juliacon

Fast, Elegant, Set-oriented Numerical Analysis using GAIO.jl

A. Herwig[1] and O. Junge[1]

[1]Technical University Munich

ABSTRACT

GAIO (Global Analysis of Invariant Objects) is a package for set oriented numerics in dynamical systems. It provides algorithms for discretisation of the Koopman operator among other uses. The Koopman operator has been of much interest in the last decade, since it can be used in order to compute, e.g., almost invariant [6], cyclic [7] and coherent sets [9] (to name just some uses). Originally written in the 90s in C, GAIO has been redesigned in Julia and is concise, while outperforming the original.

Keywords

Dynamical Systems

1. Introduction

A dynamical system can be characterised by a set of global asymptotic behavoirs. These include stability, invariance of certain regions of phase space, as well as statistical behavior of "typical" trajectories. Such topological and statistical questions (among others) can be answered using a *set-oriented* approach.
This paper presents a Julia language implementation of such a set-oriented approach to numerical analysis, encapsulated in the package GAIO.jl. The data structures and the algorithmic interface have been completely redesigned such that the code for the algorithms is very concise and close to their mathematical pseudocode. At the same time, the performance is equal or better than that of the original C and Matlab versions.

2. Dynamical Systems, Invariant Sets and Measures

Consider a continuous map $f : Q \to Q$ on some compact domain $Q \subset \mathbb{R}^d$. We recall some basic notions from dynamical systems literature required for the proceeding example.

2.1 Attractors

A set $A \subset Q$ in phase space is called *forward invariant* if $f(A) \subset A$, *backward invariant* if $f^{-1}(A) \subset A$ [1], and *invariant* if it is both forward and backward invariant. An invariant set is called *attracting* if there exists a neighborhood $U \supset A$ such that for any other neighborhood $V \supset A$ there exists an $N \in \mathbb{N}$ such that the tail $\bigcup_{k \geq N} f^k(U)$ is contained in V.

Definition 1. *The maximal invariant set contained in A is the set*

$$\mathrm{Inv}(A) = \left\{ x \in A \mid \mathcal{O}^{\pm}(x) \subset A \right\}, \tag{1}$$

where $\mathcal{O}^{\pm}(x) = \bigcup_{k \in \mathbb{Z}} f^k(\{x\})$ denotes the full orbit *of x under f.*

It follows immediately from the definition that $\mathrm{Inv}(A)$ contains all other invariant sets which are contained in A. The following proposition is important for its computation.

Proposition 1. *[8] If $A \subset Q$ is forward invariant, then $Inv(A) = \bigcap_{k \geq 0} f^k(A)$.*

This leads to a natural Ansatz for approximation by inductively tightening a cover of the maximal invariant set by finite collections of boxes. Specifically, given a partition $\mathcal{P}$ of Q and a covering $\mathcal{B}$ of the maximal invariant set by elements of $\mathcal{P}$:

(1) Refine $\mathcal{P}$ into a strictly finer partition $\mathcal{P}'$ such that $\mathrm{diam}(\mathcal{P}') \leq \theta \cdot \mathrm{diam}(\mathcal{P})$ for some fixed $\theta < 1$. Let $\mathcal{B}'$ be the (refined) covering of $\mathcal{B}$.
(2) Map the covering forward under f, i.e. cover $f(|\mathcal{B}'|)$ [2] by elements of $\mathcal{P}$. Intersect this covering with $\mathcal{B}'$.

