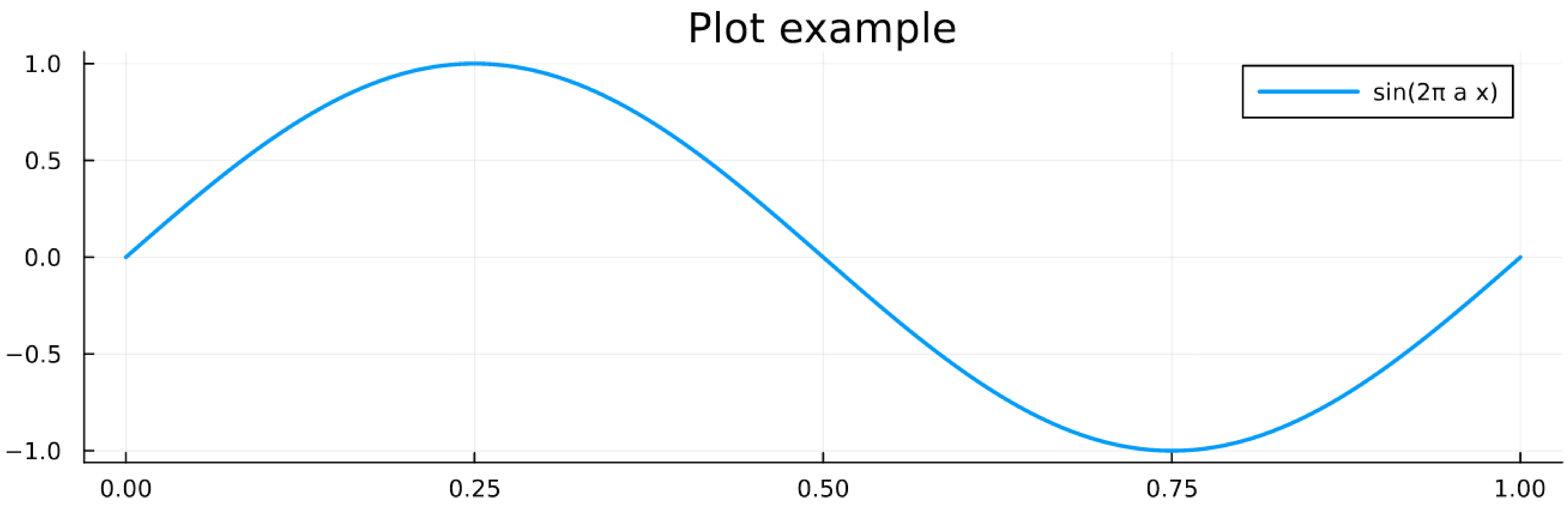
- `Pluto` is *reactive*

Pluto is a reactive Julia notebook environment

- Each cell is **always executed** in the `workspace` or `scope`
- Therefore **dependent cells react to changes**

```
k = 1 ;
```

```
ySin = sin.(2π * k * x) ;
```




```
plot(x, ySin,
    linewidth = 2,
    label = "sin(2π a x)",
    title = "Plot example",
    size =(800, 250) )
```

Reactivity is handy

```
num_cats = 2;
```

- **I have 2 cats**

```
md"- **I have $num_cats cats**"
```

- But of course **you can't do anything** ($\neq$ e.g. Jupyter, or classical scripting)

```
foo = 2;
```

```
# foo = 4;
```

- Multi lines of code should be put in: `begin ... end` blocks (variable update possible in blocks)

```
(-1.0, 0.0)
```

