DynamicalSystems.jl - A software for the exploration of chaos and nonlinear dynamics

George Datseris[1,2]

[1]Max Planck Institute for Dynamics and Self-Organization
[2]Georg-August-Universität Göttingen

June 25, 2018

George Datseris is a PhD student (candidate), supervised by Prof. Theo Geisel and Dr. Ragnar Fleischmann. He is the lead developer of ***DynamicalSystems.jl*** *and the entire JuliaDynamics GitHub organization. The development of DynamicalSystems.jl was done by George Datseris independently and without any relation to the PhD Thesis supervisors. Contributors to the software are acknowledged properly in the appropriate section. Development of* ***DynamicalSystems.jl*** *started on May 28, 2017.*

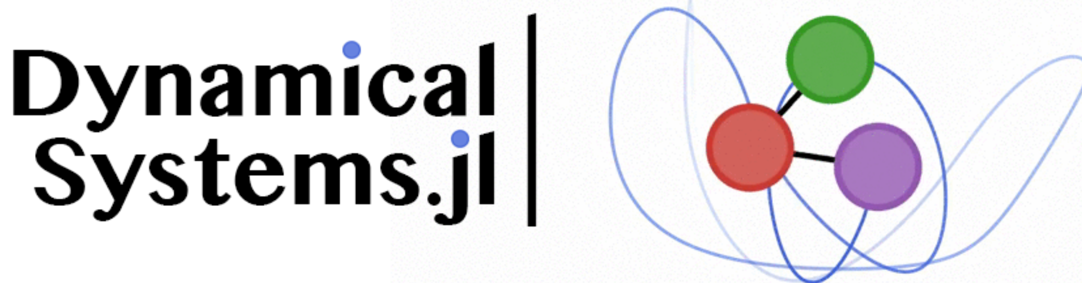

Figure 1: Logo of **DynamicalSystems.jl**.

1 Introduction & Goals

DynamicalSystems.jl [3, 2] is a software *library* for the exploration of chaos and nonlinear dynamics. It is written entirely in Julia [1], a new programming language with large potential. From a technical viewpoint, **DynamicalSystems.jl** is composed of (currently) four *packages* for the Julia language and is not an independent application.

The ultimate goal for **DynamicalSystems.jl** is to be a useful library for students and scientists working on chaos, nonlinear dynamics and in general dynamical systems. We do not want to have "just code", but also detailed descriptions and references for as many methods as possible. In addition, we strive for conciseness, transparency, performance, accuracy and reliability.

2 Full Content List

Unless specifically stated otherwise, everything described in this section works for both maps and ODEs.

1. General & flexible dynamical system definition. Defining any system typically requires 10 lines of code (see example of sec. 6.1).
2. ODEs are solved using the package DifferentialEquations.jl [7], which has vastly more features than other competing ODE solvers, see [6]. Users also have full control over all solver parameters, should they wish to.
3. Automatic computation of the Jacobian of the equations of motion through automatic differentiation. Users can still provide a hand-coded Jacobian function for increased efficiency.
4. Dedicated interface for datasets, which allows efficient computation of quantities like entropies.
5. Delay coordinates embedding, including multiple time and multiple time series embedding.
6. Poincaré surface of sections of continuous systems (for arbitrary planes).
7. Orbit diagrams (also called bifurcation diagrams) of maps.
8. Automated production of orbit diagrams for continuous systems through successive Poincaré sections.
9. Maximum Lyapunov exponent.
10. Spectrum of Lyapunov exponents.
11. Generalized (Renyi) entropy and very efficient computation method (that does *not* scale exponentially with the dimension of the dataset).
12. Permutation entropy.
13. Generalized dimensions and automated procedure of deducing them from a given curve.
14. Neighborhood estimation of points in a dataset.
15. Numerical (maximum) Lyapunov exponent of a time series/dataset.
16. Finding fixed points of any order (stable & unstable) for a map.
17. Detecting and distinguishing chaotic and regular behavior using the GALI method.
18. time series prediction using local modelling.
19. Spatio-temporal time series prediction using local modelling.
20. Spatio-temporal time series cross-prediction.

It is important to understand that since **DynamicalSystems.jl** is a Julia package, it has no GUI, dedicated windows, etc.. All functionality is provided either though scripts, or by running commands interactively in one of the many Julia consoles (REPL/Jupyter/Atom). In addition, we do not offer explicit plotting functionality. Instead, users can decide on their own how to visualize their data. In our documentation examples we chose to use the `PyPlot` (`matplotlib`) package for visualization.