A simple way to partition a hyperractangular domain $Q = [l_1, u_1] \times \ldots \times [l_d, u_d]$ is to divide it into an $N_1 \times \ldots \times N_d$ - element grid of boxes.
Consider the system of ordinary differential equations in 3 dimensions given by

$$\begin{aligned} \dot{x} &= ax + cyz \\ \dot{y} &= dy + bx + hzy \\ \dot{z} &= pz + gxy \end{aligned} \tag{2}$$

where $a, b, c, d, p, f, g, h \in \mathbb{R}$ are parameters [17]. Let f be the time-2 integral of the system, discretized using 20 steps of the standard Runge-Kutta 4th order method.

```
const a, b, c, d = 0.2, -0.01, 1.0, -0.4
const p, g, h = -1.0, -1.0, -1.0

v((x,y,z)) = @. (a,d,p)*(x,y,z) +
```

[1] $f^{-1}(A)$ denotes the preimage

[2] $|\mathcal{B}| = \bigcup_{B \in \mathcal{B}} B$

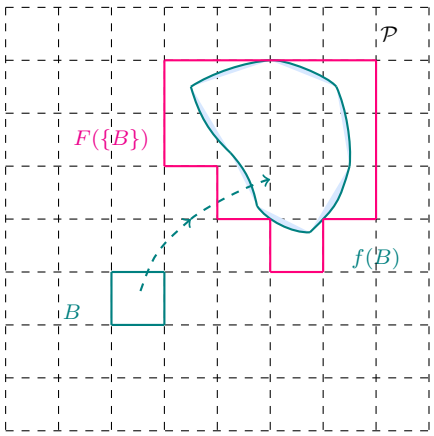

Fig. 1: Induced set-map.

```
                        (0,b*x,0) +
                        (c,h,g)*(y,z,x)*(z,x,y)

f(x) = rk4_flow_map(v, x, 0.01, 20)
```

We discretize the domain $Q = [-5, 5]^3$ into a $2 \times 2 \times 2$ grid.

```
center = (0., 0., 0.)
radius = (5., 5., 5.)
Q = Box(center, radius)
P = BoxPartition(Q, (2, 2, 2))
```

f induces a map $F : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ [3] by the relationship

$$F(\{B\}) = \{R \in \mathcal{P} \mid R \cap f(B) \neq \emptyset\} \tag{3}$$

(see Fig. 1).
GAIO.jl provides multiple methods for approximating set-wise images described in F. One intuitive method is to take a set of test points sampled uniformly within a box, and map the test points each with f. If rigorous images are required, interval arithmetic also lends itself naturally to the above setting.

```
F = BoxMap(:montecarlo, f, Q)
```

We can now implement the above algorithm, see Fig. 2. We begin with a covering of the entire domain. In each iteration we cycle through dimension on which we subdivide the box covering.

```
B = cover(P, :)
C = relative_attractor(F, B, n_steps=21)
```

The computed set is shown in Fig. 3. Many other cell-mapping and subdivision algorithms including (but not limited to)

—(un-)stable manifolds (c.f. Fig. 5)

[3] 2^S denotes the power set of S

```
function relative_attractor(F, B; n_steps)
    for k in 1:n_steps
        B = subdivide(B, k % 3 + 1)
        B = B ∩ F(B)
    end
    return B
end
```