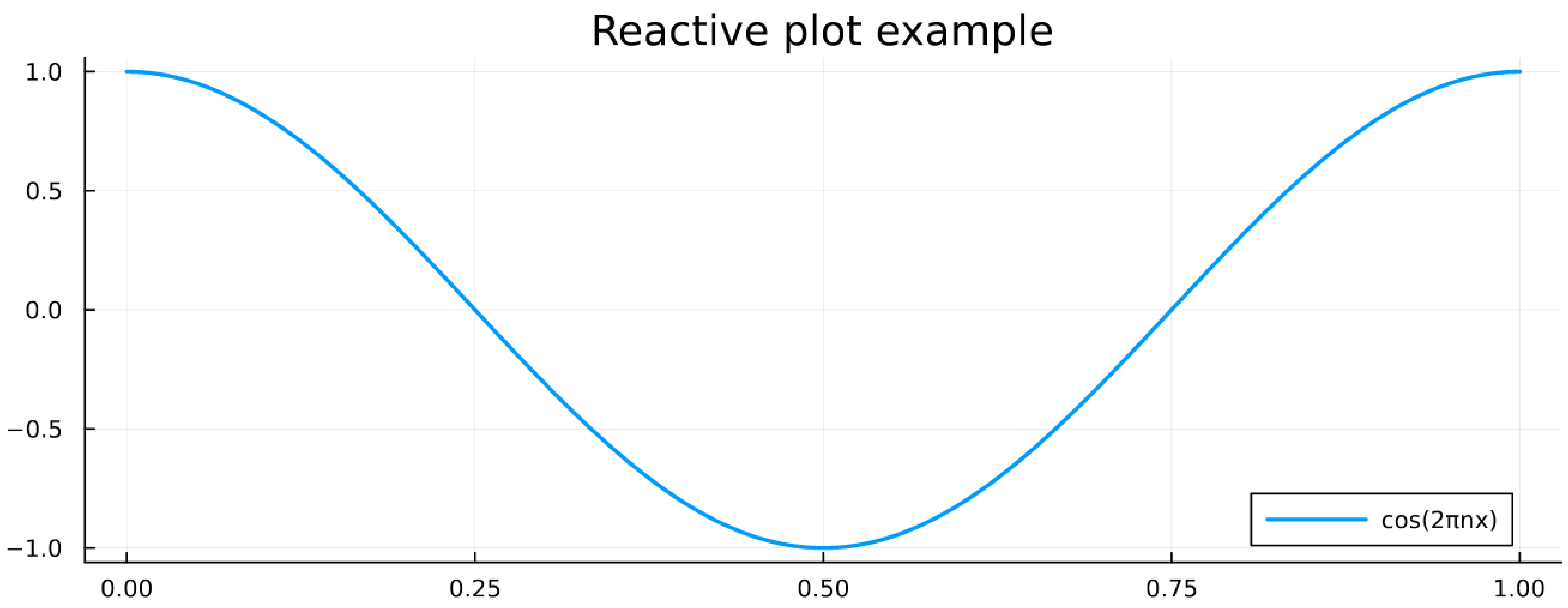
```
begin
    pi = π
    cos(pi), sin(pi)
end
```

- but not for function definition, or `if`, `for`, `while` blocks (variable update possible in blocks)

Reactivity is handy

n = 1

```
yCos = cos.(2π * n * x);
```




```
plot(x, yCos,
    linewidth = 2,
    label = "cos(2πnx)",
    title = "Reactive plot example",
    size=(800, 300))
```

Simulating differential equations

Simulating ODEs with `DifferentialEquations.jl`

Consider the predator-prey model attributed to Rosenzweig & MacArthur (1963) (see Turchin (2003), Smith (2008)).

$$\begin{cases} \dot{x} = rx\left(1 - \dfrac{x}{K}\right) - c\dfrac{x}{h+x}y \\ \dot{y} = b\dfrac{x}{h+x}y - my \end{cases}$$

```julia
using DifferentialEquations, StaticArrays
```

- `DifferentialEquations.jl` provides numerical solvers (and more)
- `StaticArrays.jl` allows use of statically sized arrays in memory that speed up integration

- **Model definition**

```julia
function model_rma(u, params, t)
    r, K, c, h, b, m = params        # unpacking
    x = u[1]                         # unpacking
    y = u[2]

    dx = r*x*(1-x/K) - c*x/(h+x)*y  # model equations
    dy = b*x/(h+x)*y - m*y

    @SVector [dx, dy]                # return derivatives as static arrays
end;
```

Initial conditions, parameters & time

- Initial conditions

```
begin
    x0 = 1.0
    y0 = 2.5
    etat0 = @SVector [x0, y0]        # packing in a Static Array
end;
```