2.1 Recommended Use

DynamicalSystems.jl has obvious applications in research. Most of the algorithms we offer are routinely used in research articles about complex systems, like for example the algorithm that computes Lyapunov exponents.

In addition, we believe that the library has great applications in education. Because the syntax is intuitive and consice, scripts that produce informative demonstrations are short and understandable. In addition, because the source code is clear and small, students can simply read it and understand how to e.g. apply and modify an algorithm for specialized applications. Lastly, we always cite relevant scientific articles that introduce and use the algorithm that we use in the source code, which means that any user that wants to learn more about the algorithm only needs to follow the citation links.

3 Highlights

In a published paper about **DynamicalSystems.jl** [3] we compare in detail our software with some well-established softwares of the field (`TSTOOL`, `E&F Chaos` and `LP-VIcode`) and point out advantages and disadvantages. In this section we summarize what we believe are the strong points of **DynamicalSystems.jl** that can make it stand out among other similar software.

- Incorporated within a large ecosystem composed of Julia, Python, R, and more. Because **DynamicalSystems.jl** is not a so-called standalone application but instead a Julia package, its input and output can be directly integrated with using other packages, for a very efficient work-flow.
  - As a trivial example, using statistical tools, random number distributions, etc. can be done on the same script, simply by adding appropriate statements like `using StatsBase`.
- Open sourced, with a free (MIT) license.
- Clear, small and easy to understand source code. **DynamicalSystems.jl** was created from the ground up with the clarity of the source code as one of the main concerns.
- If the Jacobian of the equations of motion is not provided by the user, it is computed automatically using highly-efficient automatic differentiation.
- Concise, intuitive and general: all functions work just as well with any defined dynamical system, whether it is a map or an ODE.
- Big collection of different algorithms. In addition, the high-level abstract design also allows **DynamicalSystems.jl** to be easily extendable.
- Actively maintained and constantly growing. New releases are typically out every month.
- Hosted on GitHub, making interaction of users and developers easy and straightforward.

In section 6.1 we also show some typical examples of using **DynamicalSystems.jl**.

4 Documentation

Documentation of our software can be accessed in numerous ways. The most prominent one is the official documentation page [2]. There, detailed tutorials and examples demonstrate how to use each and every aspect of **DynamicalSystems.jl**.

The second way of learning how to use the software is through the built-in “documentation string” functionality of Julia. Simply typing `?` followed by a function name displays the documentation string of the function. By convention all documentation strings of **DynamicalSystems.jl** have the following structure:

1. Function call signature.
2. Listing and description of arguments and keyword arguments.
3. Short, but informative description of the algorithm implementation.
4. References to research articles that introduced, or are highly relevant for, the algorithm.

The third and final way to lean how to use **DynamicalSystems.jl** is to either watch the video tutorial hosted on the official Julia language YouTube channel[1] or to use one of the many interactive Jupyter notebook tutorials available on GitHub[2].

[1] `https://www.youtube.com/watch?v=13hqE_1a158`, accessed on June 25, 2018
[2] `https://github.com/JuliaDynamics/tutorials`, accessed on June 25, 2018

5 Installation

Provided that you have already installed the Julia language (version 0.6)[3] you only need to run the following commands to install and use our library:

```
Pkg.add("DynamicalSystems")

using DynamicalSystems
```

which will automatically install all the dependencies (i.e. other Julia packages) necessary. The same process will work on *any* operating system.

5.1 Version

We note that **DynamicalSystems.jl** has not reached version `1.0` yet. This however wrongly points to instability. The public API and internals are both in a very stable state. We are simply waiting for version `1.0` of the Julia language to launch version `1.0` for **DynamicalSystems.jl** as well. We expect this to happen around September 2018.

6 Examples

6.1 Defining a dynamical system & computing Lyapunov exponents and Alignment Indices

DynamicalSystems.jl takes big advantage of Julia's multiple dispatch. A user first defines a central structure, a `DynamicalSystem` instance, which is then passed to any necessary function. This heavy abstraction allows the same code (from the user's perspective) to be used for any type of system, continuous or discrete, big or small.

A `DynamicalSystem` is constructed simply using the equations of motion of the system, an initial state and a parameter container[4]. Here is an example of defining the Hénon map [5]:

```
# Define equations of motion:
h_eom(x, p, t) = SVector{2}(1.0 - p[1]*x[1]^2 + x[2], p[2]*x[1])
# Define initial state:
state = zeros(2)
# Define parameter container:
p = [1.4, 0.3]

# Create DynamicalSystem instance:
henon = DiscreteDynamicalSystem(h_eom, state, p)
```

```
2-dimensional discrete dynamical system
 state:      [0.0, 0.0]
 e.o.m.:     h_eom
 in-place?   false
 jacobian:   ForwardDiff
```

Notice that the equations of motion are tied to the dynamical system instance. The parameters however, as well as the initial conditions, can be changed. We show this in one of the examples below.