Fig. 2: Relative attractor algorithm written in Julia. Compare with Fig. 10

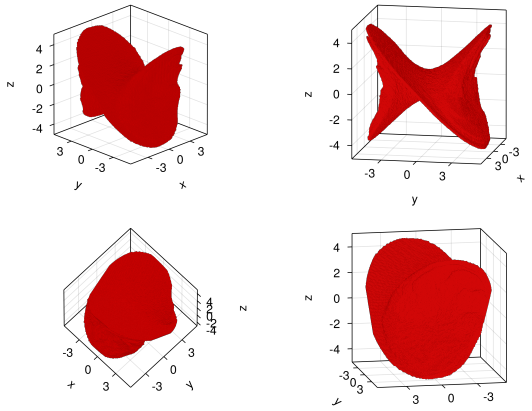

Fig. 3: Computed relative attractor of the system described in Eq. 2

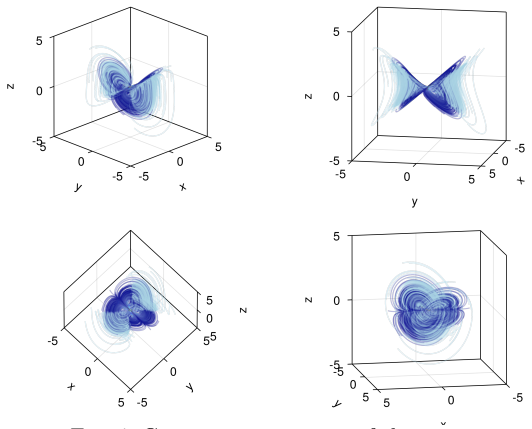

Fig. 4: Common trajectories of the system

—chain-recurrent sets
—Morse decompositions

are similarly easy to implement using GAIO.jl.

2.2 Invariant Measures

f induces a map $f_\sharp : \mathcal{M} \to \mathcal{M}$ [4] known as the *Perron-Frobenius operator* by the relation

[4] $\mathcal{M}$ denotes the space of finite, complex Borel measures on Q

$$f_\sharp\,\mu = \mu \circ f^{-1}, \tag{4}$$

i.e. the *pushforward* of μ under f. This is a bounded, positive linear operator.
A Borel measure μ over Q can be approximated by finite piecewise constant Radon-Nikodym derivatives $d\mu \approx g\,dx$. We can therefore discretize such measures using the same partitions described in section 2.1 by parameterizing

$$\begin{aligned}\mathcal{P} &= \{B_1, B_2, \ldots, B_n\}, \\ \mu &= \sum_{j=1}^{n} g_j \cdot \frac{m(B_j \cap \cdot)}{m(B_j)}.\end{aligned} \tag{5}$$

where m is the Lebesgue meaasure on $\mathbb{R}^d$.
By Krylov-Bogolyubov's theorem [13] there exists a fixed point of $f_\sharp$ or *invariant measure* $f_\sharp\,\mu = \mu$. This can be seen as quantifying the statistics of "typical" trajectories; regions of phase space visited "more often" receive more μ-mass.
The discretized approximation $F_\sharp$ should satisfy the fixpoint equation $F_\sharp\,\mu = \mu$ for each element of the partition, i.e.

$$g_i = \mu(B_i) \overset{!}{=} F_\sharp\,\mu(B_i) = \sum_{j=1}^{n} g_j \cdot \frac{m(B_j \cap f^{-1}(B_i))}{m(B_j)}. \tag{6}$$

Hence the (Markov) matrix

$$(F_\sharp)_{ij} = \frac{m(B_j \cap f^{-1}(B_i))}{m(B_j)} \tag{7}$$

results in the discrete eigenproblem $F_\sharp\,g = g$.
Convergence of the above approximation (known as *Ulam's method*) can be shown for "noisy" dynamical systems by modelling the systems as Markov processes with (arbitrarily small) diffusion. This small noise is already included in the approximation by the box coverings, which is how the method is made rigorous [7]. It remains an open question whether Ulam's method converges to the noiseless Frobenius-Perron operator.
The matrix can be computed similarly to how setwise images are computed for F, e.g. by approximating the transition probabilities $(F_\sharp)_{ij}$ using sample points [12]. We compute the matrix in GAIO.jl for the Perron-Frobenius operator over the unstable manifold of the equillibrium point $(0,0,0)^T$, and use the `eigs` function from Arpack.jl to compute a few leading eigenvalues.

```
P = BoxPartition(Q, (128,128,128))
C = cover(P, (0,0,0))
W = unstable_set(F, C)

F♯ = TransferOperator(F, W, W)
λs, evs, n_converged = eigs(F♯)
```

3. GAIO in the Julia Language

3.1 Philosophy of GAIO.jl

The data structures and algorithms that make up GAIO were originally developed in the 90's and written in C++,