- Parameters

```
begin
    r = 1.0
    K = 10.0
    c = 1.0
    # h = 2.0 is actually defined later through a Slider
    b = 2.0
    # m = 1.0 is actually defined later through a Slider

    params_rma = [r, K, c, h, b, m]      # packing
end;
```

- Integration time

Numerical integration

- Define the Cauchy problem

```
prob_rma = ODEProblem with uType SVector{2, Float64} and tType Float64. In-place: false
           timespan: (0.0, 80.0)
           u0: 2-element SVector{2, Float64} with indices SOneTo(2):
            1.0
            2.5
```

```
prob_rma = ODEProblem(model_rma, etat0, tspan, params_rma, saveat = step)
```

- Integrate

```
sim_rma = solve(prob_rma, abstol=1e-6, reltol=1e-6);
```

- Rearrange the simulation in a `dataframe`, rename data (optional)

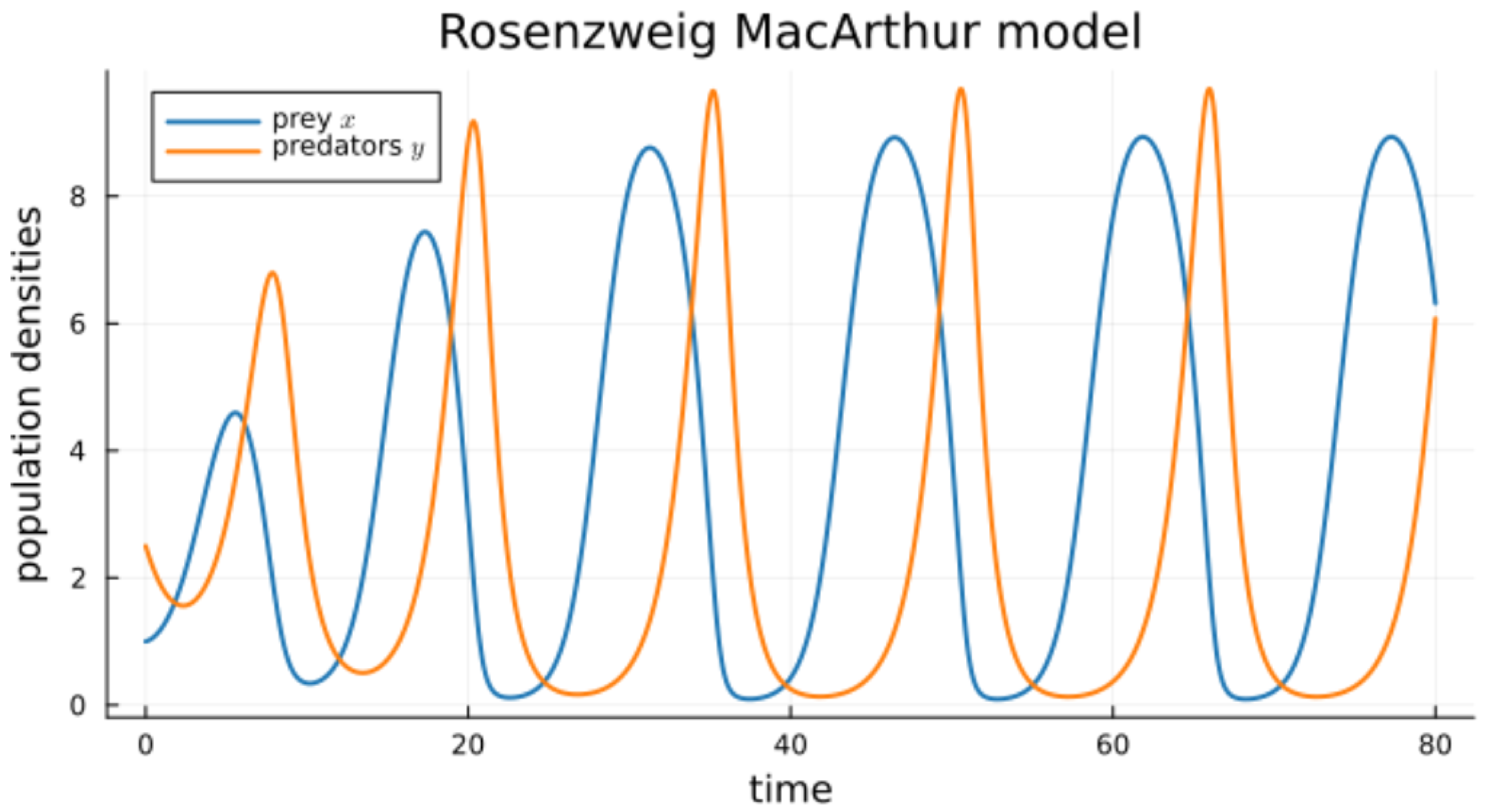
```
begin
    sol_rma = DataFrame(sim_rma)
    rename!(sol_rma, :timestamp => :time, :value1 => :x, :value2 => :y)
end;
```