The process is identical when defining a continuous system instead of a map. We also take the opportunity to show how one should define a "big" system:

[3] https://julialang.org/downloads/
[4] For more detailed examples please visit the official documentation page [2].

```julia
# Equations of motion of Henon-Helies system
function hheom!(du, u, p, t)
    du[1] = u[3]
    du[2] = u[4]
    du[3] = -u[1] - 2u[1]*u[2]
    du[4] = -u[2] - (u[1]^2 - u[2]^2)
    return nothing
end

# pass `nothing` as the parameters, because the system doesn't have any
hh = ContinuousDynamicalSystem(hheom!, [0, -0.25, 0.42081, 0], nothing)
```

```
4-dimensional continuous dynamical system
 state:      [0.0, -0.25, 0.42081, 0.0]
 e.o.m.:     hheom!
 in-place?   true
 jacobian:   ForwardDiff
```

These systems can now be passed to any function that expects a `DynamicalSystem`, for example the `trajectory` function which computes a trajectory of the system.

```julia
trajectory(hh, 100.0)
```

```
4-dimensional Dataset{Float64} with 10001 points
 0.0          -0.25       0.42081    0.0
 0.00420806   -0.249984   0.420799   0.00312486
 0.00841592   -0.249938   0.420768   0.0062489
 ⋮
 0.0832885    -0.140689   0.373744   0.289272
 0.0870229    -0.137788   0.37313    0.290786
 0.090751     -0.134873   0.372483   0.292256
```

Similarly, to compute the Lyapunov spectrum of the system we use the function `lyapunovs`:

```julia
lyapunovs(hh, 5000) # second argument is renormalization number
```

```
4-element Array{Float64,1}:
  0.0405437
  0.000994128
 -0.00104266
 -0.0404951
```

```julia
lyapunovs(henon, 5000)
```

```
2-element Array{Float64,1}:
  0.424127
 -1.6281
```

or, to compute the GALI [8] quantity and associated time vector, we use `gali`:

```julia
g, t = gali(hh, 1000.0, 4) # last argument is GALI order
g[end], t[end]
```

```
(4.772159111820687e-13, 106.13673833781026)
```

```
g, t = gali(henon, 1000, 2)
g[end], t[end]
```

```
(2.4076574067777296e-13, 14)
```

Using different parameter and/or initial condition is also straight-forward. The parameters are changed on the spot,

```
set_parameter!(henon, 2, 0.4) # set second parameter to 0.4
```

while, to use a different initial condition, you simply give it to one of the high-level functions as a keyword argument:

```
g, t = gali(henon, 1000, 2; u0 = rand(2))
g[end], t[end]
```

```
(6.245004513516506e-14, 19)
```

6.2 Attached animations

Along with this application we are attaching two animation files that highlight different aspects of our software:

- `gali_psos_henonhelies.mp4` and associated script (`.jl`). The animation shows Poincaré sections of the Hénon-Heiles system at different energies. At each energy GALI [8] is used to color-code each initial condition according to how "regular" it is (how much time does GALI need to decay).
- `roessler_Z_tspred.mp4` and associated script (`.jl`). The animation shows time series prediction of the z variable of the Roessler system. Notice that the z variable is significantly more difficult to predict than the other two, due to the fact that it is 0 for most of the time.

In the examples all plotting and animating was done through `matplotlib` and `ffmpeg`, neither of which are part of **DynamicalSystems.jl**.