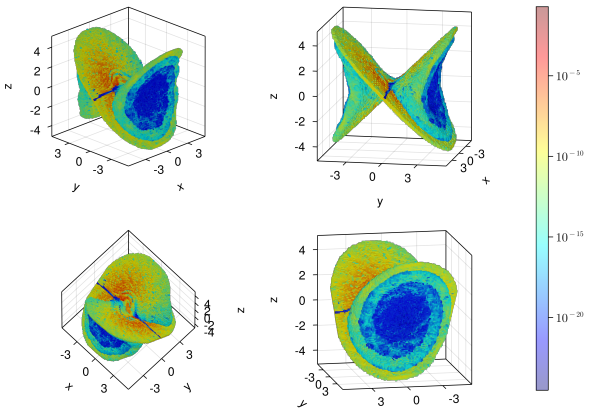

Fig. 5: Invariant measure of $F_\sharp$ supported on the unstable manifold of the equillibrium of Eq. 2 at the origin

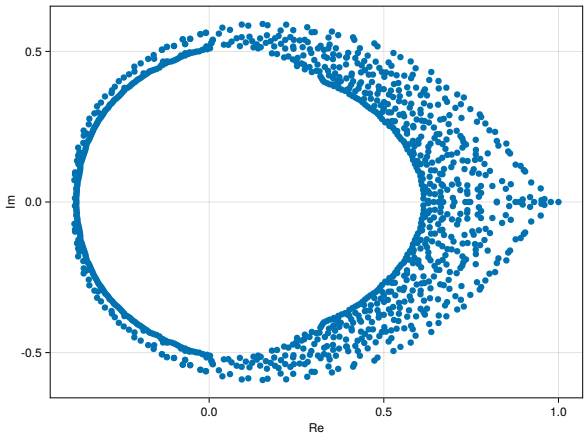

Fig. 6: Largest 1200 eigenvalues of $F_\sharp$, a 389204×389204 matrix

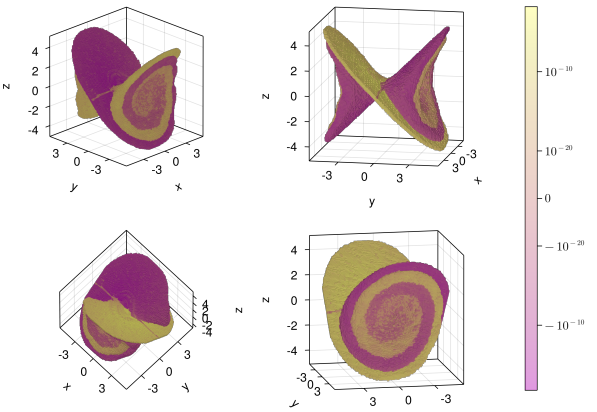

Fig. 7: Eigenmeasure corresponding to the second-largest real eigenvalue (0.978), scaled using Makie.jl's `Symlog10`

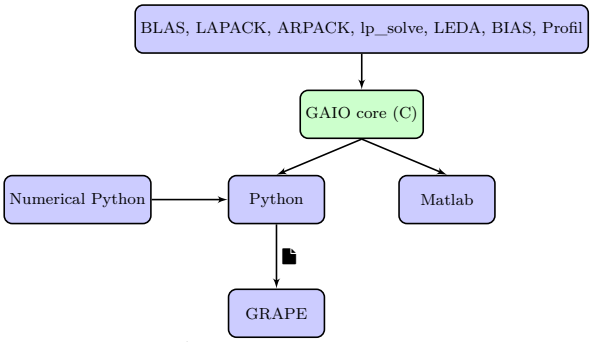

Fig. 8: An earlier software architecture

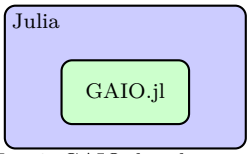

Fig. 9: GAIO.jl architecture

which was changed to C due to maintenance being cumbursome. An interface was written in Python which made use of the numerical python environment (numpy, scipy, etc.), and plotting was done by writing files wihch were read by GRAPE. Due to high demand, a second interface was written in Matlab. It comes as no surprise that this architecture was too convoluted to be sustainable (see Fig. 8). For this reason, the architecture was stripped to just the core (in C) and the programming interface (in Matlab).
GAIO was fully redesigned and rebuilt in the Julia language starting in 2020. The reason to do so (and the decision to use Julia) was twofold:

Solving the two (or three…) language problem. The cell mapping techniques require far too many function evaluations to be reasonable in an interpreted language. As with many scientific computing applications, compilation into fast machine code is necessary if one wishes to apply the methods to any real-scale problem. However, the primary userbase is mathematicians, not computer scientists. The code must therefore be easy to read, and it should be easy to convert pseudocode into usable software written in the style of a scripting language. Furthermore, the old software architecture had become difficult to maintain, even after being streamlined to just C and Matlab.

No longer needing to tradeoff simplicity and transparency. The speed of a modern compiled language on today's hardware comes with an added benefit: code does not *need* to be hyper-optimized - "mostly" optimized is good enough. GAIO originally used a tree data structure to represent partitions of phase space in such a way that each box was represented by a precise bitstring, maximally utilizing every byte of memory. While this was efficient, it made the data difficult to decipher. More generally, a decision always had to be made. Either

(a) complexity is hidden behind convenience functions, sacificing knowledge of "what's actually going on", or,

```
function relative_attractor(tree, f, steps)
    for i = 1:steps,
        tree.set_flags('all', to_be_subdivided);
        tree.subdivide(to_be_subdivided);
        b = tree.boxes(-1);
        while (~isempty(b))
            c = b(1:d);
            r = b(d+1:2*d);
            P = X*diag(r) + ones(size(X))*diag(c);
            tree.set_flags(f(P)', hit);
            b = tree.next_box(-1);
        end
        tree.remove(hit);
    end
```

Fig. 10: Relative attractor algorithm in Matlab. Compare with Fig. 2

(b) code is "transparent", allowing for extensibility while sacrificing clarity.

GAIO.jl was written with the knowledge that today, this just is not strictly necessary anymore. A memory-efficient tree structure is still offered, but the primary technique for partitioning phase space is much simpler: just use a grid with Cartesian indices - *keep it simple, stupid.*

3.2 Fitting into Julia's active scientific computing ecosystem

Another reason for the decision to use Julia for GAIO.jl is the large (and growing) open source scientific computing community. GAIO.jl makes extensive use of linear algebra routines (LinearAlgebra.jl [1]), sparse arrays (SparseArrays.jl [1], Arpack.jl [14]), graph/network routines (MatrixNetworks.jl [15]), and numerical integrators (DifferentialEquations.jl [16]), plotting ecosystems (Makie.jl [3], Plots.jl [2]), among others.
A particular example of the effectiveness of such an open source scientific computing model is CUDA.jl. The sample-point methods for cell mapping are examples of so-called *embarrassingly parallel* [10] problems. It has therefore been a long-standing desire to utilize the GPU to perform such massively parallel algorithms in GAIO. However, under the previous architecture this would have to be written in CUDA's native C interface, which suffers the issues mentioned in the preceeding section.
This is solved by the amazing work done to create CUDA.jl. The cell mapping can be written as a generic kernel whose length (in lines of code) is the same as the standard code, meaning the algorithms that make up GAIO.jl can receive up to a 200-fold [11] performance boost without ever sacrificing readability.
Furthermore, the open-source nature of Julia's scientific computing ecosystem has helped in programming GAIO.jl countless times. Since code is freely available, code reuse is common among the Julia community. Problems which have already been solved by other programmers need not be solved again.