Numerical integration

- you get the simulated solution along time, every 0.01 timesteps

	time	x	y
1	0.0	1.0	2.5
2	0.01	1.00068	2.49168
3	0.02	1.00139	2.4834
4	0.03	1.00213	2.47516
5	0.04	1.0029	2.46695
6	0.05	1.0037	2.45878
7	0.06	1.00453	2.45064
8	0.07	1.00539	2.44254
9	0.08	1.00627	2.43447
10	0.09	1.00719	2.42644
⋮ more			
8001	80.0	6.31688	6.07748

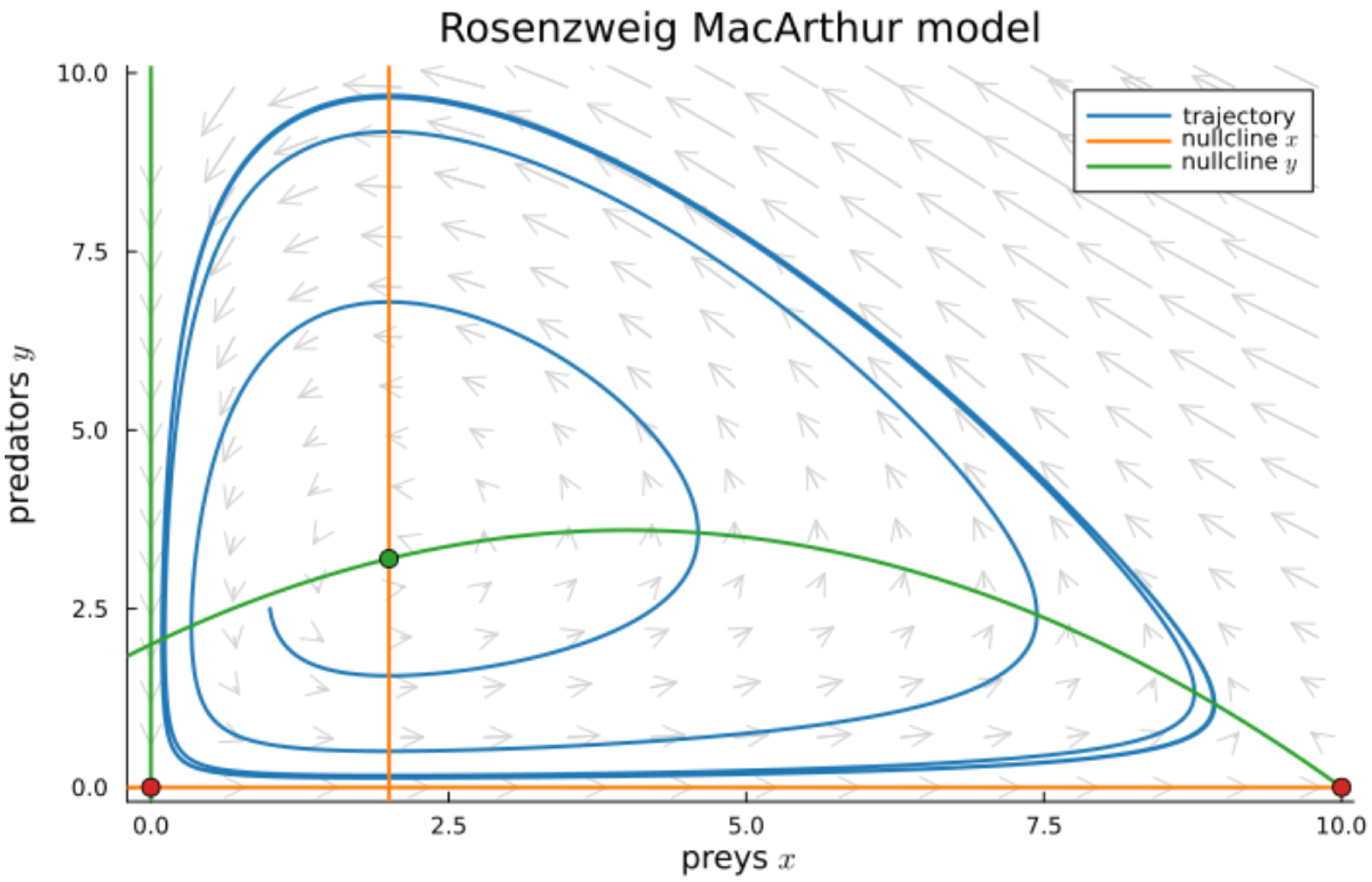
Plotting against time




```
plot(sol_rma.time, [sol_rma.x sol_rma.y],
    palette = :tab10,
    linewidth = 2,
    title = "Rosenzweig MacArthur model",
    label = ["prey " * L"x" "predators " * L"y"],  # latex strings, mardown latex is off in labels
    ylabel = "population densities",
    xlabel = "time",
    size = (650,350))
```