6.3 Source Code Simplicity: Lyapunov spectrum

In the highlights section (3) we made claims about the simplicity of our source code. To prove the point, we are presenting here the entire source code that computes the spectrum of Lyapunov exponents:

```
function lyapunovs(ds::DS{IIP, S, D}, N, Q0; Ttr = 0,
    diff_eq_kwargs = DEFAULT_DIFFEQ_KWARGS, dt = 1) where {IIP, S, D}

    T = stateeltype(ds)
    # Create tangent integrator:
    if typeof(ds) <: DDS
        @assert typeof(Ttr) == Int
        integ = tangent_integrator(ds, Q0)
    else
        integ = tangent_integrator(ds, Q0;
                                   diff_eq_kwargs = diff_eq_kwargs)
    end
    k = size(Q0)[2]; @assert k > 1

    λ::Vector{T} = _lyapunovs(integ, N, dt, Ttr)
    return λ
end

function _lyapunovs(integ, N, dt::Real, Ttr::Real)

    T = stateeltype(integ)
    t0 = integ.t
    if Ttr > 0
        while integ.t < t0 + Ttr
            step!(integ, dt)
            Q, R = qr(get_deviations(integ))
            set_deviations!(integ, Q)
        end
    end

    k = size(get_deviations(integ))[2]
    λ::Vector{T} = zeros(T, k)
    t0 = integ.t

    for i in 2:N
        step!(integ, dt)
        Q, R = qr(get_deviations(integ))
        for i in 1:k
            λ[i] += log(abs(R[i,i]))
        end
        set_deviations!(integ, Q)
    end
    λ ./= (integ.t - t0)
    return λ
end
```

This is the function that is called for both discrete and continuous systems.

The source is composed of 45 lines of code. The first `function` initializes an integrator object that evolves both the system and the tangent space at the same time. This happens on lines 8 or 10. This integrator object has different structure depending on what kind of system it is given. Its internal details however are not really

important for the `lyapunovs` function. One only needs to know how to "evolve" the integrator forwards in time (`step!(integ, dt)`), how to access the evolved deviation vectors (`get_deviations`) and how to set them to new values (`set_deviations!`).

The second function `_lyapunovs` performs all relevant computations and using a well-known algorithm to compute the Lyapunov spectrum (method "H2" of Ref. [4]). The code replicates the mathematical algorithm almost perfectly line-by-line. Please be aware that this simplicity and conciseness does not come at the cost of performance. Even though the exact same 45 lines of code are used for any system, Julia still manages to produce fast code:

```
using DynamicalSystems
henon = Systems.henon() # access pre-defined henon map

using BenchmarkTools # package that benchmarks functions
@btime lyapunovs($henon, 5000);
```

```
326.400 µs (115 allocations: 7.61 KiB)
```

The only difference between the Hénon map in the current case and the one in sec. 6.1 is that the Jacobian function is not auto-differentiated.

The result of the benchmark shows that the time required to compute the Lyapunov spectrum of the Henon map, using 5,000 QR-decomposition steps, is around 326 microseconds, on a laptop with Intel Core i5 @ 2.30 GHz processor and 8 GB of RAM running on Windows 10 (64-bit).

7 Acknowledgements

The authors would like to thank and acknowledge the following contributions

1. Jonas Isensee has implemented almost all methods related to the time series prediction functionality.
2. Chris Rackauckas is the lead developer of DifferentialEquations.jl [7]. We appreciate a lot of help with integrating the ODE solvers into our software and constant support.
3. Takafumi Arakaki has contributed the function that computes the permutation entropy and also offered helpful design comments.
4. Kristoffer Carlsson has implemented functionality for efficient nearest neighbor searches (independent of **DynamicalSystems.jl**) and has helped in integrating his package into ours.
5. Allen Hill has contributed the multiple time series delay coordinates reconstruction method.

Finally we would like to thank Ragnar Fleischmann and Ulrich Parlitz for all they knowledge of nonlinear dynamics they shared with the authors.

References

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, jan 2017.

[2] George Datseris. **DynamicalSystems.jl** documentation page. https://juliadynamics.github.io/DynamicalSystems.jl/latest/. Accessed: June 25, 2018.

[3] George Datseris. DynamicalSystems.jl: A julia software library for chaos and nonlinear dynamics. *Journal of Open Source Software*, 3(23):598, mar 2018.

[4] Karlheinz Geist, Ulrich Parlitz, and Werner Lauterborn. Comparison of Different Methods for Computing Lyapunov Exponents. *Progress of Theoretical Physics*, 83(5):875–893, 1990.

[5] M. Henon. A two-dimensional mapping with a strange attractor. *Communications in Mathematical Physics*, 50(1):69–77, feb 1976.

[6] Christopher Rackauckas. A comparison of DifferentialEquations.jl with ode solvers from different computer languages. http://www.stochasticlifestyle.com/comparison-differential-equation-solver-suites-matlab-r-julia-python-c-fortran/. Accessed: June 25, 2018.

[7] Christopher Rackauckas and Qing Nie. DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software*, 5, may 2017.

[8] Ch. Skokos, T.C. Bountis, and Ch. Antonopoulos. Geometrical properties of local dynamics in Hamiltonian systems: The Generalized Alignment Index (GALI) method. *Physica D: Nonlinear Phenomena*, 231(1):30–54, jul 2007.