4. Conclusion

The Julia package GAIO.jl is introduced via an example of a three dimensional dynamical system. The software has been redesigned and rebuilt from the ground up to balance high performance and elegance, no longer needing to rely on two languages to do so. Future work is planned to use these structures e.g. for homology computation in cubical complexes, as well as to even more tightly integrate into the existing scientific computing ecosystem e.g. DynamicalSystems.jl [4, 5] in the Julia programming language.

5. 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, 2017. doi:10.1137/141000671.

[2] Simon Christ, Daniel Schwabeneder, Christopher Rackauckas, Michael Krabbe Borregaard, and Thomas Breloff. Plots.jl – a user extendable plotting api for the julia programming language. *Journal of Open Research Software*, 2023. doi:https://doi.org/10.5334/jors.431.

[3] Simon Danisch and Julius Krumbiegel. Makie.jl: Flexible high-performance data visualization for Julia. *Journal of Open Source Software*, 6(65):3349, 2021. doi:10.21105/joss.03349.

[4] George Datseris. Dynamicalsystems.jl: A julia software library for chaos and nonlinear dynamics. *Journal of Open Source Software*, 3(23):598, mar 2018. doi:10.21105/joss.00598.

[5] George Datseris and Ulrich Parlitz. *Nonlinear dynamics: A concise introduction interlaced with code*. Springer Nature, Cham, Switzerland, 2022. doi:10.1007/978-3-030-91032-7.

[6] Michael Dellnitz and Oliver Junge. Almost invariant sets in chua's circuit. *International Journal of Bifurcation and Chaos*, 07(11):2475–2485, 1997. doi:10.1142/S0218127497001655. https://-doi.org/10.1142/S0218127497001655.

[7] Michael Dellnitz and Oliver Junge. On the approximation of complicated dynamical behavior. *SIAM Journal on Numerical Analysis*, 36(2):491–515, 1999. doi:10.1137/S0036142996313002. https://-doi.org/10.1137/S0036142996313002.

[8] Robert W Easton. Dynamical Systems. In *Geometric Methods for Discrete Dynamical Systems*. Oxford University Press, 02 1998. doi:10.1093/oso/9780195085457.003.0002. https://academic.oup.com/book/0/chapter/422180843/chapter-pdf/52433686/isbn-9780195085457-book-part-2.pdf.

[9] Gary Froyland and Kathrin Padberg-Gehle. Almost-invariant and finite-time coherent sets: Directionality, duration, and diffusion. In Wael Bahsoun, Christopher Bose, and Gary Froyland, editors, *Ergodic Theory, Open Dynamics, and Coherent Structures*, pages 171–216, New York, NY, 2014. Springer New York.

[10] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. Elsevier, 2020.

[11] April Herwig. Gaio.jl: Set-oriented methods for approximating invariant objects, and their implementation in julia, 2022. Thesis.

[12] Oliver Junge. *Rigorous discretization of subdivision techniques*, pages 916–918. World Scientific, 2000.

[13] Nicolas Kryloff and Nicolas Bogoliouboff. La theorie generale de la mesure dans son application a l'etude des systemes dynamiques de la mecanique non lineaire. *Annals of Mathematics*, 38(1):65–113, 1937.

[14] R.B. Lehoucq, D.C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics, 1998.

[15] Huda Nassar, David Gleich, Katharine Hyatt, MeherChaitanyaP, Austin Benson, Tony Kelman, Ole Kröger, Nicole Eikmeier, Elliot Saba, and Jungsoo Kim. Matrixnetworks.jl v1.0.2, 2021.

[16] Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. doi:10.5334/jors.151. Exported from https://app.dimensions.ai on 2019/05/05.

[17] Zenghui Wang, Yanxia Sun, Barend Jacobus van Wyk, Guoyuan Qi, and Michael Antonie van Wyk. A 3-d four-wing attractor and its analysis. *Brazilian Journal of Physics*, 39(3):547–553, Sep 2009. doi:10.1590/S0103-97332009000500007.