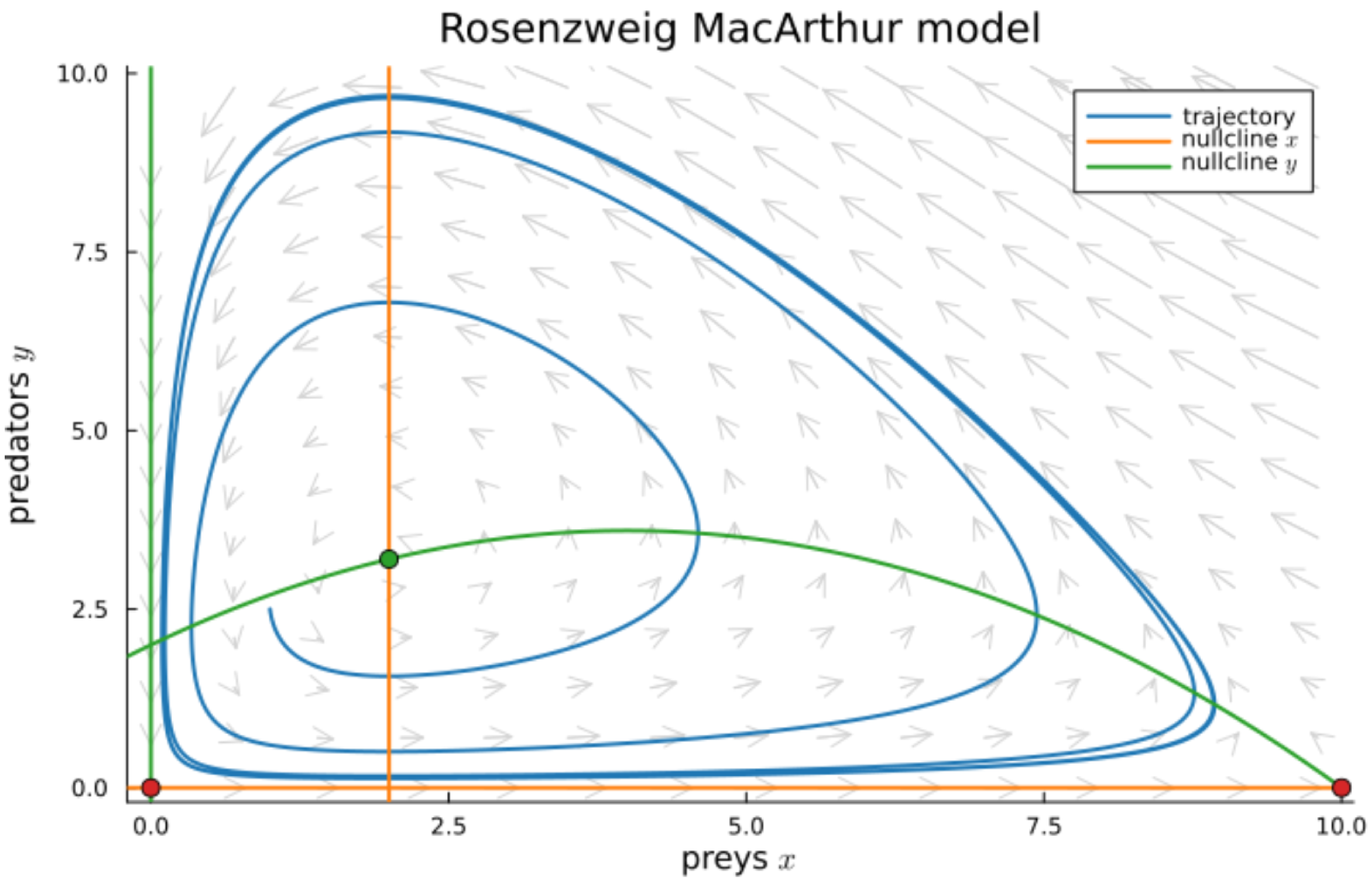
Plotting in state space

- One can have nice state space plots, but code is longer



Playing with plots in the state space

m = 1.0



Bifurcation diagram

Bifurcation diagram in function of K

- Model undergoes transcritical and Hopf bifurcations as K increases
  - analytics below Hopf bifurcation
  - **numerics** for asymptotics above Hopf bifurcation

- For a given K, simulate for a *long time* to remove transients
- From this, start a new simulation and get the `min` and `max` of the limit cycle

- K loop, and equilibria

```
begin
   K_step = 0.1

   # before transcritical
   K_plot1 = 0:K_step:m*h/(b-m)
   y_eq01 = ones(length(K_plot1)).*0

   # between transcritical and Hopf
   K_plot2 = m*h/(b-m):K_step:h+2*m*h/(b-m)
   y_eq02 = ones(length(K_plot2)).*0
   y_co2 = [r/c*(h+m*h/(b-m))*(1-m*h/(b-m)/K_p) for K_p in K_plot2]  # may have broadcasted

   # above Hopf
   K_plot3 = h+2*m*h/(b-m)-K_step/5:(K_step/10):8
   y_eq03 = ones(length(K_plot3)).*0
   y_co3 = [r/c*(h+m*h/(b-m))*(1-m*h/(b-m)/K_p) for K_p in K_plot3]; # may have broadcasted
end;
```

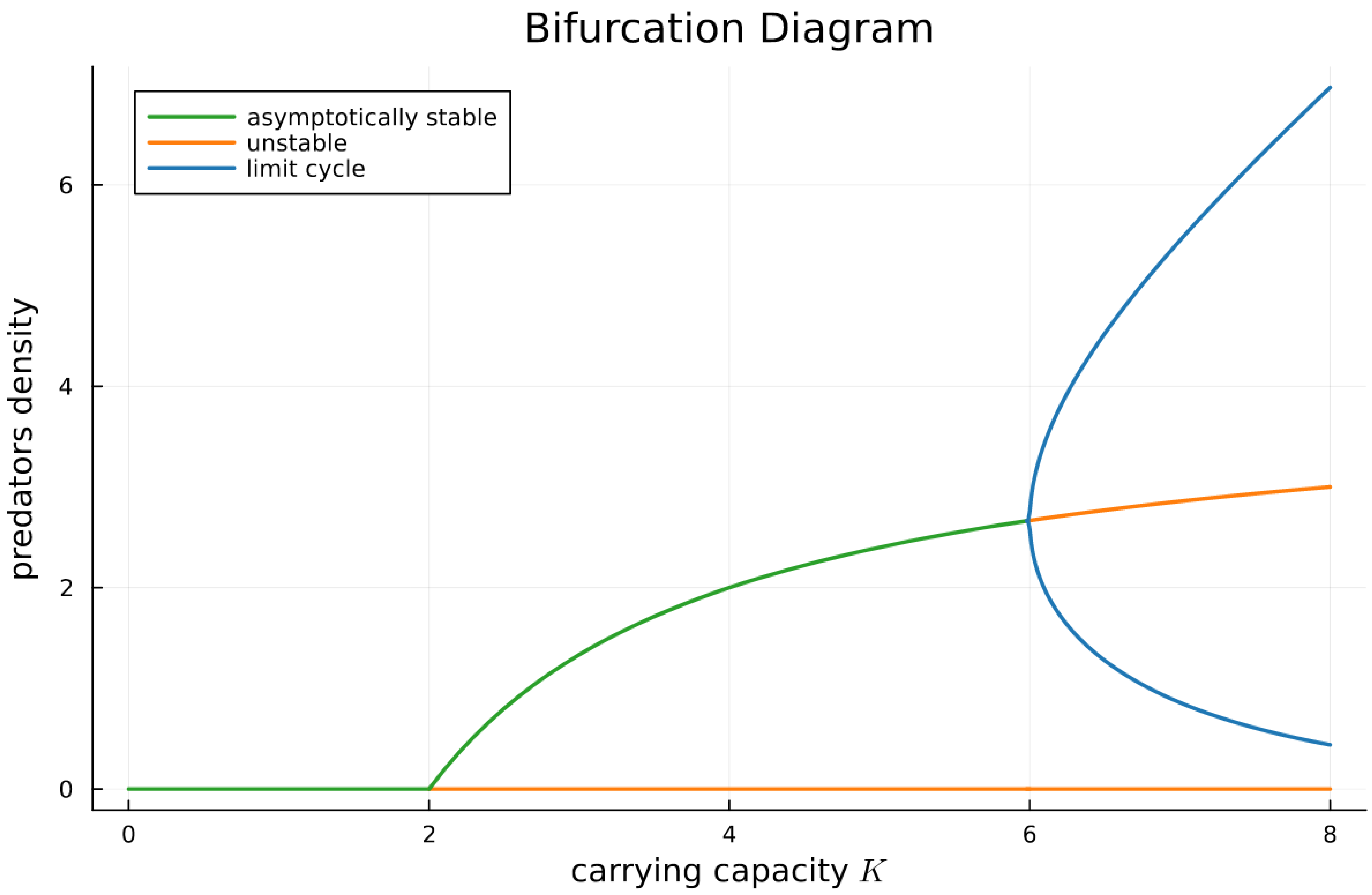
Bifurcation diagram in function of K

- Simulate transients, restart from there, and get extrema

```
begin
    # for storage
    i = 1
    y_cmin = zero(K_plot3)
    y_cmax = zero(K_plot3)

    # estimate limit cycle through loop on K
    for K_c in K_plot3                                  # loop on K values
        params_rma_cycle = [r, K_c, c, h, b, m]         # set parameters

        # transient initial value problem; simulation
        rma_trans_pbe =  ODEProblem(model_rma, etat0, t_trans, params_rma_cycle)
        post_trans2 = solve(rma_trans_pbe, save_everystep = false, save_start = false,
    abstol=1e-6, reltol=1e-6)

        # limit cycle initial value problem; simulation
        rma_cycle_pbe =  ODEProblem(model_rma, post_trans2[:,1], tspan, params_rma_cycle, saveat =
    step)
        sol_cycle = solve(rma_cycle_pbe, abstol=1e-6, reltol=1e-6)

        # get the extrema
        y_cmin[i] = minimum(sol_cycle[2,:])             # pushing is probably bad programming here
        y_cmax[i] = maximum(sol_cycle[2,:])

        i+=1
    end
end
```

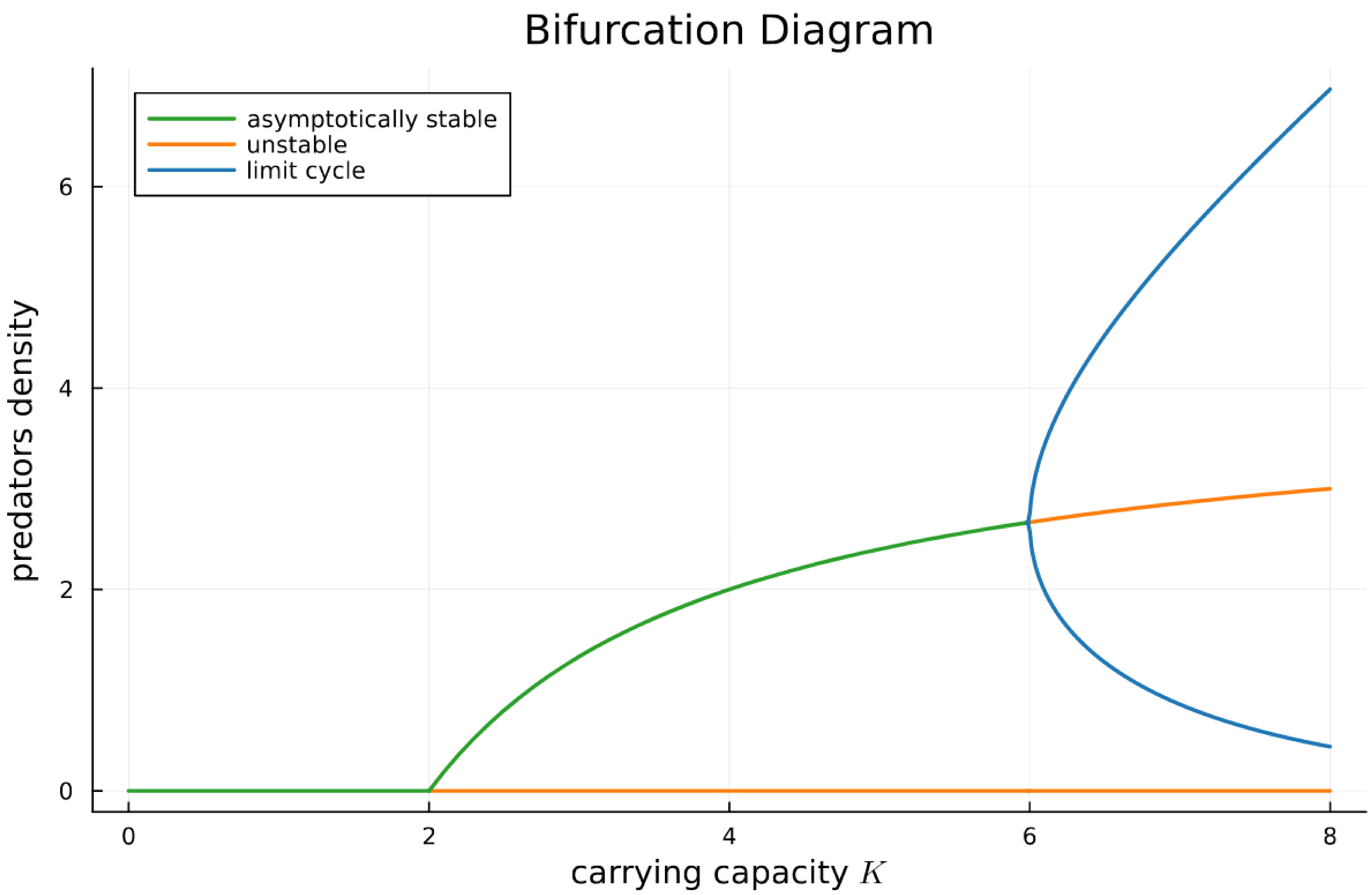
Bifurcation diagram in function of K

- After plotting everything



Playing with bifurcation diagrams

h = 2.0



Final words

Pros:

- Julia code is **easy** to learn and fun to write !
- Julia is a general purpose language, very good at scientific computing
- Julia is free software, community is growing
- (after pre-compilation) Julia is **incredibly fast** at simulating DE (and pretty much everything)
    - same bifurcation code in Python runs 2 order of magnitude slower (with my own programming skills)
- Pluto notebooks are reactive
    - reactivity is fun and useful
    - **WYSIWIG** programming : order of cell execution does not matter ($\neq$ Jupyter, scripting/ `ctrl+return`)
    - Pluto notebooks are plain Julia (text) files

Cons:

- Julia is still confidential (no colleague of mine works with it at this moment)
- *Time to first* ... can be frustrating, especially for newcomers (and sometimes *first* never comes for some reason)
- Code may need regular maintenance (present code is only 1-year old, and needed revisiting for properly running today on new `Julia` and library versions)